A VIEW OF TYPES AND PARAMETERIZATION IN PROGRAMMING LANGUAGES

by

Mark Judd

School of Computer Science
McGill University
Montréal, Québec

August, 1985

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Science

in,

Computer Science

A View of Types and Parameterization in Programming Languages

Abstract

A view of *type* in programming languages is proposed A language, M, is designed primarily as a means to illustrate this view

M's type system uses simple type fundamentals and a few, orthogonally applied combining forms to provide extensibility Following the principle of *type-completeness* ([Demers & Donahue 80a]), all identifiers have a type (including type identifiers) and all values are *first class citizens* of the language Both parameterization of types and parameters of any type (including those of type TYPE) are permitted

An *adjective* syntax is introduced to provide a *type abstraction* facility in which a type is clearly viewed as an *algebra* (i.e , as a set of operations) Type compatibility is defined in terms of algebraic compatibility. A generalized type hierarchy becomes possible in which a single entity, though belonging to one specific type, may also belong to many abstract types. *Polymorphism* is provided by allowing routines to be parameterized by such abstract types

The proposed type system permits type flexibility while maintaining strict type security

Résumé

Une vue du *type* dans les langages de programmation est proposée. Un langage, M, est conçu afin d'illustrer cette vue.

Le système de type de M utilise des règles de base et quelques formes de combinaison qui peuvent être appliquées orthogonalement pour rendre le langage extensible. Selon le principe de *type-completeness* ([Demers & Donahue 80a]), tous les identificateurs ont un type (même les identificateurs de type) et toutes les valeurs sont des *citoyens de première classe* du langage. La paramétrisation de type et des paramètres de tous types (incluant les paramètres de type TYPE) sont permis.

Une *syntaxe pour les adjectifs* est introduite pour permettre l'abstraction de type, un type est considéré comme un algèbre (comme un ensemble d'opérations). Deux types sont compatibles si leurs algèbres sont compatibles. Une hiérarchie généralisée pour les types devient possible, où une entité appartient à un seul type, mais peut aussi appartenir à plusieurs types abstraits. Le langage permet le *polymorphisme* en permettant d'utiliser les types abstraits pour paramétriser des expressions.

Le système de type proposé est sécuritaire mais flexible.

Acknowledgements

I would like to thank my thesis advisor, Prof. G V Cormack, for his helpful comments on drafts of this thesis and for his encouragement during this work Thanks also are extended to Denis Leclerc through whom most of these thoughts were distilled I am also indebted for the financial aid received through a Natural Sciences and Engineering Research Council of Canada Scholarship and through an M H Beattie Bursary

Table of Contents

CHAPTER 1

Introduction

1.1 Problem, Goal, and Focus

Computers are a revolutionizing force. They perform not only a wide variety of tasks that we do not want to do ourselves, but also those that we cannot do ourselves. They perform repetitive and mundane tasks as well as tasks that can be described as *intelligent* in some sense: pattern matching, theorem proving, inferring

What makes these machines so flexible is their ability to be programmed by *software*: an algorithm is designed to model a given situation or to solve a certain problem, this algorithm is expressed in a programming language to create a program, the program may be acted upon by a machine to produce the appropriate results. General purpose programming languages can express a large number of such algorithms.

The development of this software, however, is proving to be a *bottleneck*. Rather than being a labour-saving activity, it is labour intensive. While hardware prices are decreasing, software is proving to be more and more costly. Releasing this bottleneck has been the object of current research.

If we are to exploit the advantages of computer technology through software, we must build and manage software systems effectively. In addressing the problem, the United States Department of Defence notes ([STARS 83], p. 60)

> "The *goal* is to improve software productivity, while achieving greater system reliability and adaptability."

Productivity is an obvious goal. Reliability improves the correctness and robustness of the product. Adaptability ensures both that the software is reusable, thereby avoiding unnecessary duplication of effort, and that a particular system is maintainable or modifiable throughout its evolution.

To satisfy this goal, research has focused on providing effective *programming environments* conducive to the development and maintenance of software systems. At the heart of the environment is the programming language; it is the primary tool used for describing our models, algorithms, and solutions.

Linguists believe that the structure of language defines the boundaries of thought. The use of a particular programming language, then, though it might not prevent one from thinking certain thoughts, may facilitate or impede certain modes of thought; it may influence the class of solutions one is likely to see. The language should aid us in solving problems and in extending the class of problems that we can solve. Many researchers, therefore, have focused on the programming language as a means out of the *tar-pit* ([Brooks 75]). Any step forward in the programming language arena is a step closer to releasing the software bottleneck.

1.2 Thesis Motivation

This thesis was motivated by the perception that current type systems were both inflexible and composed of an unorthogonal application of concepts. An attempt is made to isolate the constructs inherent in other type systems and, by re-combining them orthogonally, to derive a smaller set of required primitives. Some of the inflexible qualities of existing languages can be attributed to a limiting concept of *type* and we can enhance the expressiveness of programming languages by viewing types more abstractly. Such an enhancement can be achieved without sacrificing the semantic security afforded by the type system.

1.3 Thesis Outline

In Chapter 2, the concepts inherent in current type systems are investigated. Specially noted are the abstractive facilities provided. The uses and demands programmers make of such systems are examined.

Chapter 3 defines the type fundamentals of M. The principle of type-completeness is discussed. The use of this principle ensures that all identifiers in M have a type (including type identifiers), that all values are *first class citizens* of the language, and that parameterization is universally applicable. Parameterization of types and parameters of any type (including those of type TYPE) are permitted.

Chapter 4 discusses the concept of type abstraction in M. A *type* is viewed as an algebra (i.e., as a set of operations). An abstract type encompasses all specific types which share a common (sub)algebra. This view of type abstraction supports a generalized type hierarchy in which a single entity, though belonging to one specific type, may also belong to many abstract types. A polymorphic routine in M, then, is defined as a routine that is parameterized by such an abstract type

Chapter 5 reviews the entire thesis and suggests directions for future research.

1.4 Language Sources

The languages referred to throughout this thesis are taken to be defined by the following documents: Ada†, [Ada 83]; Algol 68, [Algol68 76]; Alphard, [Alphard 81]; CLU, [Liskov et al. 79]; Euclid, [Lampson et al. 77]; FORTRAN, [FORTRAN 66]; L, [Cormack 81a], LISP, [LISP 62]; Pascal, [Jensen & Wirth 78]; Red, [Red 77]; Russell, [Demers & Donahue 79]; SIMULA, [Birtwistle et al 74]; Smalltalk, [Goldberg & Robson 83]; Tartan, [Shaw et al. 78]. In the absence of an explicit reference, these sources should be assumed. Other documents are referenced explicitly.

† Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

CHAPTER 2

Overview of Type Systems

2.1 Introduction

A type system allows programmers to define the types of entities in their world, to define an algebra on these types, and to enforce the algebra strictly The next sections develop these concepts more fully by discussing the state of type systems as embodied in current languages and by pointing out the evolution of such systems over time. Type information is shown to serve the dual purposes of abstraction and reliability both of which impact the program development process

2.2 Classification

By nature, humans are tool makers and tool users. One of their greatest tools is the ability to *abstract* *Abstraction*–"the process of separating qualities or attributes from the individual objects to which they belong"*–takes many forms. One, termed *classification*, is particularly relevant to the concept of type.

A *class* is "a number of objects, facts, or events grouped together as having common properties; a set; category; kind"*; a *type* is "a class, kind, or group sharing one or more characteristics; category"*.

To *classify*–"to arrange or put in a class or classes on the basis of resemblances or differences"*–one has to name the things in common. Simply collecting things in a set is fairly unimportant since it is the properties defined on the set that give the elements some *relation*; only in an extremely abstract mathematical sense are they related at all (i.e., being members of the same set). For instance, one has very little feel for the relation among elements of the set

$$X = \{ \text{clothes iron, Sam, encyclopedia, wedge} \}$$

* Funk & Wagnalls Standard College Dictionary.

until a property such as

$$\forall x \in X,\ x \text{ can prop open a door}$$

is given.

In programming language parlance, the values of a type are isomorphic to the elements of a set. The fact that an element t belongs to a set T implies that "t is of type T". It is membership within a certain set that gives any value its type.

Initially, types were thought of simply as sets of values. [Morris 73b] pointed out that a type was better thought of as this set of values plus the set of operations allowed on the set It is the set of operations, in fact, that provides a real interpretation of the elements. Any operation (e.g., *procedure* or *function*) defined having a parameter or result of type T contributes to the properties of the type T; it extends the concept of what it means to belong to the type T. The set of properties defined on a type is, by definition, the *algebra* for that type.

Early languages defined types that, in most cases, reflected the behaviour of the underlying machine. The notion of type in these languages was not extensible. All programs had to be written using these types whether the types were appropriate to the problem at hand or not. Modern languages include a capability to extend types by allowing simple programmer-defined types and by supplying combining forms to generate compound types. Typical examples of these combining forms are the constructors *array*, *record*, and *pointer* which can be used to construct both homogeneous and heterogeneous compound types

Reiterating, types are useful precisely because of their abstractive ability During program development, objects with distinct properties can be clearly distinguished. Knowledge about common properties can be collected in one place and named, the type name then refers to these properties Such factorization aids maintainability and readability Enforcing the distinction between types improves reliability

2.3 Type Enforcement

Most assembly languages provide an operator "+" for fixed-point addition, but do not require that its operands be, in fact, fixed-point values. Any argument supplied to this operation is simply *blessed* as being a fixed-point operand. Wild errors can easily appear within a program. Worse yet, subtle errors can appear This *blessedness* is due to the fact that most values share the same representation; the type of a particular value is

indistinguishable at run-time.

Ensuring that the type of an argument passed to a routine is compatible with the type expected is called *type-checking*. The degree of type checking inherent in a type system ranges from weak (little or no checking) to strong (total assurance of type compatibility). The point in time that this type-checking takes place can be either static (checked prior to execution) or dynamic (checked at run-time)

Early type systems provided type-checking for built-in operations only, though user-defined subroutines extended the functional capability of the language, no checking was done across these boundaries With the growing recognition that *reliability* was at least as important as the more traditional goals of efficiency and writability came a desire for stronger type-checking. The type system is one of the first lines of attack against unreliable programs in that it strengthens the semantic checking The growing breadth of type systems has occurred in an effort to enlarge the class of errors that can be detected by the compiler.

In principle, a mathematical function may be applied only to values that are in its domain of arguments. [Tennent 81] terms the application of an operation to a value that is not in its domain of arguments a *domain incompatibility* He cites the examples of dividing by zero, adding truth values, negating a character, and reading from an empty file as being typical of domain incompatibilities The main objective of type checking is to determine whether a domain incompatibility can occur.

Tennent's view is derived from the fact that ultimately the program must be run on a machine that knows nothing about types The expression $f(x)$ entails checking that the type of x is compatible with the type expected by f If the only definition of f were one that required an integer parameter, then $f(3.5)$ would be illegal since a real number is not allowed as a parameter to f. Alternatively, one may view type-checking as a check for the existence of a certain algebra. Under this view, the expression $f(x)$ entails checking that a function f exists for the type of x. Here, $f(3.5)$ is illegal since no function f exists that accepts a real parameter. The focus changes somewhat Though the two approaches seem equivalent, only the second view suffices in the presence of overloaded function names

Static type checking is often preferred to dynamic type checking First, type checks *before* execution are generally more efficient than type checks *during* execution The application of a function to a particular argument may occur several times during execution yet need only be type-checked once Second, being able to catch minor programming errors before execution simplifies program debugging and testing Detecting them at this time allows one to deal with the error when something can be done about it Third, dynamic checks require that the type of a value be inherent in the representation of a value Static

checking does not demand that one carry around this information during execution.

Dynamic checking, however, allows the system to be extremely flexible. The type of a given lexical entity is a run-time property and can differ across different executions. The validity of applying a certain operation to this entity, and indeed the meaning of the operation itself, is then dependent upon the type of the entity at that point in time. The compiler does not bind the type information before such information is available.

Even in statically type-checked systems, though, checks during execution are sometimes required. [Buckle 77] notes that when checking a program for correctness, it is often necessary to associate invariant properties with specific variables and to prove that the behaviour of these variables is compatible with their respective properties. When such a proof happens to be difficult or impossible to establish, tests are sometimes included in the program to check at run-time that the properties are indeed verified.

A *subtype* is a property of an instance of a type and serves to restrict this instance to a subset of the total values allowed for the given type. The Ada declaration

```
x: INTEGER range 1..10;
```

restricts the variable *x* to a certain subrange of the integers. Buckle's concept of a *restricted data type* allows more complex constraints to be defined, the restriction is to a more general concept of subset, not simply a subrange.

In most languages, since static determination of whether an instance is properly constrained at all times *is* difficult, subtype checking is deferred until run-time. In some instances the constraints can be guaranteed at compile-time.

Type enforcement is not always complete. Languages often provide loopholes whether inadvertently or not. FORTRAN's EQUIVALENCE, for example, and Pascal's variant records both permit circumvention of strict type security. Some programmers perceive a need for loopholes. This need can be attributed to the existence of a type system that is *too* restrictive, often due to a poor concept of type compatibility. Pascal's concept of equivalence between array types, for instance, falls into this category.

The reliability and safety provided by error detection through mechanization of type checking is an important aspect of a programmed system. More and more, programmers feel the need for strong type checking but without a corresponding loss in abstractive ability.

2.4 Type Compatibility

If type checking is to be done, one must have a concept of *type equivalence* or *type compatibility* Two common approaches are termed *name equivalence* and *structural equivalence*

Structural equivalence generally assumes that two types are equivalent if they are the same primitive type or if they both arise from the same type generator (i e , both are records or both are arrays) and the types of their components are equivalent Algol 68 additionally specifies that the names of the *selectors* of two structured modes would also have to be equivalent A type identifier, then, simply serves as an abbreviation for a representation One well-known weakness of such a scheme is that two types may *unintentionally* be equivalent It is not clear that such *impersonation* ([Morris 73a]) is all that frequent but supplying an abstract data type[1] can explicitly prevent such misuse

Name equivalence considers two entities to be of the same type if they are declared using the same (perhaps anonymous) type name This approach does eliminate any unintentional equivalence but, unfortunately, also severely restricts the abstract notion of a type Strictly enforced, all formal parameters would have to have a named type

A third, more abstract approach is termed *functional* or *behavioural equivalence* and is based more on the compatibility of an actual parameter with a corresponding formal parameter than the *equivalence* of types per se A formal type simply characterizes a set of objects that share some behavioural properties An actual parameter will be compatible with this formal type if it is an instance of this set This approach resembles structural equivalence.

Most current languages implement a hybrid of name and structural equivalence. It is the contention of this thesis that a blend of the behavioural approach with abstract data types provides a better concept of compatibility by allowing type abstractions to be expressed while enforcing distinct types where necessary

2.5 Type Hierarchies

[Carbonell 81] points out that the type hierarchies which abound in the fields of artificial intelligence, databases, and programming languages all share a central inference mechanism· inheritance of information. *Inheritance* implies that properties of a type are transmitted to

[1] See Section 2 6 for a discussion of abstract data types.

all instances of that type This inheritance can be achieved in a downward, upward, or lateral fashion

Grouping like entities together is achieved through data typing in most languages and through *classes* in Smalltalk Classes describe the properties of all instances of that class The properties of a Smalltalk class are defined by an internal state, by its recognized messages, and by the internal methods required for responding to those messages

A class may be modified to create another class This new *subclass* inherits everything about its *superclass*, the class being modified Such a modification may extend the internal state, the recognized messages, and the internal methods . The subclass refines the idea of its superclass In fact, all objects in the Smalltalk world are refinements of the most abstract class named *Object* and inherit information from this *Object* class A subclass is allowed to redefine a method described by one of its superclasses in order to tune the method

Smalltalk's world is based on the abstraction technique termed *specialization.* SIMULA embodies the same technique through its prefix classes

Ada's *derived types* show a lateral inheritance. The Ada declaration

type X is new Y,

implies that Y inherits all the properties of X (literals, aggregates, attributes, built-in and user-defined subprograms) but is a different type altogether Variables of the two types are not assignable, for instance, though values of the two types may be converted from one type to the other by built-in conversion routines.

Parameterized types, as found in Alphard, also define a hierarchical structure since the parameterized type is a generalization of each of its type instances For example, "list[x TYPE]" is a generalization of "list[Integer]" and "list[Complex]" Such a facility approximates the capabilities of a Smalltalk class; the parameterized type is akin to a superclass

Zilles ([Types 81]) notes that a definition of type is not necessarily dependent upon the set of values but rather the existence of certain operations on those values The idea of a *sortable* type simply implies that an ordering relation is defined for values of the type; integers, reals, and characters would be partitions of this type. The type hierarchy is built bottom-up by a process of *generalization* rather than *specialization*

Another simple bottom-up technique is often provided with *union* types or the related idea of a *variant record*. The *union* type is defined by enumerating its constituent types.

2.6 Abstract Data Types

With the emergence of the belief that data was at least as important as algorithms, an object-oriented view of the world and the related concept of data abstraction came into being.

[Parnas 70] developed several principles to guide the decomposition of a program into modules. One, his principle of *information hiding*, states that there should be one module for each difficult design decision in the program. The results of each decision should be hidden in a module, if this decision were later changed, only that module would need to be modified.

One common design decision is the data structure representation. In a well-modularized program there will be one module for each data structure. Any manipulation of the data structure must then be done through the procedures provided by the module because the representation of the data structure is hidden in the module. Users of the module are required to use *abstract* operations on the data structure, since the representation is hidden, no concrete operations are known. A module that provides a set of abstract operations on a data structure is termed an *abstract data type*; this approach corresponds to the vision of a type as a set of data values together with a set of operations on those values. One can tune the internals of an abstract data type without affecting users of the type. Program development is improved by encapsulating the scope of change. Several languages encorporate the idea of abstract data types explicitly.

CLU's *clusters* define abstract data types. A cluster implements a new data type consisting of a set of objects and a set of primitive operations. Within the cluster, a concrete representation is chosen for the objects of the type. Only routines defined within the cluster may access this representation directly. Alphard provides a similar technique with its *forms*.

One problem associated with the cluster approach, however, is noted by [Schwartz 78]:

> "Interesting operations will often have multiple parameters, and often several of these parameters will be logically compound objects. The cluster mechanism cannot treat these parameters symmetrically, but perforce regards one of them, call it x, as a

> principal parameter to which the operation belongs, while the others are auxiliary .. This approach is not well suited to the description of operations which use multiple logically compound parameters in relatively symmetric ways When one writes an operation involving several equally important logically complex parameters there is no unique, parameter-type-determined, place in which to put the code representing the operation Thus one important support of the cluster approach breaks down "

To overcome this sort of problem, languages now supply other encapsulation methods that do not directly define abstract data types but can be used to group logically related items together Ada's *packages*, Tartan's *modules*, and Red's *capsules* all exemplify this mechanism Visibility rules can hide internal components of such an encapsulation An Algol 68 programmer can simulate an abstract data type but the language cannot prevent its misuse

Importantly, data abstraction induces correctness into the language· the user cannot ruin the integrity of an object if allowed access only through a (presumably correct) interface to the encapsulation Reliability is enhanced

Built-in types are truly abstract data types Allowing programmer definition of abstract data types blurs the distinction between what is built-in and what is user-defined Conceptually, they are equivalent.

2.7 Summary

Programming can be reduced to three main activities. (1) classifying the various objects one deals with into sets (types), (2) defining an algebra on these sets, and (3) using the algebra to solve problems.[1] This view of programming points out the import of types· defining the types and the related algebras is all that is involved The type system is the backbone of the programming language.

Type systems are not without their drawbacks. Statically typed languages in particular seem to be too inflexible It is an inability to express the abstractions inherent in one's world that makes type systems appear too restrictive. Circumventing this inability should be the goal of any new language.

[1] Actually, the second and third activities, though perceived differently, are one in the same, one solves a problem by extending the algebra to include the solution.

CHAPTER 3

Type Fundamentals of M

3.1 Introduction

A language M has been created in order to study both the primitive and abstract concepts of type and with the primary motivation of gaining type flexibility. In the design of M, static typing was regarded as a positive force in the program development process but restrictions inherent in the type system of many current languages were recognized. M was intended to be made flexible by allowing type abstraction while maintaining static type security.

The facilities inherent in a type system were discussed in Chapter 2. This chapter discusses the fundamentals of the type system in M and shows how such facilities are provided. In Chapter 4, a more abstract view of type is defined in order to increase the expressiveness of the language.

Before discussing the specifics of M, the principle of type-completeness is explained. Through the use of this principle, M can be defined using primitive constructs that are few in number but which provide a flexible and expressive language.

3.2 Type-Completeness

To design a language with many changeable parts, it is necessary first to design a language framework that specifies how the parts must behave and how they may be composed. The idea of *type-completeness* is to require that this framework consist of a type structure which specifies the legal use of names and a few, uniformly applicable composition rules, which specify the types and meanings of composite expressions in terms of the types and meanings of their components. This idea is a fundamental tenet of M.

[Demers & Donahue 80a] argue that the idea of type-completeness is of the utmost importance in the design of programming languages. They define this idea as follows.

1. Each name, be it an identifier or an operator symbol, and each

expression in the language has a type; the type of any composite expression is composed from the types of its components

2. For each type in the type structure of the language, it is possible to write an expression in the language having this type.

3. Any expression can be parameterized with respect to any free name having any type in the expression to yield a function of an even more complex type This implies that functions must be able to have parameters of any type and to produce results of any type

Type-completeness, then, is used to simplify language design by generalizing the concepts of parameterization and declaration Lack of type-completeness forces the introduction of special mechanisms to handle cases where common combining forms would suffice. The type structure of a language is the language's real framework.

The focus of type-completeness changes the fundamental role of the language designer from one whose responsibility is to put together a large number of "features" to one who must devise a rich but small type structure and who is then forced to live within its constraints A type-complete language has a wider range of flexibility than much larger languages with many special features Both user and designer are aided by the simplicity inherent in type-completeness the user has fewer concepts to master and the designer knows that only a few combining forms will suffice thus lessening the concern about what should or should not be added to the language.

3.3 Flavour of M

M is a block-structured, type-complete, expression-oriented language very much in the spirit of Algol 68. Many of its design decisions can be traced to this language as well as to the languages Russell ([Demers & Donahue 79])[1] and L ([Cormack 81a])

Someone conversant in Algol 68 will be able to assimilate the basics of M easily since the overall styles are similar To a Pascal programmer, though the style is certainly different, the types in M parallel those in Pascal This section gives a quick feel for the language M before discussing types in depth. Two important concepts are noted: the lexical structure of M is unique and is discussed first, the difference between the creation and the naming of entities, being separate concepts in M, is then made explicit

[1] See also [Demers & Donahue 80a] and [Demers & Donahue 80b]

3.3.1 Lexical Elements

The character set in M is divided as follows:

```
alphanumeric ::= letter | digit | underscore
letter       ::= "a" | "b" | .. | "z" | "A" | "B" | .. | "Z"
digit        ::= "0" | "1" | . | "9"
underscore   ::= "_"
bracket      ::= "(" | ")" | "[" | "]" | "{" | "}" | "[[" | "]]"
                 | "≪" | "≫"
separator    ::= "," | ";"
symbol       ::= underscore | { any other printable character }
```

Each bracket and each separator form a token. Tokens are also formed by a sequence of alphanumeric characters or by a sequence of symbol characters, the tokens so formed are termed identifiers. Note that the underscore is both an alphanumeric and symbol character. Each of the following is a legal identifier.

```
foo   237   2nd   if_then_else_fi   _+_   =
```

The first four are sequences of alphanumeric characters; the last two are sequences of symbol characters.

At a lexical level, where Pascal distinguishes among identifiers, numbers, and operators, M makes no such distinction. Nevertheless, an M token may be semantically treated as an identifier, number, or operator by declaring it to be one of these.

The underscore is included as both an alphanumeric and a symbol character and plays a special role in M by allowing an extensible syntax. Extensible syntax is discussed in more detail in Section 3.4.4.

3.3.2 Program Structure

The following grammar rules apply.[1,2]

---

[1] The entire (synthesized) grammar can be found in Appendix A.

[2] The notation $\underline{x \mid y}$ denotes a (non-empty) list of x's separated by y's.

```
program     := exprList
exprList    := expr ";"+
expr        := declaration
            := ...
```

A program is simply a list of expressions. Very often; initial expressions in such a list are declarations All expressions have a type and, like Algol 68, all yield a (possibly void) value

3.3.3 Simple Values

The simplest type constructor in M is the enumeration type and is defined by the grammar rule:

```
enumType := "{" id ","+ "}"
```

The enumType expression itself defines both a new type and literals (values) for this type. The expression

```
{ false, true }
```

depicts the common notion of a boolean type. It defines a new type value–a new instance of the type TYPE; the expression is of type TYPE. Both "false" and "true" are values of this type. The instances of the type TYPE can be determined by locating all type expressions in a program; i.e , the *enumeration* of these instances is spread throughout the program.

3.3.4 Declarations and Qualified Expressions

The following grammar rules are related to declarations:[1]

```
decl          := id ":" qualifiedExpr
qualifiedExpr := typeExpr "==" expr
```

A declaration is an expression that associates a name with a value and yields a void result; it makes the name a synonym for the value As a means of redundancy, one must explicitly

[1] Italics imply a semantic constraint, not a syntactic one.

state the type of the value (In the grammar description, both *ids* and *names* are identifiers; an *id* is the defining instance of an entity; a *name* is a use of an entity defined by the corresponding *id*.)

A declaration itself creates no new entity, it simply provides another name for an existing entity The binding of the name to the value is known throughout the scope of the declaration A scope is a syntactic form in which names may be defined and over which the use of a name has meaning.

To name the boolean type mentioned previously, the following declaration would be appropriate:

Boolean TYPE ≡ { false, true }

Each entity in M may be denoted by its name or, more fully, by both its name and its type (a qualifiedExpr). Both "false" and "Boolean ≡ false" identify the first literal of the type Boolean defined above Since a single name may be used to denote many separate entities, a *type* ≡ *name* pair may sometimes be the only way to distinguish among the named entities For instance, given the following definition,

TernaryBoolean . TYPE ≡ { false, maybe, true }

"maybe" is uniquely defined as a "TernaryBoolean" but "false" may denote either "TernaryBoolean ≡ false" or "Boolean ≡ false" in a given context Using a single name like this to denote separate entities is termed *overloading* and is dealt with in more detail in Section 3.8

The following sections develop the concept of type within M more fully. The discussion deals with the functional and non-functional aspects of the language separately.

3.4 Functional Type Features

Type expressions exist in M which parallel the enumeration, record, and array types of Pascal. Such expressions define new type values–values of type TYPE. Most type expressions are usually found in declarations since they must be named to be used further; the declaration

x . TYPE ≡ ...

typifies most type declarations. The ellipsis here must be replaced by one of the various

type expressions to be described shortly.

Ideally, a type expression would be as general as any other expression in the language. In order to maintain static typing, however, some restrictions are necessary. These restrictions are described in Section 3.7.

3.4.1 Simple Types

Scalar types are the primitives of any type system; they have no subcomponents. Early languages, designed in response to the requirement for scientific and engineering applications, supply scalar numeric types such as INTEGERs and REALs Later, when the need to manipulate non-numeric data was recognized, symbolic *enumeration* types were included. The enumeration type is a mechanism for constructing a type by *enumerating* all of its possible values. Each of the enumerated values is a literal of the type.

Enumeration types are the only scalar types in M and form the simplest method of classifying entities into sets.

enumType ::= "{" id $\underline{+}$ "," "}"

Several familiar concepts may be modeled simply:

```
TrafficSignal : TYPE ≡ { Red, Yellow, Green } ;
DaysOfWeek    : TYPE ≡ { Sun, Mon, Tues, Wed, Thu, Fri, Sat } ;
Digit         : TYPE ≡ { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
Boolean       : TYPE ≡ { true, false } ;
CardSuit      : TYPE ≡ { Spades, Hearts, Diamonds, Clubs } ;
CardFace      : TYPE ≡
    { Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King } ,
```

The sets defined by enumeration types are disjoint. No one value can belong to two enumeration types. Two different values can, however, be denoted by the same identifier The Digit "3" and the CardFace "3", though two very different values, are denoted by the same identifier. These two values can be uniquely identified by the qualified expressions "Digit ≡ 3" and "CardFace ≡ 3".

There are no built-in operations for scalar types. All must be explicitly defined. If one intends an enumeration type to be ordered, the ordering must be explicitly defined. Supplying a built-in algebra is an attempt to guess at the operations inherent for a new type and may add some convenience. Nevertheless, this is not regarded as a function of a type system; it does not necessarily add to the expressiveness of the language or the flexibility of the type system.

The concept of a subtype is not fully discussed in this thesis since the motivating interest is the compile-time properties of type. Some notation is required though. The expression

$$enumTypeName_{expr_1 \quad expr_2}$$

denotes the usual subrange notation. The subtype comprises the points "$expr_1$" and "$expr_2$" and all other values of "*enumType*Name" lying between these two points. *Between-ness* is defined by the textual ordering of the lexical tokens used in defining the original enumeration type. The type expression

$$DaysOfWeek_{Mon \quad Fri}$$

depicts a subtype of the type DaysOfWeek.

The type VOID, to be discussed in Section 3.5.2, is the only built-in enumeration type. It behaves as if it had been defined by the declaration

VOID TYPE ≡ { void } ;

Some other enumeration types may be pre-defined in a standard library. Integers and reals may have to be implicitly declared. They will be treated as enumeration types in any case.

3.4.2 Product Types

In set theory, a Cartesian Product of two sets S and T, denoted $S \times T$, is the set of all ordered pairs (s,t) such that $s \in S$ and $t \in T$. This concept of a product extends to an arbitrary number of dimensions so that, in general, $S_1 \times S_2 \times \ldots \times S_n$ = the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ such that $x_1 \in S_1$, $x_2 \in S_2$, ..., $x_n \in S_n$.

M embodies the concept of a Cartesian Product simply:

productType ::= "[" id ":" *formalType*Expr $\underline{\text{,}}$ "," "]"

Examples of product types are:

Complex . TYPE ≡ [real Real, imag : Real] ,
Card TYPE ≡ [suit : CardSuit, face. CardFace] ,

The first definition not only defines a 2-dimensional space (Real × Real) named Complex but also denotes a simple algebra on this space. This algebra takes the place of *field selection* in other languages. Implicitly defined is a function from the type Complex onto Real named "real" which projects a point in Complex space onto its first axis. Similarly, "imag" is the projection onto the second axis.

Literals for product types are expressed as a comma-separated list of literals which correspond to the component types of the product. A literal for the type Cardface could be "(Jack, Hearts)" (The parentheses, though not technically required, are often present in order to parse properly; "Jack, Hearts" is acceptable.)

A product type is *not* described by an enumeration of its values, its values are implicitly garnered from the underlying constituent sets. Any list of enumerated values must be simple in that they are 1-tuples or atomic. Scalars permit a 1-space; products allow multi-space.

In several languages, elements of simple types are viewed as *values* but elements of product types are viewed as *objects*. They are both *values* to M. Rather than an object with several *components* manageable at will, M treats an instance of a product type as a single value but with some obvious *properties*. Even scalars have properties. (For instance, 1 *is* less than 2.) An implementation of the language may have to use objects to represent values, but forcing a programmer to view the world in such a manner is enigmatic.

Note that there is no restriction on the constituent types of a product type.

3.4.3 Map Types

If S and T are sets, then any subset of S × T is termed a *relation* on S × T. If, in this subset, each member of S appears exactly once as the first component of an ordered pair then the relation is termed a *function*. (The discussion of maps in this thesis is restricted to

functions only.) If (s,t) belongs to a function F, say, then t is denoted by F[s] and F is said to map s to t. The set S is termed the source; T is termed the target.

"S → T" denotes the set of all subsets of S × T which are functions. The cardinality of this set, denoted $|S \rightarrow T|$, is $|T|^{|S|}$ A particular instance of the map could be denoted by enumerating the constituent ordered pairs $\{ (s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n) \}$.

A map, then, simply expresses a relationship between sets By way of example, consider the two sets

Month = { Jan, Feb, .., Jun, ., Dec }, and
Count = { 28, 29, 30, 31 }.

There are 4^{12} different possible maps (functions) in Month → Count. One of these, the map

{ (Jan, 31), (Feb, 29), (Mar, 31), (Apr, 30), (May, 31), (Jun, 30);
(Jul, 31), (Aug, 31), (Sep, 30), (Oct, 31), (Nov, 30), (Dec, 31) },

specifies the number of days in each month of the year 1984

Maps occur frequently in programming. Some view an array, say "array (Integer range 1..5) of Colour", as a map from a set of subscripts, here Integer, into a set corresponding to the array element type, here Colour. A program itself, even, may be viewed as a map from an Input set into an Output set; the set of all (input, output) pairs expresses precisely the meaning of the program. An enumeration of all valid pairs is generally not feasible and other notations for a map must be found Instead, one may provide an algorithm for calculating the output from a given input This is the basis of computer programming and of problem solving in general.

In M, a map is defined via parameterization

mapType = "[" *formalType*Expr ","+ "]" formalTypeExpr

Where a mathematician would specify a map as "Month → Count", the alternative notation "[Month] Count" is used in M; this notation should be familiar to Algol 68 programmers. This map type comprises the 4^{12} functions from Month into Count

The syntax for a map literal follows in Ada's footsteps and uses the notation

$$\{ s_1 \Rightarrow t_1, s_2 \Rightarrow t_2, \ldots, s_n \Rightarrow t_n \}$$

instead of the usual ordered pair enumeration.

The only operation allowed for a map is *selection*. The normal functional notation is maintained though alternate syntactic forms, discussed in Section 3.4.4, are allowed

The source or target of any map is unrestricted Higher-order functions are provided by allowing the source or target to be a map type itself

M's map may be conceptualized as an array, there is no requirement for an array type constructor *per se* A parameterized construct may be seen as creating an array of constructs. a parameterized integer creates an array of integers, a parameterized procedure creates an array of procedures, each one tuned to its respective parameter. These entities are constant arrays, though, like instances of product types, they are simply values, not objects with components.

3.4.4 Expressions

This section exemplifies the syntax of M expressions, shows the types of various expressions, and exhibits the preferred style of M programs. The power of the various forms of expression is made apparent.

Expressions in M are similar to expressions in most other programming languages They are built up from the scalar elements of the type sets in a type safe manner; the arguments to the maps are of the correct type Simple expressions are

```
3          -- of type Digit
true, 6    -- of type [[ Boolean, Digit ]]
_and_      -- of type [Boolean, Boolean] Boolean, say
```

More complex expressions are constructed by supplying arguments to a map and possibly using the result as the source to a further map.

Selection from a map (termed a *function call* or *array indexing* in other languages) may be expressed using the normal functional notation or dot notation For instance, "f[x]" is equivalent to "x f". Other forms are also allowed since the underscore character serves a special purpose in M by supporting an extensible syntax Underscores actually denote the

expected form of selection expression by indicating parameter positions. This allows a natural form of expression The identifier "_+_" denotes a map with two parameters, a *call* will take the infix form "x + y" Similarly, "abs_" will take a prefix form, "_!" a postfix form, and "loop_until_pool" a matchfix form [1] Further examples follow

Form	Expression	Functional Equivalent
matchfix	if false then 3 else 4 fi	if_then_else_fi[false, 3, 4]
prefix	NOT true	NOT_[true]
postfix	x^	_^[x]
dot	obj.field	field[obj]
infix	3 * (2 + 8)	_*_[3, _+_[2,8]]
functional.	SIN[z]	SIN[z]

The use of synonym declarations is expected to be used wisely and liberally. For instance, even the "simple" expression

```
2.0 * 3 1415926 * 10 0
```

might better be expressed by the following list of expressions (an exprList)

```
( pi          Real ≡ 3 1415926,
  radius      Real ≡ 10.0;
  diameter    : Real ≡ 2 0 * radius,
  perimeter   Real ≡ pi * diameter,
  perimeter
)
```

This mirrors the style of an Algol 68 *closed clause*. The compound expression is a better documented, more understandable version of the intent to express the perimeter of a circle with a radius of 10.0 units Both expressions are of type Real; the exprList takes its type from the type of the final expression

The style of expression shown above uses synonym declarations to aid in comprehension and maintainability and is the preferred style in M All lines other than the last simply define names which are to denote values and subexpressions used within the

[1] [Leclerc 84] has implemented such a scheme for the language L. While other languages implement an extensible syntax by generating grammar rules on the fly, his scheme simply adds (scoped) lexical keywords Also see [Goguen & Meseguer 83]; they define a more general *mixfix* notation.

overall expression. These definitions aid in understanding but also allow names to be used in all places where a common subexpression is found.

> Aside: The term *single assignment*, often applied to this style of expression, is not indicative of the true semantics of the construct since no variables or assignments are involved. In M, the style is supported solely with synonym declarations. Nevertheless, the style is expressive and easily understood. LISP would be enhanced tremendously, while remaining *pure*, if it allowed definitions of this sort

3.4.5 Map Expressions

Maps play an important role in programming languages Pairwise enumeration of the elements in maps is the simplest form of definition but this enumeration is tedious and, at times, impossible Map literals, therefore, take two forms (1) pairwise enumeration, and (2) parameterized expressions. The pairwise enumeration uses Ada-like pairing notation and is used more often for maps with small domains.

$$\{ 1 \Rightarrow 1, 2 \Rightarrow 2, 3 \Rightarrow 0, 4 \Rightarrow 1, 5 \Rightarrow 2, 6 \Rightarrow 0 \}$$

M permits parameterization of any expression with the syntactic form allowed by the grammar rules

parameterizedExpr ::= "[" id ":" *formalType*Expr { "," } "]"expr

Therefore, the following three expressions denote the same map.

(1) [i . Integer$_{1\ 6}$] (i mod 3)
(2) { 1 ⇒ (1 mod 3), 2 ⇒ (2 mod 3), .. , 6 ⇒ (6 mod 3) }
(3) { 1 ⇒ 1, 2 ⇒ 2, .., 6 ⇒ 0 }

Most languages allow a form of parameterized expression but they usually restrict this parameterization solely to procedures or functions.

The "perimeter" expression shown previously was specific for a circle of radius 10.0 units It could be made more general through the use of parameterization The M expression would be.

```
[ radius: Real ]
( pi          . Real ≡ 3.1415926;
  diameter    Real ≡ 2 0 * radius,
  perimeter   Real ≡ pi * diameter;
  perimeter
)
```

This expression describes the perimeter for a circle of any size Due to its infinitude, it could not possibly be expressed as a list of "radius ⇒ perimeter" pairs The type of this expression is garnered from the types of the formal parameters and the type of the parameterized expression. Here, the type would be "[Real] Real" The names of the formal parameters and any subtype information is immaterial to the type

A typical map declaration might be

```
_nor_ : [Boolean, Boolean] Boolean ≡
```

where the ellipsis must be replaced by an expression whose type is "[Boolean, Boolean] Boolean". Such an expression will either be a map aggregate (which is quite possible in this case) or by a parameterized expression such as

```
[l: Boolean, r: Boolean] (not (l or r))
```

The full declaration would then be

```
_nor_  [Boolean, Boolean] Boolean ≡
          [l Boolean, r. Boolean] ( not (l or r) ),
```

yet this form is verbose. The following version is allowed in its place.

```
_nor_ : [l: Boolean, r: Boolean] Boolean ≡ [ .] (not (l or r)),
```

The parameter names on the left-hand side of the qualifiedExpr do not reflect on the type but may add some semantic intent. The "[...]" syntax on the right-hand side of "≡" inherits the parameter list from the left-hand side.

3.4.6 *Discriminated Union Types*

If S and T are sets, then the union of S and T, denoted S ∪ T, is the set of all elements in S or T (or both) Programming languages often include a similar concept M's syntax is

unionType ::= *type*Name | "∪"

Since the values for each type are disjoint, the programming language union is termed a *discriminated union*:[1] an element in the discriminated union of S and T can be traced to exactly one of these component sets. A discriminated union is partitioned into its component sets. This data structuring technique closely parallels the conditional or case control structure of standard languages.

Discriminated unions have been used to supply polymorphism–the ability to describe routines which are applicable across types–and to provide a variant record facility. Their necessity is lessened by a more abstract view of type in M Nevertheless, they are particularly useful as a basis for recursive data structures The following example illustrates such a type:[2]

```
Operator    TYPE ≡ { +, -, *, / },
Expr        TYPE ≡ FORWARD;
BinaryExpr  TYPE ≡ [[ left: Expr, op Operator, right: Expr ]];
Expr        TYPE ≡ Integer ∪ BinaryExpr;
x . Expr ≡ (1, +, (2, *, 3)),
```

The infinitude of values denoted by the types *BinaryExpr* and *Expr* is an implementation problem but not something about which the programmer need worry Importantly, references or pointers are not required in order to express a recursive data structure

The only operation allowed on a discriminated union type is the deUnion operation whose syntax is given by the following rules:

unionExpr ::= parameterizedExpr | "∇"
deUnionExpr = unionExpr "[" expr "]"

---

[1] *Discriminated union* is the term used by [Hoare 72]. Algol 68 calls this a *united mode.* CLU uses the term *oneof* .

[2] The keyword "FORWARD" is intended to have the same semantics as Pascal's *forward*

Each parameterizedExpr must have a single parameter and this parameter must correspond to one of the possible type alternatives in the unionType. The unionExpr must *cover* the unionType in that for each alternative in the unionType there must be a corresponding alternative in the unionExpr This operation mirrors the *tagcase* construct of CLU and the *conformity clause* of Algol 68.

For the unionType "Expr" defined above, the code

([i: Integer] *expr1* ▽ [b: BinaryExpr] *expr2*) [x]

expresses one particular deUnion expression Here, there are two alternatives corresponding to the possible type of x The alternative used will correspond to the actual type of x during execution If this type is "Integer", then the value expressed is

([i: Integer] *expr1*) [x]

which is a normal selection expression .

3.5 Non-Functional Type Features

Previous sections dealt with types in a functional world The world, however, does not seem to be totally functional; a concept of *state* does exist Programming languages, therefore, supply *variables* in an attempt to capture this state

The next sections discuss the primitives supplied by M for non-functional programming. Variables in M capture state information while procedures permit execution within a state. The operations which manipulate this state are intentionally more explicit in M than in other languages. Importantly, both variables and procedures are brought into the type structure of the language

3.5.1 Variables

[Hehner 82] notes that the concept of a reference is well understood outside the programming arena. A *title*, say "Chairman", may be associated with a particular person, this title may be transferred to another person at some point in time. The title "Chairman" never changes but rather the person to whom it refers. In this respect, the title (or name) "Chairman" is a variable reference On the other hand, a person's name is with them forever; one's name cannot be transferred to refer to another. The name of each person is not variable, it is a synonym for the actual person.

A *variable* is somewhat like a reference–hence the Algol 68 term *REF*–in that it relates a *name* with a *value*; i.e., a *name* refers to a *value*. At any given point in time, that name refers to exactly one entity but is free to refer to other entities at different points in time.

The concept of a variable in M is very much like that of CLU:

> "[CLU] variables are names used in programs to 'denote' particular objects at execution time. Unlike variables in many common programming languages, which are containers for values, CLU variables are simply names that the programmer uses to refer to objects. As such, it is possible for two variables to denote (or 'share') the same object. CLU variables are much like those in LISP, and are similar to pointer variables in other languages."

Unlike M, however,

> ". . . CLU variables are 'not' objects; they cannot be denoted by other variables or referred to by objects. Thus, variables declared within one routine cannot be accessed or modified by any other routine."

In M, variables *are* allowed to be grouped together and manipulated as a group; they *can* be denoted by other variables, they *can* be referred to by objects. M's variables subsume the concept of *pointers* inherent in many languages. In short, a variable can be treated like any other value in the language since variables are typed–variables are encorporated into the type scheme. For example, a type "VAR Integer" exists. An entity of this type is a literal of the type "VAR Integer", it may refer only to an Integer, it may be referred to by a "VAR VAR Integer".

Two operations are built in for variables. The first, the normal assignment operator ":=", is an infix operator taking a variable on the left-hand side and a value on the right. The second, the postfix operator "^" (called the deVAR operator), when applied to a variable, yields the current referent of the variable. These operators, though built-in, look very much like[1]

[1] The expression "PROC VOID" is explained in the next section.

```
_:=_ : [ VAR Integer, Integer ] PROC VOID ≡ ...
_^   " [ VAR Integer ] Integer ≡ ..
```

They work hand-in-hand. When a variable is assigned to through the assignment operator, all deVAR operations will yield the value found on the right of the assignment operator until such time as a new assignment is made

The individual elements of the type "VAR Integer" are not enumerated in the normal sense since their *enumeration* is spread across several declarations The expression "new VAR Integer" creates a new integer variable and the variable so created may be named by the normal declaration rules This declaration, strictly speaking, would have to take on the following form:

```
x : VAR Integer ≡ new VAR Integer;
```

The syntax is relaxed somewhat for this case by replacing the qualifiedExpr simply by the type of the variable itself and allowing an additional initialization phrase. The code[1]

```
x  VAR Integer ≡ new VAR Integer;
x := 5!
```

may also be expressed as

```
x : VAR Integer := 5!
```

This *sugaring* is allowed only for variable declarations Note that, unlike Algol 68 which drops a "REF" from a variable declaration in a similar sugared form, M does not drop the "VAR".

One important distinction between variables in M and variables in other languages is best illustrated by the following comparison. The code expressed below shows similar declarations of an instance of a Complex type.

[1] The "!" operator is explained in the next section.

Pascal:

```
TYPE Complex = RECORD re: Real; im: Real END;
VAR x: Complex;
```

M:

```
Complex: TYPE ≡ [[ re: Real, im: Real ]];
x: VAR Complex;
```

In Pascal, "x", "x.re", and "x.im" are all variables. Each of these may be assigned individually. In M, however, "x^.f := 3.5" would be illegal since "x^.f" is not a VAR, "x.f := 3.5" would be illegal since "f" is not a *field* of "x" (but is a *field* of "x^") The variability of a reference is restricted solely to that reference and is not distributed to the referenced components since this would treat variables as objects rather than as references to values.

Nevertheless, if *variable components* are required such a type can be described. Note the following variation on the previous Complex type

```
Complex: TYPE ≡ [[ re: VAR Real, im: VAR Real ]];
x: Complex ≡ ...;

--"x.re^" and "x.im^" are now legal
```

The initialization of "x" was purposely avoided here Again, the separation of creation from declaration creates a certain syntactic verbosity in the declaration of a compound entity with variable components. Strictly speaking, given the types

```
Product: TYPE ≡ [[ x: VAR Integer, y: VAR Real ]];
Array:   TYPE ≡ [Integer_0 .. 9] VAR Integer;
```

the following declarations would be necessary.

```
p: Product ≡ (new VAR Integer, new VAR Real);
a: Array   ≡ {0⇒new VAR Integer, ..., 9⇒new VAR Integer};
```

Where an instance is declared for a composite type whose components are all variables, the shortened declaration "id . *type*Expr" is allowed in place of the full; the "new VAR ..." expressions are made implicit. The previous declarations can be shortened to

```
p: Product;
a: Array;
```

A product type declaration is also allowed to have an initialization clause for its variable components. This initialization will occur for the corresponding components when an instance is defined. For example, given the type declaration

```
Product Type ≡ [[ x: VAR Integer := 1!, y: VAR Real := 1.0!]];
```

the declaration

```
p1: Product ≡ (new VAR Integer := 1!, new VAR Real := 1.0!);
```

may be replaced by the declaration

```
p1: Product;
```

No such initialization is allowed for a map type with variables. Assignment to each component must be on an individual basis, i.e., "a[0] := 1!; a[1] := 2!; ...". (In the next chapter, it will be shown how such an entity may be assigned to *en masse*, i.e., "a := {0 ⇒ 1, 1 ⇒ 2, ...}!")

3.5.2 Procedures

A procedure (PROC) is an expression syntactically enclosed between "≪" and "≫" brackets. The brackets delay the elaboration of the contained expression until some later point in time. An explicit use of the postfix dePROCing operator "!" forces the execution of the PROC, it replaces the *call* mechanism of some current languages

A procedure is intended to be executed within a state, the state being represented by the current references of all variables. Execution of a procedure yields the value of the contained expression, this value is usually dependent upon the state It may be, though, that only the state is to be affected, that no *value* is expressed by the elaboration of the procedure. Here, as in Algol 68, M allows an expression to return a VOID result

Two examples of procedures follow:

```
(1) ≪ 3 ≫
(2) ≪ y. Integer ≡ x^ * 2; write[y]! ≫
```

Here, the first example and the Integer expression "3" are very similar Executing this PROC always yields the Integer "3". PROCs are only interesting when there are variables

present. The second example will output the value "x^ * 2"; exactly what this value is will depend upon the state of "x" at the time the procedure is executed.

It is important that the distinction between procedures in M and procedures in other languages be made clear. Most common languages provide a procedure construct in order to

(a) name an expression/statement,
(b) parameterize an expression/statement,
(c) create a scope for local definitions, and
(d) defer binding of variables to values.

In M, these features are independent. In particular, one can parameterize a procedure in order to achieve the procedure concepts of other languages but this is not the real intent of PROCs. Simply, they are to delay execution (i.e., binding of variables to values) until a later more specific time. M's dePROCing operator is made explicit to assure that the programmer has the correct time frame in mind.

The effect of a procedure, as illustrated by the next example, should be noted carefully.[1]

```
z : VAR String := "abc "!;

  -- "&&" concatenates two Strings

Twice: String ≡ ( z^ && z^ );
  -- The value of Twice is known now.

Double: PROC String ≡ ≪ z^ && z^ ≫
  -- The value of Double is also know now but the value
  --   Double! will depend upon when Double is dePROCed.

... Twice  ... -- value is "abc abc ";
... Double! ... -- value is "abc abc ";
```

[1] In general, though no means of expressing operator precedence has currently been defined, postfix operators are assumed to have the highest precedence, followed by prefix, followed by infix. Infix and postfix are left-associative; prefix is right-associative. The built-in postfix operator "!" is given special status and has the lowest precedence of all. The expression "x := x^!" is parsed as "(x := (x^))!".

```
z := "xyz "!;
... Twice ... -- value is "abc abc ";
... Double! ... -- value is "xyz xyz ";
```

The following procedure declaration shows how an exprList can yield a PROC result.

```
random : PROC Real ≡
  ( seed : VAR Real := 0.2753168!;
    ≪ seed := f[x]!; seed^ ≫
  );
```

The example also serves to illustrate the notion of *storage* in M. "Seed" will exist when the declaration of "random" is elaborated and will be initialized at that time. This initialization is independent of any *call* to "random" The value *persists* between separate invocations. This simulates the idea of an *own* variable. The visibility of "seed" is both defined by and restricted to the exprList itself.

3.6 Parameterization

Previous sections have dealt with parameterization by discussing maps. The expansion of the parameterization subject matter here signifies its import in programming.

In M, a reliance on the concept of type-completeness ensures that parameterization and selection are universally applicable: any name can be a parameter, a parameter can be of any type, and an argument can be constructed that can be bound to any parameter. Since parameterization is the fundamental tool in any programming language for providing changeable parts in a program, it is important not to place any constraints on the forms of parameters and arguments If we want to use the same program for many different values of x, we can do so by making x a parameter. Type-completeness guarantees that this can always be done, no matter how x happens to be used in the program

Since M has brought variables, procedures, maps, and types into its type structure, it is important to discuss their relationship to parameterization. During this discussion, some seemingly natural programming concepts are shown to be unorthogonal. Some of the subtleties inherent in the language are made explicit here.

3.6.1 Types

In many languages, one often needs to create separate routines in order to process objects of different types even when the algorithms are identical. This increases complexity while degrading the clarity, modifiability, and reliability of the entire system. M does not suffer this degradation.

The definition of type-completeness implies that if we can give a name to a type, then we must be able to make that name a parameter. Type declarations exist in most languages yet type parameters do not. Newer languages, such as CLU, Alphard, and Ada, have attempted to remedy this omission but they do not approach type parameters from a type-completeness viewpoint These languages view type parameters as a compile-time feature, this being especially obvious in Ada's *generic* types. The concept of parameterized types should not be a special construct in the language but should come about naturally by using type-completeness as a design decision In M, type parameters are not grafted onto the language as an afterthought.

Some languages provide built-in parameterized types. Pascal, for instance, knows of the concept of array and allows a programmer to define arrays with any component type. Sets and files are also parameterized. The syntax, though, hides this parameterization, type parameters are identifiers preceded by the symbol *of*. [Steensgaard-Madsen 81] noted this technique and used it to provide parameterized types

Other languages provide parameterized types explicitly Alphard, for instance, CLU with its parameterized clusters, and Mary ([Holager 78]) with its *statically parameterized modes* [Tennent 77] describes a Pascal-like language with class parameters and parameterized classes Ada, Red, and Tartan simulate parameterized types via a generic facility. (In their favour, it must be noted that this restriction was forced by the Steelman ([DoD 78]) requirements.) SIMULA's class hierarchy provides an equivalent facility. Euclid's parameterized types are somewhat limited and serve to provide variant records; they cannot be parameterized by types.

No new construct is needed to implement parameterized types since parameterization is universally applicable in M. The following examples illustrate both type parameters and parameterized types.

Example: One-dimensional array type

```
1dimArray : [size: Integer, component: TYPE] TYPE ≡ [...]
            ([Integer_1 size] VAR component);

x. 1dimArray[ 10, Integer ];
y: 1dimArray[ 20, 1dimArray[ 5, Real ]];
```

Example: A "generic" list type

```
List· [t: TYPE] TYPE ≡ FORWARD;
EmptyList. TYPE ≡ { nil },
NonEmptyList: [t· TYPE] TYPE ≡ [. ] [[ car: t, cdr: List[t] ]];
List: [t· TYPE] TYPE ≡ [. ] ( EmptyList ∪ NonEmptyList[t] );

IntList. TYPE ≡ List[Integer],
```

This all said, however, it seems that type parameters normally only parameterize other types; the syntax might very well be different. Normal values can parameterize types but they simply serve to define, directly or indirectly, restricted types (subtypes). Type parameters will not normally be required for the usual parameterized entities, where the type of an entity is important, it can be garnered by other means

Chapter 4 further illustrates parameterized types.

3.6.2 Maps

Most languages allow array parameters but not all of these allow function parameters. To M, these are both maps (i.e., parameterized values) which, being declarable entities, must also be able to be passed as parameters under the concept of type-completeness. A functional programming style is well supported since higher-order functions can be defined and can be passed. For example, the declaration

```
integrate· [func: [Real] Real, low: Real, high· Real] Real ≡ ...
```

takes a function parameter over which integration will take place.

3.6.3 Variables

Parameterized variables can simulate array objects and reference-returning functions. A variable parameter simulates pass-by-reference parameter passing semantics.

3.6.4 Procedures

A PROC in M is normally used to affect or to query one's environment and must be explicitly dePROCed to be executed within this environment Taking advantage of this explicitness and of the type-completeness of M, one can define primitive control structures. (Control structures are not built into the language itself.)

```
if_then_fi :
  [ condition: Boolean, block. PROC VOID ] PROC VOID ≡ [.. ]
  ({ true ⇒ block, false ⇒ ≪ void ≫ } [ condition ]);.

while_do_od :
  [ condition  PROC Boolean, block: PROC VOID ] PROC VOID ≡ [. .]
  if condition! then ≪block!, while condition do block od!≫ fi,

if (x^ ≠ 128) then ≪ x := x^ ** 2 ≫ fi!

while ≪x^ ≠ 128≫ do ≪ x := x^ ** 2 ≫ fi!
```

Though "if_then_fi" takes a simple boolean condition, the condition for "while_do_od" is a procedure. This emphasizes the distinction between the two control structures. The "while" condition must be executed repeatedly over differing states in order to terminate.

3.7 Static Typing Restrictions

Though, ideally, types were to be *first class citizens* of M, some restrictions are necessary to ensure static typing. Specifically, a variable type is not allowed It is not statically obvious, for instance, whether the following declaration of "z" is legal.

```
X : VAR TYPE := Real!;
  ...
if condition then ≪X := Integer≫ fi!;

z : X^ := 3!;
```

A dynamically typed language would allow the declaration of "Z" and would determine its legality at run-time.

Since all types are established at compile time, all type expressions must be static One of these type expressions may be a selection expression on a parameterized type; the actual arguments in the selection must be compile-time knowable. In general, no type expression may involve deVARing or dePROCing either directly or indirectly.

3.8 Overloading

Recent languages allow *overloading*–identifying many distinct entities by the same name. [Brender & Nassi 81] note that

> " . (the overloading) facility contributes to both abstraction and namespace management Abstraction is aided because the same procedure name can be used for conceptually equivalent operations on different types of data For example, SQRT can be used for the square root operation for the various precisions of floating-point types Name management is aided because fewer names are needed, and naming conventions can be simplified or avoided "

Overloading enhances the ability to write understandable (i e , readable) programs, where the semantic distinction between two entities is minimal, so should be the syntactic distinction (The typical arithmetic operators are prime targets for overloading) Forcing different names for "conceptually equivalent operations" can be distracting, the proliferation of *max* and *min* functions in FORTRAN is one such example

Overloading is often allowed for both subprogram identifiers and enumeration literals[1] but not for variables, constants, or types. To truly overload one another, two subprograms must be distinguishable in some respect other than their name In Ada, this distinction is based on the *parameter and result type profiles* of the individual entities. In contrast, Algol 68 and Red look only at the parameter types. One entity *hides* another if they are not distinguishable by their profile Two subprograms declared in the same scope must have dissimilar profiles

[1] In Ada, enumeration literals are actually treated as parameterless functions.

When names are overloaded, some method of *overload resolution* is required to *disambiguate* or determine the entity intended While each entity is uniquely determined by a *type* ≡ *name* pair, always qualifying each name would be distracting Resolution techniques, based on the visibility rules of the language and on the type information associated with each visible entity, must infer the type of the entity from the surrounding context.

Each use of an overloaded identifier must be unambiguous In statically typed languages, if an ambiguity does exist, it must be detected at compile time Several techniques exist for overload resolution in Ada. Arguments abound on the number of passes of the parse tree required, the type of passes (bottom-up vs top-down), and efficiency in time and storage for each resolution technique [1] EL1 allows some overload resolution to take place at run-time though it takes only parameter types into account

Special rules sometimes help to enforce "conceptually equivalent operations" Overloading of operators is allowed, but since they are generally parsed as prefix or infix operations, any new definition must adhere to these parser-related restrictions In Ada, the equality operator is allowed to be overloaded but to return only a Boolean type, the inequality operator may not be overloaded and is always the negation of the corresponding equality operator

Overloading is inherent in the generics of Red: all instantiations are implicit and the name of the instantiation takes the name of the generic unit These individual instantiations are "conceptually equivalent" Though individual instantiations in Ada are not required to have the same name, common practice may be similar to Red's policy

Due to the concept of literals in M, the usual sense of overloading literals is made somewhat more complex. Since all identifiers are literals for some type, all identifiers (including those which name variables) should be allowed to be overloaded Functions are overloadable in many languages but arrays are not No such disparity exists in M since they are treated similarly In fact, in M, there are no restrictions on what may be overloaded

One further point must be made. [Dijkstra 76, p. 96] states his preference for using dot notation over functional notation In his view, *arrayX.lowBound* is better than *lowBound(arrayX)* since

"unless we introduce different sets of names for these functions

[1] See [Ada Rationale 79], [Persch et al 80], [Cormack 81b], [Baker 82]).

defined on boolean arrays and integer arrays respectively (which would be awkward) we are forced to introduce functions of an argument that may be of more than one type, something I would like to avoid as long as possible."

This does not solve the problem of name resolution but rather treats it in an inverse fashion The overloading still exists Furthermore, in M, the difference is merely syntactic

3.9 Summary of Type Features

In M, a normal enumeration type definition mechanism is provided along with generators to create product, union, and map types This shares a common ground with several languages Application of the principle of type-completeness, though, allows types, variables, and procedures in M to be typed, and also permits parameterization to be applied orthogonally Parameterized types become natural, control structures need not be built-in; expressions are more powerful.

An important distinction is made between arrays, functions, and procedures in M but this distinction is not common to other languages Functions are ordinary maps, arrays are functions whose target is a variable type (i e., a parameterized variable), procedures are not maps at all though they are often parameterized

Other factors should be noted First, a clean split is made between the *creation* and the *naming* of entities· naming provides a synonym for an (already existing) entity Second, the functional and non-functional aspects of M are also split The non-functional world is intentionally made more explicit Third, since all identifiers have a type, overloading has been generalized

This chapter has laid the groundwork for a language whose type system, with fewer primitives, surpasses the power of a language such as Pascal The next chapter builds on this foundation and a more abstract view of type to support still greater expressiveness

CHAPTER 4

Polymorphism

4.1 Introduction

Parameterization allows one to *widen* an expression to cover many similar cases. We *widen* the expression "1 * 2" to "[i: Integer] (i * 2)" to describe not just the doubling of a particular integer value, but the doubling of any integer. Parameterization implies generalization.

Most languages allow one to generalize and to describe the set of values for which the generalized expression holds. In programming languages, this set of values is described via a *formal parameter list*, the *type* of each formal parameter restricts the set of allowable actual values. If one's view of type is narrow, then parameterization is less general. By widening one's view of type, one can widen the parameterization thereby increasing the generalization.

The next section discusses a view of types in current languages and points out how this view is particularly limiting. *Polymorphism*[1] is then defined and a more general view of type is discussed. Next, the type compatibility rules in M are stated; in essence, the actual view of type in a language is defined by such rules. Finally, examples of polymorphic code are given.

4.2 Type Restrictiveness

Strong typing specifies that in any context in which an entity is used, the type of that entity must agree with the type expected. In particular, the types of actual parameters must agree with the types of the corresponding formal parameters. A narrow view of type ensures a narrow role for parameterization.

[1] See [Milner 78] for a theory of type polymorphism.

One oft-quoted criticism of Pascal is apropos. Pascal's restrictive array type is a result of two requirements

1 All types must be determinable at compile type
2 The dimensions are part of an array type

Since the dimensions of an array are part of its type, the dimensions of an actual array parameter must agree with the dimensions of the corresponding formal array parameter. This, in itself does not pose a problem Indeed, if one must make the dimensions of an array part of the type, so be it, "an array of 10 Integers" can be a type However, one should then provide a mechanism to express "an array of Integers" (where the size is not important) and "an array" (where the component type is not important either) Overspecification defeats the purpose of parameterization

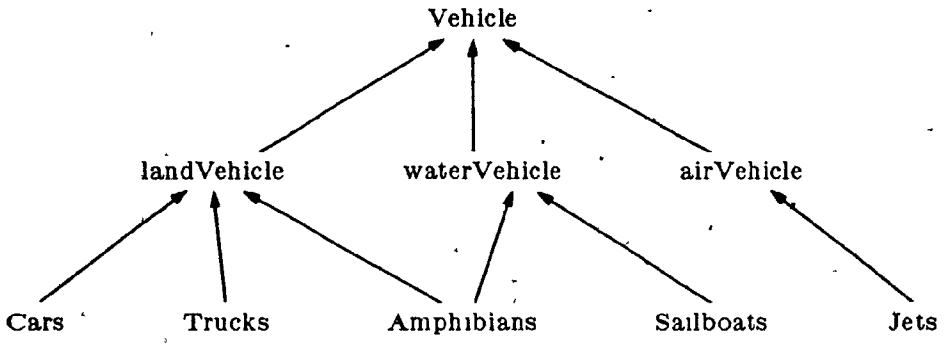
In Pascal, however, such a type abstraction mechanism is not present. If one intends to write a "sum" routine to sum an array, one must provide a separate instance of the routine for each size array. This, unfortunately, makes the language excessively cumbersome for programs that perform similar manipulations on a large number of different size arrays It is impossible to write a general array manipulation procedure in Pascal. Standardization efforts have made an attempt to solve this particular problem; several dialects have applied "fixes" (See [Kidman 78] for various proposals)

Pascal has been criticized for its lack of dynamic arrays but this is a symptom of a more general problem that most languages incur the inability to abstract the properties of an entity in which one is interested, the essential properties required should determine the type required The design of M has been motivated by the need to express such *type abstractions* One must be careful not to confuse this concept of *type abstraction* with the more common notion of an *abstract data type*

We wish to be able to write routines that can handle many different types if, in fact, these types have common properties. Though one may view "a list of Integers" as being different from "a list of Reals", say, there are operations which can be performed on "lists" themselves and upon which the types of the components have no bearing A *polymorphic* routine (*polymorph* meaning approximately "many forms") is a routine that can operate on an argument which can be one of "many types" It can subsume many instances of routines coded for different types Usually these "many types" have a certain set of properties in common and the polymorphic routine is allowed to use these properties in its definition Expressing this set of properties is the focus of the subsequent sections in this chapter

4.3 Generalized Hierarchy

The diagram below depicts the (nearly) hierarchical nature of objects in a Vehicle world.



Several important aspects must be mentioned The lowest level of this hierarchy comprises the actual entities in the Vehicle world, the sets at this level are disjoint. The levels above the bottom level denote abstractions in this world. Amphibians, being able to travel on both land and water, belong to two of the abstractions expressed on the middle level. These middle level abstractions are not disjoint. The top level here comprises all vehicles, it is the most abstract entity in the Vehicle world

It is often convenient to treat entities as belonging to one of these abstractions. There may be a single process required to register a Vehicle, for instance, independent of the actual vehicle type. The action of bailing out a waterVehicle could be described without referring specifically to reference to Amphibians or Sailboats

One cannot construct a landVehicle per se how does one assemble the parts of such an entity? One can, however, construct a Car, Truck, or Amphibian and this entity, once constructed, can be referred to as a landVehicle

The Vehicle world diagramed above, even assuming that no other specific types of vehicles really do exist, is by no means complete. Other abstractions can be inserted into the hierarchy motorizedVehicle, for example, or wheeledVehicle.

Abstractions, and a related hierarchy, are often described in a natural language by nouns and adjectives. A noun is "a word that is the name of a subject of discourse";[1] an adjective is "a word standing for the name of an attribute which being added to the name of a thing describes the thing more fully and definitely ".[2] In expressing a particular class of entities, the *preciseness* of the description (or lack thereof) can be tailored by an appropriate choice of noun and qualifying adjectives.

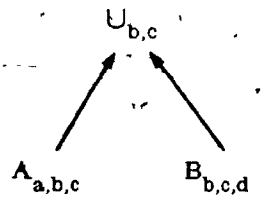
To expand on this concept, the following notation is introduced:

A, B, C, ...	denote *specific* sets of objects
a, b, c, ...	denote properties of objects
$T_{a,b,c}$	denotes a specific set "T" whose elements have the properties "a", "b", and "c"
$\cup_{u,v}$	denotes all elements in the universe with properties "u" and "v"; the set so formed encompasses all specific sets whose elements have properties which include both "u" and "v"; consequently, "$\cup$" denotes the universe of objects and comprises all specific sets

The following sets are posited:

$$A_{a,b,c} \qquad B_{b,c,d} \qquad C_{a,c,d,e}$$

Notice that these sets have properties in common. The set $\cup_{b,c}$ comprises both $A_{a,b,c}$ and $B_{b,c,d}$. A hierarchic relationship exists as depicted by the diagram:

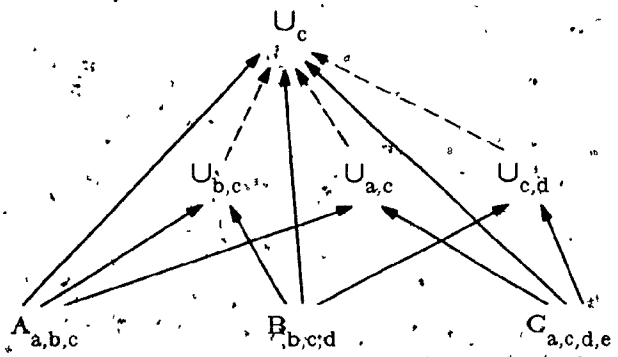


Other properties are also shared. Adding $\cup_{a,c}$, $\cup_{c,d}$, and $\cup_{c}$, the hierarchic relationship can

[1] Webster's Dictionary

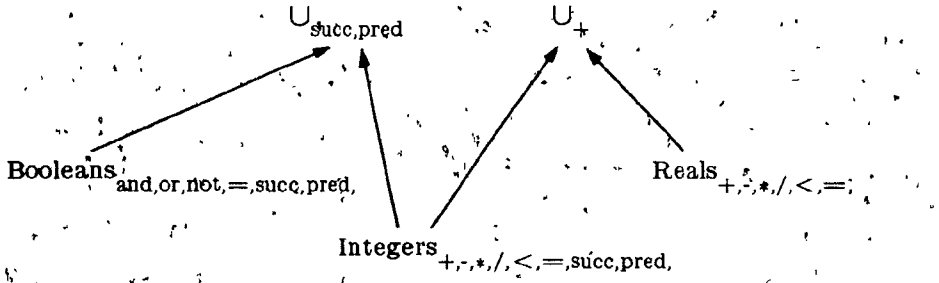
[2] Oxford English Dictionary

be drawn as shown below. Other abstractions do exist ($\cup$, $\cup_b$, even the empty $\cup_{b,e}$) but are not drawn.



A solid arrow "T ——→ $\cup$" implies that the specific set T is contained in the abstract set $\cup$. A similar relationship of containment between abstract sets is depicted by the dashed arrow.

Just as in the real world objects are classified via their properties into a generalized hierarchical structure, so may be the types in a type system. While in natural languages properties of an object are described via adjectives, in programming languages these properties are defined by the existence of a certain algebra. This type abstraction facility allows each entity to belong to many (abstract) types. Depicted below is a portion of a type world.



In this type world, the abstraction "$\cup$" would correspond to "any type". Again, notice that the two type abstractions here overlap: both comprise the Integers.

An important distinction between this *generalized hierarchy* and the type hierarchy allowed by a language such as Smalltalk must be noted. In Smalltalk, the hierarchy is built top-down. Each of its subclasses is a refinement of a single superclass. These superclasses (read "abstractions") cannot overlap. The previously diagramed Vehicle world cannot truly

exist in Smalltalk since Amphibians cannot be an instance of both a landVehicle and a waterVehicle.

Smalltalk's type hierarchy, built top-down through a process of *specialization*, is structured as a tree Each node in this tree must be named. A generalized hierarchy, built bottom-up through *generalization*, allows an expression to denote a node and permits flexibility Any possible type abstraction can be expressed An expression parameterized by such a type abstraction can be much more general. Parameterized types in M, though, do provide a hierarchical structure resembling a top-down structuring technique.

4.4 Type Abstraction

Overloading and polymorphism are related ideas in M Overloading can be used at the level of a specific type to state what are "conceptually equivalent operations"; these operations define the common properties. Taking advantage of this knowledge, one can then define a type abstraction. A polymorphic routine–one with a parameter whose type is an abstraction–then defines a "higher order operation" which is independent of any actual type.

An array of Integers is *orderable* precisely because its component type is *ordered* and the type Integer is *ordered* precisely because the relation "< [Integer, Integer] Boolean" exists. In fact, any type T, say, is ordered if the relation "< [T, T] Boolean" exists, any array of these Ts would then be orderable

Such an ordered type abstraction would be mirrored by the expression

```
?T | < [T, T] Boolean
```

and is read "any type T such that '< [T, T] Boolean' exists". ("?T" is termed an *abstract* identifier) Integers and Reals would fall in this category; indeed many types are orderable. The use of such expressions is restricted to formal parameter lists and their semantics must be discussed after the rules for type compatibility are stated. However, the further examples provided in this section should suffice to support a proper intuitive feeling for type abstraction.

Example 1: Printable Type

```
?t | stringify  [t] String
```

Since only Strings, say, can be written out, the ability to "stringify" a value (i.e , the existence of a map "stringify" from one type to the type String) makes that value printable

Example 2 Summable Type

```
?t | +   [t, t] t, additiveIdentity: t
```

Note that two "properties" are required Assuming the normal concept of Integers with "+" defined, the type Integer would then be summable if the declaration

```
additiveIdentity  Integer ≡ 0;
```

existed.

Example 3: Iterable Type

```
?t | first : t,
     last : t,
     succ : [t] t,
     ≤    : [t, t] Boolean
```

One can normally iterate over the Integers but not over the Reals. The first two properties here may be viewed as parameterless functions; they mirror Ada's built-in attributes

The designers of Euclid intended that the knowledge of *how* to enumerate the elements of some data type should generally be associated with the type (module) rather than with each loop that needs such an enumeration. In M, this *iterator* facility can be replaced by a type abstraction.

4.4.1 Type Compatibility

Though other languages have the notions of "assignability" and "type equivalence", the question of whether a given expression is type correct in M reduces to asking "Is the expression 'typeExpr ≡ expr' legal?" Type compatibility, then, must be established for each qualifiedExpr "Assignment" is treated like any other operation: it is a map selection. All selection operations, though, are defined in terms of a synonym declarations so that for the declaration "f [p pT] rT ≡ . ", "f[x]" is legal if and only if "pT ≡ x" is legal "Type equivalence" is a too strong a term Ideally, the type of an argument (T_a) should be *compatible* with the type of a formal (T_f) when the algebra defined on T_a supports at least the algebra of T_f.

Only three forms of type expressions are valid formal types (though an extension will be mentioned later):

*formalType*Expr

::= name	(Form 1)
::= name "[" expr+ "," "]"	(Form 2)
::= "[" *formalType*Expr+ "," "]" *formalType*Expr	(Form 3)

The first form corresponds to any named type, the name pinpoints a particular type value. An argument is compatible if it is an instance of this same type Enumeration, product, and union types must be named to be used since anonymous types are not allowable formal parameter types. For example, "p. {a, b, c}" cannot be a formal parameter though if the definition "T TYPE ≡ {a, b, c}" exists, "p T" can be This simple restriction allows a type matching scheme which will seem natural to programmers familiar with other languages It is important to note that different type names in M do not necessarily determine different types Many names can be synonyms for the same type though this should not be a preferred programming style

The second form signifies an instance of a parameterized type An argument is compatible if its type is a member of the same parameterized type and if the parameters supplied to the parameterized type at the declaration of the argument express the same value, i.e., are synonyms for the parameters to this formal parameterized type

The third form is a map type An argument is compatible if it itself is a map with the same number of parameters as the formal and each of these parameters match the corresponding formal parameter. In addition, the result type of the argument must match the result type of the formal This allows maps to be passable without restricting the

matching scheme to name equivalence.

Example: Form 1

```
Colour  TYPE ≡ { pink, magenta, red };
Color   TYPE ≡ { pink, magenta, red };
isPrimary  [x: Colour] Boolean ≡ .

... isPrimary[Colour≡magenta]    -- legal
... isPrimary[Color ≡magenta] .. -- illegal
```

Example: Form 2

```
1dimArray  [size Integer, component TYPE] TYPE ≡ ..

printArray  [1dimArray[5, Integer]] PROC VOID ≡ .

x1. 1dimArray[10, Integer] ≡ ..
x2: 1dimArray[20, Integer] ≡ . .
x3. 1dimArray[10, Real] ≡ .

printArray[x1]! -- legal
printArray[x2]! -- illegal, 20 is not a synonym of 10
printArray[x1]! -- illegal, Real is not a synonym of Integer
```

Example. Form 3

```
integrate. [func [Real] Real, low Real, high Real] Real ≡

.. integrate[sin, 0.0, 10.0] . -- legal (assuming a normal interpretation of "sin")
... integrate[_+_, .1, .2]    -- illegal, _+_ has wrong number of parameters
```

Though type abstraction "expressions", as alluded to earlier, do not really exist, such abstractions are expressible through a different means (but surprisingly similar syntax)

An "abstractId" is a lexical entity similar to an ordinary identifier but has an initial "?"–"?t", for instance The "baseId" corresponding to an "abstractId" is the abstractId stripped of its leading "?". In the definition of a formal parameter, an abstractId may take the place of a type name or of an argument to a parameterized type The intent of such a substitution is best realized by examples The following parameter declarations parallel the

three forms noted above but abstractIds have been introduced.

```
p1· ?t,
p2· List[?component];
p3  [?target] ?source,
```

AbstractIds minimize the importance of the actual type. An argument corresponding to the parameter "p1", for instance, can be of any type, the abstractId "?t1" acting much like a (property-less) type abstraction.[1] The parameter "p2" is matched by a "List" of any type, a List of Integers ("List[Integer]") perhaps or a List of Reals ("List[Real]"), the actual component type is of little importance The parameter "p3" is matched by a map with one parameter; again, the source and target of the map is immaterial

Type consistency throughout a parameter list can be effected by an appropriate use of baseIds and abstractIds Though a parameter, of type "?t", say, will be matched by an argument of any type, any further use of the corresponding baseId "t" within the parameter list must also be bound to the same type Note the following declaration

```
map [ p1  [?t] t ] t ≡ .
```

The parameter "p1" here would be matched by an entity whose type is "[Integer] Integer", but not one whose type is "[Integer] Real" since the individual "t"s must be consistent. Furthermore, if the argument were of type "[Integer] Integer", then the result of such a map selection will also be "Integer". The definition of "map", however, knows nothing about "t" other than that it is a type The type of the argument is inconsequential to the definition of "map"

Normally, an abstractId is used where a type expression is required It would then be of type TYPE and each use of the corresponding baseId must treated as a TYPE In a selection expression to a parameterized type (see the previous declaration of "p2"), the abstractId may take the place of something other than a type expression (though not in the case of "p2"). Parameters of a parameterized type can be of any type, the usage of the abstractId (and related baseIds) must be consistent with the corresponding declaration of the parameter for which it stands. In the expression "1dimArray[?size, ?componentType]", "size" is an Integer (as determined by the declaration of "1dimArray" above) and can only be used as such, similarly, "component" can only be used as a TYPE Since type identifiers cannot be overloaded, "1dimArray" is uniquely defined and the types of both "size" and

[1] A parallel exists in the EL1 mode "ANY" (also solely a formal type) [Wegbreit 74] terms this a *restricted union*

"component" are made obvious.

The next examples further illustrate the use of an abstractId.

Example AbstractId/BaseId Consistency

```
_<_ : [x: ?someType, y. someType] Boolean ≡ .

... 1 < 1       -- legal, someType bound to Integer
... 1 < 1.5     -- illegal, someType bound to Integer but second argument is Real
```

Example: Abstract Instance of a Parameterized Type

```
List [t TYPE] TYPE ≡

printEach: [List[Integer]] PROC VOID ≡ ...
writeEach: [List[?t]] PROC VOID ≡

intList:  List[Integer] ≡ ..;
realList: List[Real] ≡ . .;

printEach[intList]!     -- legal
printEach[realList]!    -- illegal since Real does not match Integer

writeEach[intList]!     -- legal
writeEach[realList]!    -- legal
```

We assumed earlier, however, that the type abstraction expression

```
?t | < . [t, t] t
```

was to encompass all types defined with the ordering relation "<". How is such a type matched? How do we check that the type of an argument is compatible? Certainly an Integer argument would seem to be legal but how do we know that "<" is actually defined for Integers? After all, unlike CLU, there is no "abstract data type" for Integers where we can locate such a definition. The answers to these questions are dealt with next.

4.4.2 Use-Site Binding

Normally, all identifiers not bound locally within an expression or to the parameters of that expression, need to be bound to entities more globally declared at a definition site M also allows parameters to be *use-site bound*–bound to an identifier at the site of the use of the corresponding parameterized entity Some languages term this *call-site bound*[1] This is *not* dynamic binding, rather, the identifier is still statically bound but in the environment of the user.

The example below shows the syntax and semantic effect of such a use-site binding. Explicit parameters in a formal parameter list are separated from the implicit parameters by a "|" symbol; the implicit ones, bound in the environment of the user, lie to the right of the symbol

[1] In M, supplying parameters to a procedure solely binds the parameters without executing the procedure Since, in other languages, the term *call* denotes both the binding and the execution, the term *call-site* is avoided when describing M; the term *use-site* is substituted.

Example. Use-Site Binding

```
-- Definition Site

negate [ x ?t | _-_ . [t, t] t, additiveIdentity: t] t ≡ [...]
        (additiveIdentity - x),

-- The value associated with "x" will be passed explicitly
--     from the use site
-- There must exist an entity named "_-_" at the use site whose type
--     is "[t, t] t" where "t" is the type of the argument associated
--     with "x", an entity named "additiveIdentity" must also exist
--     at the use site whose type is "t"

-----------------------------------------------------------------
-- Use Site

-- The type Integer will normally be global to all use sites since
-- it will probably be pre-defined  Assume that the normal "properties"
-- are defined  Similarly assume that Real exists

-- In order for the up-coming use of "negate" to be legal, if a declaration
-- of "additiveIdentity" is not defined or is not visible from here,
-- then it must be explicitly declared.  Assume one does not exist

additiveIdentity  Integer ≡ 0,

   negate[ -1 ]
-- legal, the additive identity for Integers is known here

   negate[ 1.0 ] .
-- illegal, no definition of "additiveIdentity  [Real, Real] Real" is visible
```

As stated previously, in checking the type compatibility of a *call* to "negate", the type corresponding to the abstractId "?t" is bound to the type of the related argument In the first use, this type is Integer The implicit parameters "_-_" and "additiveIdentity" must then have the types "[Integer, Integer] Integer" and "Integer", these definitions do, indeed, exist

Though previous discussion treated

```
?t | _-_  [t, t] t, additiveIdentity: t
```

as a type expression (a type abstraction), such an expression is really only a incomplete portion of a parameter list The definition of "negate" does not have one parameter whose type is expressed by a type abstraction expression Rather, it has three parameters–one explicit and two implicit Nevertheless, the actual concept of type abstraction is provided through an appropriate use of use-site binding and abstractIds Use-site binding is intended as a scheme whereby type information (operations, attributes) can be passed implicitly Note that type compatibility is independent of whether an argument is explicitly passed or use-site bound

4.4.9 Adjective Syntax

Semantically, neither abstractIds nor use-site binding is essential. One could simply say

```
sum   [Component   TYPE,
       array   [Integer1 10] Component,
       +   [Component, Component] Component,
       additiveIdentity   Component
      ] Component ≡
```

to declare the summation routine if parameters were allowed to depend upon previous type parameters The following use of "sum" would be typical

```
realArray  [Integer1 10] Real ≡
  sum[ Real, realArray, _+_, 0 0 ]
```

The only use of the type parameter "Component" is to ensure that other parameters involve the same type Type information is static, though, and a type parameter can deliver no extra information at runtime To lessen the verbosity of the definition and use of such maps, type information may be captured implicitly In place of the preceding declaration, the following is allowed:

```
sum   [array   [Integer1 10] ?Component,
       +   [Component, Component] Component,
       additiveIdentity   Component
      ] Component ≡
```

and then the form

```
sum[ anArray, +, additiveIdentity]
```

would typify each use The last two formal parameters of "sum", however, are actually intended to express properties of the type of first parameter Again, in order to capture this "type" information implicitly, the declaration could be changed to allow use-site binding. The above example would be declared as

```
sum  [array  [Integer1 10] ?Component |
       +  [Component, Component] Component;
       additiveIdentity  Component
      ] Component ≡
```

(note the "|") so that each use could then be simplified to the form

```
sum[ anArray ]
```

which is certainly a more abstract form; it retains the semantics of the original even though some arguments are implicit

In M, the declaration of "sum" may be simplified further through the use of *adjectives* The relevant syntax is

```
adjective    = id " " abstractId "|" properties
properties   = id " " formalTypeExpr+ ","
formalTypeExpr =
                .= adjectiveName abstractId
```

This adjective syntax mirrors that of abstract type expressions discussed in Section 4 4 Adjectives have no type, they are not values, they simply provide syntactic sugar

An *adjective*Name may be used in describing a formal parameter; the properties associated with the adjective are substituted as implicit parameters Given the following adjectives,

```
Iterable :: ?t | first : t, last . t,
                 _≤_ : [t, t] Boolean, succ : [t] t ;

Summable : ?t | _+_ . [t, t] Integer, additiveIdentity : t ,
```

the declaration

```
sum  [ array : [?Index] ?Component |
       first : Index, last  Index,
       _≤_  [Index, Index] Boolean, succ : [Index] Index,
       _+_  [Component, Component] Component,
       additiveIdentity  Component
     ] Component ≡ .
```

may be replaced by

```
sum . [ array  [Iterable ?Index] Summable ?Component ] Component ≡ .
```

The implicit parameters of each adjective are made implicit parameters of the map "sum".

The advantage of the adjective syntax is that it allows a programmer to envision the intended type abstraction. The required operations can be viewed as properties of a type rather than parameters of an expression

4.5 Polymorphic Expressions

When types, procedures, variables, and values can be passed both implicitly and explicitly, the expressions we write can be far more general. General array manipulation, for instance, becomes possible The next example sums the elements of an array Notice that the size of the array is not important All that matters is that we can iterate over its elements and that these elements are able to be summed

```
sum: [ array: [Iterable ?s] Summable ?t ] t ≡ [..]
     ( result: VAR t := additiveIdentity!,
       index: VAR s := first!,
       while ≪ index ≤ last ≫ do
           ≪ result := result^ + array[index^]!;
               index := succ[index^]!
           ≫
       od!;
       result^
     );
```

The power of polymorphism can be seen by the fact that many language-supplied operations can be viewed as polymorphic expressions The type "VAR Integer", for instance, is not defined through a VAR type generator Rather, it is an instance of the variable type family defined by the parameterized type

```
VAR_ . [component TYPE] TYPE ≡
```

The extensible syntax allows declarations to take on the natural notation "x VAR Integer", say, but this is equivalent to "x: VAR_[Integer]". Compatibility with a formal parameter whose type is "VAR_[. expr]" is determined by the parameterized type rule (see Section 4.4 1, Form 3)

An algebra exists for VARs independent of any actual component type This algebra includes the operations

```
_:=_   [variable: VAR ?t, value: t] PROC VOID ≡
_^     [variable VAR ?t] t ≡ .
```

These are provided directly by the language and cannot be explicitly defined However, the abstractions of assignment and deVARing exist across all variables regardless of component type. Other maps involving variables may be user-defined.

```
swap  [ left: VAR ?t, right VAR t ] PROC VOID ≡ [ ..]
      ≪ temp. t ≡ left^,
        left := right^!,
        right := temp!
      ≫;
```

The following definitions of "_:=_" allow "assignment" to be defined on entities that are not variables themselves but which contain variables; an array of variables can be assigned to *en masse*

```
_:=_ . [ target: [?s] VAR ?t, source: [s] VAR t ] PROC VOID ≡ [ . ]
        ( [i: s] (target[i] := source[i]^) ),
        -- copies 1 array (of variables) to another

_:=_ . [ target [?s] VAR ?t, source: [s] t ] PROC VOID ≡ [. ]
        ( [i. s] (target[i]  = source[i]) ),
        -- copies an array of constants to an array of variables
```

The following declarations describe the (built-in) parameterized type PROC and the dePROCing operator.

```
PROC_ [component  TYPE] TYPE ≡ ...
_!    [routineText: PROC ?t] t ≡ .
```

The following examples illustrate how actual routines can be written in M which other languages must build in

Example: if_then_else_fi expression

```
if_then_else_fi  [condition Boolean, Then ?t, Else  t] ≡ [ .]
                 ({true ⇒ Then, false ⇒ Else} [condition]);
```

This is not built into M It can be user-defined The previous definition of if_then_fi can be rewritten.

Example: Ordering Relation

If one method of ordering is defined, say via "_<_", other orderings may be declared simply

```
_≥_ [left: ?t, right: t | _<_ . [t, t] Boolean ] ≡ [.. ]
        (not (left < right));
```

Example: Algol 68 operations

Algol 68's "+:=" can be defined by

```
_+:=_ : [variable: VAR ?t, increment: t] PROC VOID ≡ [...]
        (variable := variable^ + increment),
```

Others, such as "-:=", may be similarly defined.

The next two examples illustrate the polymorphism inherent in parameterized types.

Example: List Length

(See definitions in Section 3.6.1.)

```
length: [list: List[?t]] Integer ≡ [...]
        (  [e: EmptyList] 0
        ▽ [n: NonEmptyList[t]] (1 + length[n.cdr])
        )  [list];
```

Example: Polymorphic Stack

```
Stack.   [Component: TYPE] TYPE ≡ [...]
         [[
            store: [Integer_1 100] VAR Component,
            1st· VAR Integer_0 100 := 0!
         ]],

push·    [stack. Stack[?t], element: t] PROC VOID ≡ [ .]
         ≪
            stack 1st + = 1!;
            stack.store[stack.1st^] .= element!
         ≫;

pop:     [stack: Stack[?t]] PROC t ≡ [...]
         ≪
            element. t ≡ stack.store[stack.1st^]^,
            stack.1st -:= 1!,
            element
         ≫,

top      [stack  Stack[?t]] t ≡ [. ] (stack store[1st^]^),

empty:   [stack: Stack[?t]] Boolean ≡ [. .] (stack.1st^ = 0);
```

4.6 Overload Resolution and Type Abstractions

Overloading is partially subsumed by an adherence to the principle of type-completeness and the use of parameterized types For instance, "push· [?t, Stack[t]] PROC VOID" subsumes the two instances of "push [Int, Stack[Int]] PROC VOID" and "push. [Real, Stack[Real]] PROC VOID" Where *generics* in Ada and Red create copies and perhaps overloaded identifiers, the polymorphism in M maintains a single abstraction Though instantiation is itself contrary to the concept of polymorphism, the implicit instantiation of Red better approximates this concept than the explicit instantiation of Ada

Unfortunately, polymorphic entities can also exacerbate the problem of overload resolution. Note the following example.

```
Female: TYPE ≡ .
Male: TYPE ≡ .

Person: [t TYPE] TYPE ≡ .

Mary. Person[Female] ≡
John Person[Male] ≡

isTalented [p Person[?t]] Boolean ≡
isTalented [p Person[Male]] Boolean ≡

. isTalented[Mary]
  -- unambiguous, can only match 1st definition of isTalented

. isTalented[John]
  -- ambiguous, matches both definitions
```

Here, it is natural to presume that the definition of "isTalented" whose formal parameter is "p: Person[Male]" is a refinement of the one whose formal parameter is "p Person[?t]" We would like the more specific definition to take precedence.

If two definitions both match a given use then we know that each definition has the same number of explicit parameters and that the corresponding explicit parameter types are related, i.e., they either are the same type or one type is an abstraction of the other–one encompasses the other. "Any type", signified by use of an abstractId, encompasses all types; a parameterized type encompasses all of its instances, a formal map type encompasses all map types with compatible parameters and result A priority of types exists: the more abstract type–the one that encompasses–is said to have lower priority than the other.

Overload resolution in M must allow for the precedence rule defined by the following algorithm:

> Given two matching definitions, find an explicit parameter position where one type has priority over the other (If no such position exists then the situation is ambiguous) Let *d1* signify the definition whose parameter type has priority, *d2* will be the other If at all explicit parameter positions, the priority of the type of *d1*'s parameter is greater than or equal to that of *d2*'s, then the definition *d1* has precedence. Otherwise, the situation is ambiguous

Such a precedence scheme gives the definition "isTalented: [p Person[Male] Boolean" precedence for the use "isTalented[John]" User-defined ":=" can be defined on a specific type and will have precedence over the built-in definition. If the declarations

```
f [x ?s, y Integer] Integer ≡
f [x Integer, y ?t] Integer ≡
```

both match a particular use ("f[1, 2]", say) then neither declaration has precedence; the first has priority for the parameter "x" while the second has priority for the parameter "y"

The rule prevents arbitration between two definitions such as

```
f [x ?t | p1, ..., pn] t ≡
f [x. ?t | p1 ] t ≡ ..
```

(The p_i's are intended to be properties of the type "t") If a particular use of *f* is ambiguous, neither definition will be given precedence even though the first seems more specific (in that we know more about the type "t") It is not apparent whether or not this sort of collision will occur naturally

4.7 Type Abstraction in Other Languages

Other languages do support the notion of a type abstraction though this is not their terminology.

In CLU, the cluster mechanism is used to implement a new data type and serves to collect a set of objects and a set of primitive operations together that will define this data type The cluster may be parameterized Type parameters are allowed and a *where* clause serves to constrain the permissible actual types The following header describes a routine which will accept an array of elements so long as the elements are orderable

```
proctype (array T]) returns T
  where T has LT proctype (T, T) returns boolean
```

CLU's *in* form allows abbreviations akin to the adjective syntax in M The previous routine header could also be declared as follows

OrderableType =
{ T | T has LT· proctype (T, T) returns boolean }

proctype (array[T]) returns T where T in OrderableType .

The declarations after the *has* keyword must mention routine types These routines, however, are not use-site bound The cluster which implements the type T must explicitly supply each routine Since it is impossible to determine *a priori* all the required operations for a given type, not all of them can be grouped in the type's cluster Though a user can define a function on a built-in type, this definition cannot be placed in the appropriate cluster The cluster mechanism finds it especially hard to place a map, say that is defined between types Use-site binding allows a degree of flexibility not permitted in a type cluster approach since *any* routine visible at a use site can be used

In Ada, an type abstraction facility is inherent in its generic type parameters

> "Generic type definitions may be array, access, or private type definitions, or one of the forms including a box ("<>") The operations available on values of a generic formal type are those associated with the corresponding generic type definition "

These generic types are akin to built-in type abstractions Table 1 depicts the syntactic form of various abstractions and how they are matched From the view of type abstraction in M, an integer type matches a discrete generic type precisely because the algebra is a superset of the algebra supplied for discrete types This relationship is guaranteed in Ada since the language built-in the algebras

Table 1. Type Abstractions in Ada

Syntactic Form	Meaning
limited private	any type
private	any type such that "=" and "=" exist
access T	any access type with base type T
(<>)	any discrete type
range <>	any integer type
digits <>	any floating point type
delta <>	any fixed point type
array (S_1, .) of T	any array of the same dimensions, index types, and component type

Parameter binding in Ada approximates that in M

> "Generic parameters are elaborated in sequence A generic parameter may only be referred to by another generic parameter of the same generic part if it (the former parameter) is a type and appears first "

This permits generic subprogram specifications to be dependent upon previously declared generic formal types Complex algebras which go beyond the "built-in" and which use more than one type can be described

Ada has a form of call-site binding If an argument corresponding to the generic formal parameter

```
with procedure SEND(    ) is <>,
```

were defaulted, M-like semantics would be attained.

In Red, a generic type parameter is used to stand for any arbitrary type and the use of "NEEDS" serves to supply an type abstraction facility An ordered type would be

```
t TYPE NEEDS <  (t, t) => Boolean
```

Red also defines this in terms of call-site binding As in Ada, a generic type parameter may be used subsequently in the definition of other generic parameters

Unfortunately, since Red's concept of overloading is restricted, so is its generic facility.

```
GENERIC
  t. TYPE
FUNC ovld(p. Boolean) => t;
  ...
END FUNC ovld,
```

This definition seemingly stands for all "ovld" functions which take one boolean parameter. Unfortunately, the definition is illegal since these "ovld" routines are indistinguishable in Red, the result is not taken into account.

Other languages show similar constructs Alphard appends a bracketed list $<p_1, p_2, ..., p_n>$ to a formal type parameter to denote the properties required of an actual type Similarly, [Gries & Gehani 77] provide a *require* $p_1, p_2, ..., p_n$ clause Both languages, however, allow only primitive operations to be specified [Demers & Donahue 80b] describe a type parameter which uses a *with* clause to list those operations which must be provided by the type A facility defined by [Jones & Liskov 76], though designed ostensibly for access control, *is* a type abstraction facility

4.8 Summary

The language M, through a bottom-up technique of generalization supports a generalized type hierarchy. The type abstractions in this hierarchy encompass any specific types which share a common (sub)algebra, an entity can logically belong to many types at once

The hierarchical structure of types is actually supported by a single type abstraction–"any type"–along with normal type matching rules Use-site binding permits type *properties* to be passed implicitly so that true *type abstraction* can be simulated

The flexibility gained by allowing type abstraction makes polymorphic programming possible, even within a statically typed language. The parameterization bandwidth is widened thereby allowing greater expression generality

CHAPTER 5

Summary

5.1 Synopsis

This thesis commenced by describing current concepts in type systems Type information, it was noted, serves the program development phase well by allowing a problem space to be partitioned into classes Properties within each class and between classes can then be expressed In programming languages, types define classes, algebras provide properties. Hierarchical structures within type systems were then discussed Describing an algebra is made simpler if one can form abstract views of type If a hierarchical structure exists, an algebra defined on the higher end of the hierarchy will hold for a greater class of values and will be, thereby, more general Hierarchies are valuable type abstraction facilities A second form of type abstraction was noted in the concept of an *abstract data type*. Although types must be represented explicitly on a machine, an abstract data type allows one to divorce oneself from this level of detail so as to view the type abstractly Type enforcement was viewed from two angles First, type checking prevents a problem solution from being described in terms that are not *meaningful* to the algebra Second, proper encapsulation of an abstract data type forces representation details to be hidden

To explore further the details of existing type systems and to challenge their designs, the language M was defined The design of a type system was perceived as the backbone of the total language design In M, orthogonality was to be achieved by adhering to the principle of type-completeness, reliability by maintaining static typing, and flexibility by permitting a less restrictive view of type

Though the basics of M's type system, by drawing on [Hoare 72]'s view, remain along classical lines, several deviations were introduced Only one form of scalar type–the enumeration type–is allowed Product types mirror ordinary record types but treat fields as contributing to the algebra of the type rather than as being components of each instance Arrays and functions are generalized to maps

A non-functional *state* is clearly separated from the functional world Variables are defined in terms of variable references rather than buckets Procedures are no longer viewed as special functions but rather as expressions to be elaborated at some later explicit point in time

Since type-completeness is an underlying tenet in M, all expressions are typed Even type expressions have a type An orthogonal application of parameterization allows user-defined parameterized types, higher-order functions, and control structures Maps further subsume the concept of *generics*. The syntax, by permitting a mixture of functional notations, allows expressions and control structures to be stated naturally

Introduced was the concept of *type abstraction* where a type can be described by a set of properties Two concepts support this idea abstract type identifiers and call-site binding An abstract type identifier used as a formal type is bound to the type of the corresponding argument Properties of this type can then be garnered by implicit call-site bound parameters An *adjective* syntax is provided to encourage the conceptualization of the intended abstraction Polymorphism is supported by both parameterized types and type abstractions

5.2 Future Directions

The design of the language M, as presented in this thesis, is not complete, some addressed areas need further scrutiny while other related fields need to be explored

One concept not addressed was that of an *abstract data type* Indeed, it may seem absurd to design a type system without such a facility In M, though, such a facility is considered to be an *information hiding* technique–a technique not so much tied to the concept of *type* as it is to *modules* and *visibility* Since the main concern was to add flexibility to the type system, the design tended towards providing *type abstractions* rather than *abstract data types*

Nevertheless, abstract data types are important and an appropriate encapsulation facility should be provided It is hoped that adding controlled visibility to the components of a product type will suffice to support such a mechanism This approach would be similar to that taken by Euclid whose modules are generalized records

Also not discussed was the topic of coercions A *type conversion* is a map from one type to another, a *coercion* is a conversion implicitly supplied If a user were able to state which conversions should be supplied implicitly, then a facility to supply automatic coercions could be meshed with overload resolution techniques Currently, every expression

in M has exactly one type and the decision not to provide coercions serves to reinforce this idea Unlike other languages, deVARing and dePROCing operations are explicit This overcomes Algol 68's ambiguity in "meekly coercible" situations Nevertheless, the explicitness may be distracting Coercions might be added but should be under programmer control DeVARing and dePROCing may be prime candidates

The decision not to supply a built-in algebra for enumeration types may be faulty Enumeration types in M lack the properties (predefined operations, attributes) defined by Ada and the programmer must necessarily define a map to ensure that the type is ordered The point made was that no algebra had to be primitive In real use, though, some facility is required to allow enumerated types to be "linearly ordered", perhaps "circularly ordered", or to remain "unordered", any necessary implicit algebra would be supplied Such a mechanism provides the added benefit of supplying "conceptually equivalent operations"

It is unclear whether the generalized overloading allowed in M will be beneficial but it is equally unclear how to justify any restrictions. Literals in many languages are overloadable M views all identifiers as being literals of one type or another and, therefore, overloadable Experience may show that some restrictions are necessary but, for now, an investigation of this generalized scheme may prove interesting

One design decision that may be loosened is that of the demand for static typing M's type system does provide a high degree of flexibility but this may still be too restrictive Indeed, we already have a degree of dynamic typing provided by union types we choose from a list of alternative types at run-time Subtypes are also a dynamic property Support of the dynamic approach does not advocate *typelessness* strong typing must still exist but may be dynamically supplied and as in EL1, should only be necessary when static typing is insufficient Harland and Gunn ([Harland & Gunn 84a Harland & Gunn 84b]) describe a dynamically typed polymorphic language and a possible architecture Their approach is flexible but lacks an explicit means of expressing type abstraction Perhaps a hybrid system, one that includes both static and dynamic type checking, may be desired in order to achieve the required mix of efficiency, flexibility, and reliability The *riskier* the execution, the more the system should be statically checked

Subtypes have not been addressed properly Though they do not affect the static properties of a type system, they should be incorporated smoothly within the overall type structure

5.3 Closing Thoughts

In the process of researching and writing this thesis, two notions became apparent First, design decisions can become intertwined. The actual number of concepts in a language, though best kept to a minimum, is less important than the orthogonality of these concepts. When concepts do not mesh cleanly, special rules are necessary, these rules become encumbrances The use of the principle of type-completeness reduces language size while increasing orthogonality

Second, ideas on polymorphism have been around for some time Why is it that a language based on such principles is not commonly available? Is it the pedantic nature of the software community, a natural lapse between idea and general acceptance, or is it, as [Holager 78] points out, that experience with polymorphic techniques has shown to be far less useful than anticipated. Time will tell Nevertheless, it is hoped that the ideas presented in this thesis make it apparent that a language embodying such principles simplifies language design while increasing language expressiveness and facilitating the programming process

References

[Ada 83]

Reference Manual for the Ada Programming Language. United States Department of Defense 1983.

[Ada Rationale 79]

Ichbiah, J.D , et. al. "Rationale for the Design of the Ada Programming Language". *SIGPLAN Notices.* v 14, n.6, June 1979.

[Algol68 76]

Revised Report on the Algorithmic Language ALGOL 68. (ed. A van Wijngaarden et al.), New York· Springer-Verlag 1976

[Alphard 81]

ALPHARD Form and Content (ed. Mary Shaw) Springer-Verlag 1981

[Baker 82]

Baker, T P. *A One-Pass Algorithm for Overload Resolution in Ada* "ACM Transactions on Programming Languages and Systems" v 4, n.4, pp.601-614, October 1982

[Birtwistle et al 74]

Birtwistle, G.M , Dahl, O.-J., Myrhaug, B., Nygaard, K. *SIMULA BEGIN* Lund: Studentlitteratur. 1974.

[Brender & Nassi 81]

Brender, R.F., Nassi, I R. "What is Ada?". *IEEE Computer.* v.14, n.6, pp.17-24, June 1981.

[Brooks 75]
Brooks, F.P., Jr. *The Mythical Man-Month* Addison-Wesley Publishing Company, Inc. 1975.

[Buckle 77]
Buckle, N "Restricted Data Types: Specification and Enforcement of Invariant Properties" In [LRDS 77], pp.68-76.

[Carbonell 81]
Carbonell, J.G "Default Reasoning and Inheritance Mechanisms on Type Hierarchies" *SIGPLAN Notices* v.16, n.1, pp.107-109, January 1981

[Cormack 81a]
Cormack, G.V *Separate Compilation and New Language Features* Ph.D thesis University of Manitoba 1981

[Cormack 81b]
Cormack, G V "An Algorithm for the Selection of Overloaded Functions in Ada" *SIGPLAN Notices* v 16, n.2, pp 48-52, February 1981

[Demers et al 78]
Demers, A J , Donahue, J E,, Skinner, G "Data Types as Values Polymorphism, Type-checking, Encapsulation". *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* pp 23-30, 1978

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

[Demers & Donahue 80a]
Demers, A.J , Donahue, J E " 'Type-Completeness' as a Language Principle" *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages* pp.234-244, 1980.

[Demers & Donahue 80b]
Demers, A.J , Donahue, J E. "Data Types, Parameters and Type Checking" *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages* pp 12-23, 1980.

[Dijkstra 76]
Dijkstra, E W *A Discipline of Programming.* Prentice-Hall, Inc 1976.

[DoD 78]
U S Department of Defense. *Department of Defense Requirements for High Order Computer Programming Languages, "Steelman"* U.S. Department of Defense. 1978

[FORTRAN 66]
Ansi Standard Fortran New York: American National Standards Institute 1966.

[Goldberg & Robson 83]
Goldberg, A., Robson, D *Smalltalk-80 The Language and Its Implementation.* Addison-Wesley, 1983

[Goguen & Meseguer 83]
Goguen, J., Meseguer, J "Programming with Parameterized Abstract Objects in OBJ" *Theory and Practice of Software Technology* (ed D Ferrari, M Bolognani, J Goguen) New York North-Holland Publishing Co pp 163-193, 1983

[Gries & Gehani 77]
Gries, P., Gehani, N "Some Ideas on Data Types in High-Level Languages" *Communications of the ACM* v.20, n 6, pp 414-420, 1977

[Harland & Gunn 84a]
Harland, D M., Gunn H I E "Polymorphic Programming I Another Language Designed on Semantic Principles" *Software-Practice & Experience.* v 14, n.10, pp.973-997, October 1984

[Harland & Gunn 84b]
Harland, D.M., Gunn H.I E "Polymorphic Programming II An Orthogonal Tagged High Level Architecture Abstract Machine" *Software-Practice & Experience* v.14, n.11, pp 1021-1046, November 1984

[Hehner 82]
Hehner, E.C R. *Programming Principles and Practice.* (partial draft) University of Toronto 1982

[Hoare 72]
Hoare, C.A.R "Notes on Data Structuring" *Structured Programming* . London Academic Press. 1972.

[Holager 78]
Holager, P "Generic Mode Facilities in Mary" *Constructing Quality Software*. (ed P.G. Hibbard, S.A. Schuman) North-Holland Publishing Company pp.117-133, 1978

[Jensen & Wirth 78]
Jensen, K , Wirth, N *PASCAL: User Manual and Report* New York Springer-Verlag, 1978

[Jones & Liskov 76]
Jones, A.K., Liskov, B.H. "An Access Control Facility for Programming Languages" Carnegie-Mellon University Technical Report, May 1976

[Kidman 78]
Kidman, B P "A Review of Proposals for Introducing Dynamic Arrays into Pascal" *Programming Language Systems* (ed M C Newey, R B Stanton, G L. Wolfendale) Canberra Australian National University Press pp 107-117, 1978

[Lampson et al 77]
Lampson, B W , Horning, J.L., London, R.L , Mitchell, J.G , Popek, G.L "Report on the Programming Language Euclid". *SIGPLAN Notices*. v.12, n.2, February 1977

[Leclerc 84]
Leclerc, D *Implementation Considerations for the Language L* Montreal McGill University 1984.

[LRDS 77]
"Proceedings of an ACM Conference on Language Design for Reliable Software". *SIGPLAN Notices* v.12, n.2, March 1977

[Liskov et al 79]
Liskov, B., Atkinson, R , Bloom, T., Moss, E., Schaffert, C , Scheifler, B., Snyder, A. "CLU Reference Manual". Massachusetts Institute of Technology Laboratory for Computer Science, October 1979.

[LISP 62]

McCarthy, J , Abrahams, P.W., Edwards, D.J , Hart, T P , Levin, M.I. *LISP 1 5 Programmer's Manual.* Cambridge. The MIT Press. 1962

[Milner,78]

Milner, R. "A Theory of Type Polymorphism in Programming" *Journal of Computer and System Sciences* v 17, n 3, pp 348-375, 1978

[Morris 73a]

Morris, J H. "Protection in Programming Languages" *Communications of the ACM* v 16, n 1, pp 15-21, January 1973

[Morris 73b]

Morris, J.H "Types are Not Sets". *ACM Symposium on Principles of Programming Languages* pp.120-124, October 1973

[Parnas 79]

Parnas, D.L "Designing Software for Ease of Extension and Contraction" *IEEE Trx on Software Engineering* v SE-5, n.2, pp.128-137, March 1979

[Persch et al 80]

Persch, G , Winterstein, G , Dausmann, M , Drossopoulou, S "Overloading in Preliminary Ada" *SIGPLAN Notices* v 15, n.11, pp 47-56, November 1980

[Red 77]

Red Language. Informal Language Specification Intermetrics 1977

[Schwartz 78]

Schwartz, J T "Program Genesis and the Design of Programming Languages". *Current Trends in Programming Methodology, vol IV, Data Structuring.* (ed. R.T. Yeh) Prentice-Hall 1978

[Shaw et al 78]

Shaw, M., Hilfinger, P., Wulf, W.A. "TARTAN Language Design for the Ironman Requirement: Reference Manual". Carnegie-Mellon University Technical Report, June 1978.

[STARS 83]
Department of Defense, USA. "Software Technology for Adaptable, Reliable Systems". *SigSoft.* v.8, n.2, April 1983.

[Steensgaard-Madsen 81]
Steensgaard-Madsen, J "A Statement-Oriented Approach to Data Abstraction" *ACM Transactions on Programming Languages.* v.3, n.1, pp 1-10, January 1981

[Tennent 77]
Tennent, R D. "On a New Approach to Representation Independent Data Classes". *Acta Informatica* v 8, pp 315-324, 1977

[Tennent 81]
Tennent, R.D *Principles of Programming Languages.* Englewood Cliffs: Prentice-Hall International, Inc. 1981.

[Types 81]
"Types" *SIGPLAN Notices.* v.16, n.1, pp.43-52, January 1981.

[Wegbreit 74]
Wegbreit, B. "The Treatment of Data Types in EL1" *Communications of the ACM.* v 17, n.5, pp.251-264, May 1974.

APPENDIX A

M Syntax

Notation. [x] denotes an optional x

x ┼ y denotes a (non-empty) list of x's separated by y's

Italics denote a semantic constraint, not a syntactic one

program ::= exprList ::= expr ┼ ","

expr ::= decl := id ":" qualifiedExpr
::= qualifiedExpr ::= *formalType*Expr "≡" expr
::= enumType := "{" id ┼ "," "}"
::= productType ::= "⟦" [id " "] *formalType*Expr ┼ "," "⟧"
::= unionType := *type*Name ┼ "∪"
::= parameterizedExpr ::= "[" explicitParms ["|" implicitParms] "]" expr
::= name
::= productExpr ::= expr ┼ ","
::= unionExpr := "[" id " " *formalType*Expr "]" expr ┼ "▽"
::= mapExpr ::= "{" expr "⇒" expr ┼ "," "}"
::= selection ::= expr "[" expr ┼ "," "]"
::= procedure ::= "≪" exprList "≫"
::= parenthesizedExpr ::= "(" exprList ")"

explicitParms ::= [id " "] *formalType*Expr ┼ ","

implicitParms ::= id ":" *formalType*Expr ┼ ","

adjective ::= id "::" abstractId "|" implicitParms

*formalType*Expr ::= any "name", "parameterizedExpr", or "selection" yielding a result of type TYPE
::= [*adjective*Name] abstractId

Both "id" and "name" are lexical identifiers, "id" simply depicts the defining instances An "abstractId", also a lexical entity, is formed by prefixing any identifier with the character "?".

A scope encloses each "exprList" and each "parameterizedExpr"

Syntactic sugar:

A right-hand side of a qualifiedExpr can inherit the parameter list from the left-hand side, the following two expressions are equivalent

x · [t1, . . , tn] T ≡ [p1· t1, , pn tn] f
x [p1. t1, . . , pn tn] T ≡ [ ] f

Special forms of lexical identifiers permit alternate *selection* expressions· "_id_", "id_", and "_id" define an "*infix*Id", a "*prefix*Id", and a "*postfix*Id", "id_1_id_2_.. _id_{n-1}_id_n" defines an identifier where "id_1" and "id_n" reflect the opening and closing identifiers for a matchfix expression while the other "id"s reflect separators *Selection* is extended to

selection ..=
:= expr *infix*Name expr
::= *prefix*Name expr
.:= expr *postfix*Name
:.= *open*Name expr, *separator*Name *close*Name