The Design of a Language for Algebraic Computation Systems

John K. Foderaro

Ph.D

Computer Science Division
Electrical Engineering and
Computer Sciences Department

Sponsors
United States Department of Energy
Advanced Research Projects Agency
System Development Foundation

Richard J. Fateman
Chairman of Committee

ABSTRACT

This thesis describes the design of a language to support a mathematics-oriented symbolic algebra system. The language, which we have named NEWSPEAK, permits the complex interrelations of mathematical types, such as rings, fields and polynomials to be described. Functions can be written over the most general type that has the required operations and properties and then inherited by subtypes. All function calls are generic, with most function resolution done at compile time. Newspeak is type-safe, yet permits runtime creation of types.

The Design of a Language for Algebraic Computation Systems

Copyright © 1983

by

John K. Foderaro

To my parents, Anthony and Rita Foderaro

Acknowledgements

I am very grateful to Richard Fateman, my research advisor and thesis supervisor, for his counsel and support. I would also like to thank the other members of my thesis committee, F. Alberto Grunbaum and Robert Wilensky.

I am grateful to Neil Soiffer and David Barton who helped me think through the ideas presented in this thesis. My thanks to those others who took the time to read and comment on my thesis: Bruce Char, Anthony Foderaro, Jim Larus, Vincent Norris, and Barry Trager.

My thanks also to Keith Sklower and Kevin Layer whose work on the Lisp system permitted me to build my prototype NEWSPEAK system.

Finally, I wish to thank my wife, Cathy, who helped me in ways too numerous to list.

This research was sponsored in part by the Department of Energy, (Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358), the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235, and by the System Development Foundation.

Table of Contents

1. Symbolic Algebra Systems....1
1.1. Introduction....1
1.2. Math-oriented vs Symbol-oriented....2
1.3. Related work....4
2. Existing Symbolic Algebra Systems....5
2.1. Introduction....5
2.2. Requirements....5
2.3. Languages in Existing Algebra Systems....7
2.3.1. Macsyma [Lisp]....7
2.3.2. Maple [C]....9
3. The Newspeak Language....10
3.1. Introduction....10
3.2. A Model of an Algebra System....10
3.3. The Newspeak Language....13
3.3.1. Object....14
3.3.2. Type....14
3.3.3. Procedures....16
3.3.4. Relations Between Types....16
3.3.5. Type-hierarchy....20
3.3.6. The Importance of Restricts....22
3.3.6.1. Satisfies....22
3.3.6.2. Extends....23
3.3.7. Restricts in other languages....24
3.3.7.1. Pascal and Ada....24
3.3.7.2. Smalltalk....24
3.3.7.3. Flavors....24
3.3.7.4. Glisp....25
3.3.8. Parameterized Types....26
3.3.9. Views of Types....26
3.3.10. Inherited Parameters....28
3.3.11. Procedures....29
3.3.12. Lex Descriptions....31
3.3.13. Type Parameters....32
3.3.14. Anonymous Restricted Types....34
3.3.15. Function Objects....34
3.3.16. Subtype....35
3.3.17. Generic Function Calls....36
3.3.18. Functional Parameter Inheritance....39
3.3.19. Distinguished Objects....40
4. The compiler....43
4.1. Type checking....43
4.2. Function database....43
4.3. Frozen Types....45
5. A Partial Implementation....46
5.1. Pointers and Object storage....46
5.2. Type Parameter Extraction....47

6. Related Languages 49
6.1. FRL 49
6.2. Capsules 49
6.2.1. Category, Functor and Domain 80
6.2.2. Lack of Category Hierarchy 82
6.2.3. Function Invocation 83
6.2.4. Package 85
6.2.5. Summary 86
6.2.6. Host Language 87
6.3. Andante and Newspad 49
6.3.1. Category, Functor and Domain 50
6.3.2. Lack of Category Hierarchy 51
6.3.3. Function Invocation 51
6.3.4. Package 54
6.3.5. Summary 55
6.3.6. Host Language 55
7. A Simple Collection of Algebraic Algorithms 57
7.1. Set, Monoid, Group 58
7.2. Ring, Integral Domain, UFD 60
7.3. Euclidean Domain 62
7.4. Polynomial 66
7.5. Using the Definitions 74
7.6. Type Conversion 74
8. Conclusions 77
8.1. Limitations 77
8.2. Future work 78

Bibliography 79

1. Symbolic Algebra Systems

1.1 Introduction

Since 1960 many languages and systems have been written to aid humans in performing computations symbolically. Some of these systems were written to compute in well-defined algebraic domains, such as Altran [Hall71] which specialized in rational function manipulation. Other systems, such as Sac-I [Collins71], were written as testbeds for the algorithms of symbolic algebra and presented the user with a large number of specialized modules, each providing an interesting set of operations in some particular domain (e.g. univariate polynomials over GF(p)). Others, such as Schoonschip [Veltman65] were useful primarily in predetermined application domains. (e.g. high energy physics). A number of systems such as Mathlab [Engelman69], Macsyma [Martin71] [Moses74], Reduce [Hearn71], and Scratchpad [Griesmer71] did not have specific limiting design objectives. By virtue of their extensibility they became known as "general purpose" symbolic algebra systems.

The algebra systems came in many forms: subroutine libraries for existing languages (Sac-I), extensions to existing languages - (Altran, Formac [Xenakis71], ABC-Algol [VanDeRiet73], Formula-Pascal [Teer78]) or complete systems (Reduce, Macsyma). Although only a few of the systems are in use today, and most of these not in widespread use, they did promote a great deal of study into the computation and representation problems involved in the manipulation of algebraic formulae.

Algebra systems have been written in a number of computer languages. Early systems were written in the popular languages of their time, Fortran (Sac-1, Altran), PL/1 (a later version of Formac), or assembler (Camal [Bourne71], Formac). Two systems which continue to be used heavily today, Macsyma and Reduce, were written in different dialects of the language Lisp in the late 1960's. In the past few years, two new algebra systems, Maple [Char83] and SMP [Cole81], have been written in the C language [Kernighan78].

The growth of the older systems has slowed to a crawl. The newer systems are growing rapidly but we fear that they too will stagnate when they reach the power of the older systems. We believe that this stagnation is a consequence of obsolescent foundations and design decisions. It is our long term goal to design a symbolic algebra system for which continuous growth will be possible. The keystone of such a system is the language in which it is written. In this thesis, we give the design rationale for such a language, define a language named Newspeak which satisfies these requirements, and indicate implementation strategies and compromises.

The language Newspeak is a unique blend of dynamic data-object creation, hierarchical data types, generic function calls, and strict compile-time type checking. It also has a novel method for specifying the data types of functional arguments. As a result, Newspeak does not suffer the high run-time cost usually associated with languages of similar expressiveness. Although Newspeak was designed to fit the needs of a symbolic algebra system, there is nothing specific to algebra systems in the language. Thus it may prove to be a useful tool for other applications which make use of hierarchical data structures (e.g. AI and VLSI design programs).

The rest of this section is devoted to describing how the system we designed differs from the existing symbolic algebra systems. In section 2 we list the requirements for our implementation language and look at the implementation languages for existing algebra systems from this perspective. In section 3 we introduce our new language, Newspeak, which has features designed to meet the needs of an algebra system. In section 4 we discuss the compilation issues. Section 5 deals with data storage issues. In Section 6 we contrast Newspeak with similar languages. Section 7 contains an annotated Newspeak program which implements polynomial manipulation algorithms in a very general form. In section 8 we summarize our work.

1.2 Math-oriented vs Symbol-oriented

Any algebra system defines representations of symbolic expressions and contains a collection of manipulative and mathematical algorithms. We call existing systems such as Macsyma, Reduce, Maple, and SMP, *symbol-oriented* because they tend to favor the manipulative processing of symbolic expressions over the execution of mathematical algorithms. Our goal is a *math-oriented* system. We will describe the distinction between these orientations first by an analogy.

Consider the task of writing a program to convert a sentence written in French to English. One solution would be to look up each French word in a French-English dictionary and replace it with the corresponding English word. Some knowledge of French conjugation would be required to locate a word in the dictionary. Such knowledge could be represented as pattern-replacement rules. An alternative solution would be to have the program read the complete sentence and convert it into an internal language-independent form. English could then be generated from the internal form. The first solution, the dictionary lookup with pattern matching, is somewhat analogous to the symbol-oriented algebra system. One characteristic of such a system is that the result may be correct for simple cases, but for complex cases or even simple but unanticipated cases, the system may produce a result which is completely wrong. The second solution, that of first trying to internalize and correctly model the input, is what a math-oriented system does. If the input is inconsistent or the system lacks the capabilities for processing it, the math-oriented system will notify the user.

We will now consider specific parts of an algebra system and how the symbol-oriented approach differs from the math-oriented one.

domain

symbol-oriented: A system of this type usually has a *general representation* for formulae. Commonly, this is a recursive tree form with the root node representing the operator and the child nodes representing the operands. This form is also used to represent programs (i.e., non-mathematical objects). The domain of programs in symbol-oriented systems are these general representation forms, which may or may not represent meaningful mathematical objects. The attitude of these systems is, "Represent anything that the user types in (that can be parsed), because it might be meaningful to some program."

math-oriented: A system of this type does not need a general representation. Such a system would be a collection of programs to manipulate representations of certain types of mathematical objects, such as integers or polynomials each of which has its own, specialized representation. The domain of such a system would be constrained to be all mathematical objects that have been included by the writing of programs to manipulate those objects. Other objects could be manipulated only after the addition of those types, and associated operations, by the programmer. The general attitude is, "Only represent those things which can be manipulated by the system. Do not allow operations which are not explicitly meaningful." We do not require axiomatic specification, for practical reasons, but the notion of axiomatization is compatible in that the structure and operations are categorized rigorously.

computation

symbol-oriented: In many systems of this type, computation proceeds primarily by pattern matching, or alternatively, by tree traversal. Since there is no guarantee that the operands are meaningful, programs in such a system generally look for certain known operand patterns and perform some operation if one is seen. By default, this processing is local in nature. The introduction of globally effective transformations (e.g. removing a common factor from numerator and denominator) is not easily supported by these techniques. In the absence of special reducible cases, programs may return a structure representing an incompletely understood object. For example, in Macsyma the differentiation function *diff* applies the rule that the derivative of a sum is the sum of the derivatives. Thus if a formula consisting of the sum of two programs (Lisp lambda expressions) were passed to *diff*, it would return the sum of *diff* of the two programs rather than report that such a request was meaningless.

Because pattern matching is expensive if not carefully guided, some symbolic systems have retreated in the direction of math systems by having specialized representations for certain classes of formulae. Macsyma includes specialized forms for rational functions, Poisson series, and Taylor series. The specialized forms represent the formulae in a certain fixed way which has been designed for efficient manipulation. Other special forms have been generated by users. Altran, based on rational functions, also has a form for simple truncated power series.

The application of a rule may trigger a long computation, as the replacement part of the rule can call an arbitrary program. The pattern match can be made expensive by requiring an expensive predicate to be applied. In Macsyma, many of the patterns are implicitly embedded in simplification programs (for efficiency) but this construction technique makes modification or debugging of these patterns and their enclosing programs very difficult. Often the model of computation is never explicitly indicated - in Macsyma it appears to be highly mutable deliberately by means of flag settings, and less deliberately by the passage of time as the program changes (i.e., it combines the worst features of declarative and procedural encodings).

math-oriented: Every object in the system has a type which determines which operations are permitted on it. If the user requests an operation on an object, the algebra system can proceed directly to the program which performs that operation, or it can report that such an operation cannot be done.

user interface

symbol-oriented: The user interface is trivial in such a overall system. There is a very close mapping between what the user types and the form used internally to store the formula. The output is basically an infix printing of the internal form with perhaps some concession to a more usual two dimensional form.

math-oriented: The user interface plays a vital role in the system. The user's input must be transformed into a valid internal object. It is important that the correct type of object is created from the input because the type will determine the valid operations on the object. Often the form the user types in to one of the existing systems is ambiguous. For example, 12 could be a member of "the integers modulo 37," or perhaps the polynomial 0*x+ 12 where the coefficients are members of the field of integers modulo 31. The input subsystem must be able to help the user select the appropriate type for his input.

There may be little correspondence between what the user sees as output and how the object is stored internally. If the users wishes to talk about subparts of an output expression, it is the output subsystem's formidable task to locate and extract or construct that subpart *and its type* from the internal form of the object.

1.3 Related work

Our work is inspired by the work of Jenks, Davenport, Barton, and Trager on Newspad [Jenks81] and the work of Barton on Andante [Soiffer81]. Andante and Newspad are languages still under design with a similar purpose to NEWSPEAK, the language described in this thesis. We will describe them in section 6.3.

2. Existing Symbolic Algebra Systems

2.1 Introduction

In this section we will present those properties which are useful in a high level computer language if it is to be used to write a math-oriented symbolic algebra system. We don't claim that a language *must* have these features, for most languages are sufficiently powerful to express any application. However, systematic approaches to a useful implementation language obviously become more useful by first providing primitives appropriate to our requirements.

2.2 Requirements

A math-oriented symbolic algebra system places some unique requirements on the language in which it is implemented. The following list of requirements will provide us with a measure to analyze the implementation languages of existing symbolic algebra systems.

interactive

An algebra system is typically interactive since it is often used as an exploratory tool. It isn't necessary to use an interactive implementation language to write an interactive system, but it does have advantages. If the implementation language is interactive, then the language environment *exists* at run time while the algebra system is running, and in fact the algebra system is just an extension of the capabilities of the interactive language. This aids greatly in debugging and it means that the algebra system needn't duplicate many of the facilities provided by a typical implementation language (such as support for input/output, memory management, exception handling and general operating system interfaces).

first-class user-defined data types

Most programming languages treat the integer and floating point types in a special, or *first-class* way. A special syntax (infix) is permitted for operations on these types, open compiling is often done, and enough about the relationships between the types is known to permit the compiler to do automatic type coercion. An algebra system deals with many different data types: polynomials in several forms, integers modulo a number, and so on. In the implementation language, these data types must be treated in a manner equivalent to that of the first class data types. The implementation language should be capable of type-checking, reading, printing, and open coding of operations on them. If the implementation language fails to treat user defined data types correctly, then the programmer of algebraic algorithms is forced to construct a language on top of the given implementation where his data types are understood (or forego the advantages provided by type checked languages). This layering of an additional language on top of an existing one has detrimental effects on the resulting system, as we see when we examine Macsyma below.

abstract data types

It is important that the unnecessary details of data types remain hidden from all programs except the programs that implement the type and thus require access to details. Morris calls this *type secrecy* [Morris73]. This insures that the implementation of a data type can be changed without concern that some piece of code depends on its current representation.

generic function calls

As was mentioned above, algebra systems create many different data types over which the

common mathematical operations such as *plus* and *times* make sense. We may write programs in which we want to add two quantities but whose *precise* types we do not know. Thus we would like to write *plus(a,b)* and let the types of *a* and *b* determine what piece of code is executed to add them. This is called a *generic function call*. Generic function calls also make sense for non-mathematical operations like *print*.

polymorphic functions

Certain algorithms work over a wide range of data types. We should have to write the algorithm only once and then declare the domain over which this algorithm is valid. If arguments of a function are permitted to have more than one type, then the function is called *polymorphic*. Often polymorphic functions are confused with generic function calls. Lisp has polymorphic functions but not generic function calls. Ada has generic function calls but not polymorphic functions.

hierarchical type checking

This is a two part requirement: first that there be type checking by the system and second that the type-checking programs be able to make use of the hierarchical relations between types in a algebra system. Type checking is a well established technique for catching common programming errors. It can also provides the compiler with information to increase the efficiency of generated code, especially generic function calls. Hierarchical typing, even in its simplest form, is found in few languages. Languages such as Pascal and Ada allow a type to be declared as a subrange of a scalar type. Languages such as Smalltalk and Flavors (in Lisp) allow a simple hierarchy of types to be created, but the compiler has very little knowledge of the hierarchy. In mathematics, the algebras of monoid, ring, field, etc, form a rather complex hierarchy. Some of the data types are parameterized, such as "integers modulo a prime p," and "polynomials over a coefficient domain D." The value of the parameter often determines how the type fits into the hierarchy. Polynomials over a field are Euclidian domains whereas polynomials over a unique factorization domain are a unique factorization domain. The implementation language for a algebra system must be able to support this complex hierarchy at compile time. The compiler can then type check expressions and resolve generic function calls.

efficiency

It is foolish to claim that language X is more efficient than language Y without estabiishing a machine and application context. Otherwise, one could build a machine whose primitives were those of language Y, and which could only run language X programs by first converting statements to language Y at some loss in speed. Therefore, in comparing efficiency, the computation model we assume is that of a simple uniprocessor with a uniform address space, such as a Motorola 68000 or a VAX (disregarding the exotic instructions). We do not explicitly pursue efficiency in our design as a separate goal, but it is implicitly of concern throughout.

uniform abstraction

The same mathematical notation that is used by grade schoolers is sometime also used in the most advanced mathematical papers. While additional symbols are used in in advanced papers, these symbols are for the most part just abbreviations and not a different language. This is unfortunately not the case in existing algebra systems. No amount of study of the Macsyma top level language will permit one to understand the underlying Lisp program to add two polynomials. The existence of two languages has a number of drawbacks. The user who really wants to understand the algebra system must learn both languages, and while he may find the top-level language easy to understand, he will prob-

ably be confused by the implementation language, especially if it isn't well suited to writing algebraic algorithms. Also, the fact that some programs have to be written in a language hidden from the user implies that there is something missing in the language accessible to the user. We want the implementation language to provide a uniform abstraction. All of the algorithms in the algebra system should be written in this language, permitting the curious user of the algebra system to understand the internals of the system.

It is important to mention that we are not suggesting that the casual user using the system in a calculator style manner be forced to use the implementation language. Rather, the serious user who wants to write programs in the system can, and perhaps should, write his programs in the single implementation language.

2.3 Languages in Existing Algebra Systems

The two most powerful generally available interactive algebra systems are Macsyma and Reduce, each of which is written in Lisp. It suffices to study just one of these Lisp-based systems. Two relative newcomers are Maple from the University of Waterloo and SMP from Cal Tech. Both of these systems are written in C (or a language close to C). Very little has been published about the internals of SMP, so we will use Maple as an example of an algebra system written in C. Because all of the existing systems are symbol-oriented and we are interested in constructing a math-oriented system, we will limit ourselves to examining how the implementation language affects the parts of the system that execute mathematical algorithms.

2.3.1 Macsyma [Lisp]

Macsyma evolved from Mathlab and from the work of Moses [Moses67] and Martin [Martin67]. It was written in Lisp (evolving with the Maclisp dialect, but later was made to run under a number of alternative Lisps). It is a collection of modules written by a number of programmers with many different styles. The interactions between modules and dependencies on particular data structures are many, with very little data abstraction being used. In fact the worst fears of the authors of Macsyma seems to have come true:

> We have grave doubts about the usefulness of large systems constructed through the haphazard contributions of unsophisticated users. Every new bit of the system must be carefully integrated with the old. [Martin71]

The reason that Macsyma has held together (from its origins in 1968 to the present) is due to the the work of a few people who understand most of the relationships between the modules. Also, the system has been rather static since 1974 or so. There are no automatic methods to insure the integrity of the system.

We believe that difficulties in understanding, modifying, and to some extent using Macsyma have to do, in part, with the use of Lisp as a implementation language. (Any criticism of 'Lisp' is suspect because there is no standard for the Lisp language. Yet, most Lisp implementations have a great deal in common. When we say that Lisp does not have some feature X, we cannot be sure that *no* implementation of Lisp has feature X. Rather we are saying that if an implementation doesn't have feature X, it can still be called Lisp without any disclaimers.) While Lisp is at its best as a prototyping system for small projects, it is not suitable for large programming projects unless the programmers use some discipline: we suggest modern programming practices such as abstract data types and well-defined module interfaces. Even though one obviously *can* write large programs in Lisp, that doesn't imply that one *should* write a large algebra system in Lisp. Let us examine how Lisp fits the requirements we made above.

Most Lisp systems are interactive. This satisfies one of our important requirements. Lisp also contains memory allocation and reclamation code (which, incidentally, Macsyma uses).

Lisp does not have first-class user-defined data types. The programmer is free to modify the evaluator and all of the relevant functions so that they look for new data types, but this is time consuming and can cause problems if more than one programmer does it independently. The lack of user-defined data types (and the fact that early programmers (c. 1968) didn't recognize the need for them) meant that Macsyma programmers created new data types out of standard objects. Two distinct methods were used. The first method uses a list whose first element is the type of the object and whose subsequent elements are the object (for example *(rat 1 2)* for 1/2). Unfortunately, knowledge of the form of each object is then spread throughout the code (for example, many functions would know that the second item in a *rat* form is the numerator). This method is expensive if the data object is small (such as an integer) since the amount of storage needed to denote the type would exceed the size of the datum itself. The second method uses a standard data type (e.g integer) in association with a global variable. An example of this is the use of the variable *modulus* to determine the meaning of Lisp integers. If *modulus* were set to seven, then integers would be treated in some circumstances as members of GF(7). This solution is very dangerous because changing *modulus* effectively changes the types of objects that have already been computed. Also, some parts of Macsyma ignore the modulus flag, as this example shows:

```
(c11) /* declare that we want to work in GF(7) */
      modulus:7;

(d11)       7

(c12) /* ask for the square root of 5, where 5 is considered
       * to be a element of GF(7)
       */
       sqrt(5);

(d12)     sqrt(2) %i
```

Note that even though we wish to compute in GF(7), Macsyma introduces an algebraic number $i\sqrt{2}$.

Abstract data types are generally implemented in Lisp by means of macro expansion of function calls that create and extract parts of the object. Macros are usually preferred over functions because macros are expanded in-line and are thus faster (and sometimes more compact). For example, *(main-var-of-poly x)* might be 'expanded' to *(car x)*. There is nothing in Lisp that prevents any function from accessing the contents of an object defined via an abstract data type; all that is necessary is to use the representation-manipulation primatives (e.g. *car, cdr, rplaca, rplacd* on lists).

Lisp does not have generic function calls, because Lisp doesn't do function resolution based on types at runtime.

Lisp does have polymorphic functions. In fact, because there is no way to specify that a function accepts only certain types of arguments, all functions in Lisp are trivially polymorphic because they accept arguments of any type.

Since Lisp doesn't have type checking or first-class user-defined data types, it certainly can't have the hierarchical type checking we desire. It is possible to declare types in Lisp programs but this type checking is quite different from that in other languages. It is used solely to tell the compiler that you believe that the value of a variable will always have a certain type, so that the compiler can open code expressions containing that variable. The compiler does not verify that your declaration is correct (in most cases it simply cannot), so as a result the compiled code may

fail in mysterious ways should the declarations be violated at runtime. The potential for failure is sufficiently pervasive that the MIT Lisp Machine design includes type checking in microcode.

2.3.2 Maple [C]

Maple is a small but surprisingly powerful symbol-oriented algebra system being written at the University of Waterloo. It is written in a cross between three languages BCPL, B and C, but we can assume for our discussion that it is written in C, the most popular of the three languages. The C language fits so few of our requirements that it wouldn't be worth considering as our implementation language. However if our goal were to write another symbol-oriented algebra system, C would not be a bad choice. The authors of Maple attribute its success partly to their willingness to try a different data structure for formulae [Char83]. The success of Lisp-based algebra systems made it seem that the best way to implement formulae were with lists and trees. The authors of Maple feel that the use of lists to represent formulae introduces an unnecessary layer of abstraction between the algebra system and the host machine. They chose to use what they call "dynamic arrays," which map more closely to a typical machine architecture. This also eliminated the need for a list processing language (like Lisp), although garbage collection and other features were re-implemented.

We believe, though, that the direction of research should be toward more math-oriented systems. C has few of the features we desire in the implementation language: it is efficient and has a very weak form of data abstraction. It would likely be difficult to convert a program such as Maple to a more math-oriented system.

3. The Newspeak Language

> The language NEWSPEAK was named after the language in George Orwell's novel *1984* [Orwell50]. In *1984* the tenets of society are engraved on the tower of the Ministry of Truth: War is Peace, Freedom is Slavery, Ignorance is Strength.

3.1 Introduction

In this section we describe the language NEWSPEAK that was designed to fulfill the requirements for a math-oriented symbolic algebra system. In interactive computer languages and environments of today such as Lisp, Smalltalk, Flavors, and Basic, compile-time type declarations are not required. It is claimed that this absence of typing gives the programmer more freedom since he need not worry about describing all the details of a program in order to get part of it running. NEWSPEAK, our new programming language, is based on the Orwellian contradiction that *typing is freedom*: requiring the NEWSPEAK programmer to declare types results in an *increase* in his freedom of expression. He can count on the *relations between types* to help him write his program and he can safely use modern programming techniques such as data abstraction and generic function calls without fear of loss of efficiency.

In the next section we present a simple model of an algebra system. In the following section we describe features of the NEWSPEAK language relevant to programming a symbolic algebra system. Constructs in NEWSPEAK will be compared with similar constructs in Smalltalk, Flavors [Weinreb81] and Glisp [Novak82]. We will then examine how NEWSPEAK might be implemented. Finally we will compare NEWSPEAK to the languages Andante and Newspad, two language with goals similar to those of NEWSPEAK.

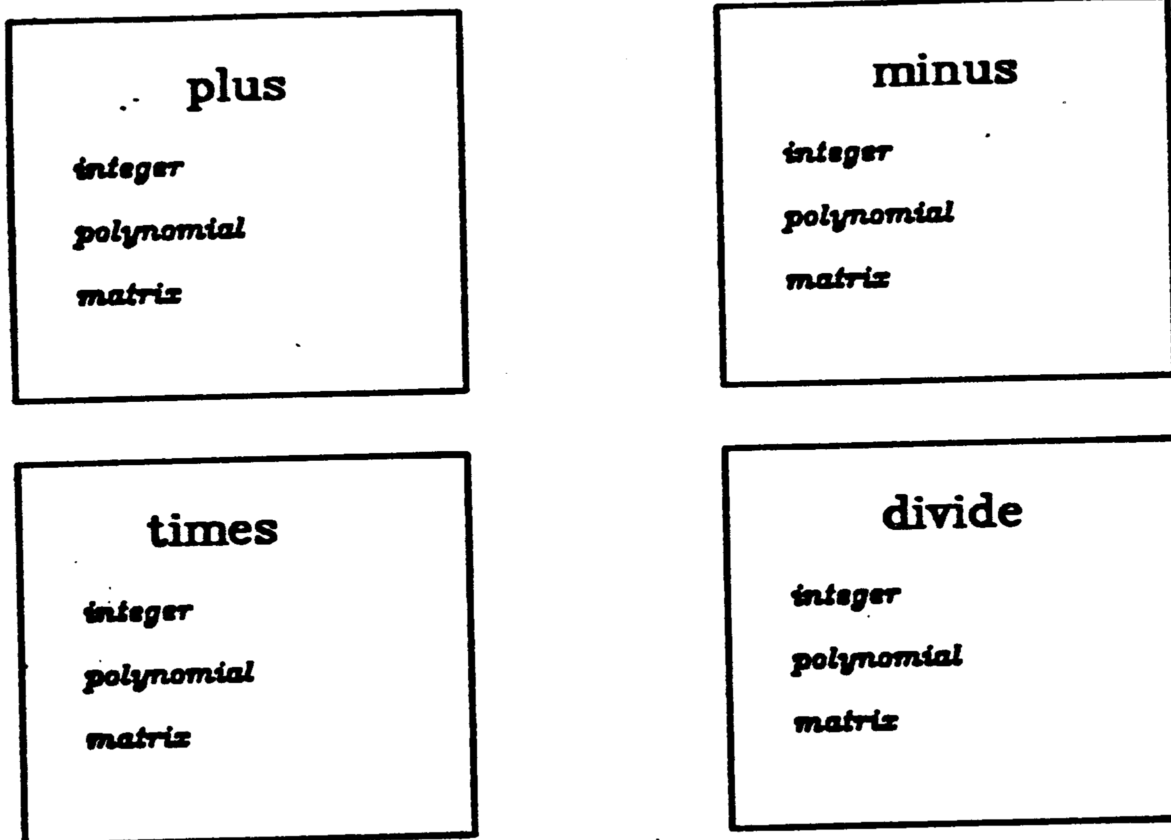
3.2 A Model of an Algebra System

The heart of an algebra system is a set of functions which operate on data objects from a set of types. The system may be organized in one of two ways, either emphasizing the operations or emphasizing the types. Figure 3.1 shows how an *operation-centered* algebra system is constructed. There are single functions for each of the operations. Each function contains code for handling each of the data types relevant to the operation. The code looks something like this:

```
plus(a,b) ::=
    if type(a) = Integer and type(b) === Integer
       then Integer_add(a,b)
    else if type(a) = Polynomial and type(b) = Polynomial
        then ...
    else if type(a) = Matrix  and type(b) = Matrix
        then ...
    ...
    else error("can't add these values ",a,b)
```

We call this "dispatch on type" programming, and this is how most symbolic algebra systems are organized. There are some easily recognized problems.

(1) knowledge of the representation of each data type is spread throughout the system. This makes it difficult to modify the representation because any program in the system may depend on it. This is alleviated to some extent by centrally defining types. For example, in Macsyma's rational function package the representation of 'coefficient' is embodied in *pcoefp* - a predicate used by polynomial-arithmetic programs to test an object for



Operation-Centered

Figure 3.1

membership in the domain "coefficient of a polynomial."

(2) adding a new data type requires modifying existing code, perhaps in numerous places. This is a delicate operation, especially if done at run-time. Consider the effect of the introduction of a date type 'interval' in the *plus* program above.

(3) it is time consuming to be continuously checking and dispatching from data types. For example, in Macsyma, arithmetic on polynomials over GF(p) is done by checking, each time a coefficient operation is executed, to see if the global variable *modulus* is non-zero.

An alternative method of organizing the system and the one we prefer is shown in figure 3.2. We call this method *type-centered* to contrast it with operation-centered. It is termed *object-oriented* in the Smalltalk and Flavors vernacular. The system is viewed a collection of types. The descriptions of operations for each type are associated with the type. This method solves some of the problems with the operation-centered approach in these ways:

(1) the knowledge of the representation of a data type is limited to the module defining the data type and operations on it.

(2) adding a new data type does not explicitly affect existing code.

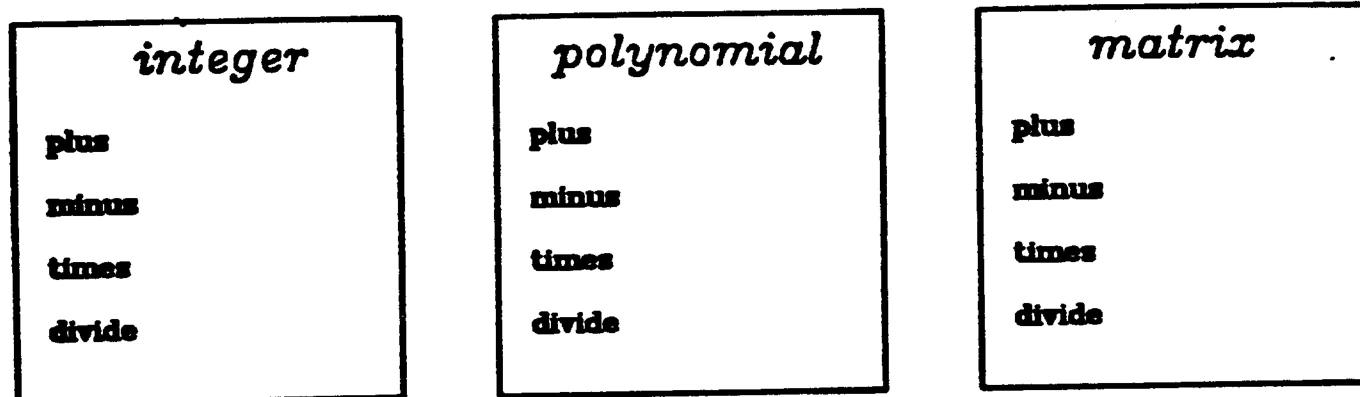
(3) each operation operates on a specific data type so that most procedures will not check the types of arguments at run-time. In some type-centered languages (e.g. Smalltalk) the selection of which function to call is always done at run-time by examining types. The time taken for the selection may be indistinguishable from that in the more traditional operation-oriented case, however there is an important difference: it is done automatically by the system (for good or ill) instead of being programmed in each function by the user. In NEWSPEAK we will see that most function selection can be done at compile-time.

A problem with the type-centered approach would seem to be that common functions may have to be written many times - once for each data type for which they are applicable. For example, almost the same Euclidean GCD algorithm would have to be written for integers and for polynomials over a field. It is important, therefore, to establish relations between the type modules to enable them to share common algorithms. The type-centered language NEWSPEAK, which we describe next, is one which permits the programmer to describe such relations in a form especially suitable for structuring of mathematical algorithms.

3.3 The Newspeak Language

We describe NEWSPEAK in a tutorial manner because we feel that it is important that the reader understand the motivation behind each of its features. Typically, we show an example of a program in a typical algebra system that can't be expressed given what has been said about NEWSPEAK to this point. Then we introduce a new NEWSPEAK feature which will enable us to write the program.

The reader should acquire the ability to *read* programs in NEWSPEAK but we will not provide complete rules for *writing* NEWSPEAK programs. Our goal is to highlight the features of NEWSPEAK that are especially appropriate for symbolic algebra systems. We illustrate these by simple examples. We begin the description of NEWSPEAK with the most basic concept, the *object*.



Type-Centered

Figure 3.2

3.3.1 Object

In the definition of object which follows, we depart from talking in the abstract because most readers will find an implementation phrase more evocative. We use the phrase *pointer to an object.* A pointer is a fixed-sized value that refers to a unique object in a specific execution of a program. We describe the form of pointers in section 5.1.

Definition: **object**

An **object** is a data value that a NEWSPEAK program can manipulate. The value is represented by a block of storage partitioned into *lex* (for lexical) fields and primitive fields. Lex fields contain pointers to objects. Each lex field has a name and can only point to objects of a certain type. Primitive fields contain values (called primitive values) which are (usually) not pointers to objects. Integers or floating-point values in the host machine's format are examples of primitive values.

3.3.2 Type

Associated with each object is another object called its *type*. An object which can serve as the type of another object is called a *type-object.* Informally, we define a type-object as an object that describes all that is common among a collection of objects. Type-objects are created in one of three ways:

- At compile time with the *deftype* declarative function. We will provide examples of this kind of 'statement' shortly.
- At run-time through a function call to the system type-generation module. This would be used to create types for the specific mathematical objects that the algebra-system user wishes to manipulate.
- At compile-time when a reference to a specific member of a parameterized type is made. Parameterized types will be discussed in section 3.3.8.

If a type-object has a name attached to it, then it is referenced in a NEWSPEAK program by surrounding it with angle brackets, as in <integer>. The brackets should be read as "the type". An integer value has an associated type-object <integer>. The type-object <integer> has <type> as its associated type-object. The type-object <type> has itself as its associated type-object. In the following discussion we will favor the use of the term *type* to the term *type-object.* In NEWSPEAK the terms are synonymous. *Type-object* will be used in those cases where we wish to emphasize that the type is itself an object.

When we use the phrase "objects of type <X>", we are describing the collection of objects whose associated type-object is <X>.

The usual way to create types is by using *deftype* function. For example, the following statement declares the type "integers modulo 5":

```
(deftype zmod5
   lex: ((val <integer>)))
```

The syntax is superficially similar to that of Lisp. Expressions are surrounded by parentheses with the main operator being the first element of the list. This *deftype* expression declares that there is a type named zmod5, and that objects of <zmod5> contain one lex field, named 'val', which points to an integer value. Nowhere is it stated that there will be only five distinct values

stored in the val field. It is up to the programs that create and manipulate <zmod5> objects to insure that the value of val inside a <zmod5> object is meaningful.

In this example and in those that follow we will ignore some details, including: when *deftype's* are permitted, how type redefinition is handled, and which pieces of code are permitted to use a type's definition. These issues are important but are independent of the ideas presented in this thesis.

Assuming that we have a type <real> of real numbers, we could define complex numbers in this way:

```
(deftype complex
   lex: ((real-part <real>)
         (imag-part <real>)))
```

Objects of <complex> thus would always have two fields, both pointers to <real> objects.

Definition: **Type**

A **type** (or **type-object**) is an object that is intended to describe all that is common for computational purposes about a collection of objects. Part of the description is explicit in the type-object, such as the names, types and locations of the lex fields in the objects of this type. The other part of the object description is an implicit collection of *properties* of the type. Properties usually pertain to semantics of functions on objects of the described type. For example, a property of <Stack> is that if S is a stack then $X = pop(push(X,S))$. A property of <Field> is that multiplication is commutative.

The properties of a type are difficult to make explicit because even for the simplest types they are often infinite in number and are not generated uniformly for all imaginable types. For <Stack>, for example, there are these properties: for each positive integer n, if you push x[1], x[2], ..., x[n] onto a stack and pop them off, the values will be x[n], x[n-1], ..., x[1]. We see two methods to explicitly represent the properties of a type: (1) declare those properties to exist which might prove useful to programs operating on objects of the type, ignoring the rest of the properties or (2) declare a complete set of axioms governing the type and use a theorem prover to deduce any needed properties. Solution (1) is used in the languages Andante and Newspad (described in section 6.3). Solution (2) would be ineffective in general, slow in practice, and counterproductive because it would be inconvenient for programmers to define all the axioms at each introduction of a new data type.

We find neither solution to be sufficiently descriptive and practical. In NEWSPEAK, the properties of types are not explicitly declared in the program. We have found a simpler, and adequate technique in which the programmer is able to declare how the properties of related types are themselves related, using a (mathematical) relation called *restricts*. The relation is based on the implicit part of the types (their properties) as well as their explicit parts (the details of their lex and primitive fields). NEWSPEAK can check that the explicit parts match, but it is the programmer's responsibility, based on his knowledge of the properties of the types, to insure that the implicit parts match. The restricts relation is described in section 3.3.4.

Other languages use the concept of a type-object which describes a collection of objects, although different terms are often used. In Smalltalk, the terms are *class* (for type-object) and *instance* (for object) The respective terms in Flavors are *flavor* and *instance*. Neither of these systems represents properties of types explicitly.

While every object has a type, it is convenient as a unifying principle to deal with types for which no objects can exist.

Definition: **domain-type**

A **domain-type** is a type for which objects can exist. Other types are called **non-domain-types**.

Non-domain-types cannot have associated objects because they are *representationless* or *parameterized* or both. A representationless type has no lex or primative field descriptions. For example,

```
(deftype object)
```

We will see that in an algebra system, types such as <Field> and <Ring> are representationless.

Parameterized types will be discussed in section 3.3.8.

3.3.3 Procedures

Programs in NEWSPEAK are called *procedures* although we may use the term *function* when we wish to emphasis that a procedure returns a value. Procedures resemble both functions and subroutines because they can return values *and* can have side effects. A procedure takes zero or more arguments and returns zero or more values, the number and types of which are fixed when the procedure is defined. A sample procedure definition for the type <zmod5> declared above is:

```
(defproc plus ((x <zmod5>) (y <zmod5>)) <zmod5>
   (new <zmod5> val (mod (plus x:val y:val) 5)))
```

The first line declares that this is a definition of the procedure *plus* which takes two <zmod5> objects (called x and y within the procedure body) and returns a <zmod5> object. The syntax *x:val* should be read "the value of the val field of the object stored in variable x", or alternatively "x's val". The first operation performed in the body is to add *x:val* and *y:val* using *plus*. Since both of these values are <integer> objects, the *plus* procedure invoked will be the one defined over <integers>. It is *not* a recursive call to the procedure named *plus* that we are defining. This call to *plus* is an example of a generic function call because the types of the operands select which *plus* is invoked. The value returned by the *plus* procedure over <integer>'s is then reduced modulo 5 and stored in the val field of a newly created <zmod5> object. The new object is returned as the value of *plus*. A *new* statement is given first a type-object (or expression returning a type-object), then a sequence of lex field name, expression, lex field name, expression and so on. If there is only one field in the object, the field name can be omitted.

The *new* statement returns a <zmod5> object from *plus* which is exactly the type that *plus* was declared to return on the first line of the defproc. It isn't necessary that the actual return type and the declared return type match exactly. In section 3.3.4 we will define a relation *restricts* between type-objects and then we can state the the type of the actual return value must be equal or *restrict* the declared return type.

The semantics of the NEWSPEAK language, as they have been described so far, are much the same as in several other languages with the possible exception of generic function calls. It could pass for Pascal with Lisp syntax. In the following sections we will describe the features of NEWSPEAK that make it unique.

3.3.4 Relations Between Types

In this section we focus on three (mathematical) relations between types: *satisfies*, *extends* and *restricts*. We define the relations, justify their definitions, and finally look at other languages to see how similar relations are used.

Definition: **Satisfies**

<A> **satisfies** <B> if the properties of <A> are a superset of the properties of <B>.

The term *satisfies* is an abbreviation for "satisfies the type property requirements of". Figure 3.3 shows some of the ways that properties of types can be related. <object> is defined to have no properties and thus is *satisfied* by all types. <B> satisfies both <A> and <object>. <C> satisfies only <object>; there is no relation between <C> and either <A> or <B>. For example, B = ordered set, A = partially ordered set, and C = finite field. An ordered set has all of the properties of a partially ordered set plus one more: all objects are comparable. The satisfies relation is transitive. The system obeys the "closed world" hypothesis: Unstated properties are assumed to be false.

Definition: **Extends**

<M> **extends** <L> if objects of <M> can be constructed from objects of <L> by adding zero or more lex or primitive fields. To be specific, if the nth value in a <L> object is a lex or primitive field named X of type <T>, then the nth value of a type <M> object is also a lex or primitive field named X of type <T>.

Figure 3.4 shows the forms of objects of types <K>, <L>, and <M>. <object> has no lex or primitive fields which means that there can never be an object whose type is <object>. This does not mean that a type cannot extend <object>; in fact all types extend <object>. In the figure, <M> extends <L> and <object>, <L> extends <object>, and <K> extends <object>. The extends relation is transitive.

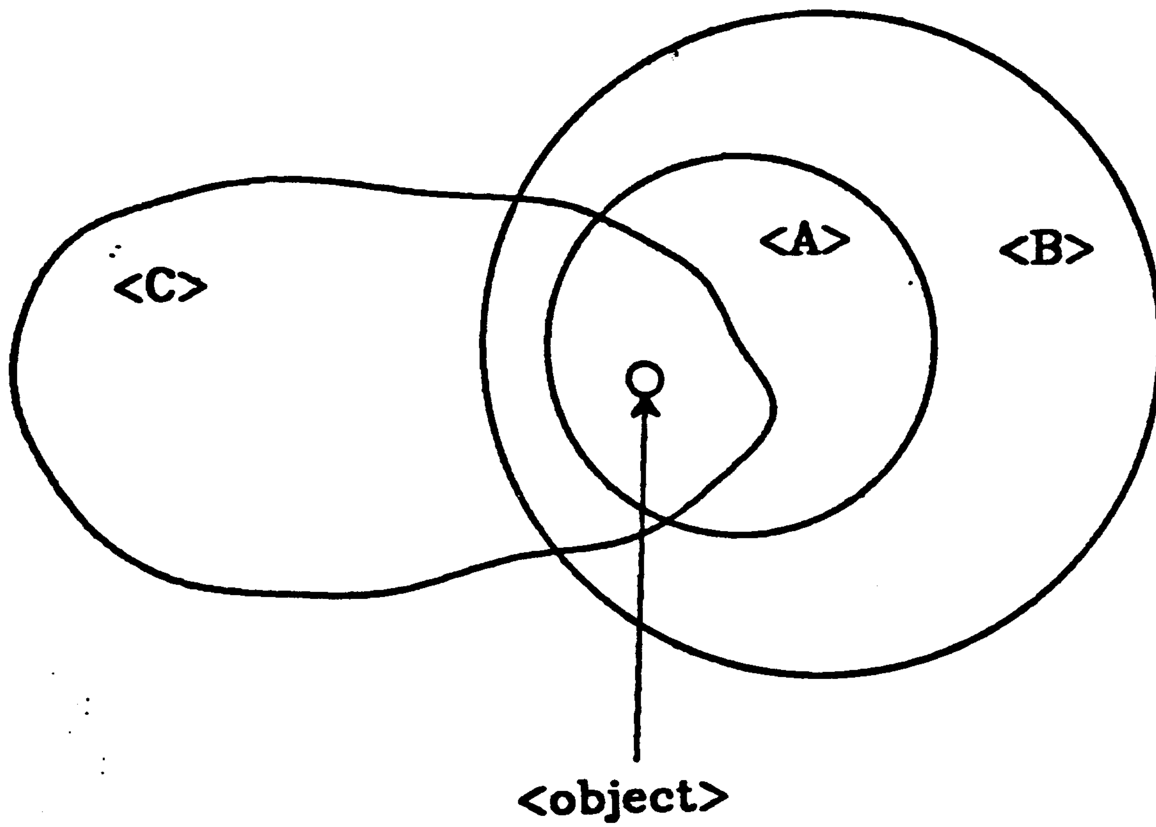
We do not specify what we allow as a property of a type. The reader will soon see the practical reasons for including properties of types. The properties we associate with types are those that provide additional mathematical and programming structure. A general rule is that properties should be independent of representation. For example, we can imagine a type <Field> with this property: There is a function *inv* such that for any non-zero object X of <Field>, *inv(X) * X* is the unit element of the <Field>. This property is independent of the representation of <Field>. (Note that we do not say "For any non-zero X there is a Y such that *X*Y=1*" as that gives us no hint as to how to find Y).

The *extends* relation (dealing with representation) is often independent of the *satisfies* relation (dealing with properties). Examples of this that we seen so far are the *push* and *pop* properties of <Stack> and the *inv** property of <Field>. When we consider those pairs of types for which extends and satisfies are both true, we define the following relation which is very important in NEWSPEAK.

Definition: **Restricts**

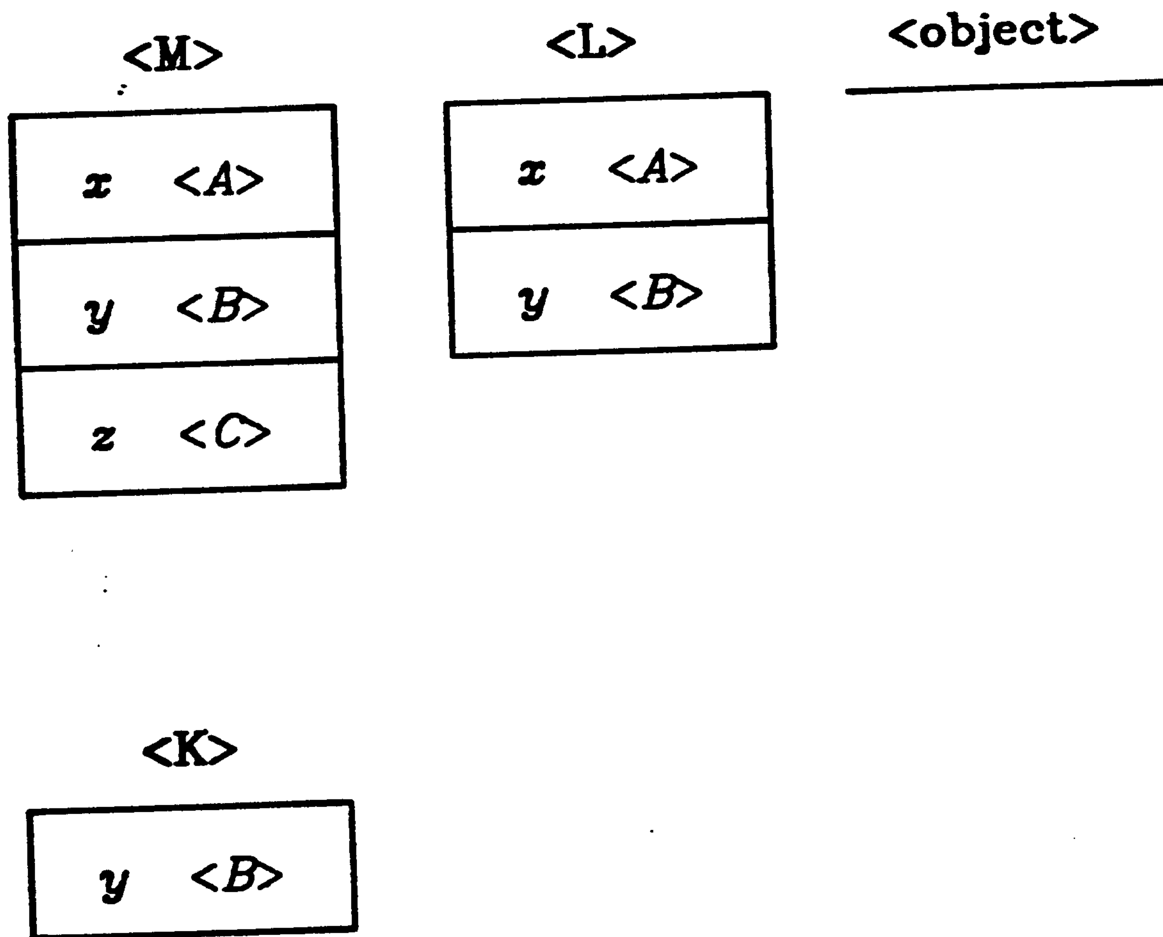
<A> **restricts** <B> if (1) <A> and <B> are not the same type, (2) <A> satisfies <B>, and (3) <A> extends <B>.

NEWSPEAK can't compute the *restricts* relation automatically because it doesn't have enough information to compute the *satisfies* relation. Therefore when the programmer defines a type, he must indicate which types it *restricts* based on his knowledge of which types it *satisfies*. The *extends* relation is then used in one of two ways. If the programmer defines the representation of the type's objects, then NEWSPEAK will verify that the *extends* relation is true between the newly



Satisfies relation

Figure 3.3



Extends relation

Figure 3.4

defined type and those types which it has been declared to *restrict*. If the programmer does not define a representation, then Newspeak will compute the smallest representation which will make the *extends* relation true (or else signal an error if such a task is impossible due to differences in the representations of the restricted types). For example, if a new type <X> were declared to restrict <L> of figure 3.4 and no representation for it were given, then a representation identical to <L>'s would be assigned to <X>. If <X> were declared to restrict both <L> and <K> then an error would be signaled because it is impossible to generate an object that extends both <L> and <K> objects.

Before showing the importance of the restricts relation, we will first describe the graph of the restricts relation, which is named the *type-hierarchy*.

3.3.5 Type-hierarchy

The restricts relation is transitive. Newspeak automatically computes the transitive closure of the restricts declarations provided by the user. For implementation reasons, the restricts relation is irreflexive and antisymmetric. This is not very limiting to the programmer because if there were a case where <A> restricted <B> and <B> restricted <A>, then <A> and <B> would be isomorphic: two names for a type with the same form and properties.

The restricts relation forms a directed acyclic graph called the *type-hierarchy*. The type <object> is at the root of the type-hierarchy. Every type except <object> restricts <object> and <object> restricts no type.

Many of the types close to <object> in the type-hierarchy will not have lex or primitive fields defined (e.g. they are representationless types). Generally, they are placeholders in the type-hierarchy representing useful collections of type properties. Types which restrict these representationless types claim to satisfy the properties represented by the representationless types but are more specific. In a symbolic algebra system, types such as <Ring> and <Field> would be representationless types. Restrictions which provide some specificity are needed before we can compute. Note that once a type is declared to contain a lex field, any type which restricts that type must have an identical lex field declared for it. Thus as one goes farther away from the root of the type-hierarchy, the sizes of the objects denoted by the types never decreases.

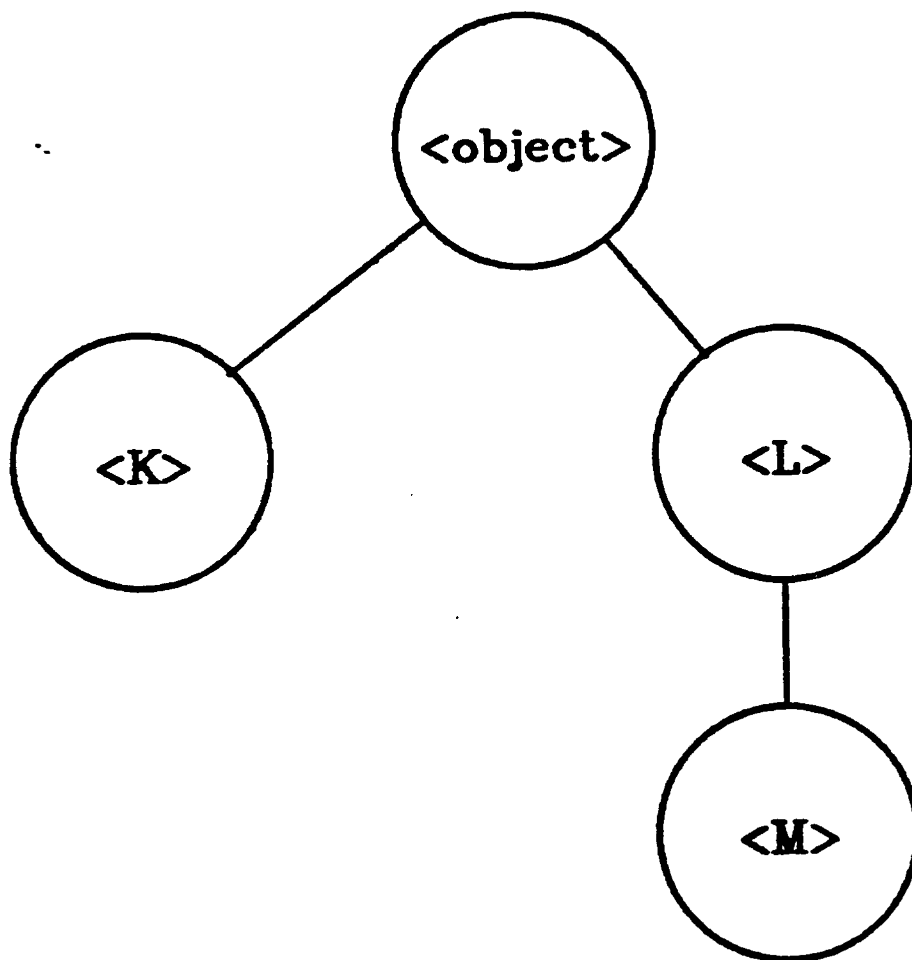
The following example shows how to define the types whose forms are shown in figure 3.4 to create the type hierarchy shown in figure 3.5.

```
(deftype L
  lex: ((x <A>)
        (y <B>))
  restricts: (( <object> )))

(deftype M
  lex: ((x <A>)
        (y <B>)
        (z <C>))
  restricts: (( <L> )))

(deftype K
  lex: ((y <B>))
```

The *restricts* clause in the deftype for <L> is optional; if a *restricts* clause is missing, the type is assumed to restrict <object>. This default is used in the deftype for <K>.



type-hierarchy

Figure 3.5

3.3.6 The Importance of Restricts

In this section, we examine the role of *restricts'* component relations: satisfies and extends. Section 3.3.6.1 will describe how the satisfies relation permits code sharing through the use of polymorphic functions. Section 3.3.6.2 will show how extends allows this to be done efficiently.

3.3.6.1 Satisfies

In an algebra system there are often many data types with similar properties. For example there are many different polynomial data types which differ by the data type of the polynomial's coefficients. Many algorithms, such as polynomial addition and multiplication, can be written independently of the coefficient's data type. Through the use of polymorphic functions, the programmer can write these algorithms once and use them for a collection of data types.

The satisfies component of the restricts relation is important because it makes code sharing officially 'correct.' If we know that <A> *satisfies* <B>, then whatever properties of <B> objects are required for a function to work correctly will be true of <A> objects.

A function which takes a type <B> argument is a *polymorphic* function since the function should be valid for any argument whose type satisfies <B>. If the argument's type also extends <B> and hence restricts <B>, the function is not only valid, but the implementation can mirror this. As an example of how the use of restricts can reduce duplicated code, we will continue the <zmod5> example (page 14). Suppose we define <zmod7>, <zmod13> and <zmod17> in the same way that we defined <zmod5>. The resulting type-hierarchy is shown in figure 3.6. We have already written the procedure *plus* for <zmod5> (page 16) and now we must duplicate that procedure for each of the other <zmodX> types (changing just the second argument to the *mod* function). We can eliminate this needless replication of code by organizing the types a bit differently, as shown in figure 3.7. We now write the *plus* procedure once for the type <Zmodn> and then use it for objects of types <zmod5>, <zmod7>, <zmod13>, and <zmod17>. In order to organize the types in such a way we need parameterized types, which we describe in section 3.3.8.

3.3.6.2 Extends

This section addresses implementation issues. In order for <A> to extend <B>, there are two requirements: <A> objects must have (at least) the same fields as <B> objects, and those fields must be in the same location in both types of objects. If we remove the second requirement, we have the *weak-extends* relation. Weak-extends represents the minimum requirement on <A> to permit <A> objects to be used where <B> objects are expected. NEWSPEAK uses extends rather then weak-extends because the guarantee of identical field order permits rapid access to the lex fields without a run-time type check, as the following example shows.

We define complex numbers in the right half plane as a restriction of <complex> (defined on page 14).

```
(deftype complexRightHP
    lex: ((imag-part <real>)
          (real-part <real>))
    restricts: (( <complex> )))
```

(Actually the type <complexRightHP> does not *satisfy* <complex> (for example, multiplication isn't closed over <complexRightHP>), so this isn't a valid restriction. However, we assume closure is irrelevant in our system, thus we pretend <complexRightHP> satisfies <complex>.)

Now consider the extends relation between <complexRightHP> and <complex>. Notice that the order of real-part and imag-part are switched in the <complexRightHP> object when compared to the <complex> object. This is an implementation inconvenience. This deftype would only be valid if we were allowing a weak-extends relation. Let us examine the conse-

Figure 3.6

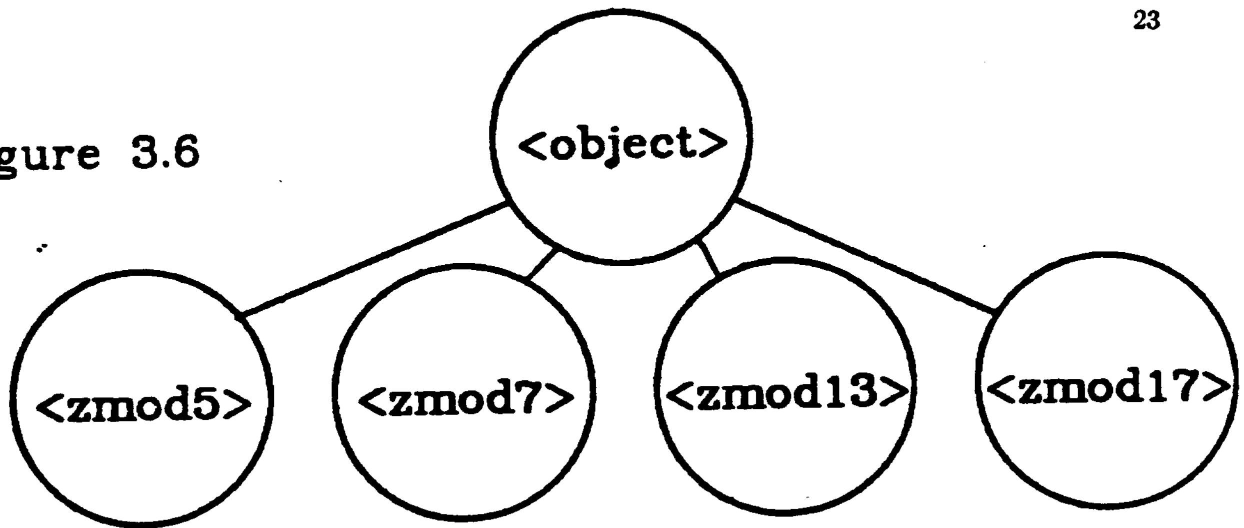
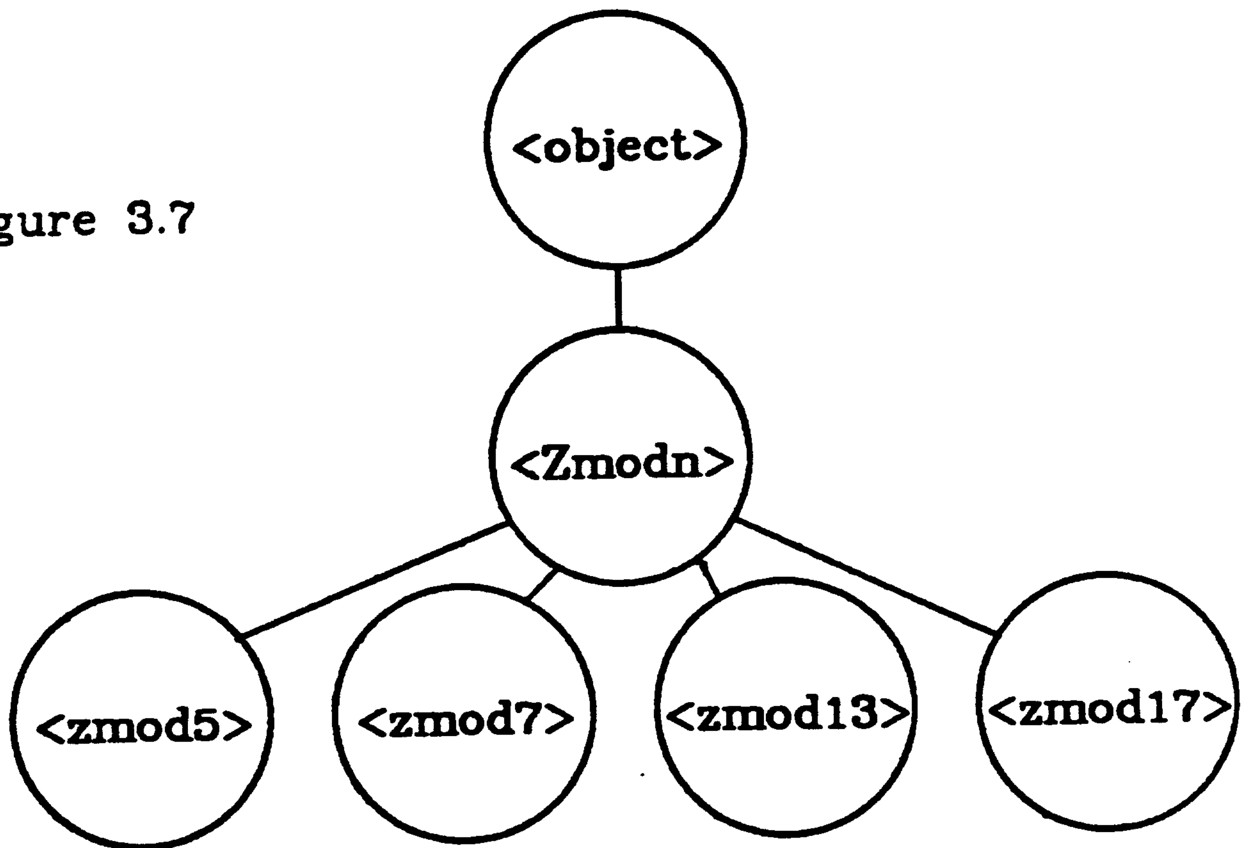


Figure 3.7



quences: The Newspeak compiler might have to compile this procedure:

```
(defproc RealPart ((x <complex>)) <real>
  x:real-part)
```

Naturally, we want RealPart to work for <complex> objects and for objects that restrict <complex>, such as <complexRightHP>. Since the real-part field is first in a <complex> object and second in a <complexRightHP> object, we must know the type of x to select the proper field. The type of x can only be determined by a run-time check - a large overhead for the simple operation of extracting a field from an object. It is to eliminate this run-time determination of a field's location that in Newspeak, we require the extends relation rather than weak-extends.

3.3.7 Restricts in other languages

Other languages have relations analogous to the restricts relation in Newspeak. The *subrange* relation found in Pascal and Ada corresponds to the extends relation of Newspeak. Smalltalk, Flavors and Glisp have relations similar to restricts. To some degree each of these languages permits polymorphic functions. Of particular concern is how a function accesses the fields of its arguments and how it insures that such accesses work even when the argument has a type which *restricts* the declared type.

3.3.7.1 Pascal and Ada

The subrange relation in Pascal and Ada declares that a given type will contain a subset of the objects of another type. It implies nothing about the properties of the types, and thus does not preserve operations. A simple example shows that the programmer should not equate the subrange relation with Newspeak's restricts relation. Suppose we want to define the type representing the positive integers. In languages like Pascal and Ada, this is done by declaring a subrange of the type integer (e.g. 1..maxint). Mimicking this, in Newspeak one might declare <positive-integer> to restrict <integer>. This is generally incorrect, at least if we wish mathematical consistency and have used the fact that <integer> is a Euclidean domain. The type <positive-integer> is merely a semigroup. A procedure written for a Euclidean domain may attempt to use the procedure which computes the additive inverse of an element, but this would fail for the <positive-integer> type. (In Newspeak <positive-integer> and <integer> would be unrelated). Therefore the Newspeak programmer must be concerned with properties more fundamentally than is the case with the subrange relation of other languages.

3.3.7.2 Smalltalk

In Smalltalk, where types are called *classes*, the analogous relation to Newspeak's restricts is named *subclass*. Each class in Smalltalk is the subclass of only one class (this is called *single inheritance*). This is a severe restriction for an algebra system, because there are often cases in which a type has the properties of more than one 'supertype'. For example, the type <Ring> is a <Monoid> for certain operations and <AbelianGroup> for others. In Newspeak, types are not limited to restricting a single type. As in Newspeak, the form of a Smalltalk object is the same as the form of an object of its superclass plus zero or more added fields. This means that a Smalltalk program that expects an object of a given type (or some restriction) can access the lex fields by using offsets from the beginning of the objects that are known at compile time.

3.3.7.3 Flavors

The Flavors language, which is built upon Lisp, is like Newspeak (and unlike Smalltalk) in that a type (called a *flavor*) can restrict more than one type (called *multiple inheritance*). When a language supports multiple inheritance, it must deal with the problem of a type restricting two

types neither of which extends the other type. This is a sticky implementation issue which appears to be avoidable in our examination of algebra system. In NEWSPEAK, if <A> restricts <B> and <C>, then it must extend both <B> and <C>; this implies that <B> extends <C> or <C> extends <B>. In Flavors, the more general weak-extends relation (page 23) is all that is required for a restriction to be valid. The Flavors system creates an object whose lex fields (or *instance variables* as they are known in Flavors) are the union of the lex fields of all the restricted types.

The implementation problems of Flavors caused by weak-extends originate in the fact that there no guaranteed fast way to find the location of a particular instance variable in an object just by knowing what type it restricts. This would appear to rule out open coding of accesses to the lex fields of a flavor object because the compiler can't predict at compile time the location of a field (as we saw on page 23). However, because Flavors is embedded in Lisp and extra work is done at the beginning and end of a call to a flavor function, a technique has been developed to make it possible to quickly access the values of the lex fields in a flavor object. In Lisp there is a global value cell for symbol objects, which can be rapidly accessed. When a function is called on a flavor object, the values of the lex fields of the flavor object are placed in the global value cells of the symbols associated with the lex fields (after saving away the old values). A function can then access the values of the lex fields rapidly. After the flavor function is called, the original values of the global variable have to be restored. The cost of making all of the lex fields accessible on every function call is high on a conventional machine, although it can be greatly reduced on machines designed with Flavor implementation in mind.

Recently, a new strategy was implemented on MIT Lisp Machines [Weinreb81] for compiled accesses to flavor instance variables. The compiler determines which variables in a function are flavor instance variables and replaces references to them with two array references: the first to a dynamically created table which indicates where that instance variable is found in the flavor object, and the second into the flavor object to access the instance variable [Stallman83]. Thus if availability of weak-extends becomes an important issue, there are techniques of modest complexity to provide this.

3.3.7.4 Glisp

Glisp permits the Lisp programmer to formally describe the Lisp data structures he uses for data. Once his data structures are described in Glisp, the programmer may access and modify elements of a data object in a representation-independent way. Glisp does not force data objects to have a certain structure, which makes it easy for Glisp code and Lisp code to be intermixed.

Glisp also has a restricts-like relation and facilities for object-oriented programming. As we explain next, the freedom Glisp provides the programmer for describing arbitrary Lisp data structures causes its restricts relation to be too weak.

Glisp permits multiple inheritance but it make no attempt to deal with the problem of restricting types with different forms. This makes inheritance a weaker organizational principle. Because Glisp does not decide what an object's representation is, it is easy for a programmer to define a type whose objects are completely different from objects of the restricted type. Glisp does not use the Flavors solution of storing lex values in global variables; instead it converts an access to a lex field of an object into a call to a standard Lisp data structure accessing function (*cadr, get, vref,* etc). A result of this design is that code for objects of a given type, <A>, will work for objects of types which restrict <A> only by deliberate data representation choice. This defeats the purpose of restriction as the principal abstraction mechanism as NEWSPEAK uses it, replacing it with much weaker data types.

3.3.8 Parameterized Types

In this section we introduce one of the non-domain types mentioned earlier: the *parameterized type*, a type which represents a collection of related types.

A parameterized type is declared as a type whose definition depends on one or more formal parameters. For example, the <Zmodn> type mentioned above depends on an <integer> value for the modulus. This type would be written:

```
(deftype Zmodn
    params: ((n <integer>))
    lex: ((val <integer>)))
```

A parameterized type is the template for a collection of types and cannot itself be the type of any object. For example, there can never be an object whose type is <Zmodn> but there can be an object of type <zmod5> defined as follows:

```
(deftype zmod5
        restricts: ((<Zmodn> (n 5))))
```

There are two important features of this deftype. Because no lex field definition is given for <zmod5>, it inherits the lex field definition from the type it restricts, <Zmodn>. The restricts clause is read "<zmod5> restricts <Zmodn> and sets the n parameter of <Zmodn> to the value 5."

In order to distinguish parameterized types from non-parameterized types, we use the convention that the names of parameterized types begin with a capital letter (except for single character names such as <A> which may be name non-parameterized types).

Smalltalk, Flavors and Glisp do not permit the user to define parameterized types and thus lack this organizational technique.

Before we can explain how to write procedures over parameterized types we will have to examine how the presence of parameterized types affects the type-hierarchy. This is the topic of the following section.

3.3.9 Views of Types

Although an object has only one type, it may be used in any function where the type of object required is one which its type restricts. When object X of <A> is used in a situation requiring an object of <B>, we say that object X is *being viewed as* a <B> object. How an object is viewed determines what operations are possible on it, what values can be extracted from the object, and what parameter values are accessible. In our example of <zmod5> restricting <Zmodn> (parameterized by n) and <Zmodn> restricting <object>, a <zmod5> object can be viewed as a <zmod5> object, a <Zmodn> object or an <object> object. Suppose the variable x contains a <zmod5> object. When viewed as a <zmod5> object, we can extract the value of lex field val with the expression *x:val*, but we cannot access the modulus (5 in this case). When viewed as a <Zmodn> object, the expression *x::n* (note the double colon) will access the modulus and the expression *x:val* will still access the val lex field. When viewed as an <object>, neither the lex field nor the parameter are accessible.

Figure 3.8 shows the type-hierarchy and the parameter and lex field accesses that are permitted depending on the declared (or viewed) type of x. It would not make sense to use the expression *x::n* when x is declared to be <zmod5> since n is not a parameter. If all we know about x is that is a <Zmodn> object, then *x::n* must be computed at run-time.

A different syntax (double colons instead of single) is used for denoting type parameters as opposed to lex values because access to this information is a fundamentally different operation. The values of type parameters are stored once within the type-object of an object whereas the lex field

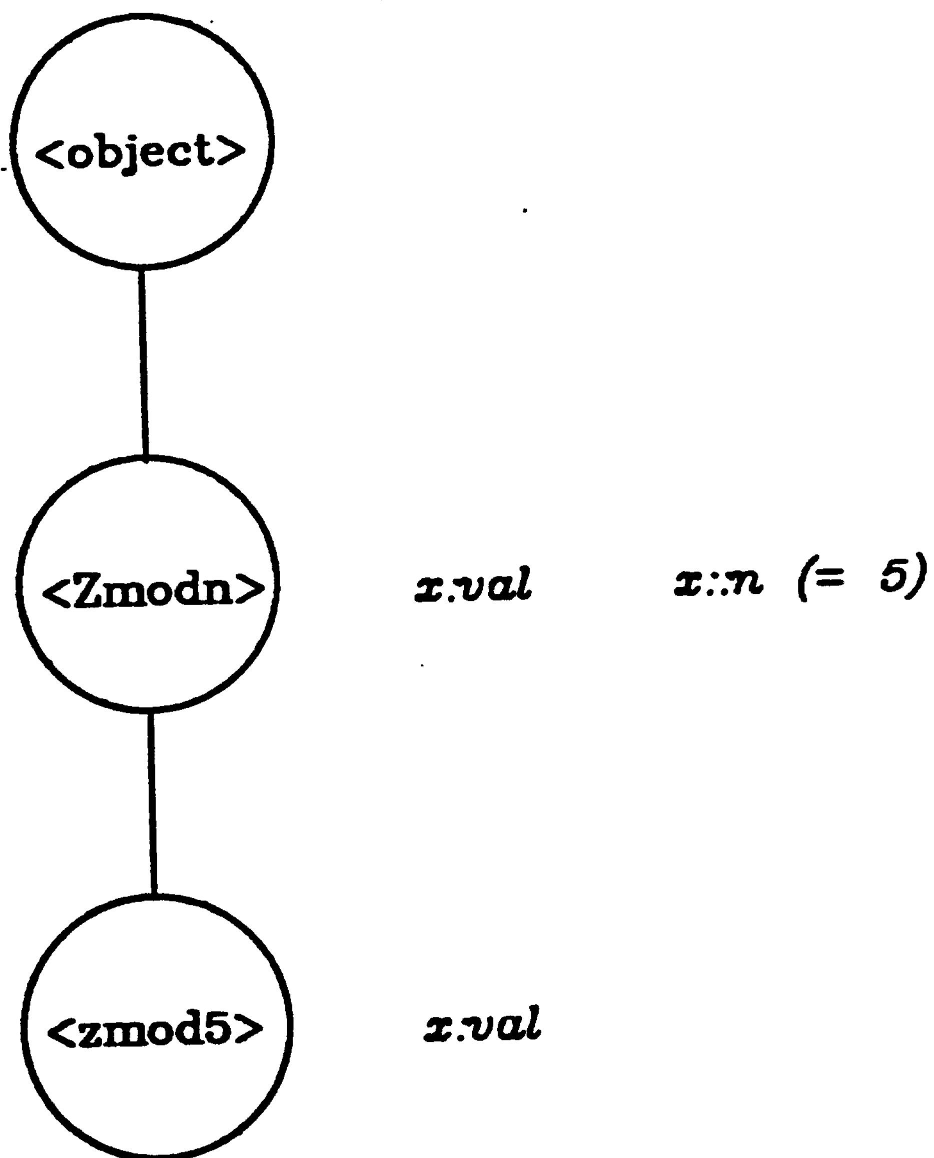


Figure 3.8

values are stored within each object.

The way an object is viewed inside a defproc statement is normally determined by the way the formal parameter or variable bound to the object is declared. It is possible to change the view by assigning the object to a variable declared differently or by using the *widen-view* and *narrow-view* statements in the NEWSPEAK language. It is always possible to widen the view of an object - that is, to declare it to be a type which is closer to the root of the type-hierarchy than its currently declared type. Narrowing the view requires that a type check be done at run-time to verify that the re-declaration is correct.

3.3.10 Inherited Parameters

When a programmer declares that parameterized-type <A> restricts parameterized-type <B> he may choose to make the value of some of <A>'s parameters equivalent to the value of some of <B>'s parameters, in effect delaying the selection of values for some parameters of <B>. For example, given the definition of the <Zmodn> type above, we may define <Zmodp> (integers modulo a prime) and <zmodp7> as

```
(deftype Zmodp
    params: ((p <integer>))
    restricts: ((<Zmodn> (n p)))

(deftype zmodp7
    restricts: ((<Zmodp> (p 7)))
```

The <Zmodp> definition declares <Zmodp> to be parameterized by one integer named p. The <Zmodp> type restricts the <Zmodn> type and the value of <Zmodn>'s n parameter is declared to be the same as <Zmodp>'s p parameter. <Zmodp> inherits the lex field named val from <Zmodn>. The <zmodp7> definition declares <zmodp7> to restrict <Zmodp>, where the value of p is the <integer> 7. <zmodp7> inherits the lex field named val from <Zmodp>.

Figure 3.9 shows the type hierarchy and the permitted parameter and lex field accesses assuming that variable x contains a <zmodp7> object. If x is viewed as a <zmodp7> object then there are no type parameters. If x is viewed as a <Zmodp> object, then *x::p* returns 7. If x is viewed as a <Zmodn> object, then *x::n* returns 7. (It is not required that we use distinct names n and p. We did so only for illustration.) The particular type parameter that is accessed is determined not only by the name but also by the declared type of the object. In section 3.3.18 we will see that the ability to inherit parameters which describe procedures will permit us to describe the common algebraic types.

3.3.11 Procedures

Before we discuss procedures in more detail, we review the basic syntax of NEWSPEAK.

5 - the <integer> whose value is 5.

x - the value of variable x. When this is used within a procedure, x must be a formal parameter or a locally declared variable. When used in the lex definition part of a deftype, x must be a parameter of the type being defined (examples of this will be given later).

<x> - the type named x.

~x - the type of the value of x. The tilde is the "type of" operator, returning the type of the ex-

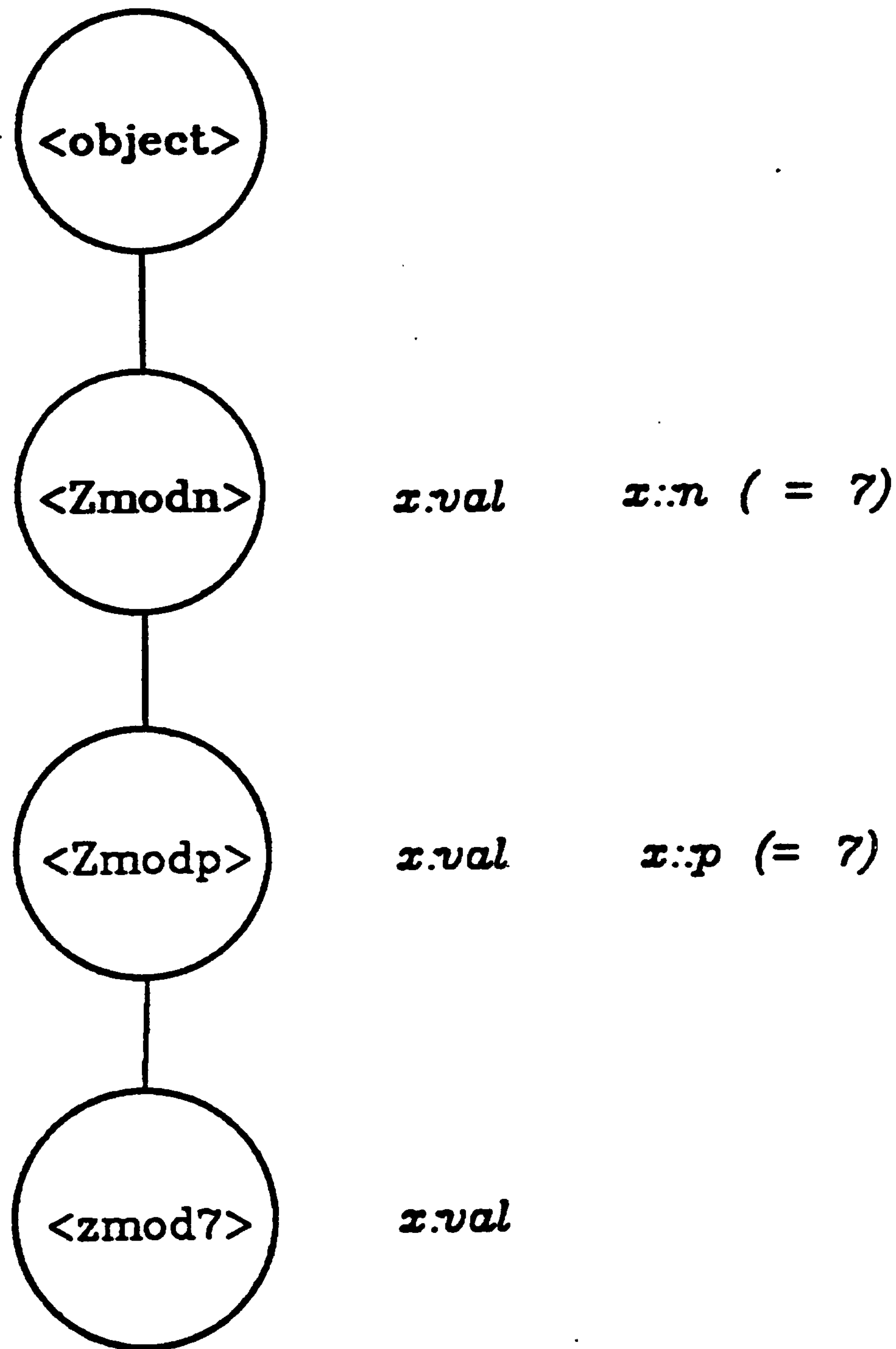


Figure 3.9

pression it precedes. When used within a defproc, this expression is computed at run-time to return the actual type of the object stored in variable x. This form is also used when declaring the types of procedures arguments and return values, where its meaning is similar (more on this later).

x:i - the value of field i of the value of variable x. In order to enforce data abstraction, it is only possible to use this form when the declared type of the variable x is open for inspection.

x::n - the value of parameter n of the type of the value of variable x. The same restrictions apply to this expression as apply to *:x:i.*

The primitive forms (tilde, lex and parameter extraction) shown above may be combined in the obvious ways (with tilde having the weakest binding). For example, *~x:i* returns the type of the lex field i of variable x.

The restricts relation forms a partial ordering on the types. As a result, two types may restrict a common type but be unrelated themselves. This leads to problems if procedures are not declared properly. As an example, we reexamine the problem of writing a single *plus* function for <Zmodn> which can be used for all restricting types (e.g. <zmod5> and <zmod13>). The naive (and incorrect) way would be this:

```
(defproc plus ((x <Zmodn>) (y <Zmodn>)) <Zmodn>
   (new <Zmodn> (mod (plus x:val y:val) x::n)))
```

There are two major problems with this procedure.

- The *new* procedure is called to create a new <Zmodn> object. Since <Zmodn> is a parameterized type and thus a non-domain-type, objects cannot be created of type <Zmodn>. (The NEWSPEAK compiler would flag such a statement as an error).

- When this procedure is called, the value of x could be a <zmod5> object and the value of y could be a <zmod13> object. There may be cases where adding objects of different moduli makes sense, but it is not our intent here.

What we want to declare is a *plus* procedure that will be called only if both arguments are the same type and that type restricts <Zmodn>. We are now faced with the problem of representing this requirement. It is tempting to generalize and say that associated with each formal parameter is an arbitrary predicate. Given a set of actual types, if all the predicates associated with the formal parameters are satisfied at run time, then the procedure can be used. The problem with this solution is that it may not be possible to determine at compile time if an arbitrary predicate would be satisfied at run time. This would lead to run-time evaluation of predicates which makes generic functions costly. NEWSPEAK's solution is to permit a user to replace the type name in the formal parameter list by one of a very select group of expressions. During compilation NEWSPEAK keeps enough information about the types of variables to be able to evaluate these expressions at compile time. The types in the formal parameter list may have one of these forms, where x is the name of another formal parameter:

<name> - the type named *name* or any restriction of it.

~x - the exact same type as formal parameter x.

x::n - the same type as the value of type parameter n of the value of formal parameter x. This is

only valid if the the type of parameter n is <type>, that is the value of parameter n must be a type-object. This will be explained shortly, after we have a chance to motivate its use.

Now we can write the plus function over <Zmodn> correctly:

```
(defproc plus ((x <Zmodn>) (y ~x)) ~x
   (new ~x val (mod (plus x:val y:val) x::n)))
```

Alternatively, we could have declared y to have the type <Zmodn> and x the type *~y*. This procedure also corrects the problem with the *new* function. It will now create an object of the same type as the actual parameter x. The expression *~x* is always sure to return a domain-type since it returns the type of an existing object.

Smalltalk, Flavors and Glisp do not have similar problems declaring the types of their arguments simply because only one argument's type is declared. Also, in these languages the type of the result is not declared.

3.3.12 Lex Descriptions

In all types defined so far, the type of each lex field has been explicitly declared. It is often the case that the type of the lex field of a parameterized type is a function of the values of the parameters. In this section we will introduce two new methods for declaring the type of a lex field. Both of these methods are needed to define the homogeneous linked-list data type, <List>.

A first attempt at writing the <List> data type might be the following:

```
(deftype List
        params: ((t <type>))
        lex: ((first t)
             (rest  <List>)))
```

The type of the 'first' field is not a specific type, but an expression: *t*. When a type is defined which restricts <List> and provides a value for t, this expression will be evaluated to determine the type of the 'first' field in the newly defined type. As for first's type when viewed as a <List> object, Newspeak assumed that it is <object>, which is the least it can assume.

This deftype is not what is intended. We would like <List> to be a (homogeneous) linked list of elements of the same type. However, the type of the 'rest' field is not declared correctly as we can see by defining a specific type of <List>:

```
(deftype list-of-integer
    restricts: ((<List> (t <integer>)))
```

As far as the type of the lex fields are concerned, this deftype is the same as the deftype:

```
(deftype list-of-integer
    lex: ((first <integer>)
         (rest  <List>)))
```

Notice that the rest field is declared to have type <List>, not <list-of-integer>. As a result, the rest field could point to an object of any type which restricts <List>, such as <list-of-real> (if such a type existed). The type <List> that we have created is a heterogeneous linked list, very much like the Lisp list data structure, not restricted to be a list of <integers>. The type

<list-of-integer> merely has as its first element an integer.

The solution we use is the special symbol *_self* in place of the type name in the lex declaration:

```
(deftype List
    params: ((t <type>))
    lex: ((first t)
          (rest _self)))
```

This states that the rest field of an object will always point to an object of the same type as the object. Thus when the <list-of-integer> type is created, the type of the rest field for the <list-of-integer> type is declared to be <list-of-integer>.

3.3.13 Type Parameters

In the <List> example, the parameter t could take on any value and <List> operations (*append, reverse,* etc) would still work. There are instances in an algebra system of types parameterized by other types where the value of a parameter determines which functions are possible. An important example is the polynomial type parameterized by the type of its coefficients. If the coefficients are from a field then it is possible to perform exact division with remainder. If the coefficients are from a unique factorization domain and hence do not have multiplicative inverses, then pseudo division must be used [Knuth81]. If the only way we could define polynomials were as follows:

```
(deftype Poly
    params: ((coefdom <type>))
    lex: ((coefficient coefdom)
          (exponent <integer>)
          (rest _self)))
```

then for many of the functions over polynomials we would have to put in repeated explicit tests for the value of coefdom. The functions (e.g. polynomial division) would then be using the "dispatch on type of coefficient" style (page 11) we wished to avoid. This problem arises because the *properties* of <Poly> are dependent on the value (not just the type) of the coefdom parameter. This clashes with our notion that a type has a single set of properties. In this case we would like to specify in the deftype the *value* of a <type> valued parameter (e.g. polynomials parameterized by a <type> valued parameter whose value is <Field> or some restriction). All type-objects have type <type> and we cannot give a subset of them a different type. For this reason, parameters of type <type> (or simply *type parameters*) are a special case in NEWSPEAK. It is possible when defining a type to specify that a parameter is a type parameter *and* that its value restricts a certain type-object. An example of such a declaration:

```
(deftype Poly-field
    params: ((coefdom <= <Field>))
    lex: ((coefficient coefdom)
          (exponent <integer>)
          (rest _self))
    restricts: (( <ED> )))          ; ED is Euclidean Domain
```

In past examples the syntax of the *params* description in a deftype was a symbol followed by a type. In this case, between the symbol and the type is a less-than-or-equal symbol (<=), which is indicative of the fact that the value of coefdom is either <Field> or some restriction of

<Field>.

The parameter declaration *(t <type>)* is equivalent to the declaration *(t <= <object>)*. The latter form is preferred, since it emphasizes the nature of the parameterized type.

There are two major ramifications of declaring a less-than-or-equal parameter like the one for <Poly-field>:

- Whenever a type is declared to restrict <Poly-field>, NEWSPEAK will check that the parameter supplied in the restriction is really a restriction of <Field>. This check is done during the generation of the new type-object.

- When, in a procedure, a variable (say x) is declared to have type <Poly-field>, the type of the expression *x:coefficient* will then be <Field>, which is the most restrictive type that can be assumed about the value of the coefficient at compile-time.

The existence of type parameters has an effect on *defprocs* too. Let us return to the <List> data type, this time defining it in the preferred way:

```
(deftype List
    params: ((t <= <object>))
    lex: ((first t)
          (rest _self)))
```

Suppose we wish to write a procedure *first(x)* that, given an object x whose type restricts <List>, returns the 'first' field of x. The type of the 'first' field of x will depend on the parameter t of x when x is viewed as a <List> object (i.e. the type will be x::t). The same syntax that we use to denote a parameter can be used to denote the result value of a defproc (recall that this is the third form mentioned in section 3.3.11).

```
(defproc first ((x <List>)) x::t
    x:first)
```

first is a procedure which takes a <List> object and returns a value whose type is the value of the t parameter of the actual argument x.

We have already seen how to write a procedure definition that returns a value of the same type as one of its actual parameters, but here is another example:

```
(defproc rest ((x <List>)) ~x
    x:rest)
```

As a final example, this is the definition of procedure *cons* which takes an object and a <List> of that type of object and returns a new <List> object:

```
(defproc cons ((x y::t) (y ~List)) ~y
    (new ~y first x rest y))
```

In summary, by means of type parameters, we can propagate information about types at compile-time which allow us to avoid type checking at inconvenient and frequent points at run-time.

3.3.14 Anonymous Restricted Types

All of the types we have discussed so far were created with deftype. Each type was given a unique name when it was defined and which is used to reference the type. When referring to a particular instance of a parameterized type it is inconvenient for the program to create a name and declare a type. For example, after we define <Zmodn> we can create <zmod6>, <zmod7> and so on with almost identical declarations. Another programmer might name these types <zmodn6> and <zmodn7>, and would end up with a new set of distinct types. To avoid these problems, a program may reference a particular member of a parameterized type simply by naming the type being restricted and the values of the type's parameters. For example, instead of defining the type <zmod6> as a restriction of <Zmodn>, one might like to refer to type "<Zmodn> with parameter 6."

Definition: **Anonymous restricted type (art)**

An **anonymous restricted type** (or **art**) is a type created by Newspeak from a parameterized type and values for its parameters. The syntax is

(art parameterized-type param-value1 param-value2 ...).

The type is called anonymous because it can't be referenced by a simple name: it is always referenced using the syntax just given. The arts of a given type are never related through restricts, but each art restricts *parameterized-type*. Furthermore, the art described by a certain *parameterized-type* and sequence of values will always refer to the same type object.

For example, *(art <Zmodn> 5)* is an art which restricts <Zmodn> with its parameter n given the value 5. Wherever *(art <Zmodn> 5)* appears in a program, it will always refer to the same type-object.

Note that given this type declaration:

```
(deftype zmod5
    restricts: ((<Zmodn> (n 5))))
```

There is no relation between <zmod5> and *(art <Zmodn> 5)*, although both restrict <Zmodn>. As a matter of good programming style, names for arts should not be introduced unnecessarily.

3.3.15 Function Objects

For every procedure we define using defproc, Newspeak associates the name of the procedure with the types of the domain and range of the procedure, and the compiled code for the procedure. The <Function> type is used for assembling this information. <Function> is parameterized by a domain and range descriptor. Objects of type <Function> contain compiled code and are stored in a data base in which the key is the procedure's name.

Because a <Function> object is parameterized by the domain and range, the user can describe procedures by creating a type which restricts the <Function> type. Newspeak provides special syntactic forms to denote <Function> anonymous restricted types. The basic syntax is *(fcn domain range)*, where domain is a sequence of type expressions similar to those that can appear in the formal parameter list of a defproc. Range is either a single type expression or a sequence of type expressions preceded by the symbol 'values'. The latter form is used to indicate that the procedure returns multiple values. Newspeak converts this special syntax into the appropriate domain and range descriptor object and then creates an *art* of <Function>. Some examples:

(fcn (<Integer> <Integer>) <Integer>) - a procedure that takes two <integer>'s and returns an <integer> result.

(fcn (<Zmodn> ~0) ~0) - a procedure which takes two identical types which restrict <Zmodn> and returns the same type. Numbers are used in the <Function> art form to refer to formal variable names. Thus the zero refers to the zeroth formal parameter to this function. This <Function> art describes a procedure like:

(defproc plus ((x <Zmodn>) (y ~x)) ~x ...and so on...)

(fcn (~1 <Zmodn>) ~1) - another way to write the previous type.

(fcn (1::t <List>) ~1) - this is how to write the type of the *cons* function of <List> defined on page 33. The *1::t* refers to value of the type of the t type parameter of the <List> object.

(fcn (<ED> ~0) (values ~0 ~0 ~0)) - a procedure which takes two identically typed <ED> objects and returns three objects of that same type. This is how the type of the extended Euclidean algorithm would be written.

3.3.16 Subtype

When we compare <Function> anonymous restricted types, we use a new relation, *subtype*, that is a superset of the *restricts* relation. As was mentioned in section 3.3.14, the arts of a given type are never related through the restricts relation. However it makes sense for a *(fcn (<zmod5>) ~0)* object to be used where a *(fcn (<Zmodn>) ~0)* is required because any function of a <zmod5> object that returns a like object can be used where a function of a <Zmodn> object returning a like object is required.

Definition: **Subtype**

<A> is a **subtype** of <B> if <A> equals <B>, <A> restricts <B>, or if both <A> and <B> are <Function> arts and the domain and range of <A> are *subdomain* and *subrange* of the domain and range of <B>.

Informally, X is a subdomain (or subrange) of Y means that corresponding elements of X are subtypes of the corresponding elements of Y. If a domain (or range) contains type expressions (rather than type-objects), then the subdomain (resp. subrange) is the one which is denotes the smaller set of types. For example, if domain X=*(~foo ~0)* and domain Y=*(~foo ~foo)*, then X is a subdomain of Y but Y is not a subdomain of X.

Now that we have defined the subtype relation, we can clarify the process NEWSPEAK uses to verify that a declared restriction of a parameterized type is valid. If type <A> is declared to restrict parameterized type <B> with actual parameter values L and M, the subtype relation is used to test if the types of L and M are legal.

We chose to define the subtype relation rather than enlarge the restricts relation because of the way we represent restricts and subtype in our current implementation. The restricts relation is explicitly represented whereas the subtype relation is computed when needed. If we were to enlarge the restricts relation we would have to deal with the case of a type added inside the type-hierarchy instead of at the leaves. This would require checking the types of existing types to see

if they should restrict the new type, and if they did the tables which represent the restricts relation would have to be enlarged. In implementations other than our current one, this process may prove inexpensive enough to do and we would then drop the subtype relation.

3.3.17 Generic Function Calls

All of the examples of Newspeak function calls we have presented to this point have been simple generic function calls. In this section we examine the mechanisms of generic function calls in Smalltalk, Flavors, Glisp, and Ada. In the next section we will introduce a type of function call called the *parameterized generic function call* which has no parallel in these other languages.

A simple generic function call consists of a function name (called a *selector*) and a sequence of zero or more arguments. Newspeak uses the selector and declared types of *all* of the arguments to determine, at compile time, the particular function to call. Because this information is known at compile time, Newspeak has the option of replacing the function call with the body of the function being called (i.e open coding the function).

In Smalltalk, each function call must have at least one argument. The selector and the actual type of only the first argument determines the function to call. In order to write generic functions which depend on the types of more then one argument (a common occurrence in an algebra-system program), the programmer must do "dispatch on type" programming, just as it is done in languages without generic functions.

Determining the actual function to call in Smalltalk is done at runtime rather than compile time. This trades interpretation-semantics flexibility against compile-time optimizations. Figure 3.10 shows a simple type-hierarchy. A function *FuncX* is defined for <A> and <C>, *FuncY* is defined for <B> and *FuncZ* is defined for <D>. *FuncY* simply calls *FuncX* on its argument, and *FuncZ* does likewise with *FuncY*. When Newspeak compiles *FuncY* it resolves the reference to *FuncX* to *FuncX* defined over <A> (for reasons we will give shortly). Thus if *FuncZ* were given a <D> object, it would call *FuncY* which would then call the *FuncX* over <A>. In Smalltalk each generic function is resolved at runtime based only on the type of the object and independent of the type over which the function was defined. Thus in Smalltalk, if *FuncZ* were given a <D> object, it would call *FuncY* which would then call *FuncX* over <C>. We believe such behavior is dangerous: the author of *FuncY* wrote it with knowledge of *FuncX* over <A>. He has no way of predicting which restrictions will be defined beneath <B>, and if programmers of those restrictions are free to redefine functions that his *FuncY* uses, then he can't be confident that his program will work for all inputs.

There *are* programs in which it is desirable to call functions which are defined by restrictions. An example we will see later is the greatest common divisor function (*gcd*) over the type of Euclidean domains. *gcd* calls a function to find the quotient and remainder of its arguments, this function being defined over a restriction of Euclidean domain (<integer>, for example). Such a call to a function defined over a restriction is handled in a different way by Newspeak and the programmer must explicitly declare which functions should be handled by restricting types. This type of call, named a *parameterized generic function call*, is described in the next section.

To summarize, in Smalltalk all generic selection is done at runtime. Thus (1) open coding cannot be done, (2) there is no way to tell at compile time the type of value returned from a function call, and (3) there isn't a way to tell if all the functions which will be needed at runtime, exist. Newspeak benefits, by contrast, from resolving generic function calls at compile time (as does Ada).

In Flavors and Glisp, both of which are embedded in Lisp, generic function calls work in the same way as they do in Smalltalk. In Glisp, the generic function calls may be open coded if certain declarations are given, but as was mentioned on page 25, this is a dangerous practice. Flavors and Glisp also permit the use of normal Lisp non-generic function calls. The calling sequences for generic and non-generic functions are different. Thus programs can become a confusing mixture of code accessible only with generic calls or only with normal calls.

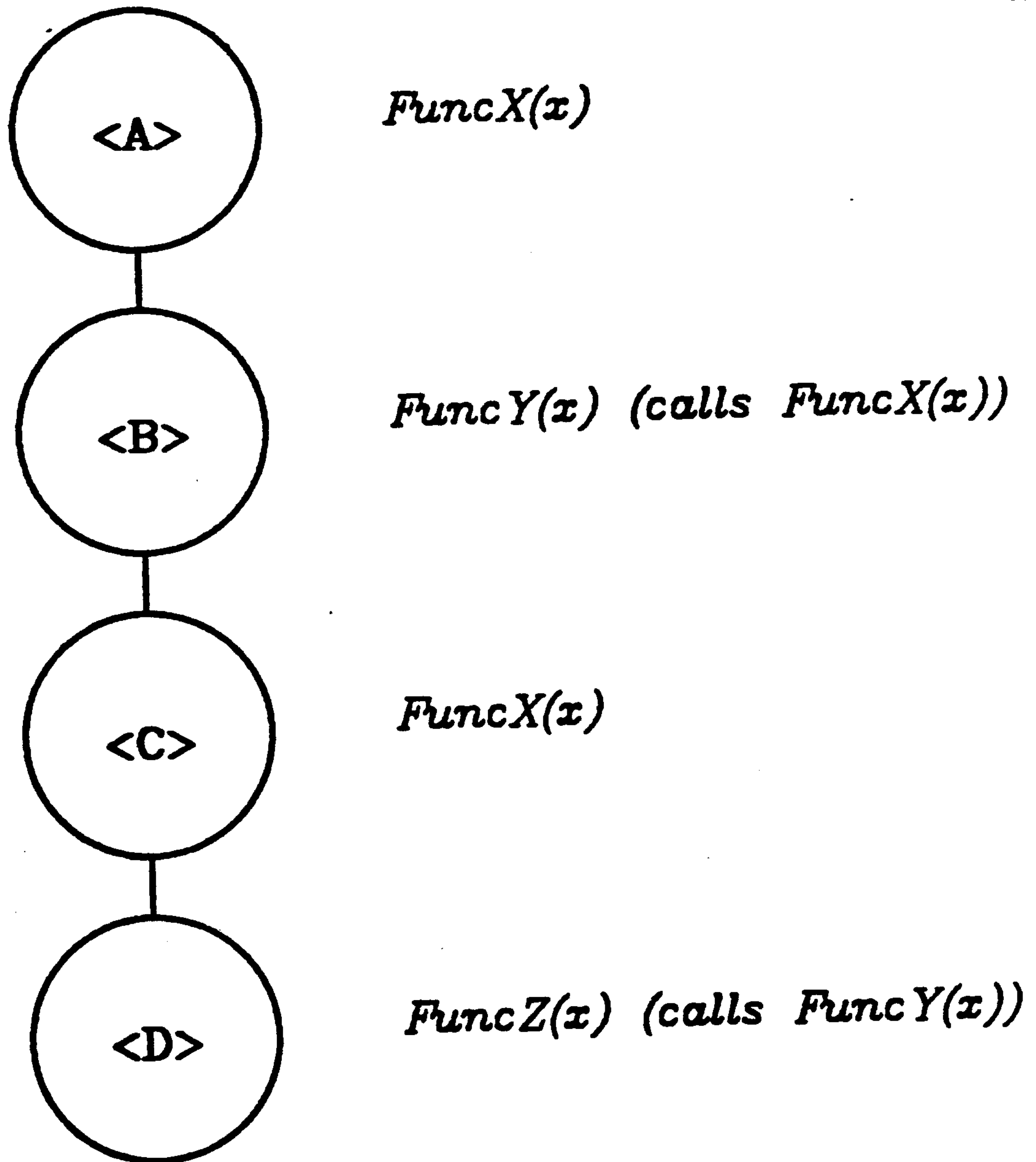


Figure 3.10

In Ada, generic function calls are handled in a different way. As in NEWSPEAK, all generic function calls are resolved at compile time. Ada uses not only the name of the procedure and the types of the arguments; it also uses the result type. For example, the user could write two functions named *plus*, both of which took two integers arguments, one returning an integer and the other a real number. Suppose *n* are *r* are integer and real variables, respectively, and the Ada compiler is given these expressions:

```
n := plus(3,4)
r := plus(3,4)
```

It would resolve each call to *plus* differently.

Before we critique Ada's method of generic function resolution, we present a prophetic statement from the *Rationale for the Design of the Ada Programming Language* [Ichbiah79]:

> We believe that the language designer should not forbid an otherwise useful facility on the grounds that it could be misused in isolated cases. He should never take the attitude of the Newspeak [Orwell50] designer:
>
> > "Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thoughtcrime impossible, because there will be no words in which to express it.
>
> Rather, he should always strive to expand the expressive power of the language, while at the same time providing more safely by the consistency of his design.

While this statement was meant to compare Ada with Orwell's Newspeak, it also applies to our NEWSPEAK, so we must defend ourselves. Our method of handling generic function calls *is* more restrictive than Ada's but we argue below it is superior. First we describe the problems with the Ada method:

- When an expression is large, it is often easiest to understand it by looking first at elementary constituents and then at successively larger subexpressions. This can be subverted in Ada because the meaning of a subexpression is ambiguous without knowledge of the context surrounding it. Given a large program which uses primarily generic functions, an expression with a few function calls and constants may have effects which confound the human reader. In NEWSPEAK, expressions can be analyzed "bottom-up".

- In Ada, an assignment statement is considered a unit as far as type checking and generic function selection is concerned. If the selection fails, the system is likely to report only that "somewhere in the statement on line N is a generic function that cannot be resolved". Closer analysis is difficult.

These are general criticisms of the Ada method of generic function resolution. If we consider these problems in combination with NEWSPEAK's other design features, the arguments against Ada's method are much stronger:

- In NEWSPEAK, all statements return values: thus an entire function can be considered as one expression for the purposes of generic function selection. If, using the Ada method, the selection failed, the compiler could only name the function and report that there is something wrong somewhere in it.

- NEWSPEAK expressions can return more than one value. While this does not rule out the Ada method, it adds to the complexity.

- As well as keeping track of the type of variables, The NEWSPEAK compiler records certain relations between the types (to be discussed on page 41). Again, it is probably possible for the compiler to maintain enough information to use the Ada method, but the human reader would be unlikely to be able to duplicate this feat. We certainly do not want to design a language where the human is at such a disadvantage.

The Newspad language (to be discussed in section 6.3) also uses Ada's method of generic function selection.

3.3.18 Functional Parameter Inheritance

A mathematical type, such as ring or integral domain, is distinguished by the functions that exist for the type and by the properties of the functions. Through the use of functional parameters, a NEWSPEAK program can express the fact that certain functions must exist. A program does not, however, represent the requirements placed on those functions, such as closure, commutivity, etc.

In the following example we will see that the invocation of a function guaranteed by its mention as a functional parameter is significantly different from a normal generic function call, although the syntax of both types of function calls are the same.

Suppose we wish to define the type <Comparable> and require that there be a function to compare for equality two <Comparable> objects. We would write it this way:

```
(deftype Comparable
    params: ((= (fcn (<Comparable> ~0) <boolean>))))
```

The = parameter is declared to be a function which takes two identically typed objects (whose type restricts <Comparable>) and returns a <boolean> (true or false) value. The <Comparable> type has no lex fields (it is representationless) and thus objects cannot exist of this type. We will see next that it *is* possible to write functions of "<Comparable> objects" - the actual objects passed to these functions will have types which restrict <Comparable>.

When we define a type that restricts <Comparable> we must provide a value for the = parameter. We can either supply the name of a procedure whose type is a subtype of *(fcn (<Comparable> ~0) <boolean>)* or we can associate this parameter with one of the parameters of the new types we are defining. The purpose of the = parameter in the <Comparable> type is to guarantee the existence of a procedure named = with the type

(fcn (<Comparable> ~0) <boolean>).

Based on that guarantee, we can write the following procedure:

```
(defproc not= ((x <Comparable>) (y ~x)) <boolean>
   (not (= x y)))
```

The call to = is the first example we have seen so far of a *parameterized generic function call.* The characteristic of the call that distinguishes it from the normal generic function call is that NEWSPEAK can't determine at compile time which procedure will be called. Instead, at run time the particular procedure to call will be determined, based on the actual type of x and y.

Before we can describe in greater detail the operation of the *not=* function, we must define a domain-type which restricts <Comparable>. We will then have objects which can be passed to the *not=* function. We next define <Zmodn> to restrict <Comparable> and then use an *art* of <Zmodn> for our domain type.

```
(deftype Zmodn
    params: ((n <integer>))
    lex: ((i <integer>))
    restricts: ((<Comparable> (= mod=))))

(defproc mod= ((x <Zmodn>) (y ~x)) <boolean>
    (= x:i y:i))
```

In the deftype for <Zmodn>, we declare that <Zmodn> restricts <Comparable> and that the = function required by <Comparable> is provided by the *mod=* function. We use the name *mod=* rather than = to make this explanation clearer. Because <Zmodn> restricts <Comparable>, we can use the function *not=* defined over <Comparable> objects for objects whose type restricts <Zmodn>. Suppose that variables a and b contain objects of type (art <Zmodn> 5). Let us follow the evaluation of the expression *(not= a b)*. The machine jumps to the *not=* function where x is bound to a and y to b. Next it evaluates *(= x y)*, which requires determining the correct = function to call. Viewing x (or y) as a <Comparable> object, the = parameter is accessed (in effect, evaluating *x::=*). In this case, the *mod=* function is value of the = parameter. The *mod=* function is then called on the values in x and y. *mod=* extracts the integers from the <Zmodn> objects and calls the = function over <integers>. The rest of the evaluation process is straightforward.

Thus, a parameterized generic function call is evaluated in two steps. First the type parameter (a <Function> object) given by the function name is extracted from the type-object. Next program contained in the <Function> object is invoked.

We will continue the discussion of function selection when we describe the function database in section 4.2. In the next section we introduce a new topic, "distinguished objects".

3.3.19 Distinguished Objects

Associated with domain-types are *distinguished objects*, useful in several contexts:

mathematical values

In the carrier set of an algebra there are usually certain members which have special characteristics. Examples are additive identities and multiplicative identities in a ring (also known as zeros and ones).

sentinel values

In languages with dynamic storage allocation, linked lists are a common data structure. Each object in a linked list contains a set of data fields and a *next* field which either points to the next object or is a sentinel indicating the end of the list. In Lisp, the sentinel is usually indicated by a pointer to the constant object nil. The nil object solution works in Lisp because Lisp is not a strongly typed language: the *next* field of a list data object in Lisp may point to any type of data object. Pascal, on the other hand, is a strongly typed language. In Pascal, there is a reserved pointer value called *nil* which indicates that the pointer doesn't point to anything. The *next* field of a Pascal data object can only point to a single type. The reserved pointer value solution was adopted so that an uninitialized pointer would not cause problems at run time by pointing to a value of an illegal type. The cost of this solution is that each time a pointer value is used, it should be checked that it isn't nil.

NEWSPEAK has the same constraints as Pascal: it must insure type correctness at run time. However the Pascal solution would be too expensive because most references in

NEWSPEAK are through pointers.

failure

There are mathematical algorithms (such as exact-quotient in a ring) which may, for certain inputs, return an indication of failure. Creating a type which includes everything of a given type plus a failure indicator is expensive and clumsy. What is needed is an object of a given type which can indicate that a failure has occurred.

Now that we we have motivated the idea of distinguished objects, we define the term:

Definition: **Distinguished object**
A **distinguished object** is an object attached to its type-object and accessible by providing the type-object and the distinguished object's name.

Each domain-type has at least one distinguished object, namely *null*, which is used for uninitialized variables of the domain-type. User programs may find this value useful as a sentinel for linked lists, or it can be used as failure indicator. The NEWSPEAK syntax to access the null object of type <X> is *(dist null <X>)*. 'dist' is an abbreviation for 'distinguished'. Although the syntax is similar to that of a function call, access to a distinguished object can always be done inline, since it only involves extracting something at a known offset within a type object.

Distinguished objects and their initial values are declared in the deftype form.

```
(deftype rational
        lex: ((num <integer>)
              (denom <integer>))
        dist: ((zero num 0 denom 1)
               (one  num 1 denom 1)))
```

Thus *(dist zero <rational>)* is a <rational> object with the *num* field containing the <integer> 0 and the *denom* field containing the <integer> 1. Note that the distinguished object *null* need not be declared because it always exists.

The existence of distinguished objects is inherited by restricting types. If we define a type <Ring> that has distinguished objects zero and one, then all types which restrict <Ring> must also have distinguished objects zero and one.

This is how we might write a function *inversep* over <Rings> which returns the <boolean> value true if its two arguments are multiplicative inverses of one another:

```
(defproc inversep ((x <Ring>) (y ~x)) <boolean>
   (= (* x y) (dist one ~x)))
```

The reasons that we used *(dist one ~x)* rather than *(dist one <Ring>)* are:

- Only domain-types have distinguished objects attached. <Ring> isn't a domain-type (it is representationless), thus does not have distinguished objects. It is possible to *declare* distinguished objects for non-domain types, but no objects will be created. The declaration is useful nevertheless because it insures that the restricting domain-types will have the declared distinguished objects.

- The type of x and y is not <Ring>, but is some restriction of <Ring>. In order to do the comparison with =, we must compare the value of the product *(* x y)* with a dis-

tinguished object of the same type.

Distinguished objects can take the place of global constants. In Lisp the global variables t and nil hold the true and false values (although any non-nil value is generally considered to represent true). In NEWSPEAK, these values are written *(dist true <boolean>)* and *(dist false <boolean>)*.

4. The compiler

4.1 Type checking

Type checking in an hierarchically typed language like NEWSPEAK is more difficult than type checking a generic-function strongly typed language like Ada. The difficulty lies in the fact that the NEWSPEAK compiler has to deal with the *declared* types of objects, knowing full well that the *actual* types of the objects may be some restriction of the declared types. Furthermore, the types of the arguments of generic functions are not expressed simply as type indicators. Instead they may also take on the forms ˜*n* or *n::par*, as was described in Section 3.3.11. This requires the compiler to maintain certain extra information about the type of each object accessible from within this procedure:

- Is the type of this object the same as the type of another object at runtime? As we saw when defining *plus* over <Zmodn> on page 30, it is not enough to know that two variables are declared <Zmodn> to be able to add them.

- Is the type of this object the same as the value of a parameter of the type of another object? This is important in order to check a function such as *cons* on page 33 for type correctness.

Furthermore, the compiler must contend with multiple values and conditional expressions whose branches return an object (or objects) of different types. The record keeping, while complex, can be done at compile time and so the cost need only be paid once.

4.2 Function database

In order to permit the compiler to select the correct function for a generic function call, all known function objects are kept ordered in a complete database. Given two keys, the name of the function and the number of arguments, a sequence of function objects can be retrieved from the database. Recall that the type <Function> has as a parameter a domain and range descriptor. In the ordering relation for function objects (objects whose types are arts of <Function>), only the domain part of the domain and range descriptor is considered. In this section we will use the term *domain* to mean the domain part of the descriptor of the type of a function object. In the function database, the function objects are ordered in this way: for a given object X and all objects Y which follow it, either the domain of object X is a subdomain of the domain of object Y or the types are unrelated (*subdomain* is defined on page 35). Given this ordering, the first function that is found whose domain is a superdomain of the domain being searched for is the correct function to use. By correct, we mean the most specific function suited for this task.

For example, if we assume the type-hierarchy in figure 4.1 and write a *print* function for <object>, <B>, and <C>, we end up with this ordering of functions with selector *print* (we only list the types of the function objects):

```
(fcn (<C>) ˜0)
(fcn (<B>) ˜0)
(fcn (<object>) ˜0)
```

Since <C> and <B> are unrelated, the relative order of their *print* functions in the data base is irrelevant, but both must precede the *print* function for <object>. Suppose the compiler is given the expression *(print x)* to compile and x has the type <D>. The compiler creates a domain expression: a list containing the type of the only argument, <D>. It then extracts from the database all *print* function objects which expect one argument (which is the sequence of three

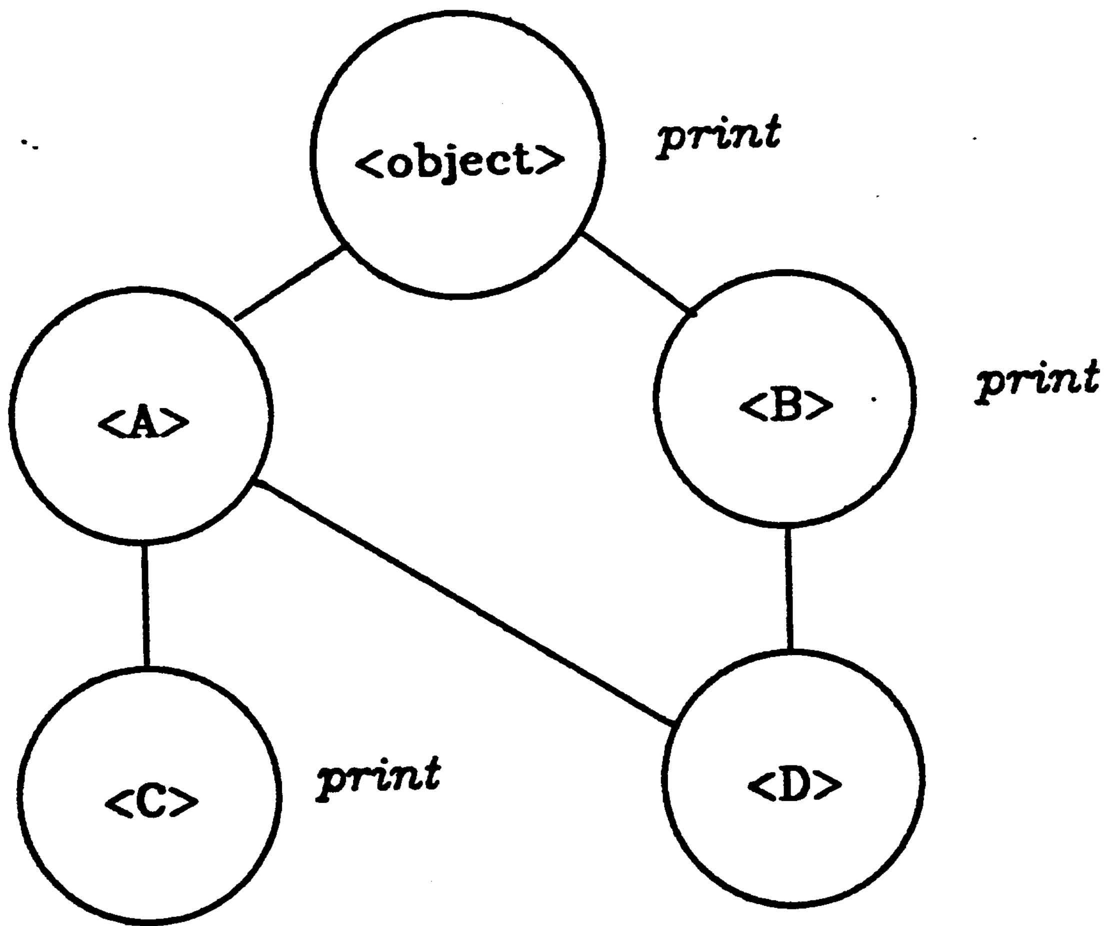


Figure 4.1

objects we listed above). It compares the given domain expression, *(<D>)*, against the domains of the function objects. When it finds the first superdomain, in this case *(<B>)*, it selects that function object.

If a type's parameter is a function object, that object is also placed in the data base but it is tagged to indicate that it is a type parameter. If the function selection process selects a type parameter, a special type of function call is generated (described in section 5.2).

4.3 Frozen Types

Suppose the *plus* function over rational numbers has this type: *(fcn (<rational> ~0) ~0)*. Now assume that there are <rational> values in x and y but due to the way they were generated, the compiler can't be sure they are the same type (as far as it can tell, their actual types may be different restrictions of <rational>). If the programmer knows that the type <rational> will never be restricted then the fact that both x and y are declared to have type <rational> is enough to indicate that they have identical types, and that type is <rational>. Because the compiler doesn't realize that their types are identical, it will not permit the function *plus* with type *(fcn (<rational> ~0) ~0)* to be called on x and y.

The problem is not a result a misdeclaration of *plus's* type: in this example we see that such a declaration occurs naturally as a result of the restriction relation and functional parameters.

```
(deftype Field
    params: ((plus (fcn (<Field> ~0) ~0))))

(deftype rational
    lex: ((num <integer>)
          (denom <integer>))
    restricts: ((<Field> (plus plus)))
```

The restricts clause of rational's deftype declares that there will be a *plus* function over rationals whose type is a subtype of <Field>'s *plus* parameter. Newspeak automatically defines the subtype to be *(fcn (<rational> ~0) ~0)*. (Newspeak only generates subtypes automatically for <Function> parameters).

We return to our original problem: we have two <rational> values that we would like to add but can't because there is no function of type *(fcn (<rational> <rational>) <rational>)*. If we declare that <rational> is a *frozen type*, that is, that it will never be restricted, then Newspeak recognizes that *(fcn (<rational> <rational>) <rational>)* is equivalent to *(fcn (<rational> ~0) ~0)* and our problem is solved. Contrast this with the *plus* function over <Zmodn>, a non-frozen type. In this case, *(fcn (<Zmodn> <Zmodn>) <Zmodn>)* is not equivalent to *(fcn (<Zmodn> ~0) ~0)*. All domains-types are assumed frozen by default when they are created, but can be explicitly unfrozen by a declaration in the deftype.

5. A Partial Implementation

In order to test the design of NEWSPEAK and demonstrate consistency, utility and completeness, we wrote a prototype system in Franz Lisp [Foderaro82]. Although it is not a complete implementation of NEWSPEAK, it is a large enough subset to permit us to run most of the programs we will present in section 7.. Lisp provided a base for rapid prototyping of a system and we were able to modify our ideas as we experimented on the language described in this thesis.

A complete NEWSPEAK system could be built entirely in NEWSPEAK itself and could be independent of Lisp. Such a construction would force us to resolve a number of problems, one of which is how to store objects in memory. We discuss some of these issues in the following subsections.

5.1 Pointers and Object storage

An object is usually implemented as a block of storage, referenced by a value called a pointer. Given an object A, it must be possible to locate the type-object for A. The method chosen for this task usually determines which dynamic memory allocation scheme is used.

The fact that objects are referenced through pointers allows objects to be passed between procedures by just passing pointers. Generally pointers fit in one machine word, making it much more efficient to pass pointers than to copy the (potentially large) objects themselves. The drawback in using pointers is the *aliasing* that results from having an object referenced from more than one location. This same situation occurs routinely in Lisp and other languages and our experience has shown that it is not a problem if the programmer follows simple rules. In Lisp, any user function can destructively modify almost any data object. In NEWSPEAK, only code that has specifically requested and received permission (at compile time) to look inside an object can destructively modify it.

In some applications, aliasing can be beneficial. It permits pieces of large data structures (such as mathematical expressions) to be shared. Localizing destructive modification makes aliasing safer.

The only requirement we have placed on a pointer to an object is that it be able to lead us to the storage for that object. Independently, we have required that every object have a type-object associated with it which is accessible from the object. We now consider strategies for *allocating objects, associating type-objects with objects,* and *creating pointers*:

Storing types within objects

Each object is a data structure that has as one component a pointer to its type-object. The pointer to an object is its actual address in memory. The advantage of using the actual address for a pointer is that it is possible to access parts of an object quickly (on most machines) by using a simple register-plus-offset operand form. The disadvantage is that each object is larger because it must contain a pointer to its type or some abbreviated indicator of its type.

Encoding the type within pointers

A typed pointer contains both the address of an object and an indicator of its type. There are two implementation problems with this.

- If a typed pointer is to reside in a machine word, then if addresses are to be large, the number of bits that can be used to indicate the type must be small. This is not a problem in most Lisps which have at most a few dozen types, but in NEWSPEAK the number of types can grow very large. The solution (used in those Lisps which have user defined data types) is to define a type code meaning "the type is actually stored in the object", thus resorting to the typed object method.

- In order to use the pointer to access a field in an object, the type information must be stripped from the pointer. In NEWSPEAK, extracting a value from an object is a very common operation, so an underlying machine addressing mode that ignores the type field would be handy. Target machines which have only simple addressing modes will require frequent use of masking or shifting instructions.

Encoding of types by association of storage blocks

The Big Bag of Pages (or bibop) storage management method [Steele77] [Foderaro81] is used in PDP-10 Maclisp and Franz Lisp (on the VAX and 68000). The pointer to an object is simply the memory address of the beginning of an object. The high order bits of the pointer can be treated as an index into a vector of type indicators called a typetable. Depending on how much space is devoted to each element of the type table, the type indicators could be pointers to type-objects or indexes into another table containing pointers to type-objects.

There are two major flaws in this method in the context of NEWSPEAK. The first is that accessing the type-object, a common operation in NEWSPEAK, would be too slow, requiring bit shifting and one or two memory accesses. The second is that if a program needs just one object of a certain type, the NEWSPEAK system would be required to allocate a whole page of objects of that type. In NEWSPEAK, there are a large number of types for which only one object is necessary (e.g. <Function> arts). If an entire page were allocated for each type, memory and address space would be poorly utilized.

Encoding of types by object table

Many Smalltalk implementations use this method. The pointer is an index into a table of pairs: an object's address and its type. All references to objects are just indices into this table. The advantages of this scheme is that it allows pointers to be small and makes object compaction easy. The drawback is that extracting an object's contents requires first going to this table to fetch the address.

Removal of type information at compile time

A pointer is the address of an object. There is no way to determine the type of an object from its pointer or from the object itself. However, NEWSPEAK would insure that for every value passed to a function, the type of the value would be passed as well (as an invisible parameter) *if* the type were needed. This type of scheme is possible in NEWSPEAK because it has strict type checking at compile time. The problems with it are: function calls may take longer if there are more parameters to pass back and forth; debugging and garbage collecting would be harder because there would be no way for these programs to determine the type of an object directly from a pointer to an object.

None of the strategies we have just examined is clearly best for NEWSPEAK, although some are quite clearly bad (e.g. bibop). We used the object table approach in our current implementation because of its simplicity. We plan to experiment with a number of different strategies in a subsequent NEWSPEAK implementation.

5.2 Type Parameter Extraction

In this section we describe how parameter values are stored in our prototype implementation. By the syntax we use for parameter extraction, we consider a parameter's value to be part of an object, not the object's type-object. For example, if x contains an object of type (art <Zmodn> 5), then we can extract 5 by *x::n* (if x is viewed as a <Zmodn> object). It ap-

pears therefore that the value 'n' is being extracted from the object x when it is really being extracted from the type-object (art <Zmodn> 5).

It takes three pieces of information to extract a type parameter: (1) the name of the parameter, (2) a type-object, <T>, to extract the parameter from, and (3) a type-object <V> which <T> restricts and in which the parameter was declared. (<V> is known as the *viewed as* type). In the above example, the parameter name is n, <T> is (art <Zmodn> 5) and <V> is <Zmodn>.

In our current implementation, each type-object <T> contains a pointer to a list of vectors. There is one vector for each parameterized type which <T> restricts. Within the vector for a particular restricted type, <V>, are the actual values of <V>'s parameters. Thus to locate the value of parameter n of <T> viewed as <V>, Newspeak must look in <T>'s list of vectors for the one associated with <V> and then extract the value associated with n. The layout of the vector of values of <V>'s parameters is determined when <V> is defined. Thus Newspeak can locate the value of a parameter within a vector with one vector reference.

In order to locate <V>'s vector, Newspeak must search the list of vectors because it cannot predict at compile time where the vector associated with a given viewed-as type will be. The searching cannot be totally avoided because Newspeak permits multiple inheritance (a type can restrict more than one type). Suppose there were a parameterized type <A> and we decided that in every type that restricted <A>, the location of the vector containing the values of <A>'s parameters would be N'th in the list. Now suppose that there were a type <B> and we decided that its vector should be N'th in the list too. If we define <C> to restrict both <A> and <B> then we have a case where both <A>'s and <B>'s parameter value vectors must be N'th in <C>'s list of parameter vectors.

In order to reduce the searching cost, we can use a number of strategies. One is to use a hash table instead of a list. This may not be too useful as the list of vectors is most likely to be small (less than 10), so the hashing could cost more than simply doing the comparisons. We must consider two costs associated with the searching: space (amount of code needed at each parameter reference), and time (cpu time needed to resolve a parameter reference). In order to save space, we want to make the searching procedure into a subroutine and access it via a "jump to subroutine" at each parameter reference location. In order to save time, we would like to avoid jumping to the searching subroutine whenever possible. One way to avoid going to the subroutine is to cache the result of the parameter lookup. The cache key would be the actual type and the value would be the viewed-as vector. There would be a separate cache block for each instance of parameter lookup in the code. The runtime mechanism for parameter lookup would be to check the actual type against the cache key and if they were identical, the cached viewed-as vector would be used. If a cache miss occurred, the standard parameter lookup would be done and the results stored in the cache. Studies of Smalltalk (in which most function calls cause a runtime table lookup based on type) show that 95% of lookups are to the same type as the preceding lookup [D'Ambrosio83]. The cache solution would cost a little more in space at each parameter reference but greatly reduce the time when a cache hit occured.

6. Related Languages

In this section we compare NEWSPEAK to a collection of related languages: FRL, Capsules, Andante and Newspad.

6.1 FRL

FRL is a hierarchical data description language embedded in Lisp used for planning and natural language understanding. It is based on Minsky's *frame* technique for representing knowledge [Minsky75].

A FRL program creates data objects called *frames* which are patterned after association lists in Lisp. Frames are stored in a database and may be interrelated. When data is stored in or retrieved from a frame, it may cause other frames to be altered or searched or a Lisp program to be run.

Datum access from a frame in FRL is thus not an atomic operation as it is in NEWSPEAK. FRL would not be suitable for implementing the low-level data types on an algebra system that we have described in the thesis (e.g. ZmodN and Polynomial). However, it may prove useful at the user-interface level of the algebra system where planning is necessary. One could model FRL in NEWSPEAK by writing special procedures to perform each datum access. Such a strategy would be work best if the procedures were written automatically from an FRL frame description.

6.2 Capsules

Capsules [Zippel83] is an object-oriented system written in Lisp by Richard Zippel, one of the authors of Macsyma. It was originally designed to serve as base language for a symbolic algebra system.

Capsules is an outgrowth of Flavors, differing mainly in how operations are associated with objects. In Flavors, the hierarchy determines which methods are callable. In Capsules, all methods have associated (explicit) properties. If an object needs an operation with a certain property, the Capsules system locates it for the user. In fact, if costs are associated with operations, then Capsules will select the least expensive operation with the required property.

The problems we see with Capsules are similar to those we have mentioned about Flavors: (1) functions are generic on only the first argument, (2) type checking isn't done (although Capsules does check to see that all required operations exist for an object). A problem unique to Capsules is that of representing properties of operations. Currently, properties are just symbols that are uninterpreted by the system. If Capsules were to be used in an algebra system, a more powerful property mechanism would be needed (e.g. parameterized properties and some reasoning scheme). We do not represent type properties explicitly in NEWSPEAK for just this reason, but in Capsules, properties must be explicitly represented in order to select the correct operators.

6.3 Andante and Newspad

The languages Andante and Newspad, like NEWSPEAK, are designed for math-oriented symbolic algebra systems. Newspad was an outgrowth of the Scratchpad project at IBM Yorktown Heights (Newspad is an abbreviation for NEW ScratchPAD). The IBM researchers, Jenks, Trager and Davenport, had goals similar to ours when they designed Newspad: algorithms should be written in their most general mathematical framework yet there should be very little performance penalty for this generality. David Barton, a graduate student at Berkeley, worked with the Newspad group for a summer. Upon his return to Berkeley he wrote a new algebra system, Andante, based on Newspad. Andante and Newspad then grew independently, but due to the interaction between the authors, a new feature in one system usually found its way into the other. According to Barton, more algebraic code has been written for Andante, but Newspad has a much higher developed user interface [Barton83]. As of this writing, both systems are still in the research stage, having never been released to the public.

Andante and Newspad are actually the names of algebra systems written on top of Lisp-based implementation languages Modes and ModLisp [Davenport80] respectively. When we refer to Andante and Newspad, we will be referring to their implementation languages, not the algebra systems themselves.

In this section we compare NEWSPEAK with Andante and Newspad. Although we studied Andante and Newspad before designing NEWSPEAK, we did not use these languages as a starting point. Instead we started from scratch (without even a base language like Lisp) and added those features that we felt were necessary to support an algebra system. The resulting language is quite distinct from Andante and Newspad yet many of the constructs in NEWSPEAK have parallels in Andante and Newspad, making it likely that we can share programs.

In order to simplify the discussion which follows, we will only compare NEWSPEAK to Newspad. Andante will be mentioned only when it significantly differs from Newspad.

6.3.1 Category, Functor and Domain

Where NEWSPEAK has the single concept of *type*, Newspad uses three concepts: *category, functor,* and *domain*. Very roughly, a category is similar to a NEWSPEAK parameterized representationless type, a functor to a NEWSPEAK parameterized type with one or more lex fields, and a domain to a NEWSPEAK type without parameters and with one or more lex fields.

The Newspad programmer defines categories and functors. Functors are 'evaluated' to produce domains. Ring is an example of a category. "Integers modulo n" is an example of functor. If the "integers modulo n" functor is evaluated with n given the value 7, it returns the domain of "integers modulo 7". Each domain is a member of one category. Each category may have many domains as members. In Andante, a domain may be a member of more than one category.

A category is a set of *function descriptors* and *attributes*. A function descriptor is a function name along with the types of the arguments it expects and the type of result it returns (much like <Function> in NEWSPEAK). The argument and result types that may be specified are limited to: a specific domain, an arbitrary domain of the category being defined, or the value of a parameter of the category definition. In the corresponding form in NEWSPEAK, there are no such limitations. Newspad can't give the user similar freedom because of the different way in which function invocation is implemented, as we shall see shortly. For example, assuming that there were categories Field and UFD, one could not describe (in the Field or UFD category definition) a function which converted an object from a Field domain to a UFD domain. Later, we will see that such multi-type definitions must be placed in something called a *package*.

Categories may be given *attributes*. The form of an attribute is either a single symbol (e.g. NoZeroDivisors) or symbol with arguments (e.g. associative('*')). Attributes list some of the properties of the category. As we mentioned on page 15, the properties of categories (types in NEWSPEAK) are often infinite in number and thus we make no attempt in NEWSPEAK to represent them explicitly. Newspad lacks a type-hierarchy so the attributes of a category are very important (even though the attributes are just a *partial* list of the properties of the type). Attributes are associated with the parameters of functors and their existence can be tested during functor evaluation. We will describe how attributes are used when we describe the definition of the polynomial type below.

Categories can be thought of as templates: they are a list of functions and attributes that must exist. Domains, as we shall see later, can be thought of as category templates filled in with the actual functions which satisfy the category's requirements.

Categories can be constructed hierarchically by using existing categories and adding functions and attributes. Unlike NEWSPEAK, the construction of categories from other categories is merely a convenience used to shorten the category definitions. A reference to category X in category Y's definition is no different than textually substituting category X's definition in place of the reference. The category hierarchy is not maintained at run-time (or even compile-time). We will see in the polynomial example below that the effect of this is that Newspad programmers may have

to duplicate the hierarchy of categories in certain functors.

Most functions are defined in functor definitions. A functor definition consists of a functor name, zero or more parameters from domains or categories, the name of a result category, possibly another functor which this one extends, a list of attributes, and a list of function definitions. When the functor is invoked with specific values for its parameters, it returns a domain which is a member of the result category. The attributes required by the result category are verified to exist in the domain created by the functor. Functions defined within a functor are constrained to operate only on values from specific domains, from domains that this functor creates, or from the domains which are parameters to the functor. Consider the functor defining the domain of rectangular matrices of dimensions M and N. The domain of the *transpose* of an M by N matrix is the domain of an N by M matrices. Since N by M is a different domain, the transpose function can't appear in a functor. Such cross-domain functions are written in *packages*, as we describe later. This problem doesn't occur in NEWSPEAK since functions may be defined over any types.

6.3.2 Lack of Category Hierarchy

We now consider the problems which occur because Newspad doesn't maintain a category hierarchy. The standard algebraic hierarchy consists of the algebras monoid, group, ring, integral domain, unique factorization domain, Euclidean domain, and field. The definitions of these algebras as types in NEWSPEAK are given in section 7.. The definitions of these algebras as categories are similar for Newspad, with the addition of attributes such as 'ufd' in the unique factorization domain category and 'gcd' in the Euclidean domain category. The difference between Newspad and NEWSPEAK is apparent when we examine how each system defines the polynomial types. As we will see in section 7., in NEWSPEAK we first define <Poly>, the type of "polynomials over coefficients in an arbitrary <Ring>". We then define *plus*, *minus* and *times* for <Poly>. Next we define a sequence of types which restrict <Poly> each assuming more about the coefficient domain: <Polycr> (polynomials over a commutative ring), <Polyid> (polynomials over an integral domain), and <Polyufd> (polynomials over a unique factorization domain). Because <Polyufd> is a unique factorization domain, we can write the *content* and *primitive-part* functions over <Polyufd>. These functions can use the definitions of *plus,minus* and *times* defined for the <Poly> type because <Polyufd> restricts <Poly>. This restriction is permitted by NEWSPEAK because the coefficient domain of <Polyufd> (a unique factorization domain), is a restriction of the coefficient domain of <Poly> (a ring).

In Newspad, there is a single polynomial functor, parameterized by a domain in the Ring category. Within the functor all polynomial functions are defined. Those functions that require a more restricted coefficient domain than Ring are preceded by a conditional statement such as "If the domain has the attribute ufd then". The programmer of the polynomial functor is reproducing the category hierarchy within the functor, using conditional statements and knowing what the attributes of each category are. This is similar to the "dispatch on type" programming technique that we avoided using generic functions, although it is not quite the same since the type dispatch is done when the functor is evaluated to produce a domain, not each time a function is called.

6.3.3 Function Invocation

Newspad, like NEWSPEAK, offers two types of function calls which we have named generic and parameterized generic. The generic function call in Newspad is used when the domains of the arguments are known. The target of such a call can be determined at compile time. In NEWSPEAK, generic function calls can be used for all types, not just the types that correspond to domains in Newspad. For example, we've seen the *not=* function defined for <Set> types on page 39. A function like that could be not be defined over the Set category in Newspad.

The second type of call, the parameterized generic function call, occurs in NEWSPEAK when a call is made to a function which is declared as a parameter of a type. In Newspad, the corresponding call is one made to a function which takes as arguments any member of the domain

created by its enclosing functor (recall that most functions are defined within functors). As an example, consider the two functors: "integers modulo n" and "polynomials over a ring". Each of these functors returns a domain of the category Ring, whose template we've (partially) drawn in figure 6.1. In order to create the domain of "polynomials with integer modulo 6 coefficients", we first pass 6 to the "integers modulo n" functor. It returns the "integers modulo 6" domain (which we will call IntMod6). We pass IntMod6 to the polynomial functor and it returns the domain of "polynomials with integer modulo 6 coefficients" (which we will call PolyIntMod6).

Figure 3.13 shows these domains. The form of each domain is a vector, with the values in the vector being pointers to the functions defined in the functor which created the domain. The location within the vector of a function with a given name is the same for each domain of a given category.

Both IntMod6 and PolyIntMod6 are domains in the Ring category so the layout of their domain vectors will be identical although the specific values in the vector will differ. There is also a place in the domain vector for pointers to other domain vectors. The PolyIntMod6 vector contains a pointer to the IntMod6 vector so that polynomial coefficient operations can be done.

We now give an example of how Newspad performs the equivalent of a parameterized generic function call in NEWSPEAK. As we've drawn Ring category template, the second location contains a pointer to the *plus* function. Suppose x and y contain objects of the domain PolyIntMod6 and we wish to evaluate *(plus x y)*. We must have access to the PolyIntMod6 vector. The second value is extracted from the domain vector (it is the particular *plus* to call), that function is invoked, passing to it the x and y values and the PolyIntMod6 vector. The plus function being invoked is defined in the "polynomial over a Ring" functor and is not specifically for "polynomials over integers modulo 6". The domain vector that is passed along with the arguments describes the particular domain over which we wish the addition to be computed. One operation that the plus function will want to perform is the addition of the coefficients of the polynomials. The IntMod6 vector is extracted from the PolyIntMod6 vector and the second location of that vector points to the *plus* function (since IntMod6 is also of the Ring category). The function is invoked and passed the coefficients and the IntMod6 vector.

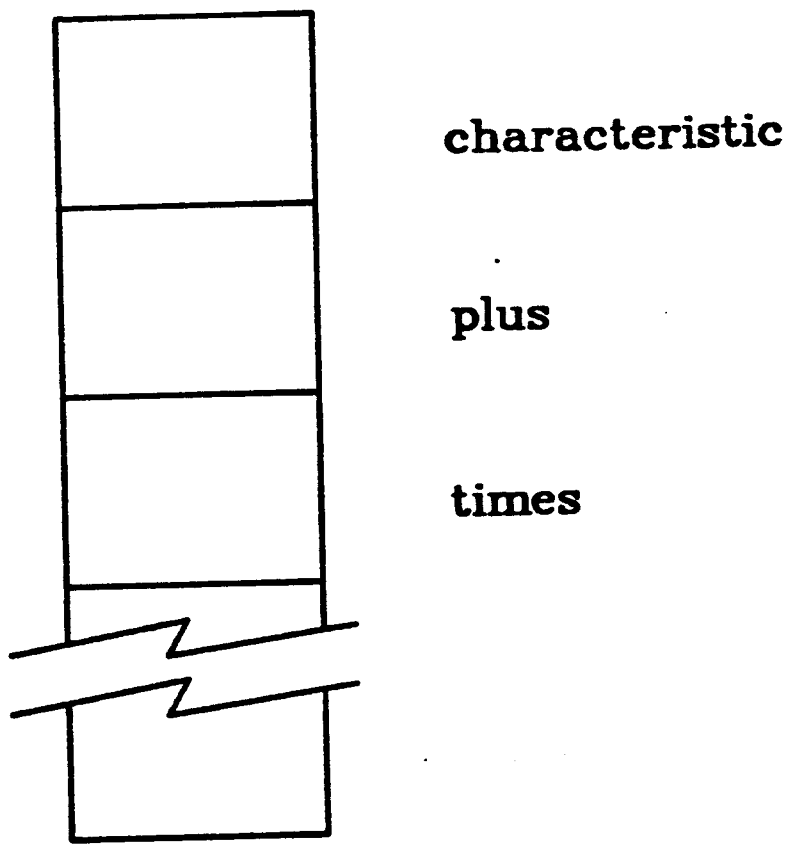
In Newspad, the appropriate domain vector is created and passed 'invisibly' to functions that require it. In Andante the user is responsible for constructing the domain vector.

From the discussion on the function calling mechanism, it is clear why functions within functors can only operate over a limited set of domains: the only parameterized generic functions which they can call are those reachable from the domain vector passed as an argument.

In NEWSPEAK, each data object (implicitly) carries around its type, and the type is a pointer into the type-hierarchy. Thus each data object has associated with it the "domain vectors" for its type and all of the types it restricts (the particular domain vector that is used depends on which type the object is viewed as). Because objects carry around their types, functions may operate on any set of data types and it will always be possible to locate the domain vectors at runtime in order to perform parameterized generic function calls.

6.3.4 Package

In order for Newspad functions to operate on objects from more than one domain (such as the transpose function mentioned earlier), they must be written in a *package* (or *capsule* in Andante) [Trager83] [Barton83]. A package is parameterized by a set of domains of specified categories and it contains functions which can operate on data from those domains. A package may contain, for example, an algorithm for exponentiation by repeated squaring. If, when a functor is evaluated, it neededs such an algorithm, it would pass the domain vector it is constructing to the package, and the package would return a new vector to be stored in the domain vector. This new vector would contain the local state information the that package required in order to run.



Ring template

Figure 6.1

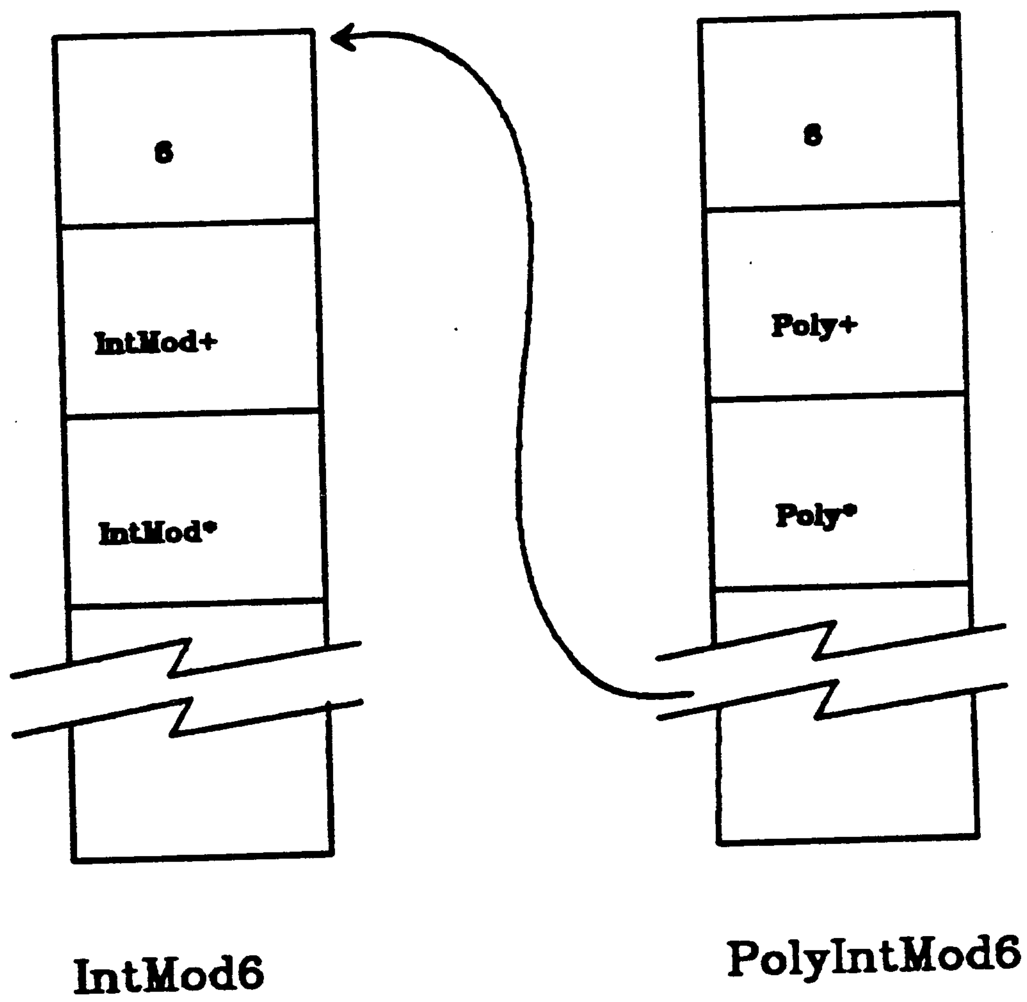


Figure 6.2

6.3.5 Summary

Newspad (and Andante) are languages well suited for the writing of programs to perform algebraic algorithms. Both systems have grown a great deal since they were last described in a published document. Our criticisms are thus based on old and incomplete information. Furthermore, we base our criticisms on how well these languages fit *our* ideas of how a system should be organized. With these caveats in mind, we proceed with a summary of the differences between NEWSPEAK and Newspad and Andante.

The advantages of Newspad's organization of functions by category and function invocation methods are:

- Access to functions is always very fast: either determined at compile time or just one vector reference at runtime. NEWSPEAK's parameterized generic function call takes longer the first time the function is located but not for subsequent calls assuming that result of the function search is cached (page 47).

- Data objects need not be typed; the domain vector contains all the information necessary to determine which functions to call. Newspad simply uses Lisp data objects without any tags to indicate their Newspad type. Newspad can do this because the Lisp system it runs in takes care of dynamic storage allocation. NEWSPEAK was not designed to work on top of Lisp, so it must put type tags on its data objects in order for garbage collection to be possible.

The problems with Newspad are:

- It has too many mechanisms: *category, functor, domain* and *capsule*, whereas NEWSPEAK has one notion of *type-object* that encompasses them all. Furthermore there are many restrictions placed on the Newspad mechanisms: functions can't be written over categories, functions within functors are limited in what they can operate on, and domains are a member of only one category.

- The hierarchy of categories isn't maintained by Newspad, thus code within functors tends to use a "dispatch on type" programming style to define the correct function based on the attributes of the functor parameter.

6.3.6 Host Language

Andante and Newspad are implemented on top of Lisp systems, Andante on top of Franz Lisp and Newspad on top of Lisp/370. NEWSPEAK is designed to be written in itself and to run on a conventional processor. We feel self-implementation is important for the following reasons.

data structures

We have more freedom to choose data structures for NEWSPEAK because we aren't constrained to use Lisp's data structures. Many Lisps have a bit vector data structure out of which we could build our objects, but we would then be forced to implement our own storage allocation and reclamation routines.

control structures

We can choose to implement novel control structures for NEWSPEAK. We already permit functions to return multiple values and we may want to add multi-processing facilities at some time. Retrofitting such features on an existing Lisp system might be difficult. Even if a Lisp already had such features it isn't clear that they would be done in a way

usable by NEWSPEAK.

uniform abstraction

We've already mentioned that Lisp does not have the type of user extensible data types that we need. If NEWSPEAK were written in Lisp, there would be a strong dividing line between the code written in NEWSPEAK and the code written in Lisp to implement NEWSPEAK. This lisp code would be opaque to the person who knew only NEWSPEAK.

better language

We have already mentioned that Lisp is a good language for prototyping systems but that there are problems with it when a program gets large. The type-centered method of organizing a system (page 13), used by NEWSPEAK, groups functions around the types they manipulate. Such modularity will help insure that a program can grow large gracefully. The modularity and software engineering related issues are perhaps the most critical issues which made earlier generation algebra systems hard to extend.

7. A Simple Collection of Algebraic Algorithms

In this section we demonstrate the expressive power of Newspeak through an extended example. Our goal is to write algorithms for polynomial GCD calculations in a general algebraic framework. Most of the code in this section just sets up the algebraic framework. In an actual algebra system, the framework would be defined once and then shared by all programs. Our type and procedure definitions are intentionally over-simple so that we can present and discuss a large number of them without getting bogged down in details.

We have been vague about the mechanisms by which *deftypes* and *defprocs* are essentially treated as Lisp defintions: put in one file and read into Newspeak. This is a detail left to the Newspeak system implementor. For the sake of the examples that follow, we can assume that all deftype and defprocs are put in one file and read in Newspeak. When a *deftype* in encountered in the file, Newspeak parses it, checks it for errors and if none are found, creates a new type-object in the hierarchy. *defprocs* are similarly checked for type consistency and then compiled, with the resulting code stored in a newly created <Function> object. The <Function> object is then added to the function database.

Figure 7.1 shows the type hierarchy that will be constructed by the declarations in this section. The types to be defined fall into two classes: representationless, and parameterized with a representation. The types on the left in the figure fall into the former class; the polynomial types on the right are of the latter class. We assume the existence of types <boolean> and <symbol> both restricting <object>, and of functions *not* over <booleans> and *eq* over <objects> (which we describe later).

7.1 Set, Monoid, Group

```
(deftype Set
   params: ( (=  (fcn (<Set> ~0) <boolean>)) )
   restricts: (( <object> )))      ; (this clause is not necessary)
```

The first type and function we define (<Set> and *!=*) have already been presented as examples (on pages 39 and 39) using different names (<Comparable> and *not=*). The names we use here, while less descriptive, are closer to the names used by Newspad and Andante. In less formal language, <Set> has one parameter, a function = which returns the <boolean> value true if given two equal objects. A mathematical set is a collection of objects with the property that no two are identical. The test for equality performed by the = function should correspond to the set theoretic notion of 'identical'.

```
(defproc != ((x <Set>) (y ~x)) <boolean>
   (not (= x y)))
```

!= is the "not equal" function. The = function which it calls is guaranteed to exist for <Set> arguments because we have defined the = parameter for <Set>.

```
(deftype Monoid
   params: ( (=    (sub-fcn <Monoid> <Set> =))
             (*   (fcn (<Monoid> ~0) ~0))
             (^   (fcn (<Monoid> <integer>) ~0))
             (f1? (fcn (<Monoid>) <boolean>)))
   restricts: ( (<Set>) )
   dist: ((zero) (one)))
```

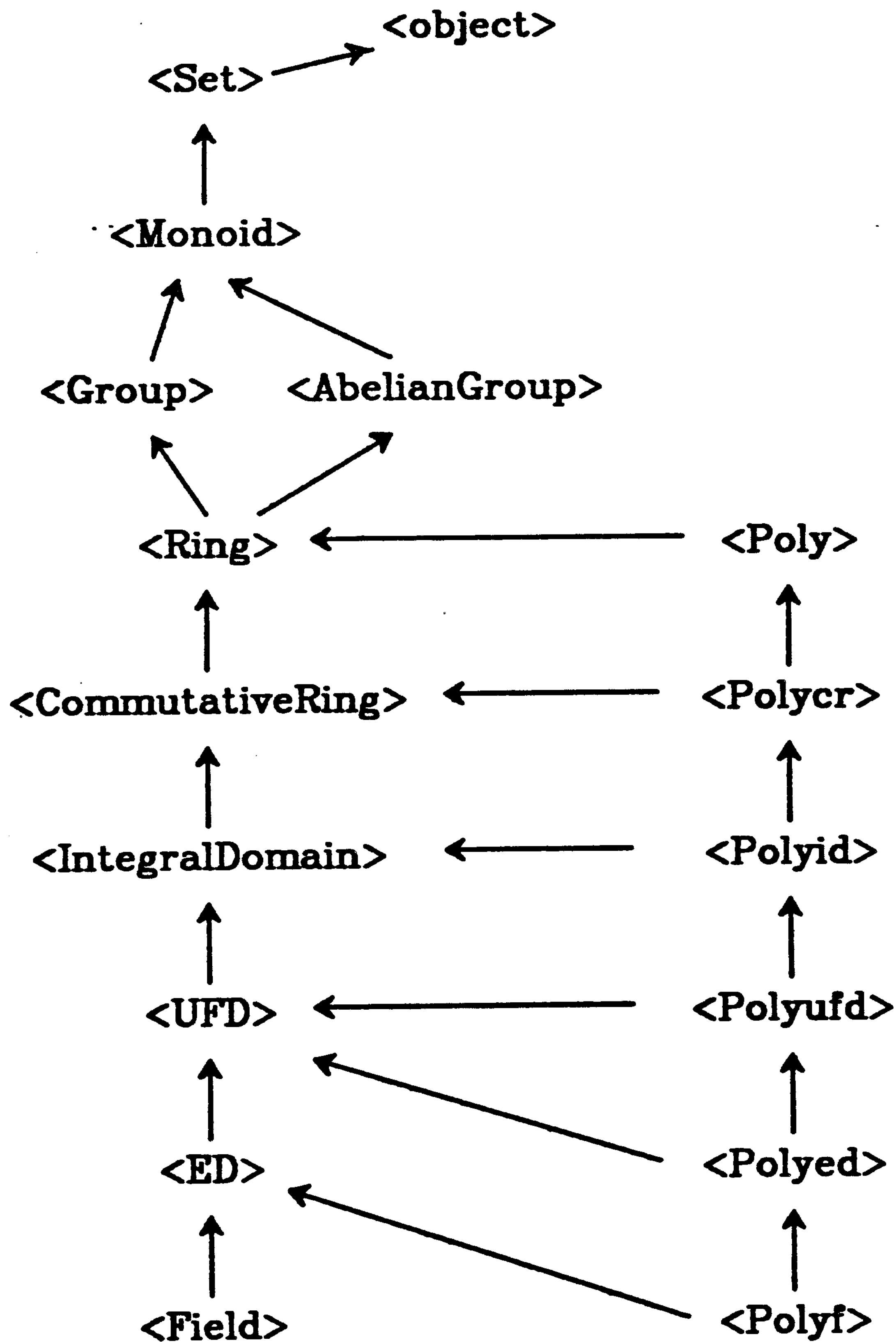


Figure 7.1

The form

```
(sub-fcn <Monoid> <Set> =)
```

a shorthand for, in this instance,

```
(fcn (<Monoid> ~0) <boolean>)
```

It is a declaration of a <Function> 'art' that has the same domain and range as the = functional parameter of <Set> with all occurrences of <Set> replaced with <Monoid>.

Because the restricts clause for <Monoid> doesn't specify a value for the <Set>'s = parameter, Newspeak makes Monoid's = parameter equivalent to the <Set>'s = parameter. This is desirable because the same function can be used for testing for equality, regardless of whether an object is viewed as a <Set> or a <Monoid>.

<Monoid> declares three new functional parameters: * for multiplication, ^ for repeated multiplication and *f1?*, a predicate which returns true if its argument is the multiplicative identity. Two distinguished elements, *zero* and *one*, are declared. The algebra monoid has only one distinguished element, the identity element, so it may appear wrong (or at least unnecessary) that two distinguished elements are declared for <Monoid>. It is done in anticipation of the types which restrict <Monoid>: The operator in a monoid, which we've named *, is treated as multiplication in some contexts, and as addition in others. We declare a *zero* distinguished element for when the operation * is considered addition and a *one* distinguished element for when it is considered as multiplication. For a given domain which restricts <Monoid>, only one of the distinguished elements would be used.

```
(deftype Group
   params: ((=   (sub-fcn <Group> <Monoid> =))
            (*   (sub-fcn <Group> <Monoid> *))
            (^   (sub-fcn <Group> <Monoid> ^))
            (f1? (sub-fcn <Group> <Monoid> f1?))
            (inv (fcn (<Group>) ~0)) )
   restricts: (( <Monoid> )))
```

The type <Group> is a restriction of <Monoid> with the addition of a functional parameter *inv* for the function which returns the inverse of an element. Because <Group> restricts <Monoid>, the existence of distinguished elements *zero* and *one* is inherited. In fact, all of the types we will define below restrict <Monoid>, so each inherits the existence of *zero* and *one*.

```
(deftype AbelianGroup
   params: ( (=   (sub-fcn <AbelianGroup> <Monoid> =))
             (+   (sub-fcn <AbelianGroup> <Monoid> *))
             (n*  (sub-fcn <AbelianGroup> <Monoid> ^))
             (f0? (sub-fcn <AbelianGroup> <Monoid> f1?))
             (-   (fcn (<AbelianGroup>) ~0)) ) ;unary inverse
   restricts: ( (<Monoid> (* + ) (^ n*) (f1? f0?)) ))
```

<AbelianGroup> is similar to <Group> except for two things: the operation is typically written as + instead of *, and the operation is commutative (which is not represented explicitly). This renaming of functions is evident in the restricts clause. If an object is viewed as an <AbelianGroup> and the + function is called, that will invoke the same function as if the object were viewed as a <Monoid> and the * function were called.

The repeated multiplication function ^ has been renamed *n** to indicate repeated addition. Also, *f1?* has been renamed *f0?*.

The -- parameter is the unary minus function.

```
(defproc - ((x <AbelianGroup>) (y ~x)) ~x
   (+ x (- y)))
```

This shows how subtraction is written using addition and unary inverse. The calls to + and -- will be compiled as parameterized generic function calls.

7.2 Ring, Integral Domain, UFD

```
(deftype Ring
   params: ( (=   (sub-fcn <Ring> <Monoid> =))
             (*   (sub-fcn <Ring> <Monoid> *))
             (^   (sub-fcn <Ring> <Monoid> ^))
             (f1? (sub-fcn <Ring> <Monoid> f1?))
             (+   (sub-fcn <Ring> <AbelianGroup> + ))
             (n*  (sub-fcn <Ring> <AbelianGroup> n*))
             (f0? (sub-fcn <Ring> <AbelianGroup> f0?))
             (-   (sub-fcn <Ring> <AbelianGroup> --))
             (exact-quotient  (fcn (<Ring> ~0) ~0))
             (unit-normal  (fcn (<Ring>) ~0))
             (unit-part    (fcn (<Ring>) ~0))
             (char  (fcn (<Ring>) <integer>))
             )

   restricts: (( <Monoid> )
               ( <AbelianGroup> )))
```

<Ring> corresponds to the possibly non-commutative algebra ring. It is the combination of a <Monoid> and an <Abelian group with the addition of a few parameters.

Recall that an element of a ring is a *unit* if it has an inverse in the ring. Two elements are *associates* if one is equal to the other multiplied by a unit. Within a set of associates, one of the members is called the *unit-normal* associate. The functional parameter *unit-normal* returns the unit-normal associate of the <Ring> object passed as a parameter. Any element of ring can be written as the product of its unit-normal and a unit. The functional parameter *unit-part* has this relationship with the *unit-normal* functional parameter: X = *unit-part(X)* * *unit-normal(X)*.

The *char* functional parameter returns the characteristic of the <Ring>. *exact-quotient* returns the quotient if exact division is possible or else it signals an error.

```
(deftype CommutativeRing
   params: ( (=   (sub-fcn <CommutativeRing> <Ring> =))
             (*   (sub-fcn <CommutativeRing> <Ring> *))
             (^   (sub-fcn <CommutativeRing> <Ring> ^))
             (f1? (sub-fcn <CommutativeRing> <Ring> f1?))
             (+   (sub-fcn <CommutativeRing> <Ring> + ))
             (n*  (sub-fcn <CommutativeRing> <Ring> n*))
             (f0? (sub-fcn <CommutativeRing> <Ring> f0?))
```

```
                    (-  (sub-fcn <CommutativeRing> <Ring> -))
                    (exact-quotient  (sub-fcn <CommutativeRing> <Ring>
                                              exact-quotient))
                    (unit-normal  (sub-fcn <CommutativeRing> <Ring>
                                           unit-normal))
                    (unit-part  (sub-fcn <CommutativeRing> <Ring>
                                         unit-part))
                    (char  (sub-fcn <CommutativeRing> <Ring> char)))

   restricts: (( <Ring> )))
```

<CommutativeRing> introduces nothing beyond <Ring> that we can represent explicitly in NEWSPEAK; however it a type with the additional property that * is commutative.

```
(deftype IntegralDomain
   params: ( (=    (sub-fcn <IntegralDomain>
                            <CommutativeRing> =))
             (*    (sub-fcn <IntegralDomain>
                            <CommutativeRing> *))
             (^    (sub-fcn <IntegralDomain>
                            <CommutativeRing> ^))
             (f1?  (sub-fcn <IntegralDomain>
                            <CommutativeRing> f1?))
             (+    (sub-fcn <IntegralDomain>
                            <CommutativeRing> + ))
             (n*   (sub-fcn <IntegralDomain>
                            <CommutativeRing> n*))
             (f0?  (sub-fcn <IntegralDomain>
                            <CommutativeRing> f0?))
             (-    (sub-fcn <IntegralDomain>
                            <CommutativeRing> -))
             (exact-quotient (sub-fcn <IntegralDomain>
                                      <CommutativeRing>
                                      exact-quotient))
             (unit-normal  (sub-fcn <IntegralDomain>
                                    <CommutativeRing>
                                    unit-normal))
             (unit-part  (sub-fcn <IntegralDomain>
                                  <CommutativeRing>
                                  unit-part))
             (char  (sub-fcn <IntegralDomain>
                             <CommutativeRing> char)))

   restricts: (( <CommutativeRing> )))
```

<IntegralDomain>'s are <CommutativeRing>'s with the additional property that zero divisors do not exist. Again, this property cannot be represented explicitly.

```
(deftype UFD
   params: ( (=    (sub-fcn <UFD> <IntegralDomain> =))
             (*   (sub-fcn <UFD> <IntegralDomain> *))
             (^   (sub-fcn <UFD> <IntegralDomain> ^))
             (f1? (sub-fcn <UFD> <IntegralDomain> f1?))
             (+    (sub-fcn <UFD> <IntegralDomain> + ))
             (n*  (sub-fcn <UFD> <IntegralDomain> n*))
             (f0? (sub-fcn <UFD> <IntegralDomain> f0?))
             (-  (sub-fcn <UFD> <IntegralDomain> -))
             (exact-quotient  (sub-fcn <UFD> <IntegralDomain>
                                          exact-quotient))
             (unit-normal  (sub-fcn <UFD> <IntegralDomain>
                                          unit-normal))
             (unit-part  (sub-fcn <UFD> <IntegralDomain> unit-part))
             (char  (sub-fcn <UFD> <IntegralDomain> char))
             (gcd  (fcn (<UFD> ~0) ~0)))

   restricts: (( <IntegralDomain> )))
```

In a unique factorization domain (UFD) the factorization of any element is unique up to associates. Furthermore, there exists a function *gcd* which returns the unique unit-normal greatest common divisor (GCD) of two elements.

7.3 Euclidean Domain

```
(deftype ED
   params: ( (=    (sub-fcn <ED> <UFD> =))
             (*   (sub-fcn <ED> <UFD> *))
             (^   (sub-fcn <ED> <UFD> ^))
             (f1? (sub-fcn <ED> <UFD> f1?))
             (+    (sub-fcn <ED> <UFD> + ))
             (n*  (sub-fcn <ED> <UFD> n*))
             (f0? (sub-fcn <ED> <UFD> f0?))
             (-  (sub-fcn <ED> <UFD> -))
             (unit-normal  (sub-fcn <ED> <UFD> unit-normal))
             (unit-part    (sub-fcn <ED> <UFD> unit-part))
             (exact-quotient  (sub-fcn <ED> <UFD> exact-quotient))
             (char  (sub-fcn <ED> <UFD> char))
             (deg<  (fcn (<ED> ~0) <boolean>))
             (quorem  (fcn (<ED> ~0) (values ~0 ~0))))

   restricts: ((<UFD>)))
```

In a Euclidean domain there is an algorithm for computing GCD's. Thus we define the function *gcd* (directly below) rather than make it a functional parameter. The *gcd* function invokes two functions, *deg<* and *quorem*, which cannot be written for the general Euclidean domain and are added as functional parameters to <ED>. The *deg<* function returns true if the Euclidean degree of the first argument is less than that of the second. We chose to use *deg<* rather than a function returning the actual degree for two reasons. The actual value of the degree is often not important. It is only important if an element has a smaller degree than another element. Also,

the degree of zero is negative infinity and we don't want to deal with the problem of representing that value in this simple example. The *quorem* function returns two values, the quotient and the remainder upon division.

This following function is the Euclidean greatest common divisor algorithm. It returns the unique unit-normal GCD of its two arguments. This is the first non-trivial program we've presented and there are many unfamiliar forms in it. We will go through it step by step.

```
(defproc gcd ((x y <ED>)) ~x
   (if (deg< x y)
      then (setq (x y) (values (unit-normal y) (unit-normal x)))
      else (setq (x y) (values (unit-normal x) (unit-normal y))))
   (do ((q (dist null ~x))
        (r (dist null ~x)))
       ((f0? y)
        x)
       (setq (q r) (quorem x y))
       (setq (x y) (values y (unit-normal r)))))
```

The types of the arguments are expressed in an abbreviated form. The first line is equivalent to

```
(defproc gcd ((x <ED>) (y ~x)) ~x
```

This is the familiar form which states that x and y have the same type and that type is <ED> (or some restriction).

The *setq, do, if* and *values* forms are modelled after like-named functions in Lisp. In NEWSPEAK they are handled by the compiler rather than being called as functions, thus these are not considered generic function calls. The only functions that *gcd* calls are *deg<*, *unit-normal, quorem* and *f0?*. All four are called as parameterized generic functions.

The form of the *if* statement is self-evident. The predicate must return a <boolean> result. In languages such as Lisp and C the predicate can return any type of value, with all values but one (nil in Lisp, zero in C) meaning 'true'. This leads to problems like the one shown in this C fragment:

```
if( x = y ) {
```

The = operator is 'assignment', not the 'equivalence' (represented ==) that was probably desired. It is a valid statement, nevertheless, so the C compiler will not flag it. In a NEWSPEAK program, to test a value against zero, the test must be written explicitly. This is a bit more work but it makes the predicate clearer and eliminates pitfalls such as the one just shown.

In the *gcd* function, the *if* statement insures that the degree of x is not less than the degree of y and converts x and y to unit-normal form. The conversion is not necessary until the end of the algorithm, but is done here (and each time around the loop) to keep the coefficient size down (a case of importance when x and y are polynomials).

The *values* statement is best understood by explanation of its implementation: it produces multiple values on the stack. The *setq* statement takes one or more values from the stack and places them in local variables. For example, the statement

```
(setq (x y) (values y x))
```

interchanges the values of x and y: first the values of y and x are stacked, then they are stored into x and y.

The *do* statement, adopted from Maclisp because of its generality, has this form:

```
(do ((var1 init1 repeat1)
     (var2 init2 repeat2))
    (end-predicate result-value)
    body1
    body2
     ...
    bodyn)
```

First new local variables *vari* are created and initialized to the values of the *initi*. The types of the values returned by the *initi* implicitly determine the types of *vari* to the compiler. Next the *end-predicate* is evaluated and if true the *result-value* is evaluated and returned from the *do*. If the *end-predicate* returns false, the *bodyi* forms are evaluated. Next, if there are any *repeati* forms, they are evaluated and their results placed in the *vari*, otherwise the value of the *vari* are unchanged. This is one pass through the loop. Evaluation returns to the *end-predicate* and continues though the loop until the *end-predicate* is satisfied or until a *return-from-do* statement is executed from within the body.

In this particular example, local variables q and r are initialized to the value of the *null* object of x's type. q and r are implicitly declared to be the same type as x. The next clause, *((f0? y) x)*, contains the end-predicate, *(f0? y)*, and result-value, *x*. This clause tests if the value is y is zero and if so returns the value of x from the *do* statement (and then from the *gcd* function). In the body of the *do*, the quotient and remainder of x divided by y are calculated and assigned to q and r. Next x takes on y's value and y takes on the unit-normal part of the remainder. The last setq could have been written

```
(setq x y)
(setq y (unit-normal r))
```

It was written as it is for stylistic reasons.

We illustrate more features via the extended Euclidean algorithm.

```
(defproc eeuclid ((a b <ED>)) (values ~a ~a ~a)
   (if (deg< a b)
     then (bind (((res1 res2 res3) (eeuclid b a)))
              (values res2 res1 res3))
     else (do (((c c1 c2) (values (unit-normal a)
                                  (dist one ~a)
                                  (dist zero ~a)))
               ((d d1 d2) (values (unit-normal b)
                                  (dist zero ~a)
                                  (dist one ~a)))
               ((q r r1 r2) (values (dist zero ~a)
                                    (dist zero ~a)
                                    (dist zero ~a)
                                    (dist zero ~a))))
              ((f0? d) (values
                          (exact-quotient
                             c1
                             (* (unit-part a) (unit-part c)))
                          (exact-quotient
```

```
                              c2
                              (* (unit-part b) (unit-part c)))
                           (unit-normal c)))
                 (setq (q r) (quorem c d))
                 (setq (r1 r2) (values (- c1 (* q d1))
                                       (- c2 (* q d2))))
                 (setq (c c1 c2) (values d d1 d2))
                 (setq (d d1 d2) (values r r1 r2)))))
```

The extended Euclidean algorithm returns values s, t and q such that q is the GCD and $s*a+t*b=q$. The particular form of this algorithm and the choice of variable names is taken from Algorithm 2.2 of [Geddes82]. The only new form introduced in this procedure is *bind*. It is used to introduce and initialize new local variables, much like *do*'s initialization part.

The *if* statement insures that a's Euclidean degree is not less than b's. If it is, *eeuclid* is called recursively and the return values swapped so the relation mentioned above exists between the result values.

The *do* statement above introduces ten local variables. They are initialized in three groups purely for stylistic reasons, apparent by comparison with the math description.

```
(deftype Field
   params: ( (=    (sub-fcn <Field> <ED> =))
             (*   (sub-fcn <Field> <ED> *))
             (^   (sub-fcn <Field> <ED> ^))
             (f1? (sub-fcn <Field> <ED> f1?))
             (+    (sub-fcn <Field> <ED> + ))
             (n*  (sub-fcn <Field> <ED> n*))
             (f0? (sub-fcn <Field> <ED> f0?))
             (-   (sub-fcn <Field> <ED> -))
             (print (sub-fcn <Field> <ED> print))
             (char (sub-fcn <Field> <ED> char))
             (inv* (fcn (<Field>) ~0)))

   restricts: (( <ED> )))
```

In a field, all elements except zero are units. Thus we add *inv**, the multiplicative inverse function, as a parameter. There are only two unit-normals elements: zero and one, making it possible to write the *unit-normal* and *unit-part* functions for all <Field>'s. Other functions are equally trivial to write over <Field>'s, as shown below.

```
(defproc unit-normal ((x <Field>)) ~x
   (if (f0? x)
      then (dist zero ~x)
      else (dist one ~x)))
```

If a value is not zero then its unit-normal associate is one, otherwise it is zero.

```
(defproc unit-part ((x <Field>)) ~x
   (if (f0? x)
      then (dist one ~x)
      else x))
```

If a value is zero, then we arbitrarily return the value one as its unit-part to make it possible for functions to safely divide by the unit-part.

```
(defproc gcd ((x y <Field>)) ~x
   (dist one ~x))
```

gcd is the unit-normal GCD of its arguments. Because the divisor must be non-zero, and the unit-normal for any non-zero element is one in a field, we always return one for the *gcd* of two fields elements.

```
(defproc exact-quotient ((x y <Field>)) ~x
   (* x (inv* y)))

(defproc quorem ((x y <Field>)) (values ~x ~x)
   (values (exact-quotient x y) (dist zero ~x)))
```

The remainder is always zero because exact division is always possible.

```
(defproc deg< ((x y <Field>)) <boolean>
   (dist false <boolean>))
```

7.4 Polynomial

All of the types defined up to now in this extended example are representationless. They form the backbone of the type hierarchy: any type, regardless of its representation, can restrict a backbone type as long as it has the required properties.

Now we introduce the polynomial type parameterized by an indeterminate (a <symbol> object) and a coefficient domain (a <type>). The type of the coefficient domain determines where the polynomial type attaches itself to the backbone of the type-hierarchy.

```
(deftype Poly
   params: ((coefdom <= <Ring>) (var <symbol>))
   lex: ((coef coefdom)
         (exp  <integer>)
         (rest _self))
   dist: ((null coef (dist zero coefdom)
                exp  (dist zero <integer>))
          (zero (sameas null))
          (one coef (dist one coefdom)
               exp  (dist zero <integer>)))
   restricts: (( <Ring> )))
```

A polynomial object is represented as a linked list of coefficients and exponents. The exponents are in decreasing order with the list terminated by the *null* object. The type of the coefficient is determined by the *coefdom* parameter of <Poly>.

Our polynomial representation is similar to the rational function form of polynomials in Macsyma. Maple uses hash tables, a distinctly different method.

The distinguished object list contains initialization expressions for the distinguished objects. The clause

```
(zero (sameas null))
```

means that *zero* and *null* will name the same object. As a result a polynomial may be considered to terminate with a *zero* rather than a *null* object, making functions like = below look clearer. We consider the equivalence of distinguished objects to be an interim solution, necessary because NEWSPEAK does not (yet) permit a program to conveniently select the list terminating object.

Because <Poly> restricts <Ring>, there are a number of functions we must write:

```
(defproc f1? ((x <Poly>)) <boolean>
   (= x (dist one ~x)))

(defproc f0? ((x <Poly>)) <boolean>
   (eq x (dist zero ~x)))
```

f1? tests if its argument is the multiplicative identity of the ring <Poly>. *f0?* tests if its argument is the additive identity. Notice that *f1?* uses = whereas *f0?* uses *eq*. *eq* is a function defined over <object>s which returns true if the arguments are the exact same object (because objects are referenced by pointers, this is done by checking if the pointer values are the same). The = function, analogous to *equal* in Lisp, is more complicated: it checks whether its arguments represent the same value, generally by recursively checking if the lex fields represent the same value. The = function for <Poly> is defined next.

The code for <Poly> operations insures that the zero polynomial is always represented by the *zero* distinguished object, thus *f0?* can use *eq* instead of the slower =. The polynomial whose value is one is not always represented by the distinguished object *one* (although we could have written the code to make this so). Thus we must use = to test for the value one.

```
(defproc = ((x y <Poly>)) <boolean>
   (if (f0? x)
      then (f0? y)
    elseif (f0? y)
      then (dist false <boolean>)
    elseif (= x:coef y:coef)
      then (if (= x:exp y:exp)
              then (= x:rest y:rest)
              else (dist false <boolean>))
      else (dist false <boolean>)))
```

The = function recursively checks if the coefficients and exponents are equal in the polynomials x and y. It is a good example of the expressive power of generic function calls. Note that there are

three calls to = within the procedure. The first call, checking the equality of the coefficients, will require a parameterized generic function call. The second call, that of the exponents, is a call to the = function over <integer> objects - a function which can be determined at compile time (and perhaps even open compiled). The third call to = is a recursive call to the procedure being defined. Such a call is *tail recursive*, that is the value returned by this recursive invocation of = is returned by the original call to =. We can expect to replace the tail recursive call by a little variable juggling and a jump to the top of the = function, in a good implementation.

Each call to = is automatically handled in a different and efficient way and the various mechanisms used are invisible to the programmer who simply writes his code in the most obvious way.

```
(defproc + ((x y <Poly>)) ~x
  (if (f0? x)              ; if either argument is zero
    then y                 ; return the other argument
   elseif (f0? y)
    then x
    else ; we only have to add if we find terms with
        ; the same exponent
        (if (> x:exp y:exp)
          then (new ~x
                  coef x:coef
                  exp  x:exp
                  rest (+ x:rest y))
         elseif (> y:exp x:exp)
          then (new ~x
                  coef y:coef
                  exp  y:exp
                  rest (+ x y:rest))
          else (bind ((tempval (+ x:coef y:coef)))
                 ; check for the case of the coefficients
                 ; canceling. Don't include terms with zero
                 ; coefficients and non zero exponents
                 (if (f0? tempval)
                   then (+ x:rest y:rest)
                   else (new ~x
                          coef (+ x:coef y:coef)
                          exp  x:exp
                          rest (+ x:rest y:rest)))))))
```

The polynomial addition routine makes use of the property that the exponents are in decreasing order. The polynomial it constructs as an answer may contain new objects and may share parts of an argument's object.

```
(defproc - ((x <Poly>)) ~x
  (if (f0? x)
    then x
    else (new ~x
            coef (- x:coef)
```

```
                    exp  x:exp
                    rest (-- x:rest))))
```

The function -- negates the polynomial by recursing down the polynomial negating the coefficients.

```
(defproc term* ((coef x::coefdom) (exp <integer>) (x <Poly>)) ~x
   (if (f0? x)
     then x
     else (new ~x
             coef (* x:coef :coef)
             exp  (+ x:exp  :exp)
             rest (term* coef exp x:rest))))
```

*term** multiplies a polynomial by a monomial implicitly given as the first two arguments. This is a utility function used by polynomial *.

```
(defproc term/ ((coef x::coefdom) (exp <integer>) (x <Poly>)) ~x
   (if (f0? x)
     then x
     else (new ~x
             coef (exact-quotient x:coef :coef)
             exp  (- x:exp  :exp)
             rest (term/ coef exp x:rest))))
```

term/ divides a polynomial by monomial. The monomial should divide evenly. This is also a utility function so it doesn't check to make sure that the exponents in the newly created polynomial objects are non-negative.

```
(defproc * ((x y <Poly>)) ~x
   (if (f0? x)              ; if either multiplicand is zero, just return zero
     then x
    elseif (f0? y)
       then y
     else ; otherwise multiply y by leading coef of x and add result
          ; the the result of multiplying the rest of x times y.
          (+  (term* x:coef x:exp y)
              (* x:rest y))))
```

In this polynomial multiplication function, we treat one polynomial as a set of monomials and find the sum of the product of the each monomial with the other polynomial. This is a somewhat inefficient method, we use it here for its simplicity.

```
(defproc unit-normal ((x <Poly>)) ~x
   (exact-quotient x (unit-part x)))

(defproc unit-part ((x <Poly>)) ~x
   (new ~x
        coef (unit-part x:coef)
        exp 0))
```

The units of the polynomial type are the units of the coefficient domain.

```
(defproc char ((x <Poly>)) <integer>
   (char x:coef))
```

The characteristic of the polynomial type is the characteristic of the coefficient domain.

```
(defproc degree ((x <Poly>)) <integer>
   (if (f0? x)
     then (error |degree of zero poly|)
     else x:exp))
```

The degree of the polynomial is the exponent of the leading term. If the polynomial is zero, the degree is commonly given as minus infinity. Since we don't want to worry about adding infinities to the integer type, we consider asking the degree of a zero polynomial to be an error.

```
(defproc deg< ((x y <Poly>)) <boolean>
   (if (f0? x)
     then (not (f0? y))
     else (< x:exp y:exp)))
```

```
(defproc exact-quotient ((x y <Poly>)) ~x
   (if (f0? y)
     then (error |exact-quotient by zero|)
    elseif (f0? x)
     then x
    elseif (deg< x y)
     then (error |exact-quotient can't be done|)
     else (do ((q (dist null ~x)))
              ((deg< x y)
               (if (not (f0? x))
                 then (error |exact-quotient not exact |))
               :q)
              (bind ((quot (new ~x
                                coef (exact-quotient x:coef y:coef)
                                exp  (- x:exp y:exp)))
                     (rem (- x (* quot y))))
                (setq q (+ q quot))
                (setq x rem)))))
```

exact-quotient should only be called when it is known that the division is possible. The division is performed by repeated subtraction.

```
;; polynomial over a commutative ring
(deftype Polycr
   params: ((coefdom <= <CommutativeRing>) (var <symbol>))
   restricts: (( <CommutativeRing> )
```

```
               ( <Poly> )))

;; polynomial over an integral domain
(deftype Polyid
   params: ((coefdom <= <IntegralDomain>) (var <symbol>))
   restricts: (( <IntegralDomain> )
               ( <Polycr> )))

;; polynomial over a unique factorization domain
(deftype Polyufd
   params: ((coefdom <= <UFD>) (var <symbol>))
   restricts: (( <UFD> )
               ( <Polyid> )))
```

When we restrict the coefficient domain, the polynomials take on more properties. A polynomial over a commutative ring is a commutative ring itself. All of the functions defined over <Poly> also work over <Polycr>, <Polyid> and <Polyufd>.

```
(defproc content-recur ((so-far poly::coefdom) (poly <Polyufd>))
                        poly::coefdom
   (if (or (f1? so-far) (f0? poly))
      then so-far
      else (content-recur (gcd so-far poly:coef)
                          poly:rest)))

(defproc content ((x <Polyufd>)) x::coefdom
   (if (f0? x)
      then (dist one x::coefdom)
      else (content-recur x:coef x:rest)))
```

When the coefficients of a polynomial restrict a unique factorization domain, we can use the fact that the *gcd* function is defined over the coefficients. The *content* function returns the GCD of the coefficients (notice that the value returned is in the coefficient domain).

```
(defproc pp ((x <Polyufd>)) ~x
   (if (f0? x)
      then x
      else (term/ (content x) 0 x)))
```

The primitive part (*pp*) of a polynomial is the purely polynomial part, that is, the polynomial with the content removed.

```
(defproc content-pp ((x <Polyufd>)) (values x::coefdom ~x)
   (if (f0? x)
      then (values (dist zero x::coefdom) (dist zero ~x))
      else (bind ((c (content x)))
              (values c (term/ c 0 x)))))
```

When computing the primitive part, the content is calculated. If a program requires both the content and the primitive part, calling *content-pp* is faster than calling *content* and *pp* separately since the content need only be calculated once.

```
(defproc pquorem ((x y <Polyufd>)) (values ~x ~x)
   (if (f0? y)
     then (error |pseudo poly divide by zero|)
     else (setq x (* x (new ~x
                          coef (^ y:coef (+ (- (degree x)
                                               (degree y))
                                            1))
                          exp 0)))
          (do ((q (dist null ~x)))
              ((deg< x y)
               (values q x))
              (bind ((quot (new ~x
                              coef (exact-quotient x:coef y:coef)
                              exp (- x:exp y:exp)))
                    (rem (- x (* quot y))))
                (setq q (+ q quot))
                (setq x rem)))))
```

pquorem returns the quotient q and remainder r of the pseudo division of x by y. Pseudo division differs from normal division is that the dividend is multiplied by the leading coefficient of the divisor enough times to insure that each division step will be exact (see page 2-27 of [Geddes82]).

```
(defproc gcd ((a b <Polyufd>)) ~a
   (if (f0? a)
     then b
    elseif (f0? b)
     then a
     else (if (deg< a b)
             then (setq (a b) (values b a)))
          (do (((c-cont c) (content-pp (unit-normal a)))
               ((d-cont d) (content-pp (unit-normal b)))
               (q (dist null ~a))
               (r (dist null ~a)))
              ((f0? d)
               (term* (gcd c-cont d-cont) 0 c))
              (setq (q r) (pquorem c d))
              (setq (c d) (values d (pp r))))))
```

Because <Polyufd> restricts <UFD>, it must supply a GCD function to satisfy the *gcd* parameter of <UFD>. This *gcd* function is the primitive polynomial remainder sequence Euclidean GCD (primitive PRS GCD) - algorithm 2.3 of [Geddes82].

```
;; polynomials over a Euclidean domain
(deftype Polyed
   params: ((coefdom <= <ED>) (var <symbol>))
```

```
   restricts: (( <UFD> )
               ( <Polyufd> )))

;; polynomials over a Field
(deftype Polyf
   params: ((coefdom <= <Field>) (var <symbol>))
   restricts: (( <ED> )
               ( <Polyed> )))
```

Polynomials over a field have additional useful properties resulting from to the divisibility of their coefficients. Because they restrict the Euclidean domain type (<ED>), they can use the Euclidean GCD (presented on page 63).

```
(defproc unit-normal ((x <Polyf>)) ~x
   (if (f0? x)
      then x
    elseif (f1? x:coef)
      then x      ; already monic
      else (exact-quotient x (new ~x
                                  coef x:coef
                                  exp  (dist zero ~x:exp)))))

(defproc unit-part ((x <Polyf>)) ~x
   (if (f0? x)
      then (dist one ~x)
      else (new ~x
                coef x:coef
                exp  (dist zero ~x:exp))))
```

In the <Polyf> type, all non-zero coefficients are units. The unit-normal polynomial is monic (if non-zero, the leading coefficient is one).

```
(defproc quorem ((x y <Polyf>)) (values ~x ~x)
   (if (f0? y)
      then (error |poly divide by zero|)
    elseif (f0? x)
      then (values x x)
      else (do ((q (dist null ~x)))
               ((deg< x y)
                (values q x))
               (bind ((quot (new ~x
                                 coef (exact-quotient x:coef y:coef)
                                 exp  (- x:exp y:exp)))
                      (rem (- x (* quot y))))
                 (setq q (+ q quot))
                 (setq x rem)))))
```

Because the coefficient domain is a field, we can do real division (instead of pseudo-division) without failure as long as the divisor is non zero.

```
(defproc exact-quotient ((x y <Polyf>)) ~x
   (bind (((q r) (quorem x y)))
      (if (not (f0? r))
         then (error |polyf exact quotient not exact |)
         else q)))
```

The *exact-quotient* function should only be called when it is known that the divisor evenly divides the dividend.

7.5 Using the Definitions

Now that the polynomial types are defined, a program can create and manipulate polynomials. In order to create type of polynomials in x over the Eudidean domain of integers, a specific type could be created with *deftype*, or the 'art' form could be used to create a nameless type: *(art <Polyed> <integer> x)*.

If the ring <Matrix> were defined, the type of "polynomials in x over matrices" could generated in a similar way: (art <Poly> <Matrix> x). Macsyma's standard programs are unable to work with such a data type because their polynomial operations assume coefficient commutativity.

7.6 Type Conversion

In algebraic algorithms such as the Hensel lifting algorithm (page 74 of [Fateman78]) it is necessary to convert a data object from one type to another. The target type may not be known at compile-time, in which case it is be necessary to create the type at run-time before doing the conversion. This task is difficult in the usual strongly-typed system, and thus our ability to handle this is a test of the convenience of NEWSPEAK. One way to do this is illustrated in the following example: We define the type of "polynomials in one variable over integers modulo n" and show how conversions are done between such types with different moduli.

```
(deftype PolyZmodn
   params: ((n <integer>)
            (var <symbol>))
   restricts: ((<Polycr> (coefdom (art <Zmodn> n)))))
```

<PolyZmodn> is parameterized by n, the modulus, and var, the indeterminant. The restricts clause contains a new form: *(art <Zmodn> n)*. This is a *delayed type expression.* When <PolyZmodn> is restricted and a value is supplied for n, the expression (art <Zmodn> n) will denote a type-object. For the purposes of verifying that (art <Zmodn> n) is a valid value for the coefdom parameter, NEWSPEAK assumes that the expression is simply <Zmodn>. We've assumed that <Zmodn> was defined to restrict <CommutativeRing> (otherwise NEWSPEAK will not permit this *deftype* because coefdom must be a type which restricts <CommutativeRing>).

The first conversion function defined is that between <Zmodn> types:

```
(defproc cvt ((x <Zmodn>) (y <Zmodn>)) ~y
   (new ~y (mod x:val y::n)))
```

The *cvt* function takes two objects whose types are possibly different restrictions of <Zmodn> and returns the first object converted to the type of the second object. The arguments to this

function may appear counterintuitive - one probably expects *cvt* to take as arguments an object and a desired type. The problem with passing the desired type is that no information is declared about the values the type can have: it can't be assumed that the type is a restriction of <Zmodn>, or even a domain-type. By passing in a representative object of the desired type: (1) it is assured that the desired type is a domain-type, (2) the parameters of the desired type can be accessed (they can't be accessed from the type-object itself by using the double-colon notation), and (3) the range of values the desired type can take on can be represented (e.g. that it restricts <Zmodn>).

A similar conversion function can be written for <PolyZmodn>:

```
(defproc cvt ((x <PolyZmodn>) (y <PolyZmodn>)) ~y
   (if (f0? x)
     then (dist zero ~y)
     else (new ~y
                coef (cvt x:coef y:coef)
                exp  x:exp
                rest (cvt x:rest y))))
```

There are two calls to *cvt* within this function: the first to a different *cvt* over <Zmodn> for the coefficient and the second a recursive call to *cvt* over <PolyZmodn>.

This function copies the entire polynomial, replacing each link in the linked-list with a link of the new type (and converting the types of the coefficients too). Copying is required because each link of the polynomial is 'tagged' with the polynomial's type. While this makes arithmetic functions easy to write recursively, it also makes type conversion more expensive. We discuss an alternative method of representing polynomials below.

An example of a function which uses the *cvt* function defined above is this:

```
(defproc lift-square ((x <PolyZmodn>)) <PolyZmodn>
   (cvt x (dist null (art <PolyZmodn> (* x::n x::n) x:var))))
```

The <PolyZmodn> object passed as an argument is converted to another <PolyZmodn> type, this one with a modulus which is the square of the modulus of the original object's type. The expression

```
(art <PolyZmodn> (* x::n x::n) x:var)
```

is another example of a delayed type expression. For the purpose of compilation, NEWSPEAK assumes that the type is <PolyZmodn> (thus it can successfully do the generic function lookup for the *cvt* function).

The final example is a function to do the first stage lift of the linear Hensel algorithm. (Lifting from p^n to p^{n+1} introduces no additional concepts but would be a somewhat longer program.) We are given $V(x)$ and $W(x)$, relatively prime monic polynomials with coefficients in the field of integers modulo a prime p (i.e. elements of $Z_p[x]$), and $U(x)$, an element of $Z[x]$. They have this relation:

$$V(x)\cdot W(x) \equiv U(x) \bmod p$$

We wish to determine polynomials $Vnew(x)$ and $Wnew(x)$ in $Z_{p^2}[x]$ such that

$$Vnew(x)\cdot Wnew(x) \equiv U(x) \bmod p^2$$

This defproc computes these polynomials:

```
(defproc hensel-lift
  ((u (art <Polyed> <integer>)) (v w <PolyZmodn>) (p <integer>))
  (values <PolyZmodn> <PolyZmodn>)

  (bind (; create a and b in Zp[x] such that
         ; a*v + w*u = 1   in Zp[x]
         ((a b dummy) (eeuclid v w))
         ; create null object of new type: Zp^2[x]
         (objnew (dist null (art <PolyZmodn> (^ p 2))))
         ; create new polynomials in Zp^2[x]
         ; from those in Zp[x].
         (aup (cvt a objnew))
         (bup (cvt b objnew))
         (vup (cvt v objnew))
         (wup (cvt w objnew))
         ; calculate c = ((v*w - u)/ p^2) in Zp^2[x]
         (c (term/ (- (* vup wup) (cvt u objnew))
                   (new ~objnew:coef val (^ p 2))
                   0))
         ; convert c from Zp^2[x] to  Zp[x]
         (cdown (cvt c a))
         ; determine the quotient (q) and remainder (anew) of
         ;   a*c/w   in Zp[x]
         ((q anew) (quorem (* a cdown) w))
         ; set bnew to b*c + q*v  in Zp[x]
         (bnew (+ (* b cdown) (* q v))))
    ; return Vnew(x) and Wnew(x)
    (values ; Vnew = v - p*bnew  in Zp^2[x]
            (+ vup
               (term* (cvt bnew objnew)
                      (new ~objnew:coef val (- p))
                      0))
            ; Wnew = w - p*anew   in Zp^2[x]
            (+ :wup
               (term* (cvt anew objnew)
                      (new ~objnew:coef val (- (^ p 2)))
                      0)))))
```

The arguments to *hensel-lift* are $U(x)$, $V(x)$, $W(x)$, and the prime p. The function must work with polynomials in three domains: $Z_p[x]$, $Z_{p^2}[x]$ and $Z[x]$. The *cvt* function we defined earlier is used to convert polynomials between domains. (Also, a *cvt* function from $Z[x]$ to $Z_n[x]$ is used even though it hasn't been defined in this section. It is very similar to the *cvt* function from $Z_n[x]$ to $Z_m[x]$ that we've already shown.) As we mentioned earlier, *cvt* creates a totally new polynomial, an operation that is expensive in both time and space, especially since most of the conversions are done between domains where the coefficients (viewed as integers) do not have to be recalculated (e.g. from $Z_p[x]$ to $Z_{p^2}[x]$). It is tempting, therefore, to maintain the modulus inside the polynomial object itself and represent the coefficents as <integer>'s. Then type conversion would be merely a destructive modification of the modulus value. In NEWSPEAK, the programmer may use such a technique but he is then taking on much of the type checking burden himself.

8. Conclusions

The goal of our research was the design of a language to support a math-oriented symbolic algebra system. While symbol-oriented algebra systems can solve many problems, their lack of mathematical rigor promotes blunders. Furthermore their lack of structure makes addition of new knowledge difficult.

In order to design a math-oriented symbolic algebra system, we need a language with the ability to represent the complex interrelations of mathematical types. The only languages with sufficiently powerful hierarchical data typing facilities lack the strict compile-time type checking that we feel is necessary for a large program such as a symbolic algebra system. As a result, NEWSPEAK was designed to combine type hierarchies *and* compile-time type checking. We discovered that the normal benefits of compile-time type checking (e.g. efficient execution and type security) can even be obtained in a language such as NEWSPEAK, where the precise types of variables are not known at compile-time. Strict typing also enhances the programmer's ability to specify the input and output data types for his functions.

For the particular domain of symbolic algebra systems, NEWSPEAK is especially appropriate. It has these important features:

- Its ability to represent the hierarchy of mathematical types enables one to write programs over the most general types and have those programs be inherited by appropriate types.

- Types may be parameterized by other types, permitting types such as "polynomials over a ring" to be described.

- The syntax for function calls is the same, whether a generic or parameterized generic call is being made. The user need not be aware of the difference.

- The language is type safe, permitting the compiler to generate code without run-time consistency checks.

- Types can be created at runtime. The algebra system constructed on top of NEWSPEAK will be interactive, and the user may want to extend the system by adding new data types.

- While the mathematical algorithms will be the core of the algebra system we plan to build, the outer layers will be made up of rather mundane programs which don't require a complex type hierarchy. The most important requirement for the mundane code is that it be compiled efficiently and not suffer from generalities introduced into the language to satisfy the mathematical programs. This is the case in NEWSPEAK.

8.1 Limitations

Due to its extensibility, NEWSPEAK doesn't suffer the major limitation of a non-extensible language: a fixed set of first-class data types. A limitation of NEWSPEAK which is shared by other high level languages such as Lisp and Smalltalk is that the precise form of data objects isn't a fixed part of the language. Thus NEWSPEAK would not be suitable for applications such as systems programming where the exact form of data objects is important. (One *could* write a program in another language which would transform a NEWSPEAK object into any necessary form, as has been done with Franz Lisp.)

8.2 Future work

We have described in this thesis the core of the NEWSPEAK language. Before it is completely implemented, a number of other issues must be resolved. One of the most important is insuring consistency between separately compiled modules. Changes in the type hierarchy, especially near the top, can affect modules which depend on types defined lower down. This will require maintaining a database of dependencies between modules and types.

Strict type checking is a powerful asset to NEWSPEAK but it is likely to be troublesome to the programmer. Type checking permits NEWSPEAK to move most of the cost associated with programming within a type-hierarchy to compile-time. The programmer may be intimidated by having to write programs which satisfy strict typing rules, especially if he is unclear about the characteristics of the types he is working on. It is vitally important to the success of the NEWSPEAK language as a tool for prototyping systems that a programming environment be built around the language. Such an environment would help the user with typing problems and with the debugging of his programs.

We must also establish rules which determine when an object should be treated as atomic and when its contents are visible. We cannot simply forbid programs outside of an object's type-defining module to see inside the object, because it may be necessary to write functions to convert from one type to another (e.g. polynomials from factored to unfactored form).

The parameters of type-objects are not mutable (alterable) by programs. There are times in an algebra system when mutable state variables are required. For example, when defining a sparse multivariate-polynomial type, we would like to have available an ordered list of the indeterminants in the polynomial, this list being subject to change when new variables are introduced. We plan on adding mutable type parameters to NEWSPEAK.

We also plan to add variant records, also known as union types. The sparse multivariate polynomials type just mentioned could use such a facility (where each coefficient is either a member of the coefficient domain or else a polynomial). A variant record is not required if the user is willing to allocate space for both types of values, and to insure that his program can tell which variant is valid. It is for the latter reason that we feel that NEWSPEAK, not the programmer, should handle variants records. It can insure that a program only accesses the valid variant field.

A large algebra system may be best written as a set of independent processes communicating via a byte stream or shared memory. While many operating systems provide multiprocessing facilities, the methods and capabilities vary from system to system. We do not want to tie NEWSPEAK to a particular operating system or machine so we will consider adding multiprocessing to NEWSPEAK itself.

When we proceed to a full scale implementation of NEWSPEAK, the language is certain to grow. It will be our goal to maintain the semantics that we have presented in this thesis.

Bibliography

Barton83. Barton, D., *private communication*, (1983)

Bourne71. Bourne, S. and Horton, J., *The Design of the Cambridge Algebra System*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 134-143

Char83. Char, B. et al, *The Design of Maple: A Compact, Portable and Powerful Computer Algebra System*, Research Report CS-83-06, University of Waterloo, (April 1983)

Cole81. Cole, C., Wolfram, S. et al, *SMP, A Symbolic Manipulation Program*, California Institute of Technology, (1981)

Collins71. Collins, G., *The Sac-I System: An Introduction and Survey*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 144-152

D'Ambrosio83. D'Ambrosio, B., *Smalltalk-80 Language Measurement-Dynamic Use of Compiled Methods*, Proceedings of CS292r, CS Division, University of California at Berkeley, (April 1983)

Davenport80. Davenport, J. and Jenks, R., *ModLisp*, Proceedings of the 1980 Lisp Conference, (1980), pp. 65-74

Engelman69. Engelman, C., *Mathlab 68*, Information processing 68, North-Holland, Amsterdam, (1969), pp. 462-467

Fateman78. Fateman, R., *CS 292s Draft notes*, (1978)

Foderaro81. Foderaro, J. and Fateman, R., *Characterization of VAX Macsyma*, Proceedings of 1981 ACM Symposium on Symbolic and Algebraic Computation, (1981), pp. 14-19

Foderaro82. Foderaro, J. and Sklower, K., *The Franz Lisp Manual*, CS Division, EECS Department, University of California at Berkeley, (1982)

Geddes82. Geddes, K., *Algebraic Algorithms for Symbolic Computation (draft notes)*, University of Waterloo, (Jan. 1982)

Goldberg83. Goldberg, A. and Robson, D., *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, (1983)

Griesmer71. Griesmer, J. and Jenks, R., *Scratchpad/1 - An Interactive Facility for Symbolic Mathematics*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 42-58

Hall71. Hall, A., *The Altran System for Rational Function Manipulation - A Survey*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 153-157

Hearn71. Hearn, A., *Reduce 2: A System and Language for Algebraic Implementation*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 128-133

Ichbiah79. Ichbian et al, *Rationale for the Design of the Ada Programming Langauge*, SIGPLAN Notices, ACM, (June 1979)

Jenks81. Jenks, R. and Trager, B., *A Language for Computational Algebra*, Proceedings of 1981 ACM Symposium on Symbolic and Algebraic Computation, pp. x-y

Kernighan78. Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice-Hall, (1978)

Knuth81. Knuth, D., *The Art of Computer Programming, Volume 2*, Addison-Wesley, (1981)

Martin67. Martin, W., *Symbolic Mathematical Laboratory*, MIT (MAC-TR-36), (1967)

Martin71. Martin, W. A. and Fateman, R. J., *The Macsyma System*, Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, (1971), pp. 59-75

Minsky75. Minsky. M., *A Framework for Representing Knowledge*, in P. H. Winston (Ed.) The Psychology of Computer Vision, McGraw-Hill, (1975)

Morris73. Morris, J., *Types are not Sets*, ACM Symposium on Principles of Programming Languages, (1973), pp. 120-124

Moses67. Moses, J., *Symbolic Integration*, MIT (MAC-TR-47), (1967)

Moses74. Moses, J., *Macsyma - The Fifth Year*, Proceedings Eurosam 74 Conference, (August 1974)

Novak82. Novak, G., *Glisp User's Manual*, CS Dept, Stanford University, (November 1982)

Orwell50. Orwell, G., *1984*, Harcourt, Brace and Co., (1950)

Roberts77. Roberts, R. and Goldstein, I., *The FRL Manual*, The MIT A.I. Laboratory, (1977)

Soiffer81. Soiffer, N., *A Perplexed User's Guide to Andante*, UCB internal memo, (December 1981)

Stallman83. Stallman, R., *personal communication*, (1983)

Steele77. Steele, G., *Data Representations in PDP-10 MacLisp*, Proceedings of the 1977 Macsyma Users' Conference, (1977), pp. 203-214

Teer78. Teer, F., *Formula Manipulation and Pascal*, University of Amsterdam, (1978)

Trager83. Trager, B., private communication, (1983)

VanDeRiet73. Van de Riet, R., *ABC Algol, A Portable Language for Formula Manipulation Systems*, Mathematisch Centrum, (1973)

Veltman65. Veltman, M., *Schoonschip, A CDC 6600 Programme for Symbolic Evaluation of Feynman Diagrams*, CERN, (1965)

Weinreb81. Weinreb, D. and Moon, D., *Lisp Machine Manual*, MIT, (1981)

Xenakis71. Xenakis, J., *The PL/1 - Formac Interpreter*, Proceedings of the Second Symposium on Sym-

bolic and Algebraic Manipulation, ACM, (1971), pp. 153-157

Zippel83. Zippel, R., *Capsules*, MIT, (1983)