Typeful Programming

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Abstract

There exists an identifiable programming style based on the widespread use of type information handled through mechanical typechecking techniques.

This *typeful* programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming.

The main purpose of this paper is to show how typeful programming is best supported by *sophisticated* type systems, and how these systems can help in clarifying programming issues and in adding power and regularity to languages.

We start with an introduction to the notions of types, subtypes and polymorphism. Then we introduce a general framework, derived in part from constructive logic, into which most of the known type systems can be accommodated and extended. The main part of the paper shows how this framework can be adapted systematically to cope with actual programming constructs. For concreteness we describe a particular programming language with advanced features; the emphasis here is on the combination of subtyping and polymorphism. We then discuss how typing concepts apply to large programs, made of collections of modules, and very large programs, made of collections of large programs. We also sketch how typing applies to system programming; an area which by nature escapes rigid typing. In summary, we compare the most common programming styles, suggesting that many of them are compatible with, and benefit from, a typeful discipline.

Appears in: Formal Description of Programming Concepts, E.J.Neuhold, M.Paul Eds., Springer-Verlag, 1991.

SRC Research Report 45, May 24, 1989. Revised January 1, 1993.

Contents

1. Introduction

2. Typeful languages

2.1. Relevant concepts

2.2. Theory and practice

2.3. Why types?

2.4. Why subtypes?

2.5. Why polymorphism?

3. Quantifiers and subtypes

3.1. Kinds, types, and values

3.2. Signatures and bindings

3.3. The Quest language

4. The kind of types

4.1. Introduction

4.2. Basic and built-in types

4.3. Function types

4.4. Tuple types

4.5. Option types

4.6. Auto types

4.7. Recursive types

4.8. Mutable types

4.9. Exception types

5. Operator kinds

5.1. Type operators

5.2. Recursive type operators

6. Power kinds

6.1. Tuple subtypes

6.2. Option subtypes

6.3. Record and variant types

6.4. Higher-order subtypes

6.5. Bounded universal quantifiers

6.6. Bounded existential quantifiers

6.7. Auto subtypes

6.8. Mutable subtypes

6.9. Recursive subtypes

7. Large programs

7.1. Interfaces and modules

7.2. Manifest types and kinds

7.3. Diamond import

8. Huge programs

8.1. Open systems

8.2. Closed systems

8.3. Sealed systems

9. System programs

9.1. Dynamic types

9.2. Stack allocation

9.3. Type violations

10. Conclusions

10.1. This style

10.2. Other styles

10.3. Acknowledgments

11. Appendix

11.1. Syntax

11.2. Type rules

11.3. Library interfaces

1. Introduction

There exists an identifiable programming style which is based on the widespread use of type information, and which relies on mechanical and transparent typechecking techniques to handle such information. This *typeful* programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming. Hence, it makes sense to discuss this programming style in a way that is relatively independent of particular flow-of-control paradigms, such as the ones just mentioned.

Let us see more precisely what typeful programming is, and what it is not. The widespread use of type information is intended as a partial specification of a program. In this sense, one can say that typeful programming is just a special case of program specification, and one can read *type* as a synonym for *specification*, and *typechecking* as a synonym for *verification* in this discussion. This view fits well with the *types as propositions* paradigm of axiomatic semantics, and the *propositions as types* paradigm of intuitionistic logic.

However, typeful programming is distinct from program specification in some fundamental ways. As we noted already, there must be a mechanical way of verifying that type constraints are respected by programs. The slogan here is that *laws should be enforceable:* unchecked constraining information, while often useful for documentation purposes, cannot be relied upon and is very hard to keep consistent in large software systems. In general, systems should not exhibit constraints that are not actively enforced at the earliest possible moment. In the case of typechecking the earliest moment is at compile-time, although some checks may have to be deferred until run-time. In contrast, some specifications can be neither typechecked nor deferred until run time, and require general theorem-proving (e.g., in verifying the property of being a constant function).

Another emphasis is on transparent typing. It should be easy for a programmer to predict reliably which programs are going to typecheck. In other words, if a program fails to typecheck, the reason should be apparent. In automatic program verification, it may be hard in general to understand why a program failed to verify; at the current state of the art one may need to have a deep understanding of the inner workings of the verifier in order to correct the problem.

The scope of typeful programming, as defined above, is limited. Typechecking will never merge with program verification, since the former requires mechanical checking and the latter is undecidable. We may however attempt to reduce this gap, on the one hand by integrating specifications as extensions of type systems, and on the other hand by increasing the sophistication of type systems. We intend to show that the latter road can already lead us quite far towards expressing program characteristics.

The scope of typeful programming would also have another major limitation if we required programs to be completely statically typed. A statically typed language can be Turing-complete, but still not be able to express (the type of) an embedded *eval* function; this is important in many areas, and is just a symptom of similar problems occurring in compiler bootstrapping, in handling persistent data, etc. There are interesting ways in which statically checked languages can be extended to cover *eval* functions and other similar situations. The flavor of typeful programming is preserved if these extensions involve run-time type checks, and if these dynamic checks have a good relationship with corresponding static checks. Hence, typeful programming advocates *static typing*, as much as possible, and *dynamic typing* when necessary; the strict observance of either or both of these techniques leads to *strong typing*, intended as the absence of unchecked run-time type errors.

The main purpose of this paper is to show how typeful programming is best supported by *sophisticated* type systems, and how these systems can help in clarifying programming issues and in adding power and regularity to languages. To a minor extent, the purpose of the paper is to motivate the use of typing in programming, as is done in the first few subsections, but in fact we take almost for granted the benefits of *simple* type systems.

How should we go about explaining and justifying sophisticated type systems? One expository approach would involve listing and comparing all the different language design problems and solutions that have led to increasingly powerful notions of typing. This approach, however, may produce a very fragmented description of the field, and might not reveal much more information than can be gathered from language manuals and survey articles. Comparisons between existing languages may help in designing new languages, but may equally easily help in perpetuating existing design deficiencies.

Another expository approach would rigorously describe the formal system underlying powerful forms of typeful programming. In fact, some such systems have been around for at least fifteen years with only partial impact on programming. Many of the ideas had to be rediscovered in different contexts, or had to be translated into languages in order to become understandable to a general computer science audience. The problem here is that it is not easy to extract practical languages from formal systems. Moreover, formal expositions tend to highlight hard theoretical properties of small formalisms, instead of engineering properties of large languages with many interacting features.

The approach we take consists in underlining basic concepts in the concrete context of a single programming language. Advances in semantics and type theory have revealed much hidden unity in features used in practical languages; these features often constitute special cases, restricted versions, engineering compromises, combinations, or even misunderstandings of more fundamental notions. Our example language attempts to cover and integrate most of the features found in typeful languages, while providing a direct mapping to the basic concepts. Inevitably, some common notions are transmuted or left out, and the exposition is tinted by one particular language style and syntax. Hopefully, readers will see though this thin disguise of basic concepts more easily than they would through a formal system or a series of existing languages.

We start with a general introduction to typeful languages, then we switch to a single concrete language. The central part of the paper has the form of a language manual with many motivating examples. In the final sections we return to a more general look at properties and uses of typeful languages.

2. Typeful languages

In this section we discuss some properties of typeful languages and we justify type systems from a software methodology point of view.

2.1. Relevant concepts

We shall try to cover, in a unified type-theoretical framework, a number of concepts that have emerged in various programming languages. The emphasis here is on language constructs such as the following that have been developed for writing large or well-structured programs.

Higher-order functions (functions which take or return other functions) are an important structuring tool; they can help in abstracting program behavior by enhancing the abstraction power of ordinary functions. They are present in most common languages, but are often severely limited. Many languages prevent them from being declared within other functions, and most languages forbid them from being returned by other functions. Higher-order functions form the essence of the λ-calculus [Barendregt 85]. It seems that the first programming language to define them correctly was Iswim [Landin 66] and today the major language to implement them correctly is Standard ML [Milner 84]. Even in their weaker forms, they are considered an essential structuring tool in many areas of system programming, such as operating systems.

Abstract types come from the desire to hide irrelevant program information and to protect internal program invariants from unwanted external intervention. An abstract type is an ordinary type along with a set of operations; the structure of the type is hidden and the provided operations are the only ones authorized to manipulate objects of that type. This notion was well embedded in CLU [Liskov et al. 77] [Liskov Guttag 86] and formed the basis for the later development of modular programming languages. (This notion of abstraction is more restrictive than the general notion of algebraic abstract types [Futatsugi Goguen Jouannaud Meseguer 85].)

Polymorphism is the ability of a function to handle objects of many types [Strachey 67]. In ad hoc polymorphism a function can behave in arbitrarily different ways on objects of different types. We shall ignore this view here, and consider only *generic* polymorphism where a function behaves in some uniform way over all the relevant types. The two forms of generic polymorphism are *parametric* polymorphism, where uniform behavior is embodied by a type parameter, and *subtype* polymorphism, where uniform behavior is embodied by a subtype hierarchy. The first and prototypical language for parametric polymorphism is ML; the mechanisms we adopt later are more closely related to Russell [Demers Donahue 79] and Pebble [Burstall Lampson 84]. The notion of subtype polymorphism we consider first appeared in Simula67 [Dahl Nygaard 66] (single inheritance) and is more closely related to the one in Amber [Cardelli 86] (multiple inheritance). A major aim here is to unify parametric and subtype polymorphism in a single type system.

Subtyping is a relation between types, similar to the relation of containment between sets. If A is a subtype of B, then any object of type A is also an object of type B. In other words, an object of A has all the properties of an object of B. The latter statement is close to the definition of *inheritance* in object-oriented languages (although inheritance is more strictly related to behavior than typing). Subtyping over record-like types can emulate many properties of inheritance, but subtyping is a more abstract notion, because it applies to all types, and because it does not involve a built-in notion of *methods*.

Modules and interfaces were introduced in Mesa [Mitchell Maybury Sweet 79], and then perfected in Modula2 [Wirth 83]. Standard ML embodies today's most advanced module system. Modules are the major structuring concept, after functions, in modern languages. They are very similar to abstract types, but add the notion of imported identifiers (which are declared in other interfaces) thereby evading the strict block-structure of statically scoped languages. Interfaces contain the names and types of (some of) the operations defined in a module, along with names of abstract types. Since modules and interfaces are self-contained units (i.e., they refer to no external identifiers, except other interfaces), they can be used as units of compilation.

2.2. Theory and practice

The conceptual framework for typeful programming is to be found in various theories of typed λ-calculi [Reynolds 74] [Martin-Löf 80]; in particular, we were inspired by Girard's system $F\omega$ [Girard 71] and by the theory of *Constructions* [Coquand Huet 85] [Hyland Pitts 87]. This collection of theories, generically referred to as *type theory*, studies very expressive type structures in the framework of constructive logic.

More often than not, these theoretical structures have direct correspondence in programming constructs; this is not accidental, since computing can be seen as a branch of constructive logic. Similar or identical type structures and programming constructs have often been discovered independently. One can also extrapolate this correspondence and turn it into a predictive tool: if a concept is present in type theory but absent in programming, or vice versa, it can be very fruitful to both areas to investigate and see what the corresponding concept might be in the other context.

Theoretical understanding can greatly contribute to the precision and simplicity of language constructs. However, a programming language is not just a formal system. Many practical considerations have to be factored into the design of a language, and a perfect compromise between theory and practice is rarely achieved. Theorists want something with nice properties. Engineers want something useful with which to build systems that work most of the time. A good language has nice properties, is useful, and makes it easy to build systems that work all the time.

Here is a list of practical considerations which should enter the mind of any language designer. The enforcement of these considerations may cause divergency between practical languages and their theoretical background.

Notation. Notation is very important: it should help express the meaning of a program, and not hide it under either cryptic symbols or useless clutter. Notation should first be easy to *read* and look at; more time is spent reading programs than writing them. As a distant second, notation should be easy to *write*, but one should never make notation harder to read in order to make it easier to write; environment tools can be used to make writing more convenient. Moreover, if large programs are easy to read for a human, they are likely to be easy to parse for a computer (but not vice versa, e.g. for Lisp).

Scale. One should worry about the organization of large programs, both in terms of notation and in terms of language constructs. Large programs are more important than small programs, although small programs make cuter examples. Moreover, most programs either go unused or become large programs. Important scale considerations are whether one can reuse parts of a large program, and whether one can easily extend a large program without having to modify it. A surprisingly common mistake consists in designing languages under the assumption that only small programs will be written; for example languages without function parameters, without proper notions of scoping, without modules, or without type systems. If widely used, such languages eventually suffer conceptual collapses under the weight of ad hoc extensions.

Typechecking. The language should be typable so that typechecking is feasible. In addition, typechecking should be relatively efficient and errors easy to understand. If at all possible, typechecking should be decidable; otherwise good and predictable heuristics should exist.

Translation to machine language. This should be possible (some languages can only be interpreted). The translation process should be relatively efficient, and the produced code should have a simple relationship to the source program.

Efficiency. The translated code should be efficient; that is, one should avoid language constructs that are hard to implement efficiently or require excessive cleverness on the part of the compiler. Clever optimizations are rarely if ever implemented, and they don't compose easily.

Generality. A useful language (sometimes called a "real" language) should be able to support building a wide variety of systems. For this it should be both theoretically complete and practically complete. The former condition can be expressed as *Turing-completeness:* the ability of a language to express every computable function. The latter condition is harder to quantify, but here are some criteria which make a language more and more "real". Practical completeness is the ability of a language to conveniently express its own (a) interpreter; (b) translator; (c) run-time support (e.g. garbage collector); (d) operating system. Of course this classification is very subjective, but in each case a person claiming a given classification for a given system should be willing to make personal use of the resulting system. The language we describe later falls mainly in category (b), but we shall discuss how to extend it to category (c).

2.3. Why types?

There are many different reasons for having types; these have been discussed in the literature (for a review see [Cardelli Wegner 85]). Here we focus on a particular reason which is relevant to the practical design considerations described in the previous section. Types here are motivated from a *software methodology* point of view:

Types are essential for the ordered evolution of large software systems.

Large software systems are not created in a day; they evolve over long periods of time, and may involve several programmers. The problem with evolving software systems is that they are (unfortunately):

• *Not correct:* people keep finding and fixing bugs, and in the process they introduce new ones. Of course the goal is to make the system correct, but as soon as this is achieved (if ever), one immediately notices that the system is:

• *Not good enough:* either the system is too slow, or it uses too much space. Of course the goal is to make the system efficient, but this requires new changes and we are back to the previous point. If, eventually, the system becomes good enough, then very likely it is:

• *Not clean enough:* since future evolution is always to be considered, the system may require restructuring for better future maintenance; this again can introduce new bugs and inefficiencies. However, if a system ever becomes correct, efficient, and well-structured, this is the ideal moment to introduce new functionality, because systems are always:

• *Not functional enough:* as new features are added, the cycle begins again. In the unlikely event that all these goals are achieved, the system is by now usually obsolete. Evolution is the *normal* situation for large software systems.

This picture of evolving software suggests the mildly paradoxical concept of *software reliability*. Normally one says that a program is either correct or incorrect; either it works or it does not work. But like any complex organism, a large software system is always in a transitional situation where most things work but a few things do not, and where fewer things work immediately after an extensive change. As an analogy, we say that hardware is *reliable* if it does not break too often or too extensively in spite of *wear*. With software, we say that an evolving system is *reliable* if it does not break too often or too extensively in spite of *change*. Software reliability, in this sense, is always a major concern in the construction of large systems, maybe even the main concern.

This is where type systems come in. Types provide a way of *controlling evolution*, by partially verifying programs at each stage. Since typechecking is mechanical, one can guarantee, for a well designed language, that certain classes of errors cannot arise during execution, hence giving a minimal degree of confidence after each change. This elimination of entire classes of errors is also very helpful in identifying those problems which cannot be detected during typechecking.

In conclusion, types (and typechecking) increase the *reliability* of evolving software systems.

2.4. Why subtypes?

As in the previous section, we look at subtypes from a *software methodology* point of view:

Subtypes are essential for the ordered extension of large software systems.

When a software system is produced, there are usually many users who want to extend its functionality in some way. This could be done by asking the original developers to modify the system, but we have seen that this causes unreliability. Modifications could be carried out directly by the users, but this is even worse, because users will have little familiarity with the internals of the system. Moreover, changes made by the users will make the system incompatible with future releases of the system by the original developers.

Hence, various degrees of customizability are often built into systems, so that they can be extended from the *outside*. For example, the procedural behavior of a system can be extended by allowing users to supply procedures that are invoked in particular situations.

Subtypes provide another very flexible extension mechanism that allows users to extend the data structures and operations provided by the system. If the system provides a given type, users may create a subtype of it with specialized operations. The advantage now is that the old system will recognize the new subtypes as instances of the old types, and will be able to operate on them.

Hence, subtypes increase the reliability of systems because they provide a way of increasing functionality without having to change the original system. Subtyping by itself, however, does not guarantee that the new extensions will preserve the system's internal invariants. The extended system will be more reliable, and more likely compatible with future releases of the base system, if some abstraction of the base system has been used in building the extended system. Hence, it is very important that the subtyping and abstraction features of a language interact nicely.

Through the construction of type hierarchies, subtyping also provides a way of organizing data which benefits the structuring of large systems independently of reliability considerations.

2.5. Why polymorphism?

We have seen how types are important for the construction of large systems, but type systems are sometimes too constraining. We can imagine a type system as a set of constraints imposed on an underlying untyped language. The type system imposes discipline and increases reliability, but on the other hand it restricts the class of correct programs that can be written. A good type system, while providing static checking, should not impose excessive constraints.

We now describe a simple untyped language so we can discuss some kinds of untyped flexibility we may want to preserve in a typed language. Here `x` are variables, `k` are constants, `l` are labels, and `a`,`b` are terms:

Construct	Introduction	Elimination
variable	`x`	
constant	`k`	
function	**fun**`(x)b`	`b(a)`
tuple	**tuple** l_1=a_1..l_n=a_n **end**	`b.l`

The *introduction* column shows ways of creating instances of a given construct; the *elimination* column shows ways of using a given construct. Most programming features can be classified this way.

This is a very flexible language, since it is untyped. But there is a problem: the features in the elimination column may *fail*; an application `b(a)` fails if `b` is not a function, and a tuple selection `b.l` fails if `b` is not a tuple, or if it is a tuple without a field labeled `l`.

These failure points are exactly what makes software unreliable, because they can occur unpredictably in an untyped language. The purpose of a type system is to prevent them from happening by statically analyzing programs.

There are two special kinds of flexible behavior we want to preserve in a type system. In the untyped language we can apply a function to elements of different types and in many situations this will not cause failures, for example when applying a pairing function:

```
(fun(x) tuple first=x second=x end)(3)
(fun(x) tuple first=x second=x end)(true)
```

A type system preserving this flexibility is said to support *parametric polymorphism*, because this can be achieved by passing types as parameters, as we shall see later regarding more interesting examples.

A different kind of flexibility is exhibited by functions operating on tuples; if a function accepts a given tuple without failing, it also accepts an extended version of that tuple, because it just ignores the additional tuple components:

```
(fun(x)x.a)(tuple a=3 end)
(fun(x)x.a)(tuple a=3 b=true end)
```

A type system preserving this kind of flexibility is said to support *subtype polymorphism* because this can be achieved, as we shall see, by defining a subtype relation on tuples.

Our type system will eliminate the failure points above, while preserving both kinds of flexibility.

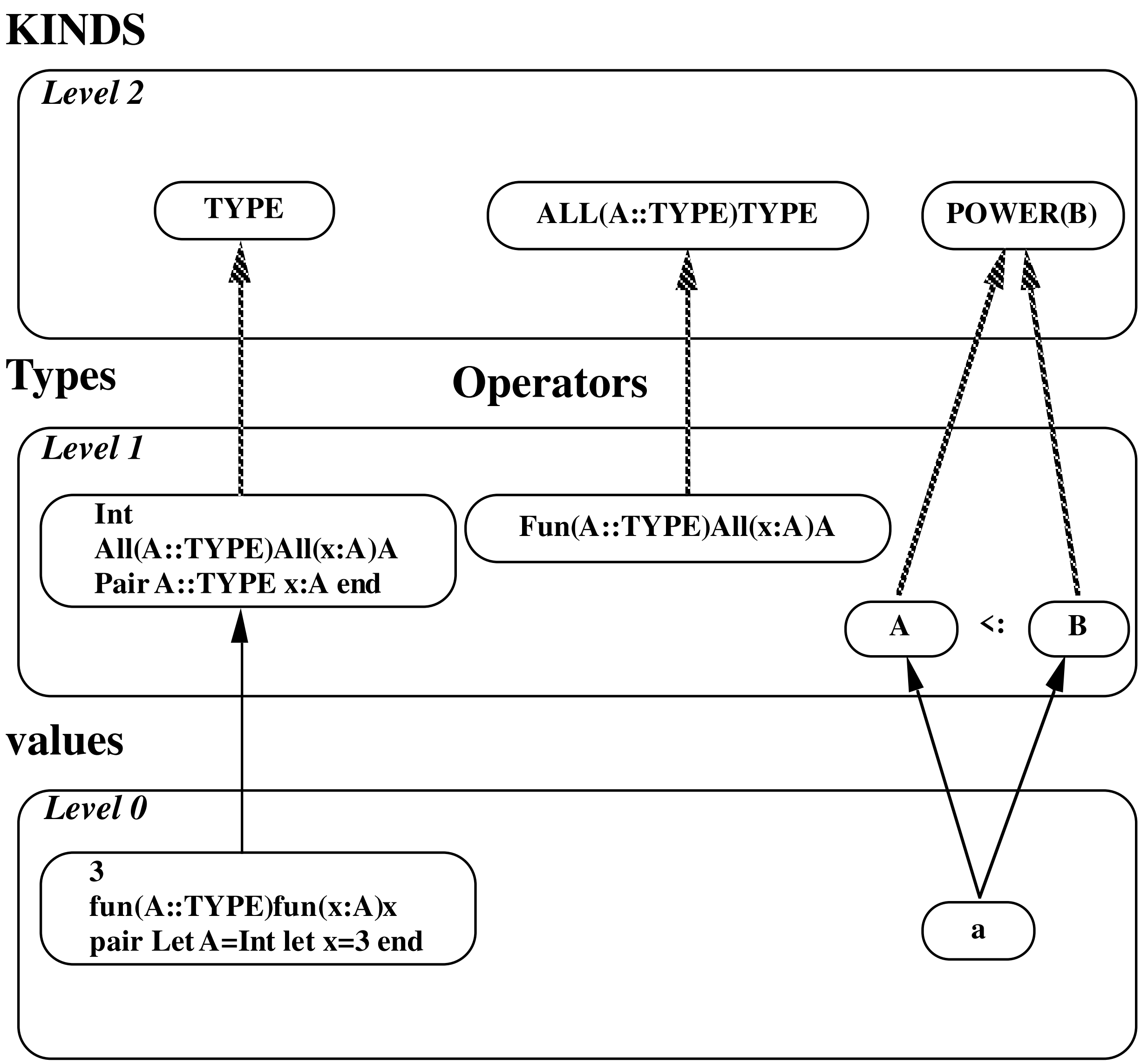
3. Quantifiers and subtypes

In this section we begin introducing one particular type system characterized by a three-level structure of entities. This structure has driven many of the design decisions that have been made in a language called Quest, which is used in the following sections to illustrate basic and advanced typing concepts.

3.1. Kinds, types, and values

Ordinary languages distinguish between *values* and *types*. We can say that in this case there are two *levels* of entities; values inhabit *Level 0*, while types inhabit *Level 1*.

We shall be looking at a language, Quest, that adds a third level above types; this is called the level of *kinds*, or *Level 2*. Kinds are the "types" of types, that is they are collections of types, just like types are collections of values. The collection of all types is the *kind* `TYPE` (which is not a type). We use `a:A` to say that a value `a` has a type `A`, and `A::K` to say that a type `A` has a kind `K`.



In common languages, two levels are sufficient since the type level is relatively simple. Types are generated by ground types such as integers, and by operators such as function space and cartesian product. Moreover, there is normally only one (implicit) kind, the kind of all types, so that a general notion of kinds is not necessary.

Types in the Quest language have a much richer structure, including type variables, type operators, and a notion of type computation. In fact, the Quest type level is itself a very expressive λ-calculus, as opposed to a simple algebra of type operators. Moreover, this λ-calculus is "typed", where the "types" here are the kinds, since we are one level "higher" than normal. This should become clearer by the end of this section.

The level structure we adopt is shown in the level diagram. Values are at Level 0. At Level 1 we have types, intended as sets of values, and also type operators, intended as functions mapping types to types (e.g. the `List` type operator is a function mapping the type `Int` to the type of lists of integer). At Level 2 we have kinds which are the "types" of types and operators; intuitively kinds are either sets of types or sets of operators.

In more detail, at the value level we have *basic values*, such as `3`; *(higher-order) functions*, such as **`fun`**`(x:A)x`; *polymorphic functions*, such as **`fun`**`(A::TYPE)`**`fun`**`(x:A)x`; *pairs* of values, such as **`pair`** `3 true` **`end`**; and *packages* of types and values, such as **`pair Let`** `A=Int` **`let`** `x:A=3` **`end`**.

Note that already at the value level types are intimately mixed with values, and not just as specifications of value components: polymorphic functions take types as arguments, and packages have types as components. This does not mean that types are first-class entities which can play a role in value computations, but it does mean that types are not as neatly separated from values as one might expect.

At the type level we find, first of all, the types of values. So we have *basic types*, such as `Int`; *function types*, such as **`All`**`(x:A)A` (functions from `A` to `A`, usually written `A->A`); *polymorphic types*, such as **`All`**`(A::TYPE)`**`All`**`(x:A)A`; *pair types*, such as **`Pair`** `x:Int y:Bool` **`end`**; and *abstract types*, such as **`Pair`** `A::TYPE x:A` **`end`**.

At the type level we also find *operators*, which are functions from types to types (and from operators to operators, etc.) but which are not types themselves. For example **`Fun`**`(A::TYPE)`**`All`**`(x:A)A` is an operator that given a type `A` returns the type **`All`**`(x:A)A`. Another example is the parametric `List` operator, which given the type `Int` produces the type `List(Int)` of integer lists.

At the kind level we have the kind of all types, `TYPE`, and the kinds of operators, such as **`ALL`**`(A::TYPE)TYPE`, which is the kind of `List`.

The right-hand side of the level diagram illustrates subtyping. At the type level we have a subtyping relation, written `A<:B` (`A` is a subtype of `B`); if a value has type `A`, then it also has type `B`. At the kind level we have the `POWER` kinds for subtyping; if `B` is a type, then `POWER(B)` is the kind of all subtypes of `B`, in particular we have `B::POWER(B)`. Moreover, if `A<:B` then `A::POWER(B)`, and we identify these two notions.

We also have a relation of *subkind*, written `<::` (never actually used in programs), such that if `A<:B` then `POWER(A)<::POWER(B)`, and also `POWER(A)<::TYPE` (the latter means that the collection of subtypes of `A` is contained in the collection of all types).

A structure such as the one in the level diagram is characterized by the *quantifiers* it incorporates. (These are deeply related to the quantifiers of logic [Martin-Löf 80], but we shall not discuss this connection here.) A quantifier is intended as a type construction that binds variables. There are four possible *universal* quantifiers (classifying function-like objects) and four possible *existential* quantifiers (classifying pair-like objects). These basic quantifiers are *binary*: they involve a variable ranging over a first type or kind in the scope of a second type or kind. The number four above comes from all the possible type/kind combinations of such binary operators; the following examples should make this clear.

In our level structure we have three universal quantifiers (two flavors of **`All`** plus **`ALL`**), and two existential quantifiers (two flavors of **`Pair`**, of which only one appears in the diagram):

- **All**(*x*:*A*)*B*
  This is the *type* of functions from values in *A* to values in *B*, where *A* and *B* are types. The variable *x* can appear in *B* only in special circumstances, so this is normally equivalent to the function space *A*->*B*. Sample element: **fun**(x:Int)x.
- **All**(*X*::*K*)*B*
  This is the *type* of functions from types in *K* to values in *B*, where *K* is a kind, *B* is a type, and *X* may occur in *B*. Sample element: **fun**(A::TYPE)**fun**(x:A)x.
- **ALL**(*X*::*K*)*L*
  This is the *kind* of functions from types in *K* to types in *L*, where *K* and *L* are kinds, and *X* may occur in *L*. Sample element: **Fun**(A::TYPE)A.
- **Pair** *x*:*A* *y*:*B* **end**
  This is the *type* of pairs of values in *A* and values in *B*, where *A* and *B* are types. The variable *x* can appear in *B* only in special circumstances, so this is normally equivalent to the cartesian product *A*#*B*. Sample element: **pair let** x=true **let** y=3 **end**.
- **Pair** *X*::*K* *y*:*B* **end:**
  This is the *type* of pairs of types in *K* and values in *B*, where *K* is a kind, *B* is a type, and *X* may occur in *B*. Sample element: **pair let** X=Int **let** y:X=3 **end**.

For symmetry, we could add a third existential quantifier **PAIR** *X*::*K* *Y*::*L* **end** (a *kind*), but this turns out not to be very useful. There are very specific reasons for not admitting the other two possible quantifiers (the *kind* of functions from values to types, and the *kind* of pairs of values and types, where the types may depend on the values) or the unrestricted forms of **All**(*x*:*A*)*B* and **Pair** *x*:*A* *y*:*B* **end** where *x* may occur in *B*. These quantifiers make compilation very problematic, and they are very regretfully but deliberately omitted.

Before proceeding, we should summarize and clarify our notation.

NOTATION

- We use lower case identifiers (with internal capitalization for compound names) for Level 0, capitalized identifiers for Level 1, and all caps for Level 2. For example, **let** introduces values, **Let** introduces types, and **DEF** introduces kinds.

- We use the Courier font for programs, Courier Bold for some program keywords, and Courier Italic for program metavariables. In particular: *x*,*y*,*z* range over value variables; *a*,*b*,*c* range over value terms; *X*,*Y*,*Z* range over type (or operator) variables; *A*,*B*,*C* range over type (or operator) terms; *U*,*V*,*W* range over kind variables; *K*,*L*,*M* range over kind terms. Note that this does not apply to program (non-meta) variables: for example, we often use A (Roman) as a type variable.

- We use *a*:*A* to say that a value *a* has a type *A*; *A*::*K* to say that a type *A* has a kind *K*; *A*<:*B* to say that *A* is a subtype of *B*; and *K*<::*L* to say that *K* is a subkind of *L*.

These conventions have already been used implicitly in this section, and will be used heavily hereafter.

3.2. Signatures and bindings

The quantifiers we have examined in the previous section are all *binary*; they compose two types or kinds to produce a new type or kind. In programming languages it is much more convenient to use *n-ary* quantifiers, so that we can easily express functions of *n* arguments and tuples of *n* components. N-ary quantifiers are achieved through the notions of *signatures* and *bindings* [Burstall Lampson 84].

A *signature* is a (possibly empty) ordered association of kinds to type variables, and of types to value variables, where all the variables have distinct names. For example:

`A::TYPE  a:A  f:`**`All`**`(x:A)Int`

This is a signature declaring a type variable, a value variable of that type, and a value variable denoting a function from objects of that type to integer. Note that signatures introduce variables from left to right, and such variables can be mentioned after their introduction.

A *binding* is a (possibly empty) ordered association of types to type variables and values to value variables, where all the variables have distinct names, for example:

`Let`` A::TYPE=Int  `**`let`**` a:A=3  `**`let`**` f:`**`All`**`(x:A)Int=`**`fun`**`(x:Int)x+1`

This is a binding associating the type `Int` and the values `3` and **`fun`**`(x:Int)x+1` with the appropriate variables. Bindings introduce variables from left to right; for example `a` could be used in the body of `f`. In some cases it is desirable to omit the variables and their related type information; the binding above then reduces to:

`:Int  3  `**`fun`**`(x:Int)x+1`

The colon in front of `Int` is to tell a parser, or a human, that a type is about to follow.

We can now convert our binary quantifiers to n-ary quantifiers as follows, where S is a signature and A is a type (and pairs become tuples):

n-ary universals:	**All**(S)A
n-ary existentials:	**Tuple** S **end**

NOTATION

- We use S as a metavariable ranging over signatures, and D as a metavariable ranging over bindings.

- The following abbreviations will be used extensively for components of signatures and bindings, where S_i are signatures, L is a kind, B is a type and b is a value:

Signatures:

$x_1,..,x_n$:B	for	x_1:B..x_n:B
$X_1,..,X_n$::L	for	X_1::L..X_n::L
x(S_1)..(S_n):B	for	x:**All**(S_1)..**All**(S_n)B
X(S_1)..(S_n)::L	for	X::**ALL**(S_1)..**ALL**(S_n)L

Bindings:

let x(S_1)..(S_n):B=b	for	**let** x=**fun**(S_1)..**fun**(S_n):B b
Let X(S_1)..(S_n)::L=B	for	**Let** X=**Fun**(S_1)..**Fun**(S_n)::L B

Signatures and bindings can be used in many different places in a language, and hence make the syntax more uniform. The positions where signatures and bindings appear in a surrounding context are here shown underlined; note the use of some of the abbreviations just introduced:

Signatures in bindings:
```
Let A::TYPE=Int  let a:A=3  let f(x:A):Int=x+1
```

Signatures in formal parameters:
```
let f(A::TYPE a:A f(x:A):Int):Int = ...
let f(_):Int = ...
```

Signatures in types:
```
All(A::TYPE a:A f(x:A):Int) A
All(_) A
Tuple A::TYPE a:A f(x:A):Int end
Tuple _ end
```

Signatures in interfaces:
```
interface I import ...
export
   A::TYPE
   a:A
   f(x:A):Int
end
```

Bindings at the top-level:
```
Let A::TYPE=Int;  let a:A=3;  let f(x:A):Int=x+1;
:Int;  3;  fun(x:Int) x+1;
_;
```

Bindings in actual parameters:
```
f(Let A::TYPE=Int  let a:A=3  let f(x:A):Int=x+1)
f(:Int  3  fun(x:Int) x+1)
f(_)
```

Bindings in tuples:
```
tuple Let A::TYPE=Int  let a:A=3  let f(x:A):Int=x+1 end
tuple :Int  3  fun(x:Int) x+1 end
tuple _ end
```

Bindings in modules:
```
module m:I import ...
export
   Let A::TYPE=Int
   let a:A=3
   let f(x:A):Int=x+1
end
```

Interfaces and modules will be discussed later.

3.3. The Quest language

The design principles and ideas we have discussed so far, as well as some yet to be discussed, are incorporated in a programming language called Quest (a quasi-acronym for Quantifiers and Subtypes). This language illustrates fundamental concepts of typeful programming, and how they can be concretely embedded and integrated in a single language. The language will be described in detail in the following sections, where it will provide a basis for further discussions and examples.

The example language is still speculative in some parts; the boundary between solid and tentative features can be detected by looking at the formal syntax in the Appendix. Features that have been given syntax there have also been implemented and are relatively well thought out. Other features described in the paper should be regarded with more suspicion.

Quest is used throughout the rest of the paper. The three major sections to follow deal with: a) the kind of types, including basic types, various structured types, polymorphism, abstract types, recursive types, dynamic types, mutable types and exception types; b) the kinds of operators, including parametric, recursive, and higher-order type operators; and c) the kinds of subtypes, including inheritance and bounded quantification. Then follow two sections on programming with modules and interfaces, seen as values and types respectively, and a section on system programming from the point of view of typing violations.

The main unifying concepts in Quest are those of type quantification and subtypes. However, a number of important type constructions do not fall neatly in these classes (e.g. mutable type and exceptions types), and they are added on the side. The important point is that they do not perturb the existing structure too much.

Here is a brief, general, and allusive overview of Quest characteristics; a gradual introduction begins in the next section.

Quest has three levels of entities: 1) values, 2) types and operators, 3) kinds. Types classify values, and kinds classify types and type operators. Kinds are needed because the type level is unusually rich.

Explicit type quantification (universal and existential) encompasses parametric polymorphism and abstract types. Quantification is possible not just over types (as in ordinary polymorphic languages) but also over type operators and over the subtypes of a given type.

Subtyping is defined inductively on all type constructions (e.g. including higher-order functions and abstract types). Subtyping on tuple types, whose components are ordered, provides a form of single inheritance by allowing a tuple to be viewed as a truncated version of itself with fewer final components. Subtyping on record types, whose components are unordered, provides a form of multiple inheritance, by allowing a record to be viewed as a smaller version of itself with an arbitrary subset of its components.

There are user-definable higher-order type operators and computations at the type level. The typechecker includes a normal-order typed λ-calculus evaluator (where the "types" here are actually the kinds).

A *generalized correspondence principle* is adopted, based on the notions of signatures and bindings. Landin proposed a correspondence between declarations and formal parameters. Burstall and Lampson proposed a correspondence between interfaces and declarations, and between parametric modules and functions [Burstall Lampson 84]. In Quest there is a correspondence between declarations, formal parameters, and interfaces, all based on a common syntax, and between definitions, actual parameters, and modules, also based on a common syntax.

Evaluation is deterministic, left-to-right, applicative-order. That is, functions are evaluated before their arguments, the arguments are evaluated left to right, and the function bodies are evaluated after their arguments. In records and tuples, the components are evaluated left to right, etc. Conditionals, cases, loops, etc. evaluate only what is determined by their particular flow of control. All entities are explicitly initialized.

The general flavor of the language is that of an interactive, compiled, strongly-typed, applicative-order, expression-based language with first-class higher-order functions and imperative features. Type annotations are used rather heavily, in line with existing programming-in-the-large languages.

In viewing signatures and bindings in terms of quantifiers, it is natural to expect alpha-conversion (the renaming of bound variables) to hold. Alpha-conversion would however allow signatures to match each other independently of the names of their components; this is routine for signatures in functions but very strange indeed for signatures in tuples, and it would create too many "accidental" matches in large software systems. Hence, Quest signatures and bindings must match by component name (and of course by component order and type). Some negative aspects of this choice are avoided by allowing component names in signatures and bindings to be *omitted*; omitted names match any name for the purpose of typechecking, and therefore provide a weak form of alpha-conversion.

We conclude this section with some further notes about Quest syntax.

Comments are enclosed in "`(*`" and "`*)`" and can be nested. A comment is lexically equivalent to a blank. Quotes (`'`) enclose character literals, and double quotes (`"`) enclose string literals.

Curly brackets "`{`" and "`}`" are used for grouping value, type, and kind expressions (e.g. to force operator precedence). Parentheses "`(`" and "`)`" are used in formal and actual parameter lists, that is for function declarations, definitions and applications.

The syntax has virtually no commas or semicolons. Commas are used only in lists of identifiers, and semicolons are used only to terminate top-level sentences, or whole modules. Otherwise, blank space is used. Indentation is not required for disambiguation, but it greatly improves readability because of the absence of normal separators. The resulting syntax is still easy to parse because of widespread use of initial and final keywords.

There are two lexical classes of identifiers: alphanumeric (letters and numerals starting with a letter) and symbolic (built out of `!@#$%&*_+=-|\`:<>/?`). Reserved keywords belonging to either lexical class are not considered identifiers. Alphanumeric identifiers are never used in infix position (e.g. `f(x y)`, but not `x f y`). Symbolic identifiers are always infix, but can also be used as prefix or stand-alone (e.g. `x+y`, `+(x y)`, or `{+}`).

New infix operators can be declared freely (e.g. **`let`** `++(x,y:Int):Int = 2*{x+y}`). There is no declaration of infix status. (It is not needed because of the strict lexical conventions.) There is no declaration of infix precedence. All infix operators, including the built-in ones, have the same precedence and are right-associative. This decision has been made because it is difficult to specify the relative precedence of operators in a meaningful way, and because it is equally difficult to remember or guess the relative precedence of operators when new infixes can be introduced.

There is no overloading of literals or operators, not even built-in ones. For example, `2` is an integer, `2.0` is a real, `+` is integer plus, and `real.+` is real plus, obtained through a built-in interface.

The style conventions we adopt are as follows. Value identifiers (including function names and module names) are in lower case, with internal capitalization for compound names. Type identifiers (including interface names) are capitalized. Kind identifiers are all caps. Keywords are capitalized in roughly the same way (e.g. "**`Tuple..end`**" is a tuple type, and "**`tuple..end`**" is a tuple value). Flow-control keywords (such as "**`begin..end`**" and "**`if..then..else..end`**") are in lower case. Keywords are pretty-printed in boldface, and comments in italics.

The infix `:` sign means "has type". The infix `::` sign means "has kind". The infix `<:` sign means "is subtype of". The = sign is used to associate identifiers with values, types, and kinds. The = sign is *not* used as the boolean test for equality: the constructs `x` **`is`** `y` and `x` **`isnot`** `y` are used instead. The `:=` sign is used as the assignment operator.

4. The kind of types

The kind of types is the only kind that is supported by ordinary programming languages such as Pascal. Hence, this section will include many familiar programming constructs. However, we also find some relatively unusual programming features, namely polymorphism and abstract types in their full generality. The Quest language, introduced in the previous section, is used throughout.

Basic types provide no new insights, but we have to describe them patiently since they are essential for examples. Function types use the general notion of signatures (universally quantified) to represent explicit polymorphism. Tuple types similarly use general signatures (existentially quantified) to represent abstract types.

Another quantifier, summation of signatures over a finite set, accounts for the familiar disjoint union types. Infinite summation of signatures over types corresponds to dynamic typechecking and supports persistent data. Recursive types allow the construction of lists and trees. Mutable types (assignable variables and fields, and arrays) have to be handled carefully to maintain type soundness. Finally, exception types take care of exceptions.

4.1. Introduction

Quest is an interactive compiled language. The user inputs a phrase to evaluate an expression or to establish a binding, then the system produces an answer, and so on. This level of interaction is called the *top level* of the system. We use the following notation to illustrate the top-level interaction: user input is preceded by a "•" sign; system output is preceded by a "»" sign, is italicized, and is shown in a smaller font. The output portions are occasionally omitted when they are obvious.

```
•   let a = 3;
»   let a:Int = 3
•   a+4;
»   7 : Int
```

We recall that **`let`** is used for introducing values, and **`Let`** for introducing types; other forms of bindings will be explained as they arise.

The concepts of modules and interfaces are basic. Interfaces are used not only to organize user programs, but also to organize the basic system. Many routines that would normally be considered "primitives" are collected into "built-in" interfaces (interfaces whose implementation is provided by the basic system). These include features as complex as input/output, but also features as basic as Ascii conversions.

By convention, if an interface is dedicated to implementing a single type, or has a main type, that type is given the simple name "`T`", since this will always be qualified by the module name which will normally be expressive enough (e.g. `list.T`).

4.2. Basic and built-in types

The basic types are `Ok`, `Bool`, `Char`, `String`, `Int`, `Real`, `Array`, and `Exception`. In a language with modules there is, in principle, no need for basic types because new basic types can be introduced by external interfaces (some of which may be known to the compiler, so that they are implemented efficiently). However, it is nice to have direct syntactic support for some commonly used constants and operators; this is the real reason for having special types designated "basic" (this need could perhaps be eliminated if the language supported syntactic extensions).

Types that are not basic but are defined in interfaces *known* to the compiler, are called *built-in*. The respective modules and interfaces are also called *built-in*. Examples of built-in interfaces are `Reader`, `Writer`, and `Dynamic`, described in the Appendix. Other built-in interfaces can also be defined as needed by implementations.

Ok

`Ok` is the type with a single constant `ok` and no operations. This type is sometimes called *void* or *unit* in other languages, and is used in expression-based languages as the type of expressions that are not really meant to have any value (e.g. assignment statements).

Bool

`Bool` is the type with constants `true` and `false`. Boolean operations are the infix `\/` (or) and `/\` (and), and the function `not`; they evaluate all their arguments. The equality or inequality of two values is given by the forms *`x`*

is *y* and *x* **isnot** *y*, which return a boolean. This is-ness predicate is defined as ordinary equality for `Ok`, `Bool`, `Char`, `Int`, and `Real`, and as "same memory location" for all the other types including `String`. Abstract types should always define their own equality operation in their interface.

A basic programming construct based on booleans is the *conditional*:

```
•  if true then 3 else 4 end;
»  3 : Int
```

The conditional can take many forms. It may or may not have the **then** and **else** branches (missing branches have type `Ok`), and additional **elsif** branches can be cascaded.

Two other conditional constructs are provided:

a **andif** b	same as:	**if** a **then** b **else** false **end**
a **orif** b	same as:	**if** a **then** true **else** b **end**

These are useful to express complex boolean conditions when not all subconditions should be evaluated. For example, assuming `a`,`n`, and `m` are defined:

```
•  if {n isnot 0} andif {{m/n}>0} then a:=m
   elsif {n is 0} orif {m<0} then a:=0
   end;
»  ok : Ok
```

This conditional never causes division by zero.

Char

`Char` is the type of Ascii characters, such as `'a'`, which are syntactically enclosed in single quotes[1]. Operations on characters are provided through a built-in module `ascii` of built-in interface `Ascii` (this is written `ascii:Ascii`). Incidentally, this is an example of how to organize basic operations into interfaces in order not to clutter the language: one might forget that `char` is an Ascii operation and confuse it with a variable or function, but on seeing `ascii.char` the meaning should be obvious.

String

`String` is the type of strings of Ascii characters, such as `"abc"`, enclosed in double quotes[2]. The operations on strings are provided through a built-in module `string:StringOp`, but there is also a predefined infix operator <> for string concatenation:

```
•  let s1="concat" and s2="enate";
»  let s1:String = "concat"
   let s2:String = "enate"
•  s1 <> s2;
»  "concatenate" : String
```

1 Backslash can be used to insert some special characters within character quotes: \n is newline, \t is tab, \b is backspace, \f is form feed, \' is single quote, and \\ is backslash; backslash followed by any other character is equivalent to that character.

2 As with characters, backslash can be used to insert some special characters within string quotes: \n is newline, \t is tab, \b is backspace, \f is form feed, \" is double quote, and \\ is backslash; backslash followed by any other character is equivalent to that character.

This example also illustrates simultaneous bindings, separated by **and**.

Another library module `conv:Conv` contains routines for converting various quantities to strings; this is useful for printing.

Int

`Int` is the type of integer numbers. Negative numbers are prefixed by a tilde, like ~3. The integer operators are the infix +, −, *, /, % (modulo), <, >, <=, and >=.

Real

`Real` is the type of floating point numbers. Real constants are formed by an integer, a dot, a positive integer, and optionally an exponent part, as in `3.2` or `~5.1E~4`. To avoid overloading integer operations, the real operators are then infix ++, −−, **, //, ^^ (exponent), <<, >>, <<=, and >>=.

```
•   2.8 ++ 3.4;
»   6.2: Real
```

Conversions and other operations are provided through the built-in modules `real:RealOp` and `trig:Trig`.

Others

Arrays, exceptions, readers, writers, and dynamics are discussed in later sections, or in the Appendix.

4.3. Function types

Function types are normally realized, implicitly or explicitly, by an "arrow" operator requiring a domain and a range type. Here we adopt an operator (actually, a quantifier) requiring a domain *signature*; this is more general because the signature may introduce type variables, which can then occur in the range type.

A function type has the form:

```
All(S)A
```

This is the type of functions with parameters of signature *S* and result of type *A*:

```
fun(S):A b          or, abbreviated          fun(S)b
```

Depending on the form of *S* and *A*, functions can be simple, higher order (if *S* or *A* contain function types), or polymorphic (if *S* introduces type variables).

Simple functions

Here is a simple function declaration:

```
•   let succ = fun(a:Int):Int a+1;
»   let succ(a:Int):Int = <fun>
•   succ(3);
»   4 : Int
```

As discussed earlier, these declarations are usually abbreviated as follows:

```
• let succ(a:Int):Int = a+1;
» let succ(a:Int):Int = <fun>
```

Here is a function of two arguments; when applied it can take two simple arguments, or any binding of the correct form:

```
• let plus(a:Int b:Int):Int = a+b;
» let plus(a:Int b:Int):Int = <fun>
• plus(3 4);
» 7 : Int
• plus(let a=3 let b=a+1);
» 7 : Int
```

Recursive functions are defined by a **`let rec`** binding:

```
• let rec fact(n:Int):Int =
    if n is 0 then 1 else n*fact(n-1) end;
» let fact(n:Int):Int = <fun>
```

Simultaneous recursion is achieved by **`let rec`** followed by multiple bindings separated by **`and`**. All the entities in a recursive definition must syntactically be *constructors*, that is functions, tuples, or other explicit data structures, but not function applications, tuple selections, or other operations.

Higher-order functions

There is not much to say about higher-order functions, except that they are allowed:

```
• let double(f(a:Int):Int)(a:Int):Int = f(f(a));
» let double(f(a:Int):Int)(a:Int):Int = <fun>

• let succ2 = double(succ);
» let succ2(a:Int):Int = <fun>
• succ2(3);
» 5 : Int
```

Note that a function body may contain non-local identifiers (such as `f` in the body of `double`), and these identifiers are retrieved in the appropriate, statically determined scope.

Polymorphic functions

Polymorphic functions are functions that have types as parameters, and that can later be instantiated to specific types. Here is the polymorphic identity function:

```
• let id(A::TYPE)(a:A):A = a;
» let id(A::TYPE)(a:A):A = <fun>
• id(:Int)(3);
» 3 : Int
```

The identity function can be instantiated in many interesting ways. First, it can be instantiated to a simple type, like `Int` to obtain the integer identity:

```
•   let intId = id(:Int);
»   let intId(a:Int):Int = <fun>
```

It can be applied to the integer identity by first providing its type, the integer endomorphism type:

```
•   Let IntEndo = All(a:Int)Int;
»   Let IntEndo::TYPE = All(a:Int)Int
•   id(:IntEndo)(intId);
»   <fun> : All(a:Int)Int
```

It can even be applied to itself by first providing its own type:

```
•   Let Endo = All(A::TYPE)All(a:A)A;
»   Let Endo::TYPE = All(A::TYPE)All(a:A)A
•   id(:Endo)(id);
»   <fun> : All(A::TYPE)All(a:A)A
```

Polymorphic functions are often *curried* in their type parameter, as we have done so far, so that they can be conveniently instantiated. However, this is not a requirement; we could also define the polymorphic identity as follows:

```
•   let id2(A::TYPE a:A):A = a;
»   let id2(A::TYPE a:A):A = <fun>
•   id2(:Int 3);
»   3 : Int
```

A slightly more interesting polymorphic function is `twice`, which can double the effect of any function (including itself):

```
•   let twice(A::TYPE)(f(a:A):A)(a:A):A = f(f(a));
»   let twice(A::TYPE)(f(a:A):A)(a:A):A = <fun>

•   let succ2 = twice(:IntEndo)(succ);
»   let succ2(a:Int):Int = <fun>
```

Type systems that allow polymorphic functions to be applied to their own types are called *impredicative*. The semantics of such systems is delicate, but no paradox is necessarily involved.

4.4. Tuple types

Tuple types are normally intended as iterated applications of the cartesian product operator. Our tuple types are formed from *signatures*; this is more general because signatures can introduce type variables which may occur in the following signature components. These tuple types can be interpreted as iterated existential quantifiers.

A tuple type has the form:

```
Tuple S end
```

This is the type of tuples containing bindings *D* of signature *S*:

tuple *D* **end**

Depending on the form of *S*, we can have a simple tuple type or an abstract type (if *S* introduces type variables).

Simple tuples

A simple tuple is an ordered collection of values. These values can be provided without names:

```
•  tuple 3 4 "five" end;
»  tuple 3 4 "five" end : Tuple :Int :Int :String end
```

Then one way to extract such values is to pass the tuple to a function that provides names for the components in its input signature. But frequently, tuple components are named:

```
•  let t: Tuple a:Int b:String end =
      tuple
         let a = 3
         and b = "four"
      end;
»  let t:Tuple a:Int b:String end = tuple let a=3 let b="four" end
```

Then the tuple components can be extracted by name via the *dot notation*:

```
•  t.b;
»  "four": String
```

Note that a tuple may contain an arbitrary binding, including function definitions, recursive definitions, etc. An empty tuple type is the same as the `Ok` type, and the empty tuple is the same as `ok`.

Abstract tuples

An abstract type is obtained by a tuple type which contains a type and a set of constants and operations over that type.

```
•  Let T = Tuple A::TYPE a:A f(x:A):Int end;
»  Let T::TYPE = Tuple A::TYPE a:A f(x:A):Int end
```

This abstract type can be implemented by providing a type, a value of that type, and an operation from that type to `Int`:

```
•  let t1:T =
      tuple
         Let A::TYPE = Int
         let a:A = 0
         let f(x:A):Int = x+1
      end;
»  let t1:T =
     tuple Let A::TYPE=<Unknown> let a:A=<unknown> let f(x:A):Int=<fun> end
```

Abstract types and their values are not printed, in order to maintain the privacy implicit in the notion of type abstraction. Functions are not printed because they are compiled and their source code might not be easily available.

An abstract type can be implemented in many different ways; here is another implementation:

```
•   let t2:T =
       tuple
          Let A::TYPE = Bool
          let a:A = false
          let f(x:A):Int = if x then 0 else 1 end
       end;
»   let t2:T =
      tuple Let A::TYPE=<Unknown> let a:A=<unknown> let f(x:A):Int=<fun> end
```

What makes type `T` abstract? Abstraction is produced by the rule for extracting types and values from the tuples above: the identity of the type is never revealed, and values of that type are never shown:

```
•   :t1.A;
»   :t1.A :: TYPE
•   t1.a;
»   <unknown> : t1.A
```

Moreover, an abstract type does not match its own representation:

```
•   t1.a + 1;
»   Type Error: t1.A is not a subtype of Int
```

Although the representation type must remain unknown, it is still possible to perform useful computations:

```
•   t1.f(t1.a);
»   1 : Int
```

But attempts to mix two different implementations of an abstract type produce errors:

```
•   t1.f(t2.a);
»   Type Error: t2.A is not a subtype of t1.A
```

This fails because it cannot be determined that `t1` and `t2` are the same implementation of `T`; in fact in this example they are really different implementations whose interaction would be meaningless.

An abstract type can even be implemented as "itself":

```
•   let t3:T =
       tuple
          Let A::TYPE = T
          let a:A = t2
          let f(x:A):Int = x.f(x.a)
       end;
»   let t3:T =
      tuple Let A::TYPE=<Unknown> let a:A=<unknown> let f(x:A):Int=<fun> end
```

This is another instance of *impredicativity* where a value of an abstract type can contain its own type as a component.

4.5. Option types

Option types model the usual *disjoint union* or *variant record* types; no unusual features are present here. An option type represents a choice between a finite set of signatures; objects having an option type come with a tag indicating the choice they represent. More precisely, an option type is an ordered named collection of signatures, where the names are distinct. An object of an option type, or *option*, is a tuple consisting of a tag and a binding. An option of tag *x* and binding *D* has an option type *T* provided that *T* has a tag *x* with an associated signature *S*, and the binding *D* has signature *S*.

```
•   Let T =
       Option
          a
          b with x:Bool end
          c with x,y:String end
       end;
•   let aOption = option a of T end;
•   let bOption = option b of T with let x = true end;
•   let cOption = option c of T with "xString" "yString" end;
```

Since the choices in an option type are ordered, there is an operator `ordinal(o)` returning the 0-based ordinal number of an option as determined by its tag[3]. Note that the tag component of an option can never be modified.

The option tag can be tested by the `?` operator (see examples below).

The option contents can be extracted via the `!` operator. This may fail (at run-time) if the wrong tag is specified. If successful, the result is a tuple whose first component is the ordinal of the option.

```
•   aOption?a;
»   true : Bool
•   bOption?a;
»   false : Bool
•   bOption!a;
»   Exception: '!'Error
•   bOption!b;
»   tuple 1 let x=true end : Tuple :Int x:Bool end
•   bOption!b.x;
»   true : Bool
```

A more structured way of inspecting options is through the **`case`** construct, which discriminates on the option tag and may bind an identifier to the option contents:

```
•   case bOption
       when a then ""
       when b with arm then
          if arm.x then "true" else "false" end
```

3 Conversely, an option can be created from an ordinal number as **`option`** `ordinal(n)` **`of`** `T` **`..`** **`end`**, where `n` is determined at run-time; this form is allowed only when all the branches of an option type have the same signature.

```
        when c with arm then
            arm.x <> arm.y
        else "??"
    end
»   "true": String
```

A **`when`** clause may list several options, then the following are equivalent:

```
when x,y with z then a          when x with z then a
                                when y with z then a
```

where the two occurrences of `z` in the right-hand side must receive the same type.

If an option type starts directly with a **`with`** clause, it means that the components of that clause are common to all the branches. That is, the following types are equivalent:

```
Option                          Option
    with x:Bool end                 a with x,y:Bool end
    a with y:Bool end               b with x,z:Bool end
    b with z:Bool end           end;
end;
```

However, values of these types behave slightly differently; in the former case, the common components can be extracted directly by the dot notation, without first coercing to a given label by `!`.

Option types will be discussed again in conjunction with subtypes.

4.6. Auto types

There are some situations in programming where it is necessary to delay typechecking until run-time. These situations can be captured by a new form of quantification: an infinite union of signatures indexed by types, whose use requires dynamic typechecking. This quantifier resembles both option types and abstract types. Like an option type it is a union with run-time discrimination, but the union is infinite. Like an abstract type it is a collection of types and related values, but the types can be dynamically inspected.

Automorphic values (auto values, for short) are values capable of describing themselves; they contain a type that is used to determine the rest of their shape. The related *automorphic types* (auto types) consist of a type variable *X* and a signature *S* depending on *X*; they represent the union of all the signatures *S* when *X* varies over all types or, as we shall see, over all the subtypes of a given type.

```
•   Let UniformPair =
        Auto A::TYPE with fst,snd:A end;
•   let p:UniformPair =
        auto :Bool with false true end;
»   let p:UniformPair = auto :Bool with false true end
```

Restriction: the type component of an auto value must be a *closed* type: it cannot contain any free type variables.

It is possible to discriminate on the type component of an automorphic object through a construct similar to **case**; this is named after the *inspect* construct in Simula67, since, as we shall see, it can be used to discriminate between the subtypes of a given type.

```
•   inspect b
       when Bool with arm then arm.fst\/arm.snd
       when Int with arm then (arm.fst*arm.snd) isnot 0
    end
»   true: Bool
```

The types in the **when** clauses of inspect must all be closed types. The branch that is selected for execution (if any) is the first branch such that the type in the auto value is a subtype of the type in the branch. An **else** branch can be added at the end of the construct.

Automorphic types will be discussed again in conjunction with subtypes and with dynamic types.

4.7. Recursive types

Recursive types are useful for defining lists, trees, and other inductive data structures. Since these are often parametric types, they will be discussed in the section on type operators.

Here we discuss a rather curious fact: recursive types allow one to reproduce what is normally called *type-free* programming within the framework of a typed language. Hence no "expressive power" (in some sense) is lost by using a typed language with recursive types. Again, this shows that a sufficiently sophisticated type system can remove many of the restrictions normally associated with static typing.

S-expressions are Lisp's fundamental data structures. These can be described as follows (for simplicity, we do not consider atoms):

```
•   Let Rec SExp::TYPE =
       Option
          nil
          cons with car,cdr:SExp end
       end
»   Let Rec SExp::TYPE = Option nil  cons with car,cdr:SExp end end
```

The usual Lisp primitives[4] can now be defined in terms of operations on options and tuples; this is left as an exercise.

Another interesting recursive type is the "universal" type of "untyped" functions:

```
•   Let Rec V::TYPE = All(:V)V
»   let Rec V::TYPE = All(:V)V
```

We can now define the universal untyped combinators `K` and `S`:

```
•   let K(x:V)(y:V):V = x
»   let K:V = <fun>
```

4 I.e., "pure" Lisp primitives. To define `rplaca` see the next section.

```
•  let S(x:V)(y:V)(z:V):V = x(z)(y(z))
»  let S:V = <fun>
```

Note however that Quest is a call-by-value language; as an exercise, define a call-by-name version of the definitions above, and encode a non-strict conditional expression. (Hint: `Let Rec V::TYPE = All()All(:V)V`.)

It is also interesting to notice that recursive types by themselves imply the presence of general recursive functions. As an exercise, define the fixpoint combinator on `V` (both the call-by-name and the call-by-value versions) without using **`let rec`**.

Two recursive types are equivalent when their *infinite expansions* (obtained by unfolding recursion) form equivalent infinite trees. Recursive type definitions describe finite graphs with type operators at the nodes, hence it is possible to test the condition above in finite time, without actually generating infinite structures [Amadio Cardelli 90].

4.8. Mutable types

Imperative programming is based on the notion of a mutable global store, together with constructs for sequencing the operations affecting the store. Mutability interacts very nicely with all the quantifiers, including polymorphism[5], showing that the functional-style approach suggested by type theory does not prevent the design of imperative languages.

Var types

Mutable store is supported by two type operators: **`Var`**`(A)` is the type of mutable identifiers and data components of type `A`; and **`Out`**`(A)` is the type of write-only parameters of type `A`.

(These operators are not in fact first-class, in the sense that values of type **`Var`**`(A)` or **`Out`**`(A)` are not first-class values, and types like **`Var`**`(`**`Var`**`(A))` cannot be produced. However, when discussing type rules it is often useful to think of **`Var`** and **`Out`** as ordinary operators; the above restrictions are for efficiency reasons only, and not for typing reasons.)

An identifier can be declared mutable by prefixing it with the **`var`** keyword in a binding or signature; it then receives a **`Var`** type.

```
•  let var a=3;
»  let var a:Int = 3
```

The answer `let var a:Int = 3` above should be read as meaning that `a` has type **`Var`**`(Int)`.

When evaluating an expression of type **`Var`**`(A)` (or **`Out`**`(A)`), an automatic coercion is applied to obtain a value of type `A`.

```
•  a;
»  3 : Int
```

The `:=` sign is used as the assignment operator; it requires a destination of type **`Var`**`(A)` and a source of type `A`, and returns `ok` after performing the side-effect:

5 The problems encountered in ML [Gordon Milner Wadsworth 79, page 52] are avoided by the use of explicit polymorphism. However, we have to take some precautions with subtyping, as explained later.

```
• a:=5;
» ok : Ok
• a+1;
» 6 : Int
```

Functions cannot refer directly to global variables of **Var** type; these must be either passed as in and **out** parameters, or embedded in data structures to be accessed indirectly. This restriction facilitates the implementation of higher-order functions. Here is an example of two functions sharing a private variable that is accessible to them only (sometimes called an *own* variable):

```
• let t: Tuple f,g():Int end =
    begin
      let a = tuple let var x=3 end
      tuple
        let f():Int = begin a.x:=a.x+1 a.x end
        let g():Int = begin a.x:=a.x+2 a.x end
      end
    end;
» let t:Tuple f():Int g():Int end =
    tuple let f = <fun> let g = <fun> end
• t.f();
» 4: Int
• t.g();
» 6: Int
```

Data structure components can be made mutable by using the same **var** notation used for variables:

```
• let t = tuple let var a=0 end;
» let t = tuple let var a:Int = 0 end
• t.a := t.a+1;
» ok : Ok
```

Sequencing and iteration

A syntactic construct called a *block* is used to execute expressions sequentially and then return a value. A block is just a binding `D` (hence it can declare local variables) with the restriction that its last component, which is the result value, must be a value expression. The type of a block is then the type of its last expression.

Value-variables declared in a block `D:A` may not appear free in `A` (as tuple variables from which an abstract type is extracted), since they would escape their scope.

The most explicit use of a block is in the **begin** `D` **end** construct, but blocks are implicit in most places where a value is expected, for example in the branches of a conditional.

Iteration is provided by a **loop .. exit .. end** construct, which evaluates its body (a binding) repeatedly until a syntactically enclosed **exit** is encountered. The **exit** construct has no type of its own; it can be thought of as an exception **raise exit as** `A` **end**, where `A` is inferred. A **loop** construct always returns `ok`, if it terminates, and has type `Ok`.

In addition, **while** and **for** constructs are provided as abbreviations for loop-exit forms.

Here is an example of usage of blocks and iteration:

```
• let gcd(n,m:Int):Int =
      begin
         let var vn=n
         and var vm=m
         while vn isnot vm do
            if vn>vm then vn:=vn-vm end
            if vn<vm then vm:=vm-vn end
         end
         vn
      end;
» let gcd(n,m:Int):Int = <fun>
• gcd(12 20);
» 4 : Int
```

Out types

Formal parameters of functions may be declared of **Out** type by prefixing them with the **out** keyword. This makes them into write-only copy-out parameters.

When passing an argument corresponding to an **out** parameter, there must be an explicit coercion into an **Out** type. This is provided by the **@** keyword, which must be applied to a mutable (**Var** or **Out**) entity: this is meant to be suggestive of passing the location of the entity instead of its value.

```
• let g(x:Int out y:Int):Ok = y:=x+1;
• begin g(0 @t.a) t.a end;
» 1 : Int
```

For convenience, it is also possible to convert a non-mutable value to an **out** parameter by creating a location for it on-the-fly, as in `g(0 let var y=0)`; this is also abbreviated as `g(0 var(0))`. In other words, an **out** entity can be obtained from a **Var** or **Out** entity by **@**, or from an ordinary value by **var**. This description was very operational; the deeper relations between **Var** and **Out**, and the type rules for passing **Var** and **Out** parameters, are explained in the section on subtyping.

Polymorphism and `Out` parameters can be combined to define the following function:

```
• let assign(A::TYPE out a:A b:A):Ok = a:=b;
• let var a=3;
» assign(:Int @a 7);
```

this has the same effect as an assignment operation.

Array types

Array types `Array(A)` describe arrays with elements of type *A*, indexed by non-negative integers. Array types do not carry size information, since array values are dynamically allocated. Individual arrays are created with a given size, which then never changes.

Array values can be created with a given size and an initial value for all the elements, or from an explicit list of values. They can be indexed and updated via the usual square-bracket notation, and an `extent` operator is provided to extract their size.

```
• let a:Array(Int) = array of 0 1 2 3 4 5 end;
• let b:Array(Bool) = array of(5 false);
• b[0] := {a[0] is 0};
```

The array `a` has the 6 listed elements. The array `b` is defined to have `5` elements initialized to `false`.

We have used here *listfix* syntax, which is an enumeration of values enclosed in **`of..end`**, or a size-init pair enclosed in **`of`**`(..)`. Although **`array`** is the basic listfix construct, additional listfix functions can be defined. They are interpreted according to the following abbreviations:

f **of .. end**	for	*f*(**array of .. end**)
f **of**(*n a*)	for	*f*(**array of**(*n a*))

Hence, all we need to get a listfix function is to define an ordinary function with an array parameter:

```
•   let sum(a:Array(Int)):Int =
       begin
          let var total=0
          for i = 0 upto extent(a)-1 do
             total := total+a[i]
          end
          total
       end;
»   let sum(a:Array(Int)):Int = <fun>

•   sum of 0 1 2 3 4 end;
»   10 : Int
•   sum of(5 1);
»   5 : Int
```

The complete set of operations on arrays is provided by the built-in module `arrayOp:ArrayOp`. It is interesting to examine the polymorphic signatures of these operations.

4.9. Exception types

Exceptions, when used wisely, are an important structuring tool in embedded systems where anomalous conditions are out of the control of the programmer, or where their handling has to be deferred to clients. Again it is comforting to notice that exceptions, being a control-flow mechanism, do not greatly affect the type system. In fact, exceptions can be propagated together with a typed entity, and in this sense they can be typechecked.

Many primitive operations produce exceptions when applied to certain arguments (e.g. division by zero). User-defined exceptions are also available in our language. Exceptions in Quest are treated differently than in most languages; there is an explicit notion of an exception *value* which can be created by the **`exception`** construct, and then bound to a variable or even passed to a function to be eventually *raised*.

```
•   let exc1: Exception(Int) =
       exception integerException:Int end;
•   let exc2: Exception(String) =
       exception stringException:String end;
```

The exception construct generates a new unique exception value whenever it is evaluated. The identifier in it is used only for printing exception messages; the type is the type of a value that can accompany the exception. Exceptions can be raised with a value of the appropriated type via the **`raise`** construct:

```
•  raise exc1 with 3 end;
»  Exception: integerException with 3:Int
```

Exceptions can be caught by the **try** construct that attempts evaluating an expression. If an exception is generated during the evaluation it will be matched against a set of exceptions, optionally binding an identifier to the exception value. If the exception is not among the ones considered by the **try** construct, and if there is no **else** branch, the exception is propagated further.

```
•  try
      raise exc1 with 55 end
      when exc1 with x then x+5
      when exc2 then 1
      else 0
   end;
»  60: Int
```

Exceptions propagate along the dynamic call chain.

5. Operator kinds

Operators are functions from types to types that are evaluated at compile-time. There are also higher-order operators that map types or operators to other types or operators. All languages have built-in operators, such as function spaces and cartesian products, but very few languages allow new operators to be defined, or restrict them to first-order (types to types). Higher-order operators embody a surprising expressive power; they define one of the largest known classes of total functions [Girard 71], and every free algebra with total operations (booleans, integers, lists, trees, etc.) is uniformly representable in them [Böhm Berarducci 85] (see the example below for booleans). Because of this, we believe they will turn out to be very useful for parametrization and for carrying out compile-time computations.

An operator has the form:

```
Fun(S)A
```

where *S* is a signature introducing only type variables, and *A* is a type.

The kind of an operator has the form:

```
ALL(S)K
```

where *K* is a kind.

5.1. Type operators

The simplest operator is the *identity* operator, which takes a type and returns it; here it is defined in the normal and abbreviated ways.

```
•  Let Id::ALL(A::TYPE)TYPE = Fun(A::TYPE) A;
•  Let Id(A::TYPE)::TYPE = A;
»  Let Id(A::TYPE)::TYPE = A
```

```
• :Id(Int);
» :Int :: TYPE
• let a:Id(Int) = 3;
» let a:Int = 3
```

The *function space* and *cartesian product* (infix) operators take two types and return a type:

```
• Let ->(A,B::TYPE)::TYPE = All(:A)B;
• Let #(A,B::TYPE)::TYPE = Tuple fst:A snd:B end;
• let f:{Int#Int}->Int = fun(p:Int#Int)p.fst+p.snd;
```

Note the *omitted* identifier in `All(:A)B`; this is to make typings such as `{fun(x:A)B}:A->B` legal for any identifier `x`, hence allowing `f` above to typecheck. Omitted identifiers and alpha-conversion were briefly discussed in the section "The Quest language".

The *optional* operator is for data components which may or may not be present:

```
• Let Opt(A::TYPE)::TYPE =
     Option
        none
        some with some:A end
     end;
```

Operator applications are evaluated with call-by-name, although this is not very important since all computations at the operator level terminate. (There are no recursive operators.) Some interesting functional programs can be written at the operator level (**DEF** here introduces a kind):

```
• DEF BOOL = ALL(Then,Else::TYPE)TYPE;
• Let True::BOOL = Fun(Then,Else::TYPE) Then;
• Let False::BOOL = Fun(Then,Else::TYPE) Else;
• Let Cond(If::BOOL Then,Else::TYPE)::TYPE = If(Then Else);
```

Convince yourself that `Cond` above is really a conditional at the type level.

5.2. Recursive type operators

There are no recursive type operators, in order to guarantee that computations at the type level always terminate. However, there are recursive types which can achieve much of the same effect. For example, one might expect to define parametric lists via a *recursive operator* as follows:

```
• (* Invalid recursive definition:
  Let Rec List(A::TYPE)::TYPE =
     Option
        nil
        cons with head:A tail:List(A) end
     end;
  *)
```

Instead, we have to define `List` as an *ordinary operator* that returns a *recursive type*:

```
•  Let List(A::TYPE)::TYPE =
      Rec(B::TYPE)
         Option
            nil
            cons with head:A tail:B end
         end;
»  Let List(A::TYPE)::TYPE =
      Rec(B::TYPE) Option nil  cons with head:A tail:B end end
```

Here we have used the recursive type constructor **Rec**(*A*::*K*) *B* which has the restriction that the kind *K* must be `TYPE` (or a power kind, see next section).

Here are the two basic list constructors:

```
•  let nil(A::TYPE):List(A) =
      option nil of List(A) end
   and cons(A::TYPE)(head:A tail:List(A)):List(A) =
      option cons of List(A) with
         head tail
      end;
```

The other basic list operations are left as an exercise.

6. Power kinds

Most of the quantifier and operator structure described so far derives from Girard's system *Fw* [Girard 71]. The main new fundamental notion in Quest is the integration of subtyping with type quantification. This allows us to express more directly a wider range of programming situations and programming language features, as well as to introduce new ones.

In this section we define a subtyping relation between types, written `<:`, such that if *x* has type *A* and *A*`<:`*B* then x has also type *B*. Hence, the intuition behind subtyping is ordinary set inclusion. Subtyping is defined for *all* type constructions, but normally holds only among types based on the same constructor. We do not have non-trivial subtype relations over basic types, although it is possible to add them when the machine representations of related types are compatible.

We can then talk of the collection of all subtypes of a given type *B*; this forms a kind which we call `POWER(`*B*`)`, the power-kind of *B*. Then *A*`::POWER(`*B*`)` and *A*`<:`*B* have the same meaning; the former is taken as the actual primitive, while the latter is considered an abbreviation. Subtyping induces a subkind relation, written `<::` , on kinds; basically, `POWER(`*A*`)<::POWER(`*B*`)` whenever *A*`<:`*B*, and `POWER(`*A*`)<:: TYPE` for any type *A*. Then we say that a signature *S* is a *subsignature* of a signature *S'* if *S* has the same number of components as *S'* with the same names and in the same position, and if the component types (or kinds) of *S* are subtypes (or subkinds) of the corresponding components in *S'*. Moreover, we say that *S"* is an *extended subsignature* of *S'* if *S"* has a prefix *S* which is a subsignature of *S'*. The formal subtyping rules for a simplified Quest language are in the Appendix.

Our notion of subtyping can emulate many characteristics of what is commonly understood as inheritance in object oriented languages. However, we must keep in mind that inheritance is a fuzzy word with no unequivocally established definition, while subtyping has a technical meaning, and that in any case subtyping is only one component of a full treatment of inheritance. Hence, we must try and keep the two notions distinct to avoid confusion.

The most familiar subtyping relation is between tuple types; this idea comes from object oriented programming where a *class* `A` is a *subclass* of another class `B` if the former has all the *attributes* (*methods* and *instance variables*) of the latter.

We interpret an *object* as a tuple, a *method* as a functional component of a tuple, a *public instance variable* as a modifiable value component of a tuple, a *private instance variable* as a modifiable own variable of the methods (not appearing in the tuple), a *class* as a function generating objects with given methods and instance variables, and finally a *class signature* as a tuple type. We do not discuss the interpretation of *self* here, which is in itself a very interesting and complex topic [Cook Hill Canning 90] .

If `A` is a subclass of `B`, then `A` is said to *inherit* from `B`. In object oriented programming this usually (but not always) means that objects of `A` inherit the methods (functional fields) of `B` by sharing the method code, and also inherit the instance variables (data fields) of `B` by allocating space for them.

In our framework only signatures are in some sense inherited, not object components. Inheritance of methods can be achieved manually by code sharing. Since such sharing is not enforced by the language, we acquire flexibility: a class signature can be implemented by many classes, hence different instances of the same class signature can have different methods. This confers a dynamic aspect to method binding, while not requiring any run-time search in the class hierarchy for method access.

6.1. Tuple subtypes

The subtyping rule for tuples is as follows. A tuple type `A` is a subtype of a tuple type `B` if the signature of `A` is an extended subsignature of the signature of `B`.

Remember that fields in a tuple or tuple type are ordered, hence tuple subtyping emulates *single-inheritance* hierarchies where the hierarchical structure always forms a tree (or a forest).

Simple tuples

Here is an example of a subtyping hierarchy on tuple types:

```
•   Let Object =
        Tuple age:Int end;
•   Let Vehicle =
        Tuple age:Int speed:Int end;
•   Let Machine =
        Tuple age:Int fuel:String end;
```

which produces the following subtyping diagram:

```
Vehicle        <:
                         Object
Machine        <:
```

Here are a couple of actual objects:

```
•   let myObject:Object =
        tuple let age=3 end;
```

```
•   let myVehicle:Vehicle =
       tuple let age=3 let speed=120 end;
```

Since `myVehicle:Vehicle` and `Vehicle<:Object`, we have that `myVehicle:Object` (this is called the *subsumption* rule). Hence functions expecting objects will accept vehicles:

```
•   let age(object:Object):Int = object.age;
•   age(myVehicle);
»   3 : Int
```

This example illustrates the flexibility inherent in subtypes. Note that inheritance (i.e. subtyping) is automatic and depends only on the structure of types; it does not have to be declared. Note also that the `age` function could have been defined before the `Vehicle` type, but still would work on vehicles as soon as these were defined.

Abstract tuples

Since tuples may be used to form abstract types, we immediately get a notion of *abstract subtypes*: that is, subtyping between abstract types. For example:

```
•   Let Point =
       Tuple
          A::TYPE
          new(x,y:Int):A
          x,y(p:A):Int
       end;
•   Let ColorPoint =
       Tuple
          A::TYPE
          new(x,y:Int):A
          x,y(p:A):Int
          paint(p:A c:Color):Ok
          color(p:A):Color
       end;
```

where a new color point starts with some default color and can then be painted in some other color. Here `ColorPoint <: Point`, hence any program working on points will also accept color points, by subsumption.

6.2. Option subtypes

The subtyping rule for option types is as follows: an option type *A* is a subtype of an option type *B* if *B* has all the tagged components of *A* (according to their names and positions) and possibly more, and the component signatures in *A* are extended subsignatures of the corresponding component signatures in *B*. Again, components in an option type are ordered.

For example, if we define the following *enumeration* types:

```
•   Let Day =
       Option mon tue wed thu fri sat sun end;
•   Let WeekDay =
       Option mon tue wed thu fri end;
```

then we have:

```
WeekDay   <:   Day
```

It is important to notice that, unlike enumeration types in Pascal, here we can freely extend an existing enumeration with additional elements, while preserving type compatibility with the old enumeration. Similarly, after defining a tree-like data type using options, one can add new kinds of nodes without affecting programs using the old definition. Of course the old programs will not recognize the new options, but their code can be reused to deal with the old options.

6.3. Record and variant types

Components of tuple and option types are ordered; this has the advantage that tuple selection can be performed in constant time (typically in one machine instruction), and case discrimination can also be performed in constant time (via a branch table). However, it is interesting to examine versions of tuple and option types that are unordered; these are called *record* and *variant* types respectively. Record selection and variant discrimination cannot be performed quite as efficiently (the difference in practice is a small constant), but have interesting properties.

Records are named collections of values, where all the names are distinct. Unlike tuples, record components are unordered, they must always be named, and they cannot introduce types[6].

Record x_1:A_1 .. x_n:A_n **end**

This is the type of a record containing (a permutation of) the required named components:

record x_1:A_1=a_1 .. x_n:A_n=a_n **end**

For example, here is a record type and a function operating on it. Field selection is achieved by the dot notation, as in tuples:

```
• Let T = Record a:Int b:Bool end;
» Let T::TYPE = Record a:Int b:Bool end
• let f(r:T):Int = if r.b then r.a else 0 end;
» let f(r:T):Int = <fun>
• f(record b=true a=3 end);
» 3 : Int
```

As for tuples, a record type *A* is a subtype of a record type *B* if *A* has all the components of *B* (according to their names) and the component types in *A* are subtypes of the corresponding component types in *B*. But since record types are unordered, subtyping hierarchies built out of record types can form arbitrary directed acyclic graphs instead of trees, hence emulating *multiple inheritance* among class signatures.

Note that although record types are unordered, typechecking does not blow up combinatorially: record types can be sorted by label name, and matching of record types is then still a linear process. The same applies to variant types.

Adapting our previous example:

```
• Let Object =
    Record age:Int end;
```

6 Types in tuple are useful only because they create dependencies based on order.

- **Let** `Vehicle =`
  Record `age:Int speed:Int` **end;**
- **Let** `Machine =`
  Record `age:Int fuel:String` **end;**
- **Let** `Car =`
  Record `age:Int speed:Int fuel:String` **end;**

This produces the following subtyping diagram:

```
            <:      Vehicle      <:
       Car                           Object
            <:      Machine      <:
```

Because of multiple inheritance, it is not possible to compute statically the displacement of a field of a given name in an arbitrary record. Hence, some form of run-time lookup is required. This can be implemented rather efficiently through caching techniques that remember where a given name was found in a record "last time", and do a full lookup only when this fails. With this scheme, the majority of accesses are still done in constant time (maybe 5 machine instructions), and there is gentle degradation when type hierarchies become more complex and cache hits tend to decrease [Cardelli 86].

A variant type is an unordered named collection of types, where the names are distinct. A variant is a tagged value, with the value matching the proper branch of the relevant variant type.

Variant x_1:A_1 .. x_n:A_n **end**

variant *x* **of** *A* **with** *a* **end**

Otherwise, variants and variant types are similar to options and option types, along with the `?` and `!` operators and the **case** construct[7].

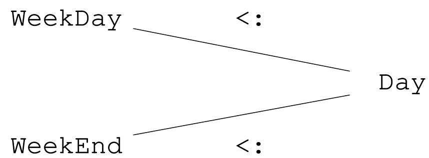
A variant type *A* is a subtype of a variant type *B* if *B* has all the components of *A* (according to their names) and the component types in *A* are subtypes of the corresponding component types in *B*. Again, components in a variant type are unordered.

For example, if we define the following *enumeration* types:

- **Let** `Day =`
  Variant `mon,tue,wed,thu,fri,sat,sun:Ok` **end;**
- **Let** `WeekDay =`
  Variant `mon,tue,wed,thu,fri:Ok` **end;**
- **Let** `WeekEnd =`
  Variant `sat,sun:Ok` **end;**

then we have:

7 The `ordinal` operator is not provided on variants.



Because of lack of order, it is not possible to compile an exact branch table for the case statement. However, efficient caching techniques can still be used: one can remember where a given variant was dispatched "last time" in a case statement, and make a linear search through case branches only if this fails. Again, this degrades gracefully if some precautions are taken, such as sorting the case branches during compilation.

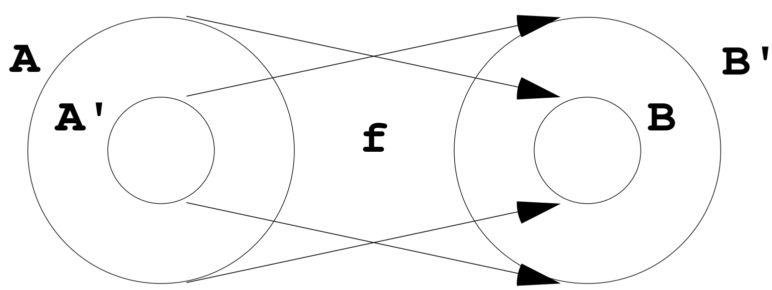
6.4. Higher-order subtypes

In ordinary mathematics, a function from `A` to `B` can also be considered as a function from `A'` to `B'` if `A'` is a subset of `A` and `B` is a subset of `B'`. This is called *contravariance* (in the first argument) of the function space operator.

This also makes good programming sense: this rule is used in some form by all the object-oriented languages that have a sound type system. It asserts that a function working on objects of a given type will also work on objects of any subtype of that type, and that a function returning an object of a given type can be regarded as returning an object of any supertype of that type.

Hence the subtyping rule for function spaces asserts that **`All`**`(x:A)B <: `**`All`**`(x:A')B'` if `A'<:A` and (assuming `x:A'`) `B<:B'`.

By adopting this rule for functions, we take "ground" subtyping, at the data-structure level, and "lift" it to higher function spaces. Since object-oriented programming is based on ground subtyping, and



functional programming is based on higher-order functions, we can say that the rule above unifies functional and object-oriented programming, at least at the type level.

6.5. Bounded universal quantifiers

Subtyping increases flexibility in typing via subsumption, but by the same mechanism can cause loss of type information. Consider the identity function on objects; when applied to a car, this returns an object:

```
•   let objectId(object:Object):Object = object;
•   objectId(myCar);
»   record age=3 end : Object
```

Here we have unfortunately forgotten that `myCar` was a `Car`, just by passing it through an identity function.

This problem can be solved by providing more type information:

```
•   let objectId(A<:Object)(object:A):A = object;
•   ObjectId(:Car)(myCar);
»   record age=3 speed=120 fuel="gas" end : Car
```

Note the signature `A<:Object`; `objectId` is now a polymorphic function, but not polymorphic in any type; it is polymorphic in the subtypes of `Object`. It takes a type which is any subtype of `Object`, then a value of that type, and returns a value of that type.

The signature `A<:Object` is called a *bounded (universal) quantifier*; in fact it is not a new kind of signature because it can be interpreted as an abbreviation for `A::POWER(Object)`.

Bounded universal quantifiers have the effect of integrating polymorphism with subtyping, and they provide more expressive power than either notion taken separately.

6.6. Bounded existential quantifiers

Bounded quantifiers can also be used in tuple types, where they are called *bounded existential quantifiers*. They provide a weaker form of type abstraction, called partial type abstraction; here for example we have an abstract type `A` which is not completely abstract: it is known to be an `Object`:

```
•   Let U =
       Tuple A<:Object a:A f(x:A):Int end;
```

It can be implemented as any subtype of `Object`; in this case it is implemented as a `Vehicle`.

```
•   let u:U =
       tuple
          Let A<:Object = Vehicle
          let a:A = myVehicle
          let f(x:A):Int = x.speed
       end;
```

The interaction between type abstraction and inheritance is a delicate topic; in many languages inheritance violates type abstraction by providing access to implementation details of superclasses. Here we have mechanisms that smoothly integrate abstract types and subtyping: the partially abstract types above, and the abstract subtypes we have seen in a previous section.

6.7. Auto subtypes

An auto type is a subtype of another auto type if the respective components (with matching names and order) are in the subtype or subkind relation.

The first component of an auto type can be any subkind of `TYPE`; it is therefore possible to form a sum of a restricted class of types:

```
•   Let AnyObject = Auto A<:Object with a:A end;
•   let aCar:AnyObject = auto :Car with myCar end
```

Then, the **`inspect`** construct can be used to discriminate at run-time between the subtypes of a given type, in this case between the subtypes of `Object`. This operation is widely used in object-oriented languages such as Simula67, Modula-3, and Oberon.

6.8. Mutable subtypes

The best way to explain the relations between **Out** and **Var** types is to take **Out** as a primitive operator, and consider **Var**(A) to be defined as a *type conjunction* [Reynolds 88]:

Var(A) intended as **In**(A) $\cap$ **Out**(A),

This means that **Var**(A) is the type of locations that have *both* type **In**(A) and type **Out**(A). Here **In**(A) is intended as the type of locations from which we can only read an A, and **Out**(A) as the type of locations to which we can only write an A. Hence **Var**(A) is the type of read-write locations.

Type conjunction has the following subtyping properties:

$A\cap B <: A$
$A\cap B <: B$
$A<:B \wedge A<:C \;\Rightarrow\; A <: B\cap C$

from which we may also derive (using reflexivity and transitivity of <:, and writing $A<:>B$ for $A<:B \wedge B<:A$):

$A\cap A <:> A$
$A\cap B <:> B\cap A$
$A\cap(B\cap C) <:> (A\cap B)\cap C$

$A<:C \;\Rightarrow\; A\cap B <: C$
$A <: B\cap C \;\Rightarrow\; A<:B$
$A<:B \wedge C<:D \;\Rightarrow\; A\cap C <: B\cap D$

The conversions between A and **In**(A) are rather trivial; they are performed by **let** variable declarations (A->**In**(A)) and by variable access (**In**(A)->A), with subtyping rule $A<:B \Rightarrow$ **In**(B)<:**In**(B). For simplicity, in Quest we identify the types A and **In**(A), and take their conversions as implicit. This amounts to assuming **In**(A)<:A and A<:**In**(A).

The **Out** types are much more interesting. First note that we assume neither A<:**Out**(A) nor **Out**(A)<:A (this would amount to **In**(A)<:**Out**(A) and **Out**(A)<:**In**(A)). The fundamental subtyping rule for **Out** is suggested by the following reasoning: if every A is a B by subtyping, then a place where one can write a B is also a place where one can write an A. This means that **Out** is contravariant:

(1) $A<:B \;\Rightarrow\;$ **Out**(B)<:**Out**(A)

This has a nice consequence: **out** parameters appear in contravariant positions (on the "left" of function arrows), but as parameters they should behave covariantly since they are really function results. The contravariant rule for **Out** has the effect of neutralizing the contravariant rule for functions.

From the subtyping rules given so far we can easily infer:

(2') **Var**(A) <: A
(3') **Var**(A) <: **Out**(A)

with the more general forms:

(2) A<:B ⇒ **Var**(A)<:B
(3) B<:A ⇒ **Var**(A)<:**Out**(B)

and, putting the last two together:

(4) (A<:B) ∧ (B<:A) ⇒ **Var**(A)<:**Var**(B)

Since **Var**(A)<:A the conversion from the former to the latter is implicit and automatic; nothing special is needed to fetch the contents of a mutable location. This rule also says that any structure that can be updated can be regarded as a structure that cannot be updated, and passed to any function that will not update it.

Since **Var**(A)<:**Out**(A), it becomes possible to pass a mutable location as an **out** parameter, but the explicit conversions **@** and **var** are required to prevent the automatic fetching of mutable locations described above[8]:

```
• let var a = 3;
• let f(x:Int out y:Int):Ok = y := x;
               (* i.e. f:All(x:Int y:Out(Int))Ok *)
• f(3 @a);     (* a becomes 3 *)
• f(3 var(0)) (* dummy location initially 0 becomes 3 *)
```

Now these conversions can be given a precise typing:

```
@:   Out(A)->Out(A)
var: A->Var(A)
```

In the example, they have been used with types:

```
@:   Var(Int)->Out(Int)  :>  Out(Int)->Out(Int)
var: Var(Int)->Out(Int)  :>  Int->Var(Int)
```

In-out **var** parameters could be used in addition to **out** parameters, but because of (4) they would be less flexible from a subtyping point of view. For simplicity, they are left out of the language[9].

The following example shows that relaxing rule (4) to A<:B ⇒ **Var**(A)<:**Var**(B) is semantically unsound:

```
• let r =
     tuple
        let var a =
           tuple let b=3 end
     end;
• let f(r:Tuple var a:Tuple end end):Ok =
     r.a := tuple end;
```

8 These conversions are being required more for making programs easier to read than out of algorithmic necessity.
9 Note: the implicit conversion **Var**(A)<:A is problematic for a natural implementation of in-out parameters as pointer-displacement pairs.

- `f(r);        (* Typechecks! *)`
- `r.a.b;       (* Crash! *)`

Here the invocation of `f(r)` removes the `b` field from inside `r`, but `r` maintains its old type. Hence, a crash occurs when attempting to extract `r.a.b`. Note how everything typechecks under the weaker subtyping rule for **`Var`**.

Finally, the subtyping rule for arrays is given by considering them, for the purpose of subtyping, as functions from integers to mutable locations:

`Array(A)` regarded as **`All`**`(:Int)` **`Tuple :Var`**`(A)` **`end`**

Hence we obtain the rule:

(5) $(A<:B) \wedge (B<:A) \Rightarrow$ `Array(A)<: Array(B)`

which requires the `Array` operator to be both covariant and contravariant.

6.9. Recursive subtypes

As for type equivalence, two recursive types are in subtype relation when their infinite expansions are in subtype relation. (Again, this condition can be tested in finite time.)

We can therefore produce hierarchies of recursively defined types. The following example also involves the subtyping rules for **`Var`** types, defined in the previous section.

```
• Let List(A::TYPE)::TYPE =
     Rec(List::TYPE)
        Option
           nil
           cons with head:A tail:List end
        end;

• Let VarList(A:TYPE)::TYPE =
     Rec(VarList::TYPE)
        Option
           nil
           cons with var head:A var tail:VarList end
        end;
```

Then we obtain, for example, `VarList(Int) <: List(Int)` by a non-trivial use of the `Var(A)<:A` rule. Note that `VarList <: List` does not make sense since subtyping is defined between types, not between operators.

7. Large programs

The usefulness, even necessity, of typeful programming is most apparent in the construction of large systems. Most serious programming involves the construction of large programs, which have the property that no single person can understand or remember all of their details at the same time (even if they are written by a single person).

Large programs must be split into modules, not only to allow compilers to deal with them one piece at a time, but also to allow people to understand them one piece at a time. In fact it is a good idea to split even relatively small programs into modules. As a rule of thumb, a module should not exceed a few hundred lines of code, and can be as small as desired.

A lot of experience is required to understand where module boundaries should be located [Parnas 72]. In principle, any part of a program which could *conceivably* be reused should form a module. Any collection of routines which maintain an internal invariant that could be violated by careless use should also form a module. And almost every user-defined data type (or collection of closely related data types) should form a module, together with the relevant operations. Module boundaries should be located wherever there is some information that can or should be hidden; that is, information not needed by other modules, or information that can be misused by other modules.

7.1. Interfaces and modules

Module boundaries are called *interfaces*. Interfaces declare the types (or kinds) of identifiers supplied by modules; that is, they describe how modules may plug together to form systems. Many interfaces provide one or more abstract data types with related operations; in this case they should define a consistent and complete set of operations on the hidden types. Other interfaces may just be a collection of related routines, or a collection of types to be used by other modules. Interfaces should be carefully documented, because they are the units of abstraction that one must understand in order to understand a larger system.

Both interfaces and modules may *import* other interfaces or modules; they all *export* a set of identifiers. A module *satisfies* (or *implements*) an interface if it provides all the identifiers required by the interface; a module may also define additional identifiers for its own internal purposes.

In Quest, interfaces and modules can be entered at the top level, although normally they will be found in separate files. Each interface, say `A`, can be *implemented* by many modules, say `b` and `c`. Each module specifies the interface it implements; this is written as `b:A` and `c:A` in the module headings:

```
• interface A              • module b:A
  import ..                  import ..
  export                     export
     ..                         ..
  end;                       end;
```

Both interfaces and modules may explicitly *import* other interfaces and modules in their headings (the interface implemented by a module is implicitly imported in the module). The following line imports: interfaces `C`, `D`, and `E`; module `c` implementing `C`; and modules `d1` and `d2` both implementing `D`.

```
import  c:C  d1,d2:D  :E
```

Imported modules are just tuples; values and types can be extracted with the usual "dot" notation. Imported interfaces are just tuple types. Since modules are tuples, they are first class: it is possible to pass modules around, store them in variables, and choose dynamically between different implementations of the same interface.

One of the main pragmatic goals of modules and interfaces is to provide separate compilation. When an interface is evaluated, a corresponding file is written, containing a compiled version of the interface. Similarly, when a module is evaluated, a file is written, containing a compiled version of the module. When interfaces and modules are evaluated, all the imported interfaces must have been compiled, but the imported modules do not have to be. The import dependencies of both modules and interfaces must form a directed acyclic graph; that is, mutually recursive imports are not allowed to guarantee that the linking process is deterministic.

Modules are *linked* by importing them at the top level[10].

- **`import`** *`b:A;`*

» *`let b:A = ..`*

At this point all the interfaces and modules imported directly by *b*, *A*, and recursively by their imports, must have been compiled. The result is the definition at the top level of a tuple *b*, of type *A*, from which values and types can be extracted in the usual fashion.

A compiled module is like a function closure: nothing in it has been evaluated yet. At link time, the contents of compiled modules are evaluated in the context of their imports. In the linking process, a single copy of each module is evaluated (although it may be imported many times). Evaluation happens depth-first along import chains, and in the order in which modules appear in the import lists.

Version checking is performed during linking, to make sure that the compiled interfaces and modules are consistent.

7.2. Manifest types and kinds

When programming with modules, it is very convenient to define a type in an interface and then have other modules and interfaces refer to that definition. But so far we have shown no way of doing this: types can be defined in bindings (module bodies) where they are not accessible to other modules, and signatures (interfaces) can only contain specifications of abstract types[11].

Hence a separate mechanism is provided to introduce type and kind definitions in signatures; such types and kinds are then called *manifest*. This should be seen just as a convenience: these types and kinds could be expanded at all their points of use, and everything would be the same. In particular, manifest entities appearing in signatures and bindings are (conceptually) removed and expanded before typechecking.

We augment our syntactic classes as follows:

• Signatures: they can now contain manifest kinds and types, introduced by the **`DEF`** and **`Def`** keywords respectively. The identifiers thus introduced can be mentioned later in the signature. These definitions are particularly useful in signatures forming tuple types (e.g. interfaces), so that these definitions can be later extracted from the tuple type.

`DEF` *`U=K`*	is a manifest kind definition.
`Def` *`X::K=A`*	is a manifest type definition. (Similarly for **`Def Rec`**.)

• Kinds: if *X* is a type identifier bound to a tuple type (e.g. an interface), and *U* is a kind variable introduced by **`DEF`** in that tuple type, then:

`X_U`	is a kind.

10 In view of dynamic linking, it might be nicer to introduce **`import`** as a new form of binding that would not be restricted to the top level.

11 In fact we could write *`X<:A`* (a specification of a partially-abstract type) in an interface, thereby making *X* publicly available as (a subtype of) the known type *A*. However according to our signature matching rules, any implementation of such an interface should then "implement" *X* , probably just by duplicating *A*'s possibly complex definition.

• Types: if A denotes a tuple type, and X is a type variable declare by **Def** in that tuple type, then:

A_X is a type.

The "dot" notation ($x.Y$ and $x.y$) is used to extract types and values out of binding-like structures, while the "underscore" notation (X_U and A_X) is used to extract manifest kinds and types out of signature-like structures.

7.3. Diamond import

In module systems that admit multiple implementations of the same interface, there must be a way of telling when implementations of an interface imported through different paths are the "same" implementation [MacQueen 84].

This is called the diamond import problem. A module `d` imports two modules `c` and `b` which both import a module `a`. Then the types flowing from `a` to `d` through two different import paths are made to interact in `d`. In other words, if `a:A` and `b:A`, then `a.T` must not match `b.T`, unless we know statically that `a` and `b` are the same *value* (the same implementation of the interface `A`).

In the Quest context (without parametric modules), this is solved by assuming a global name space of externally compiled modules, so that `a.T` matches `b.T` precisely when `a` and `b` are the same identifier.

- ```
  interface A
  export
     T::TYPE
     new(x:Int):T
     int(x:T):Int
  end;
  ```
- ```
  module a:A
  export
     Let T::TYPE = Int
     let new(x:Int):T = x
     and int(x:T):Int = x
  end;
  ```
- ```
  interface B
  import a:A
  export
     x:a.T
  end;
  ```
- ```
  module b:B
  import a:A
  export
     let x = a.new(0)
  end;
  ```
- ```
  interface C
  import a:A
  export
     f(x:a.T):Int
  end;
  ```
- ```
  module c:C
  import a:A
  export
     let f(x:a.T):Int = a.int(x)+1
  end;
  ```
- ```
  interface D
  export
     z:Int
  end;
  ```
- ```
  module d:D
  import b:B c:C
  export
     let z = c.f(b.x)
  end;
  ```

Note that the application `c.f(b.x)` in module `d` typechecks because the `a` imported by `b` and the `a` imported by `c` are the "same" implementation of the interface `A`, since `a` is a global external name.

To illustrate the correspondence between interfaces and signatures, and between modules and bindings, we can rephrase the diamond import example as follows.

- ```
  Let A::TYPE =
     Tuple
        T::TYPE
        new(x:Int):T
        int(x:T):Int
     end;
  ```
- ```
  let a:A =
     tuple
        Let T::TYPE = Int
        let new(x:Int):T = x
        and int(x:T):Int = x
     end;
  ```
- ```
  Let B::TYPE =
     Tuple
        x:a.T
     end;
  ```
- ```
  let b:B =
     tuple
        let x = a.new(0)
     end;
  ```
- ```
  Let C::TYPE =
     Tuple
        f(x:a.T):Int
     end;
  ```
- ```
  let c:C =
     tuple
        let f(x:a.T):Int = a.int(x)+1
     end;
  ```
- ```
  Let D::TYPE =
     Tuple
        z:Int
     end;
  ```
- ```
  let d:D =
     tuple
        let z = c.f(b.x)
     end;
  ```

In this case, `c.f(b.x)` typechecks because the types of `c.f` and `b.x` both refer to the *same* variable `a` which is lexically in the scope of both `c` and `b`.

8. Huge programs

When programs first started becoming *large*, composed of hundreds of procedures, it became necessary to process them in chunks, both for practical implementation considerations and for better structuring. Eventually this trend led to modules and interfaces.

Today programs are starting to become *huge*, composed of hundreds of interfaces, and the same problems are reappearing one level up. Modules and interfaces, as we have considered them so far, have a flat structure; when there are too many of them it is hard to figure out the organization of a system. More importantly, a group of interfaces may be intended to be private to a given subsystem, and should not be accessible to other subsystems.

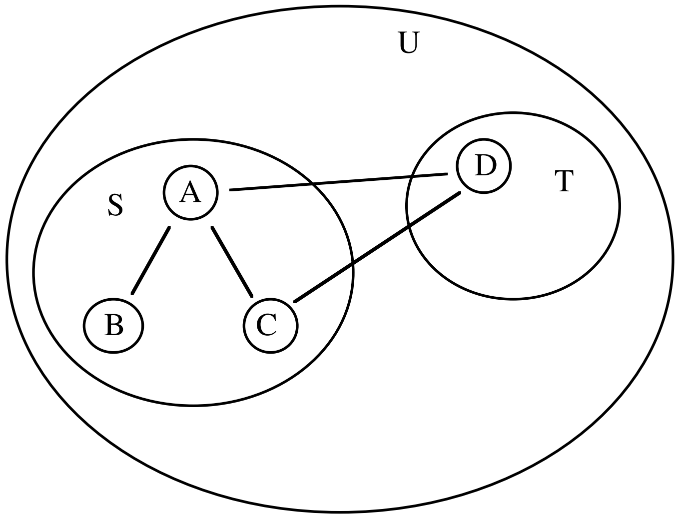
It would clearly be desirable to be able to group interfaces into systems which could then be grouped into larger systems, and so on. We cannot just syntactically nest modules and interfaces and hope to solve the problem: separate compilation is then prevented. Also remember that the flat structure of modules is an advantage because it escapes the strict block scoping rules.

In this section, we examine a way of describing *systems of interfaces*, while remaining compatible with our previous notions of modules and interfaces. One of the main goals is to be able to reorganize the structure of a subsystem (including its inner interfaces) without affecting the other subsystems that use it; this is similar to a separate compilation criterion for modules.

We classify systems as *open*, *closed*, and *sealed*. To make an analogy with hardware, an open system is like a hardware box without a cover; anybody can plug wires into it. A closed system is a hardware box with a cover but with expansion slots: one can plug wires only into the outside connectors, but one can also add a new piece of hardware (with related external connectors) that has internal access to the box. Finally, a sealed system can be used only through the provided connectors.

8.1. Open systems

Consider the following system organization, illustrated in the diagram. A system `U` is composed of two (sub-) systems `S` and `T`. System `S` is composed of three interfaces `A`, `B`, and `C` (where `A` imports `B` and `C`) with corresponding modules `a`, `b`, and `c`. System `T` is composed of a single exported interface `D`, which imports `A` and `C`.



We express this arrangement by the following notation:

- **system** U
  end;
- **system** S **of** U
  end;
- **system** T **of** U
  end;
- **interface** A **of** S
  import :B :C
  export ...
  end;
- **interface** B **of** S
  export ...
  end;
- **interface** C **of** S
  export ...
  end;
- **interface** D **of** T
  import :A :C
  export ...
  end;

Each interface can import all other interfaces directly, without mentioning which systems they belong to; this seems necessary to facilitate system reorganizations. Moreover, a new interface `E` could join system `T` just by claiming to belong to it.

The reason these systems are open is that there are no restrictions regarding membership or visibility. Their structure can be reorganized very easily just by changing membership claims and without affecting unrelated parts; this flexibility is important in large evolving systems. At the same time, the membership claims provide some degree of structuring.

8.2. Closed systems

The next step is to restrict visibility across system boundaries; system `S` may declare that only interfaces `A` and `B` and module `a:A` are exported, thereby preventing `D` from importing `C`. Similarly, system `U` may declare that only the interface `A` of `S` is available outside `U`. In general, when an interface `I` imports another interface `J`, either there must be no closed system barriers between I and J, or if there are such barriers, J must be exported through all of them.

- ```
  system U
  export :A of S
  end;
  ```
- ```
  system S of U
  export a:A :B
  end;
  ```
- ```
  interface D of T
  import :A
  export    ...
  end;
  ```

A closed system still does not prevent interfaces from spontaneously joining it. Moreover we allow interfaces to be members of multiple systems. Hence interface D could counteract the closure of system S by claiming to belong to S too, thereby being able again to import C:

- ```
  interface D of T,S
  import  :A :C
  export    ...
  end;
  ```

What is the point of closing a system and still allowing interfaces to join it? The idea is that in joining a system one explicitly declares the intention of depending on its internal structure, while simply importing an interface provided by a system declares the intention of not depending on any implementation details of that system.

Note that we still have a single name space for interfaces and modules. This is a doubtful feature, but this way interfaces and modules can be moved from one system to another without the programmer having to modify all their clients.

8.3. Sealed systems

The final step is then to prevent "unauthorized" membership in a system. To seal system S, a line is added that explicitly lists all the interfaces (and modules) that are allowed to belong to S, in this case A, B and C (and modules a and c). At this point D is again cut out of S and prevented access to C, although D could be added to the component list of S if desired:

- ```
  system S of U
  components a:A :B c:C
  export a:A :B
  end;
  ```

A sealed system provides a solution to the problem of implementing large abstract data types. In ordinary module systems, if the implementation of an abstract type T spans several modules, say m1 and m2, then the representation of T must be made public through an interface, say I, to be imported by m1 and m2. This way the type ceases to be abstract, because everybody can see T through interface I.

The solution now is to add a new interface J exporting a real abstract type T and its operations; the implementation of J realizes T by I_T, imported from I, and implements the operations by importing them from m1 and m2. Then everything is wrapped into a sealed system that exports only J.
  ```

- ```
  interface I of W
  export
     Def T = ...
  end;
  ```
- ```
  module m:I
  export
  end;
  ```
- ```
  interface I1 of W
  import I
  export
     f: ...
  end;
  ```
- ```
  module m1:I1
  import I
  export
     let f = ...
  end;
  ```
- ```
  interface I2 of W
  import I
  export
     g: ...
  end;
  ```
- ```
  module m2:I2
  import I
  export
     let g = ...
  end;
  ```
- ```
  interface J of W
  export
     T::TYPE
     f,g: ...
  end;
  ```
- ```
  module n:J
  import :I m1:I1 m2:I2
  export
     Let T = I_T
     let f = m1.f and g = m1.g
  end;
  ```
- ```
  system W
  components m:I m1:I1 m2:I2 n:J
  export n:J
  end;
  ```

The process of closing a system may reveal unintentional dependencies that may have accumulated during development. The process of sealing a system may reveal deficiencies in the system interface that have to be fixed.

It is expected that, during its evolution, a software system will start as open to facilitate initial development. Then it will be closed when relatively stable interfaces have been developed and the system is ready to be released to clients. However, at this stage developers may still want to have easy access to the closed system, and they can do so by joining it. When the system is finally quite stable it can be sealed, effectively forming a large, structured abstract type, for example an operating system or file system interface.

9. System programs

As we mentioned in the introduction, a language cannot be considered "real" unless it allows some form of low-level programming; for example a "real" language should be able to express its own compiler, including its own run-time system (memory allocation, garbage collection, etc.). Most interesting systems, at some point, must get down to the bit level. One can always arrange that these parts of the system are written in some other language, but this is unsatisfactory. A better solution is to allow these low-level parts to be written in some variation (a subset with some special extensions) of the language at hand, and be clearly marked as such. An extreme situation is the infamous "assembly language insert", but one can find much better solutions that are relatively or completely implementation-independent, that provide good checking, and that localize responsibility when unpredicted behavior results. Some such mechanisms are considered in this section.
  ```

9.1. Dynamic types

Static typechecking cannot cover all situations. One problem is with giving a type to an *eval* function, or to a generic *print* function. A more common problem is handling in a type-sound way data that lives longer than any activation of the compiler [Atkinson Bailey Chisholm Cockshott Morrison 83]. These problems can be solved by the introduction of *dynamic* types, here realized by a built-in module `dynamic:Dynamic` (see the Appendix).

Objects of type `Dynamic_T` should be imagined as pairs of a type and an object of that type. In fact, `Dynamic_T` is defined as the type **`Auto`** `A::TYPE` **`with`** `a:A` **`end`**, and many of the relevant operations are defined in terms of operations on auto types. In particular, the type component of a `Dynamic_T` object must be a closed type, and it can be tested at run-time through **`inspect`**.

One can construct dynamic objects as follows:

```
•   let d3:Dynamic_T = dynamic.new(:Int 3);
»   let d3:Dynamic_T = auto :Int with 3 end
```

These objects can then be *narrowed* to a given (closed) type via the `dynamic.be` operation. If the given type matches the type contained in the dynamic, then the value contained in the dynamic is returned. Otherwise an exception is raised (narrowing is just a special case of **`inspect`**):

```
•   dynamic.be(:Int d3);
»   3: Int
•   dynamic.be(:Bool d3);
»   Exception: dynamicError
```

The matching rules for narrowing and inspecting are the same as for static typechecking, except that the check happens at run-time.

Since an object of type dynamic is self-describing it can be saved to a file and then read back and narrowed, maybe in a separate programming session:

```
•   let wr = writer.file("d3.dyn");
•   dynamic.extern(wr d3);          (* Write d3 to file *)
•   writer.close(wr);
...
•   let rd = reader.file("d3.dyn");
•   let d3 = dynamic.intern(rd);     (* Read d3 from file *)
```

The operations `extern` and `intern` preserve sharing and circularities within a single dynamic object, but not across different objects. All values can be made into dynamics, including functions and dynamics. All dynamic values can be *externed*, except readers and writers; in general it is not meaningful to extern objects that are bound to input/output devices.

9.2. Stack allocation

Memory allocation in Quest is mostly dynamic; variables are kept on a stack but they normally refer to heap-allocated data structures such as tuples, functions, etc. Heap memory is reclaimed by garbage collection.

In a language based on dynamic storage allocation and garbage collection, how does one write a garbage collector? The solution involves identifying an allocation-free subset of the language, and writing the garbage collector in that subset. In other words, one must distinguish between *stack allocation*, where storage is easily reclaimed, and *heap allocation*, which requires collection.

Ordinary languages make a distinction between memory structures and pointers to memory structures. A structure denoted by a variable is always stack allocated, while a structure denoted by a pointer can be either stack or heap allocated. This allows fine control on what goes where, but requires programmers to be conscious of this distinction even when everything is heap allocated. In this situation the default, so to speak, is stack allocation and explicit actions are required to achieve heap allocation.

Quest takes the opposite default. Normally, every object is a pointer to a dynamically allocated structure. This leads to more transparent programs for symbolic manipulation situations. For system programming, however, the distinction must be reintroduced. In Quest this requires some explicit actions.

The mechanism we adopt is superficially similar to the one used for **Var** types: variables can be declared of **Still** types. Values of **Still**(*A*) type are values of type *A* allocated "flat", without an additional pointer referring to them, hence they must be part of some other structure, such as the stack or a tuple in the heap[12].

A stack-allocated variable can be declared by a **still** keyword:

```
•  Let A = Tuple x,y:Int end;
»  Let A::TYPE = Tuple x,y:Int end
•  let still a:A = tuple let x=3 let y=5 end;
»  let still a:A = tuple let x=3 let 7=5 end
```

Variables of **Still** types must be passed *by-alias*, as we have seen for **Var** types:

```
•  let f(still a:A):Int = a.x+a.y;
»  let f(still a:A):Int = <fun>
•  f(@a);
»  8: Int
```

Restrictions have to be imposed on what is legal with **Still** types.

- Global variables of functions cannot have **Still** types. (This could create dangling references.)
- Functions or expressions cannot return objects of **Still** type. (This too could create dangling references.)
- Objects of **Still** types cannot be the source or target of an assignment; only their components can. (Such assignment would have a very different semantics than ordinary assignment, and would cause problems because polymorphic functions would not know the size of such objects.)
- Objects of **Still** type cannot be compared by **is** or **isnot**. (Same reason as for assignment.)
- Objects of types **Still**(String), **Still**(Array(*A*)), and **Still**(**All**(*S*)*A*) must be initialized to compile-time known sizes.

There is no reason to declare variables of types such as **Still**(Int), since integers are always allocated "flat" anyway, but this is not forbidden.

The only subtyping rule is **Still**(*A*)**<:Still**(*B*) if *A***<:***B*.

An important property of this mechanism is that removing the **Still**, **still**, and **@** annotations does not change the semantics of a program; hence these annotations could be seen as optimization suggestions to the compiler.

Hence, a garbage collector could be written by using only **Still** types and basic types; if asked, the compiler could then check that there is really no need for heap allocation in a given module. The latter property is true of a module only if it is true of the module body and of all the modules it imports.

---

12 None of **Var**(**Var**(A)), **Var**(**Still**(A)), **Still**(**Var**(A)), or **Still**(**Still**(A)) are allowed.

Something as low-level as a garbage collector may actually require some additional tricks, and these are discussed in the next section.

9.3. Type violations

Most system programming languages allow arbitrary type violations, some indiscriminately, some only in restricted parts of a program. Operations that involve type violations are called *unsound.* Type violations fall in several classes:

Basic-value coercions. These include conversions between integers, booleans, characters, sets, etc. There is no need for type violations here, because built-in interfaces can be provided to carry out the coercions in a type-sound way.

Bit and word operations. These involve operations such as bit-field operations, bit-wise boolean operations, unsigned arithmetic and comparisons, etc. Again, these operations can be provided through a sound built-in `Word` interface.

Address arithmetic. If necessary, there should be a built-in (unsound) interface, providing the adequate operations on addresses and type conversions. Various situations involve pointers into the heap (very dangerous with relocating collectors), pointers to the stack, pointers to static areas, and pointers into other address spaces. Sometimes array indexing can replace address arithmetic.

Memory mapping. This involves looking at an area of memory as an unstructured array, although it contains structured data. This is typical of memory allocators and collectors.

Metalevel operations. This is the most fundamental use of type violations. When an incremental compiler has compiled a piece of code, the compiler must *jump to it* to execute it. There is no way to do this in a type-sound way, since the soundness of this operation depends on the correctness of the entire compiler. Once such execution has returned a value, such a value must be printed according to the static type information: this requires more unsound operations. The returned value can be made into a dynamic and printed accordingly, but the validity of this dynamic value depends on the correctness of the typechecker.

In the Appendix, we suggest a particular type violation mechanism via the built-in interface `Value`, which manipulates raw words of memory. But whatever the type violation mechanisms are, they need to be controlled somehow, lest they compromise the reliability of the entire language.

Cedar-Mesa and Modula-3 use the following idea. Unsound operations that may violate run-time system invariants are called *unsafe*. Unsafe operations can only be used in modules that are explicitly declared unsafe. If a module is declared safe, the compiler checks that (a) its body contains no unsafe operations, and (b) it imports no unsafe interfaces.

We adopt essentially the same scheme here, except we use the keyword **`unsound`**[13]. Unsound modules may advertise an unsound interface. However, unsound modules can also have ordinary interfaces. In the latter case, the programmer of the unsound module guarantees (i.e. proves or, more likely, trusts) that the module is not actually unsound, from an external point of view, although internally it uses unsound operations. This way, low-level facilities can be added to the system without requiring all the users of such facilities to declare their modules unsound just because they import them.

13 Soundness is the property one proves to show that a type system does not permit type violations. The word *unsafe* has a slightly different meaning in the languages above, since some unsound operations are considered safe if they do not violate run-time invariants.

The only unsound operations in Quest are provided by built-in unsound interfaces such as `Value`. Unsound modules can import (implementations of) unsound interfaces and may have unsound or sound interfaces. Sound modules can import only (implementations of) sound interfaces. (These implementations may be unsound.) Sound interfaces are allowed to import (implementations of) unsound interfaces since soundness can be compromised only by operations, not by their types. Finally, the top level is implicitly sound.

The main advantage of this scheme is that if something goes very wrong the responsibility can be restricted to unsound modules.

10. Conclusions

We have illustrated a style of programming based on the use of rich type systems. This is not new in general, but the particularly rich type system we have described, based on type quantifiers and subtypes, extends the state of the art. This rich type structure can account for functional, imperative, algebraic, and object-oriented programming in a unified framework, and extends to programming in the large and, with care, to system programming.

10.1. This style

Typeful programming is based on the notion of *statically checkable properties of a program*. Many such properties can be considered, but currently we know best how to implement static checks involving type properties. Furthermore, we know that not all properties can be statically checked, because of undecidability problems. We also know from practice that such properties must be easily definable so that programmers can predict which programs are legal. The combination of these two facts tells us that one should not consider arbitrarily complex static properties, but only relatively simple ones.

Type systems, even powerful ones, are relatively simple to understand and implement, and relatively predictable. They provide the best known practical compromise between typeless programming and full program verification.

10.2. Other styles

We have focused here on a particular programming style. There are many other programming styles, most of which are compatible with this one, some of which are not; we briefly analyze their relation to typeful programming.

Typeless programming. Truly typeless languages include Assemblers, BCPL and, to a minor extent, C; these languages regard raw memory as the fundamental data structure. Large systems programmed in this style become very hard to debug and maintain, hence we dismiss these languages as obsolete, undisciplined, and unreliable. Although C has a relatively rich notion of typing, the type enforcement is too weak to lead predictably to robust systems.

Type-free programming. This concerns languages with some notion of run-time typing (including strong typing) but no static typing (Lisp, APL, Smalltalk). Unlike the previous category, large systems built this way can be very easily debugged. Nonetheless, this practice does not lead to robustness, reliability or maintainability. Bugs can easily be *fixed*, but cannot easily be *prevented*, and systems cannot be easily restructured. These systems seem to become rapidly incomprehensible with size [Weinreb Moon 81], since no static structure is imposed on them.

Type-free programming is often advocated for beginners [Kemeny Kurtz 71], but languages like ML and Miranda [Turner 85] have demonstrated that some powerful type systems, if desired, can be made completely unobtrusive through type inferencing techniques, and can actually help trap many of the mistakes beginners make.

Functional programming. Languages for functional programming have traditionally been untyped, but most of the recent ones are typed [Turner 85]. Higher-order function types account for this style of programming.

Imperative programming. Virtually all imperative languages (the main exceptions being Assemblers and Basic) have some form of typing. See Modula-2 [Wirth 83] for a modern, highly structured type system. Types are not normally involved in controlling side-effects, but see [Lucassen Gifford 88] which proposes an interesting way to use quantifiers for this purpose.

Object-oriented programming. The original object-oriented language, Simula67 [Dahl Nygaard 66], was typed. Smalltalk [Krasner 83] abandoned typing, but the most recent object-oriented languages are again typed [Stroustrup 86] [Cardelli Donahue Glassman Jordan Kalsow Nelson 89] [Schaffert Cooper Bullis Kilian Wilpolt 86] [Wirth 87]. The notion of subtyping is necessary to capture many of the fundamental notions of object-oriented programming.

Relational programming. There is no reason in principle why languages like Prolog [Kowalski 79] should not be typed. In fact, these languages manipulate very highly structured data that could be typed in a natural way. Some efforts have been made in this direction [Mycroft O'Keefe 84], but this seems to be an open problem.

Algebraic programming. This is a programming style where data abstraction is considered fundamental (object-oriented programming also has this aspect). The notion of abstract types we have described is fundamentally the same as in CLU [Liskov 77], and has strong relations with OBJ2's [Futatsugi Goguen Jouannaud Meseguer 85] (although we handle only *free* algebras, without equations). The idea that existential quantifiers accurately model abstract types comes from [Mitchell Plotkin 85]. Virtually all algebraic programming is conducted in the many-sorted case, which makes it naturally typed.

Concurrent programming. There is a different flavor of concurrent programming that fits with each of the programming styles above; concurrency is, in a sense, orthogonal to style. The question of how types interact with concurrency is an interesting one. On one hand, there seems to be very little relation between the two: concurrency has to do with flow of control, while type systems normally avoid flow of control questions. On the other hand, attempts have been made to use type information to control concurrency [Strom Yemini 83] [Andrews Schneider 88]. In some areas (e.g. shared memory concurrency) static checks are badly needed in order to build reliable systems. Maybe type systems could provide them; this is an open problem.

Programming in the large. This is a programming discipline for organizing large software systems, based on the notions of modules and interfaces [Mitchell Maybury Sweet 79] [Wirth 83]. Large systems are kept consistent by checking that all the interfaces fit together: this is done by typechecking plus version checking. An interface is basically the type of a module, and we have described how modules and interfaces correspond to tuples and tuple types.

System programming. This is a programming discipline where some forms of type violations are necessary. However, many common system programming languages have some notion of typing [Harbison Steele 84] [Mitchell Maybury Sweet 79] [Wirth 83]. This is because, especially in this area, unchecked programs can have errors that are extremely difficult to track down, and typing (or quasi-typing) is necessary to maintain sanity.

Database programming. Quest's `Dynamic` types provide a rudimentary form of data persistence [Atkinson Bailey Chisholm Cockshott Morrison 83]. More interestingly, set and relation types, as defined in [Buneman Ohori 87], can be integrated in the Quest type system [Cardelli 88], and they interact very nicely with subtyping. Relational algebra operators (generalized to higher-order) can then be introduced.

We conclude that the need for typing is characteristic of any serious form of programming, and not of any particular programming style.

10.3. Acknowledgments

Thanks for various discussions to: Roberto Amadio (rules for recursive types), Pierre-Louis Curien (typechecking algorithms), Peppe Longo (models of subtyping), and Benjamin Pierce (dynamic types).

11. Appendix

11.1. Syntax

Id ranges over non-terminal identifiers; A and B range over syntactic expressions.

<table>
<tr><td>Id ::= A</td><td>the non-terminal identifier Id is defined to be A</td></tr>
<tr><td>Id</td><td>a non-terminal symbol</td></tr>
<tr><td>".."</td><td>a terminal symbol ("" is the empty input)</td></tr>
<tr><td></td><td></td></tr>
<tr><td>ide</td><td>an alphanumeric identifier token (`A..Z`, `a..z`, `0..9`, with initial letter) or infix</td></tr>
<tr><td>infix</td><td>a symbolic identifier token (`!@#$%&*_+=-|\`:<>/?`)</td></tr>
<tr><td>char</td><td>a character token (`'a'`, with escapes `\n \b \t \f \\ \'`)</td></tr>
<tr><td>string</td><td>a string token (`"abc"`, with escapes `\n \b \t \f \\ \"`)</td></tr>
<tr><td>int</td><td>an integer token (`2`)</td></tr>
<tr><td>real</td><td>a real token (`2.0`)</td></tr>
<tr><td>A B</td><td>means A followed by B (binds strongest)</td></tr>
<tr><td>A | B</td><td>means A or B</td></tr>
<tr><td>[A]</td><td>means ("" | A)</td></tr>
<tr><td>{A}</td><td>means ("" | A {A})</td></tr>
<tr><td>(A)</td><td>means A</td></tr>
</table>

```
Program ::=
    {[Interface | Module | Linkage | Binding] ";"}

Interface ::=
    ["unsound"] "interface" ide ["import" Import] "export" Signature "end"

Module ::=
    ["unsound"] "module" ide ":" ide ["import" Import] "export" Binding "end"

Linkage ::=
    "import" Import

Import ::=
    {[ideList] ":" ide}

Kind ::=
    ide |
    "TYPE" |
    "POWER" "(" Type ")" |
    "ALL" "(" TypeSignature ")" Kind |
    ide "_" ide |
    "{" Kind "}"

Type ::=
    ide {"." ide} |
    "Ok" | "Bool" | "Char" | "String" | "Int" | "Real" |
    "Array" "(" Type ")" | "Exception" |
    "All" "(" Signature ")" Type |
    "Tuple" Signature "end" |
    "Option" OptionSignature "end" |
    "Auto" [ide] HasKind "with" Signature "end" |
    "Record" ValueSignature "end" |
    "Variant" ValueSignature "end" |
    "Fun" "(" TypeSignature ")" [HasKind] Type |
    "Rec" "(" ide HasKind ")" Type |
    Type "(" TypeBinding ")" |
    Type infix Type |
    Type "_" ide |
    "{" Type "}"
```

```
Value ::=
    ide |
    "ok" | "true" | "false" | char | string | integer | real |
    "if" Binding ["then" Binding] {"elsif" Binding ["then" Binding]} ["else" Binding] "end" |
    "begin" Binding "end" |
    "loop" Binding "end" | "exit" |
    "while" Binding "do" Binding "end" |
    "for" ide "=" Binding ( "upto" | "downto" ) Binding "do" Binding "end" |
    "fun" "(" Signature ")" [":" Type] Value |
    Value "(" Binding ")" |
    Value (infix | "is" | "isnot" | "andif" | "orif" | ":=") Value |
    "tuple" Binding "end" |
    "auto" (["let" ide [HasKind] "=" ] ":") Type "with" Binding "end" |
    "option" (ide | "ordinal" "(" Value ")") "of" Type ["with" Binding] "end" |
    "record" ValueBinding "end" |
    "variant" ["var"] ide "of" Type ["with" Value] "end" |
    Value ("." | "?" | "!") ide |
    "case" Binding CaseBranches "end" |
    ("array" | Value) "of" (Binding "end" | "(" Binding ")") |
    Value "[" Value "]" [ ":=" Value ] |
    "inspect" Binding InspectBranches "end" |
    "exception" ide [":" Type] "end" |
    "raise" Value ["with" Value] ["as" Type] "end"|
    "try" Binding TryBranches "end" |
    "{" Value "}"

Signature ::=
    { TypeSignature |
      ["var" | "out"] IdeList (HasType|ValueFormals) | HasMutType }

TypeSignature ::=
    { "DEF" KindDecl |
      "Def" ["Rec"] TypeDecl |
      [IdeList] HasKind }

ValueSignature ::=
    {["var"] IdeList HasType}

OptionSignature ::=
    ["with" Signature "end"] {IdeList ["with" Signature "end"]}

Binding ::=
    { "DEF" KindDecl |
      "Def" ["Rec"] TypeDecl |
      "Let" ["Rec"] TypeDecl |
      "let" ["rec"] ValueDecl |
      "::" Kind |
      ":" Type |
      "var" "(" Value ")" |
      "@" Value |
      Value }

TypeBinding ::=
    { Type }

ValueBinding ::=
    {["var"] ide "=" Value}

KindDecl ::=
    ide "=" Kind |
    KindDecl "and" KindDecl
```

```
TypeDecl ::=
    ide [HasKind | TypeFormals] "=" Type |
    TypeDecl "and" TypeDecl

ValueDecl ::=
    ["var"] ide [HasType | ValueFormals] "=" Value |
    ValueDecl "and" ValueDecl

TypeFormals ::=
    {"(" TypeSignature ")"} HasKind

ValueFormals ::=
    {"(" Signature ")"} ":" Type

CaseBranches ::=
    {"when" IdeList ["with" ide [":" Type]] "then" Binding}
    ["else" Binding]

InspectBranches ::=
    {"when" Type ["with" {IdeList [":" Type]}] "then" Binding}
    ["else" Binding]

TryBranches ::=
    {"when" Binding ["with" ide [":" Type]] "then" Binding}
    ["else" Binding]

HasType ::=
    ":" Type

HasMutType ::=
    ":" Type | ":" "Var" "(" Type ")"

HasKind ::=
    "<:" Type | "::" Kind

IdeList ::=
    ide | ide "," IdeList
```

Operators:

niladic:	`Ok Bool Char String Int Real ok true false`
prefix monadic:	`not extent ordinal`
infix:	`is isnot andif orif`
	`/\ \/ + - * / % < > <= >= <> ++ -- ** // ^^ << >> <<= >>=`

Keywords:

```
ALL DEF POWER TYPE All Array Auto Def Exception Fun Let Option Out Rec Record Tuple Var Variant
and array as auto begin case do downto else elsif end exception exit export for fun if import
inspect interface let loop module of option out raise rec record then try tuple unsound upto var
variant when while with ? ! : :: <: := = _ @
```

Notes:

The `@` keywords can appear only in actual-parameter bindings.
Bindings evaluated for a single result must end with a value component (modulo manifest declarations).
Bindings in a listfix construct may begin with a type (the type of array elements) and must then contain only values.
Recursive value bindings can contain only *constructors*, i.e. functions, tuples, etc.

11.2. Type rules

A complete semantics of Quest would take many pages; we think it could be written using Structural Operational Semantics [Plotkin 81], also using the techniques developed in [Abadi Cardelli Pierce Plotkin 89] for dynamic types. See [Harper Milner Tofte 88] for a full formal language definition in this style.

This section contains the type rules for *Miniature Quest*, which is a language with scaled down notions of values, types, kinds, bindings, and signatures, but which presents the essential concepts.

Syntax

Signatures (S):
- ∅
- S, X::K
- S, x:A

Bindings (D):
- ∅
- D, X::K=A
- D, x:A=a

Kinds (K, L, M):
- TYPE
- ALL(X::K)L
- POWER(A)

Types and operators (A,B,C; type identifiers X,Y,Z):
- X
- All(S)A
- Tuple(S)
- Fun(X::K)A   A(B)
- Rec(X::TYPE)A

Values (a,b,c; value identifiers x,y,z):
- x
- fun(S)a   a(D)
- tuple(D)   bind S = a in b
- rec(x:A)a

Here we use the construct "bind S = a in b" which binds the components of the tuple "a" to the identifiers in "S" in the scope "b". In full Quest we use instead the notation "x.Y" and "x.y" to extract types and values out of a tuple denoted by an identifier "x". Then a program fragment "let x = a .. x.Y .. x.y..." corresponds to "bind ..Y::K..y:A.. = a in .. y .. Y...". The type rule for "bind" prevents the type identifiers in "S" from occurring in the type of the result. Similarly, in full Quest types like "x.Y" are not allowed to escape the scope of "x". The formal relation between "bind" and "x.Y"-"x.y" is studied in [Cardelli Leroy 90].

Judgments

$\vdash S$ sig	S is a signature	$S \vdash D \therefore S'$	D has signature S'
$S \vdash K$ kind	K is a kind	$S \vdash A::K$	A has kind K
$S \vdash A$ type	A is a type (same as $S \vdash A::TYPE$)	$S \vdash a:A$	a has type A
$S \vdash S' <\therefore S''$	S' is a subsignature of S"	$S \vdash S' <\therefore> S''$	equivalent signatures
$S \vdash K <:: L$	K is a subkind of L	$S \vdash K <::> L$	equivalent kinds
$S \vdash A <: B$	A is a subtype of B (same as $S \vdash A::POWER(B)$)	$S \vdash A <:> B$	equivalent types

Notation

S S' is the concatenation (iterated extension) of S with S'. Signatures and bindings are ordered sequences; however we freely use the notation $X \in dom(S)$ (type X is defined in S), $x \in dom(S)$ (value x is defined in S). Similarly for bindings.

E{X←A} and E{x←a} denote the substitution of the type variable X by the type A, or of the variable x by the value a, within an expression E of any sort. For a binding D, E{D} is defined as follows: E{∅} = E; E{D', X::K=A} = E{X←A}{D'}; E{D', x:A=a} = E{x←a}{D'}.

E[E'] indicates that E' is a given subexpression of expression E. Then E[E"] denotes the substitution of (a particular occurrence of) E' by E" in E. Here E, E' and E" are expressions of any sort.

A type C is *contractive* in a (free) type variable X [MacQueen Plotkin Sethi 86], written C↓X , if and only if C is either: a type variable different from X, a function or tuple type, an operator application whose reduced form is contractive in X, or a recursive type whose body is contractive in X. The body of a legal recursive type must also be contractive in the recursion variable. From a type whose recursion bodies are all contractive in their recursion variables, we can construct a well-formed regular (infinite) tree.

Equivalence

$$\frac{S \vdash A :: K}{S \vdash A <:> A} \qquad \frac{S \vdash K \text{ kind}}{S \vdash K <::> K} \qquad \frac{\vdash S\ S' \ \text{sig}}{S \vdash S' <\therefore> S'}$$

$$\frac{S \vdash A <:> A'}{S \vdash A' <:> A} \qquad \frac{S \vdash K <::> K'}{S \vdash K' <::> K} \qquad \frac{S \vdash S' <\therefore> S''}{S \vdash S'' <\therefore> S'}$$

$$\frac{S \vdash A <:> A' \quad S \vdash A' <:> A''}{S \vdash A <:> A''} \qquad \frac{S \vdash K <::> K' \quad S \vdash K' <::> K''}{S \vdash K <::> K''} \qquad \frac{S \vdash S' <\therefore> S'' \quad S \vdash S'' <\therefore> S'''}{S \vdash S' <\therefore> S'''}$$

Congruence

$$\frac{S \vdash A[B] :: K \quad S \vdash B <:> B'}{S \vdash A[B] <:> A[B']} \qquad \frac{S \vdash A[K] :: L \quad S \vdash K <::> K'}{S \vdash A[K] <:> A[K']} \qquad \frac{S \vdash A[S'] :: L \quad S \vdash S' <\therefore> S''}{S \vdash A[S'] <:> A[S'']}$$

$$\frac{S \vdash K[A] \text{ kind} \quad S \vdash A <:> A'}{S \vdash K[A] <::> K[A']} \qquad \frac{S \vdash K[L] \text{ kind} \quad S \vdash L <::> L'}{S \vdash K[L] <::> K[L']} \qquad \frac{S \vdash K[S'] \text{ kind} \quad S \vdash S' <\therefore> S''}{S \vdash K[S'] <::> K[S'']}$$

$$\frac{\vdash S\ S' \ \text{sig} \quad S \vdash A <:> A'}{S \vdash S'[A] <\therefore> S'[A']} \qquad \frac{\vdash S\ S'[K] \text{ sig} \quad S \vdash K <::> K'}{S \vdash S'[K] <\therefore> S'[K']} \qquad \frac{\vdash S\ S'[S''] \text{ sig} \quad S \vdash S'' <\therefore> S'''}{S \vdash S'[S''] <\therefore> S'[S''']}$$

Inclusion

$$\frac{S \vdash A <:> B}{S \vdash A <: B} \qquad \frac{S \vdash K <::> L}{S \vdash K <:: L} \qquad \frac{S \vdash S' <\therefore> S''}{S \vdash S' <\therefore S''}$$

Subsumption

$$\frac{S \vdash a : A \quad S \vdash A <: B}{S \vdash a : B} \qquad \frac{S \vdash A :: K \quad S \vdash K <:: L}{S \vdash A :: L} \qquad \frac{S \vdash D \therefore S' \quad S \vdash S'' <\therefore S''}{S \vdash D \therefore S''}$$

Conversion

$$\frac{S, X::K \vdash B :: L \quad S \vdash A :: K}{S \vdash (\text{Fun}(X::K)B)(A) <:> B\{X \leftarrow A\}} \qquad \frac{S \vdash \text{Rec}(X::\text{TYPE})A \text{ type}}{S \vdash \text{Rec}(X::\text{TYPE})A <:> A\{X \leftarrow \text{Rec}(X::\text{TYPE})A\}}$$

$$\frac{S \vdash A <:> C\{X \leftarrow A\} \quad S \vdash B <:> C\{X \leftarrow B\} \quad C \downarrow X}{S \vdash A <:> B}$$

Signatures

$$\vdash \emptyset\ \mathrm{sig} \qquad \frac{S \vdash K\ \mathrm{kind} \quad X \notin \mathrm{dom}(S)}{\vdash S, X{::}K\ \mathrm{sig}} \qquad \frac{S \vdash A\ \mathrm{type} \quad x \notin \mathrm{dom}(S)}{\vdash S, x{:}A\ \mathrm{sig}}$$

Bindings

$$\frac{\vdash S\ \mathrm{sig}}{S \vdash \emptyset \therefore \emptyset} \qquad \frac{S \vdash D \therefore S' \quad S \vdash A\{D\} :: K\{D\}}{S \vdash D, X{::}K = A \ \therefore\ S', X{::}K} \qquad \frac{S \vdash D \therefore S' \quad S \vdash a\{D\} : A\{D\}}{S \vdash D, x{:}A = a \ \therefore\ S', x{:}A}$$

Kinds

$$\frac{\vdash S\ \mathrm{sig}}{S \vdash \mathrm{TYPE}\ \mathrm{kind}} \qquad \frac{S \vdash K\ \mathrm{kind} \quad S, X{::}K \vdash L\ \mathrm{kind}}{S \vdash \mathrm{ALL}(X{::}K)L\ \mathrm{kind}} \qquad \frac{S \vdash A\ \mathrm{type}}{S \vdash \mathrm{POWER}(A)\ \mathrm{kind}}$$

Types and Operators

$$\frac{\vdash S, X{::}K, S'\ \mathrm{sig}}{S, X{::}K, S' \vdash X :: K} \qquad \frac{S\ S' \vdash A\ \mathrm{type}}{S \vdash \mathrm{All}(S')A\ \mathrm{type}} \qquad \frac{\vdash S\ S'\ \mathrm{sig}}{S \vdash \mathrm{Tuple}(S')\ \mathrm{type}}$$

$$\frac{S, X{::}K \vdash B :: L}{S \vdash \mathrm{Fun}(X{::}K)B :: \mathrm{ALL}(X{::}K)L} \qquad \frac{S \vdash B :: \mathrm{ALL}(X{::}K)L \quad S \vdash A :: K}{S \vdash B(A) :: L\{X \leftarrow A\}} \qquad \frac{S, X{::}\mathrm{TYPE} \vdash A\ \mathrm{type} \quad A \downarrow X}{S \vdash \mathrm{Rec}(X{::}\mathrm{TYPE})A\ \mathrm{type}}$$

Values

$$\frac{\vdash S, x{:}A, S'\ \mathrm{sig}}{S, x{:}A, S' \vdash x : A} \qquad \frac{S\ S' \vdash a : A}{S \vdash \mathrm{fun}(S')a : \mathrm{All}(S')A} \qquad \frac{S \vdash a : \mathrm{All}(X')A \quad S \vdash D \therefore S'}{S \vdash a(D) : A\{D\}}$$

$$\frac{S \vdash D \therefore S'}{S \vdash \mathrm{tuple}(D) : \mathrm{Tuple}(S')} \qquad \frac{S \vdash a : \mathrm{Tuple}(S') \quad S \vdash B\ \mathrm{type} \quad S\ S' \vdash b : B}{S \vdash \mathrm{bind}\ S' = a\ \mathrm{in}\ b \ :\ B} \qquad \frac{S, x{:}A \vdash a : A}{S \vdash \mathrm{rec}(x{:}A)a : A}$$

SubSignatures

$$\frac{S \vdash S' <\therefore S'' \quad S\ S' \vdash K <:: L}{S \vdash S', X{::}K \ <\therefore\ S'', X{::}L} \qquad \frac{S \vdash S' <\therefore S'' \quad S\ S' \vdash A <: B}{S \vdash S', x{:}A \ <\therefore\ S'', x{:}B}$$

SubKinds

$$\frac{S \vdash K' <:: K \quad S, X{::}K' \vdash L <:: L'}{S \vdash \mathrm{ALL}(X{::}K)L \ <::\ \mathrm{ALL}(X{::}K')L'} \qquad \frac{S \vdash A\ \mathrm{type}}{S \vdash \mathrm{POWER}(A) \ <::\ \mathrm{TYPE}} \qquad \frac{S \vdash A \ <:\ B}{S \vdash \mathrm{POWER}(A) \ <::\ \mathrm{POWER}(B)}$$

SubTypes

$$\frac{S \vdash S'' <\therefore S' \quad S\ S'' \vdash A' <: A''}{S \vdash \mathrm{All}(S')A' \ <:\ \mathrm{All}(X'')A''} \qquad \frac{\vdash S\ S'\ S''\ \mathrm{sig} \quad S \vdash S' <\therefore S'''}{S \vdash \mathrm{Tuple}(S'\ S'') <: \mathrm{Tuple}(S''')}$$

$$\frac{S \vdash \mathrm{Rec}(X{::}\mathrm{TYPE})A\ \mathrm{type} \quad S \vdash \mathrm{Rec}(Y{::}\mathrm{TYPE})B\ \mathrm{type} \quad S, Y{::}\mathrm{TYPE}, X <: Y \vdash A <: B}{S \vdash \mathrm{Rec}(X{::}\mathrm{TYPE})A <: \mathrm{Rec}(Y{::}\mathrm{TYPE})B}$$

11.3. Library interfaces

Here is a list of supplied modules; their interface specification follows. These modules are pre-linked at the top level, but must be explicitly imported by any module which uses them.

```
arrayOp: ArrayOp          (* Array operations *)
ascii: Ascii              (* Ascii conversions *)
conv: Conv                (* String conversions *)
dynamic: Dynamic          (* Dynamically typed values *)
int: IntOp                (* Integer number operations *)
list: List                (* Polymorphic lists *)
reader: Reader            (* Input operations *)
real: RealOp              (* Real number operations *)
trig: Trig                (* Trigonometry, to be defined *)
string: StringOp          (* String operations *)
value: Value              (* Arbitrary value operations. Unsound! *)
writer: Writer            (* Output operations *)
```

```
interface ArrayOp
export
  error:Exception(Ok)
     (* Raised when an operation cannot be carried out. *)
  new(A::TYPE size:Int init:A):Array(A)
     (* Create a new array of given size, all initialized to init. *)
  size(A::TYPE array:Array(A)):Int
     (* Return the size of an array, same as "size(array)" (predefined). *)
  get(A::TYPE array:Array(A) index:Int):A
     (* Extract an array element, same as "array[index]". *)
  set(A::TYPE array:Array(A) index:Int item:A):Ok
     (* Update an array element, same as "array[index] := item". *)
end;

interface Ascii
export
  error:Exception(Ok)
     (* Raised when an operation cannot be carried out. *)
  char(n:Int):Char
     (* Return the character of ascii encoding n.
        Raise error if n<0 or n>255. *)
  val(c:Char):Int
     (* Return the ascii encoding of character c. *)
end;

interface Conv
export
  okay():String
     (* Return the string "ok". *)
  bool(b:Bool):String
     (* Return "true" if b is true, "false" otherwise. *)
  int(n:Int):String
     (* Return a string representation of the integer n
        (preceded by '~' if negative). *)
  real(r:Real):String
     (* Return a string representation of the real r
        (preceded by '~' if negative). *)
```

```
    char(c:Char):String
        (* Return a string containing the character c in single quotes,
           with backslash encoding if necessary. *)
    string(s:String):String
        (* Return a string containing the string s in double quotes,
           with backslash encoding wherever necessary. *)
end;

interface Dynamic
import reader:Reader writer:Writer
export
    Def T = Auto A::TYPE with :A end
        (* A pair of an arbitrary object and its type. *)
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    new(A::TYPE a:A):T
        (* Package an object of any type into an object of type T. *)
    be(A::TYPE d:T):A
        (* If the object d was generated from and object a of type A,
           then return a, otherwise raise error. *)
    copy(d:T):T
        (* Make a complete copy of a dynamic object. *)
    extern(wr:writer.T d:T):Ok
        (* Write a representation of a dynamic object to a writer. *)
    intern(rd:reader.T):T
        (* Read a dynamic object from a reader. *)
end;

interface IntOp
export
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    minInt,maxInt:Int
        (* The most negative and most positive representable integers. *)
    abs(n:Int):Int
        (* Absolute value. *)
    min,max(n,m:Int):Int
        (* Min and max. *)
end;

interface List
export
    T::ALL(A::TYPE)::TYPE
        (* If list:List is an implementation of this interface,
           then l:list.T(A) is a list of items of type A. *)
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    nil(A::TYPE):T(A)
    cons(A::TYPE :A :T(A)):T(A)
    null(A::TYPE :T(A)):Bool
    head(A::TYPE :T(A)):A
    tail(A::TYPE :T(A)):T(A)
    length(A::TYPE :T(A)):Int
    enum(A::TYPE :Array(A)):T(A)
        (* "list.enum of ... end" returns the list of elements "...". *)
end;
```

```
interface Reader
export
   T::TYPE
      (* A reader is a source of characters. *)
   error:Exception(Ok)
      (* Raised when an operation cannot be carried out. *)
   input:T
      (* The standard input reader. *)
   file(name:String):T
      (* Make a reader out of a file, given its file name. *)
   more(reader:T):Bool
      (* Test whether there are more characters to be read. *)
   ready(reader:T):Int
      (* Counts the number of characters that can be read without blocking;
         the end-of-stream marker counts as 1. Never blocks. *)
   getChar(reader:T):Char
      (* Read a character from a reader. *)
   getString(reader:T size:Int):String
      (* Read a string of given size from a reader. *)
   getSubString(reader:T string:String start,size:Int):Ok
      (* Read a string of given size from a reader and store it in
         another string at a given position. *)
   close(reader:T):Ok
      (* Close a reader; operations on it will now fail. *)
end;

interface RealOp
export
   error:Exception(Ok)
      (* Raised when an operation cannot be carried out. *)
   minReal,maxReal:Real
      (* The most negative and most positive representable reals. *)
   negEpsilon,posEpsilon:Real
      (* The most positive representable negative real, and
         the most negative representable positive real. *)
   e:Real
   int(n:Int):Real
   floor,round(r:Real):Int
   abs,log(r:Real):Real
   min,max(r,s:Real):Real
end;

interface StringOp
export
   error:Exception(Ok)
      (* Raised when an operation cannot be carried out. *)
   new(size:Int init:Char):String
      (* Create a new string of give size, all initialized to init. *)
   isEmpty(string:String):Bool
      (* Test whether a string is empty. *)
   length(string:String):Int
      (* Return the length of a string. *)
   getChar(string:String index:Int):Char
      (* Extract a character from a string. *)
   setChar(string:String index:Int char:Char):Ok
      (* Replace a character of a string. *)
   getSub(source:String start,size:Int):String
      (* Extract a substring from a string. *)
```

```
   setSub(dest:String destStart:Int
      source:String sourceStart,sourceSize:Int):Ok
      (* Replace a substring of a string. *)
   cat(string1,string2:String):String
      (* Concatenate two strings (same as "<>"). *)
   catSub(string1:String start1,size1:Int
      string2:String start2,size2:Int):String
      (* Concatenate two substrings. *)
   equal(string1,string2:String):Bool
      (* True if two strings have the same size and contents. *)
   equalSub(string1:String start1,size1:Int
      string2:String start2,size2:Int):Bool
      (* True if two substrings have the same size and contents. *)
   precedes(string1,string2:String):Bool
      (* True if two strings are equal or in lexicographic order. *)
   precedesSub(string1:String start1,size1:Int
      string2:String start2,size2:Int):Bool
      (* True if two substrings are equal or in lexicographic order. *)
end;

unsound interface Value
export
   T::TYPE
      (* The type of an arbitrary value. *)
   error:Exception(Ok)
      (* Raised when an operation cannot be carried out. *)
   length:Int
      (* Implementation-dependent size of a value in implementation-dependent
         units. *)
   new(A::TYPE a:A):T
      (* Convert anything to a value. *)
   be(A::TYPE v:T):A
      (* Convert a value to anything. Unsound! *)
   fetch(addr:T displ:Int):T
      (* Fetch the value at location addr+displ in memory. *)
   store(addr:T displ:Int w:T):Ok
      (* Store a value at location addr+displ in memory. Unsound! *)
end;

interface Writer
export
   T::TYPE
      (* A writer is a sink of characters. *)
   error:Exception(Ok)
      (* Raised when an operation cannot be carried out. *)
   output:T
      (* The standard output writer. *)
   file(name:String):T
      (* Make a writer out of a file, given its file name. *)
   flush(writer:T):Ok
      (* Flush any buffered characters to their final destination. *)
   putChar(writer:T char:Char):Ok
      (* Write a character to a writer. *)
   putString(writer:T string:String):Ok
      (* Write a string to a writer. *)
   putSubString(writer:T string:String start,size:Int):Ok
      (* Write a substring of a given string to a writer. *)
```

```
    close(writer:T):Ok
        (* Close a writer; operations on it will now fail. *)
end;
```

References

[Abadi Cardelli Pierce Plotkin 89] M.Abadi, L.Cardelli, B. Pierce, G.D.Plotkin: **Dynamic Typing in a Statically Typed Language**, Proc. POPL 1989.

[Amadio Cardelli 90] R.M.Amadio, L.Cardelli: **Subtyping recursive types**, DEC SRC Report, to appear.

[Andrews Schneider 88] G.R.Andrews, F.B.Schneider: **Concepts and notations for concurrent programming**, Computer Surveys, Vol. 15, No. 1, March 1983.

[Atkinson Bailey Chisholm Cockshott Morrison 83] M.P.Atkinson, P.J.Bailey, K.J.Chisholm, W.P.Cockshott, R.Morrison: **An approach to persistent programming**, Computer Journal 26(4), November 1983.

[Barendregt 85] H.P.Barendregt: **The lambda-calculus, its syntax and semantics**, North-Holland 1985.

[Böhm Berarducci 85] C.Böhm, A.Berarducci: **Automatic synthesis of typed λ-programs on term algebras**, Theoretical Computer Science, 39, pp. 135-154, 1985.

[Buneman Ohori 87] P.Buneman, A.Ohori: **Using powerdomains to generalize relational databases**, submitted for publication.

[Burstall Lampson 84] R.M.Burstall, B.Lampson: **A kernel language for abstract data types and modules**, in Semantics of Data Types, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 86] 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. Lecture Notes in Computer Science n. 242, Springer-Verlag, 1986.

[Cardelli 88] L.Cardelli: **Types for Data-Oriented Languages**, Proceedings of the First Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988.

[Cardelli Donahue Glassman Jordan Kalsow Nelson 89] L.Cardelli, J.Donahue, L.Glassman, M.Jordan, B.Kalsow, G.Nelson: **Modula-3 report (revised)**, Report #52, DEC Systems Research Center, November 1989.

[Cardelli Leroy 90] L.Cardelli, X.Leroy: **Abstract types and the dot notation**, Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods, Israle, April 90.

[Cardelli Longo 90] L.Cardelli, G.Longo: **A semantic basis for Quest**, Proceedings of the 6th ACM LISP and Functional Programming Conference, Nice, France, June 1990.

[Cardelli Wegner 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

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

[Cook Hill Canning 90] W.Cook, W.Hill, P.Canning: **Inheritance is not subtyping**, Proc. POPL'90.

[Courcelle 83] B.Courcelle: **Fundamental properties of infinite trees**, Theoretical Computer Science, 25, pp. 95-169, 1983.

[Dahl Nygaard 66] O.Dahl, K.Nygaard: **Simula, an Algol-based simulation language**, Comm. ACM, Vol 9, pp. 671-678, 1966.

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

[Futatsugi Goguen Jouannaud Meseguer 85] K.Futatsugi, J.A.Goguen, J.P.Jouannaud, J.Meseguer: **Principles of OBJ2**, Proc. POPL 1985.

[Girard 71] J-Y.Girard: **Une extension de l'interprétation de Gödel à 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.

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

[Harbison Steele 84] S.P.Harbison, G.L.Steele Jr.: **C, a reference manual**, Prentice Hall 1984.

[Harper Milner Tofte 88] R.Harper, R.Milner, M.Tofte: **The definition of Standard ML - Version 2**, Report LFCS-88-62, Dept. of Computer Science, University of Edinburgh, 1988.

[Hyland Pitts 87] J.M.E.Hyland, A.M.Pitts: **The theory of constructions: categorical semantics and topos-theoretic models**, in Categories in Computer Science and Logic (Proc. Boulder '87), Contemporary Math., Amer. Math. Soc., Providence RI.

[Kemeny Kurtz 71] J.G.Kemeny, T.E.Kurtz: **Basic Programming**, John Wiley & Sons, 1971.

[Kowalski 79] R.Kowalski: **Logic for problem solving**, North-Holland 1979.

[Krasner 83] G.Krasner(Ed.): **Smalltalk-80. Bits of history, words of advice**, Addison-Wesley, 1983.

[Landin 66] P.J.Landin: **The next 700 programming languages**, Comm ACM, Vol. 9, No. 3, 1966, pp. 157-166.

[Liskov et al. 77] B.H.Liskov et al.: **Abstraction Mechanisms in CLU**, Comm ACM 20,8, 1977.

[Liskov Guttag 86] B.H.Liskov, J.Guttag: **Abstraction and specification in program development**, MIT Press, Cambridge, MA, 1986.

[Lucassen Gifford 88] J.M.Lucassen, D.K.Gifford: **Polymorphic Effect Systems**, Proc. POPL '88.

[MacQueen 84] 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 Plotkin Sethi 86] D.B.MacQueen, G.D.Plotkin, R.Sethi: **An ideal model for recursive polymorphic types**, Information and Control 71, pp. 95-130, 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.

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

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

[Mycroft O'Keefe 84] A.Mycroft, R.A.O'Keefe: **A polymorphic type system for Prolog**, Artificial Intelligence 23, pp. 295-307, North Holland, 1984.

[Parnas 72] D.L.Parnas: **On the criteria to be used in decomposing systems into modules**, Communications of the ACM, Vol. 15, no. 12, pp. 1053-1058, December 1972.

[Plotkin 81] G.D.Plotkin: **A structural approach to operational semantics**, Report DAIMI FN 19, Computer Science Department, Aarhus University, 1981.

[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 88] J.C.Reynolds: **Preliminary design of the programming language Forsythe**, Report CMU-CS-88-159, Carnegie Mellon University, 1988.

[Schaffert Cooper Bullis Kilian Wilpolt 86] C.Schaffert, T.Cooper, B.Bullis, M.Kilian, C.Wilpolt: **An introduction to Trellis/Owl**, Proc. OOPSLA'86.

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

[Strom Yemini 83] R.Strom, S.Yemini: **NIL: an integrated language and system for distributed programming**, Proc. SIGPLAN'83 Symposium on Programming Language Issues in Software Systems, 1983.

[Stroustrup 86] B.Stroustrup: **The C++ programming language**, Addison-Wesley 1986.

[Turner 85] D.A.Turner: **Miranda: a non-strict functional language with polymorphic types**, in Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science No. 201, Springer-Verlag , 1985.

[Weinreb Moon 81] D.Weinreb, D.Moon: **Lisp machine manual**, Symbolics Inc., 1981.

[Wirth 83] N.Wirth: **Programming in Modula-2**, Texts and Monographs in Computer Science, Springer-Verlag 1983.

[Wirth 87] N.Wirth: **From Modula to Oberon, and the programming language Oberon**, Report 82, Institut für Informatik, ETH Zürich, 1987.