A Typed, Applicative Programming Environment

Peter Grogono

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

February 1985

ABSTRACT

A Typed, Applicative Programming Environment

Peter Grogono, Ph.D.
Concordia University, 1985

The traditional tools, an editor, a compiler, and a debugger, are no longer adequate for program development. This perception has lead to the introduction of "programming environments" for the development of programs in various languages. In this thesis, we introduce a new kind of programming environment designed for the rapid production of reliable software. The environment is based on a family of high-level, applicative languages, collectively called Dee, that combine features traditionally associated with both compiled and interpreted languages. Distinctive features of the environment include an explicit multi-level structure, a rich type system, modular program construction, data abstraction, and incremental program development.

Acknowledgments

John Dee (1527-1608) was a British philosopher whose creative work spanned mathematics, logic, alchemy, and magic. The quotations at the beginning of each chapter of this thesis are taken from Dee's best known work, the preface to the first English translation of Euclid's **Elements of Geometry** [Dee].

I have received support, technical assistance, and encouragement from many sources during the preparation of this thesis. Concordia University awarded me a Graduate Fellowship and le Fonds F.C.A.C. provided further financial assistance. The Department of Computer Science at Concordia University provided a stimulating milieu in which to study and learn. I have been privileged to attend meetings of IFIP Working Groups during my research. I am grateful to Hendrik Boom for introducing me to WG 2.1, to Mark Rain for introducing me to WG 2.4, and to the Chairmen of these groups, Robert Dewar and Gerhart Goos respectively, for invitations to the meetings. Several members of these groups, in particular Bengt Nordstrom, Mark Rain, Paul Rouse, David Turner, and Paul Voda, have discussed the work described herein with me. Jonathan Seldin offered helpful comments on the type system of Dee. C.A.R. Hoare provided useful criticisms of an early version of Dee presented at ETH in Zurich [Grogono 1984]. The members of my Examining Committee, Hendrik Boom, Clement Lam, Jaroslav Opatrny, and Paul Voda have provided continued and welcome assistance. Finally, V.S.

Alagar, who supervised both my Master's Thesis and the present dissertation, has been a patient listener and critic as well as an invaluable source of ideas.

Table of Contents

"The fruit and gain which I require for these my pains and travail, shall be nothing else, but only that you, gentle reader, will gratefully accept the same: and that you may thereby receive some profit: and moreover to excite and stir up others learned, to do the like, and to take pains in that behalf."

1 Introduction

A programming **environment** is a collection of hardware and software that assists a programmer in the design and implementation of a program. We are accustomed to programming environments that support either a compiled language with an editor and a compiler or an interpreted language with an interpreter. In this thesis, we discuss the design of an environment that could assist programmers in the way that a text-processing system assists writers or a symbolic algebra system assists mathematicians. A programming environment must be based on a programming language or family of programming languages, and the design of the languages, in turn, must be based on a methodology. We begin by discussing strategies for programming language design.

1.1 Strategies and Principles for Programming Language Design

There are various strategies for the design of programming languages. The strategies vary in importance and generality, and not all of them can be applied to one language. Our discussions of strategies are of necessity brief, and much has been omitted. We do not, for example, discuss the roles of syntax and orthogonality in language design. We introduce several principles important for language design and discuss their advantages and disadvantages both in general and with reference to existing programming languages. The application of these principles to the design of a language helps to ensure that the language is in fact amenable to the production of verifiable and reliable programs.

1.1.1 Abstraction Mechanisms

The development of programming languages and programming methodologies used to be treated as orthogonal issues. The application of a particular method to a particular problem yielded an abstract program that could be hand-translated into any convenient language. Wirth embodied a methodology in Pascal [Jensen 1978], a language intended to facilitate top-down development with stepwise refinement [Wirth 1971]. This methodology has been widely accepted but has resisted formalization [Vessey 1984]. The introduction of abstract data types brought method and language still closer, but in this case formalization has been accomplished while practice has not yet been convincing.

The work on abstract data types focussed attention on the importance of **abstraction mechanisms** in programming languages. An abstraction mechanism serves three purposes. It supports **encapsulation**, that is, the separation of specification from implementation (information hiding [Parnas 1971]); it provides a method for naming and parametrizing objects; and it enables information to be **localized**. The built-in abstractions of a programming language should be based on mathematical concepts, such as functions, sets, and relations, rather than on machine-oriented concepts, such as goto-statements, assignments, and arrays. There should be a small number of fundamental ways in which programmers can construct their own abstractions from the built-in abstractions of the language.

All languages provide some form of function abstraction. The ubiquity of functional abstraction in programming languages was recognized early by McCarthy, Landin, and Strachey [McCarthy 1960] [Landin 1965] [Strachey 1966]. McCarthy's work lead to LISP [McCarthy 1962], and Strachey's ideas, coupled with Scott's theory of domains, lead to denotational semantics [Scott 1977]. APL [Iverson 1980] and LISP were the earliest languages founded on the mathematical concept of functionality. Languages such as LISP and SASL [Turner 1976] illustrate the role of higher-order functions in programming.

Set abstraction did not appear in programming languages until comparatively recently, although set notation has long been used for the description of algorithms. Languages such as CIP-L [Bauer 1983], KRC [Turner 1981], and SETL [Dewar 1981] provide set abstraction. A relation can be modelled by a function whose values are sets, but this leads to cumbersome notation. A better solution seems to be to incorporate relations into a language as a primitive construct, as in Prolog [Kowalski 1979]. Prolog, however, has the converse problem that functional dependencies cannot be expressed in a simple way. (There is a corresponding problem with functional dependencies in a relational database.) Combining functions and relations in a single language appears to be a fruitful topic for investigation [Voda 1984b].

Abstraction mechanisms must be introduced in a disciplined way. Rules for naming and referencing objects are particularly important. We must provide comprehensive management for the names introduced by abstraction by adopting simple scope rules and making provision for qualified names. (Early implementations of LISP were notably inadequate in this respect.)

The "uniform reference" principle requires that the denotation of an object should not depend on its implementation. As an example of uniform reference, we may consider selectors. In LISP, a selector always has the form (F X). In Pascal, however, we might use any of f(x), x[f], x.f, or x^.f, according to the representation chosen. On the other hand, a programmer should be able to provide multiple representations. This can be done in "flavoured" dialects of LISP and in some object-oriented languages.

An important characteristic of a programming language is the ease with which problems can be represented in it at an appropriate level of abstraction. The "level" of a language is determined to a large extent by its built-in abstractions. CIP-L, Mary2 [Rain 1984], and SETL are "wide-spectrum" languages that support the expression of programs at all stages of development, from specification to low-level, imperative code. The designer of a wide-spectrum language has to provide either an extremely rich environment or a meta-programming language that enables programmers to construct their own task-specific environments. Although LISP is not a wide-spectrum language, experience with LISP has shown that transformations are simplified if the problem is expressed in a language that can be used as its own metalanguage [Sandewall 1978]. An alternative strategy for very high-level languages is to exclude low-level features from the language altogether. If the programmer has no low-level control, the system must be capable of making its own representation and implementation decisions.

Multi-level systems provide an alternative to wide-spectrum language design. Berkling has proposed a system with five levels [Berkling 1981].

1. Algebra, transformations, partial evaluation, naming, higher order functions, variable-free programming.
2. lambda calculus, beta-reduction, literal substitution.
3. Expressions, trees, pre-order representation and traversal.
4. Constructors, atoms, lambda-variables, primitive functions, definitions.
5. Stacks.

Dee, the principal topic of this thesis, is also a multilevel system. Comparing Berkling's systems to Dee, we find that, roughly, L_1 in Dee corresponds to Berkling's levels 1 and 2, L_2 to levels 2, 3, and 4, and L_4 to level 5. (The significance of L_1, L_2,... is explained Section 1.2.)

1.1.2 Values and Objects

Mathematics deals with values. Values are abstractions: they cannot be created, destroyed, or changed. Computers cannot process values. Within the computer, a value is represented by an object. Objects can be created, destroyed, and changed. Most programming languages confuse values and objects. Maintaining a clear distinction between values and objects contributes to clear language design [MacLennan 1982].

The designer of a programming language has the choice of providing access to computational objects or of concealing them and providing instead access only to their values. A programming language that has an assignment operator allows access to objects. The inclusion or exclusion of assignment, or an equivalent operation, has a number of consequences concerning the ease with which we can read and write programs and the ease with which the language can be implemented efficiently.

A pure applicative language provides only values. The objects representing values can be shared in an implementation [Hoare 1975]. Referential transparency and the lack of side-effects permits a wide class of transformations and implementation optimizations. Pure applicative languages have a serious disadvantage, however, in that they have very limited mechanisms for representing states. States can be passed as arguments to a function and passed again, perhaps in modified form, to recursive calls of the same function. States can also be simulated by continuations, but the notation is cumbersome and opaque.

Object-oriented languages, such as Smalltalk [Goldberg 1983], adopt the opposite approach. Objects, rather than their abstracted values, are visible to the programmer who can explicitly create and destroy them. Object-oriented languages undoubtedly provide a simpler, and perhaps more accurate, model of the real world, but the problems of transforming and verifying programs written in object-oriented languages have yet to be studied in depth.

The usual arguments against the use of objects are based on complexity and lack of referential transparency [Backus 1979], and those against the use of values are based on efficiency. Whatever the relative merits of these arguments, there remains the fact that programmers need both values and objects.

1.1.3 Semantics

It is common practice to design a programming language and then to provide a formal description of it. The formal description is presented in the form of a semantics for the language. There has been much debate as to the relative power of various descriptive methods, and "powerful" methods have been developed to describe bizarre constructions such as jumps into a block.

A semantics ascribes a meaning to programs. It can do this by describing either what the program does or how it does it. Algebraic and axiomatic semantics belong to the first category ("what"), and operational semantics belongs to the second category ("how"). Denotational semantics bridges the gap: a denotational semantics can be as abstract or as operational as we require. The choice of semantics affects the difficulty of describing language features. Features that are easy to describe operationally tend to have straightforward implementations. Features that are easy to describe denotationally have pleasant mathematical properties but are often inefficient to implement. Examples of this dichotomy include the goto-statement in imperative languages and dynamic scoping and call-by-name in applicative languages.

Ashcroft and Wadge have made a strong case for the prescriptive use of denotational semantics [Ashcroft 1982]. There is as yet no widely-used programming language whose design was directed solely by denotational semantics. There are, however, several languages that were designed by constructing a semantics for operations on a particular class of denotable objects. These languages, which include APL, LISP, SNOBOL [Griswold 1971], and ML [Gordon 1978a], provide evidence of the effectiveness of the method.

The principle difficulty of applying the method arises when the language is implemented. Unlike procedural languages, these "abstract" languages do not have natural implementations on a von Neumann machine. We can overcome this difficulty in either of two ways. We can devise clever implementation techniques to map the language onto a von Neumann architecture or we can design a new kind of machine to close the "semantic gap" between software and hardware.

1.1.4 Types

Before applying an operation to an operand, we must ensure that the operand belongs to the domain of the operation. This condition is satisfied if the operand has the appropriate type. ALGOL 68 [van Wijngaarden 1975] and Pascal demonstrated the feasibility, the advantages, and some of the difficulties of compile-time type checking. The concept of abstract data types followed, but there are as yet few languages that support abstract data types in a fully satisfactory way. It seems natural to make the abstract data type the unit of modularity, but there are few examples of the successful application of this idea [Goguen 1984]. It is generally conceded that type declarations are unacceptable in a language intended for interactive use. This perception has led to languages such as ML and B [Meertens 1981], which do not require type declarations but infer types automatically.

Type checking a program is akin to proving a static property of the program. Type inference goes further in the sense that, if we regard the type of an object as an approximation to its value, type inference is partial evaluation. By increasing the precision of the type system we increase our knowledge of the static properties of the program at the expense of time spent compiling it. With a sufficiently precise type system, we can prove termination of the program, but only at the risk of non-termination of the compiler [Martin-Lof 1979]. The language designer must compromise between security and compiler complexity.

1.1.5 Transformation

There are three approaches to program correctness. One is to write a program and then prove it correct; another is to develop a program and a proof of its correctness concurrently; and a third is to transform a specification into an executable program. When the third approach is used, the transformation may be carried out by the system, by the user, or by the system under the direction of the user. A variation of the second method is to write a program that may be interpreted as either a proof of the existence of an object or a method for determining the value of the object [Boom 1982] [Voda 1984a].

Programming by transformation is in some ways orthogonal to the other strategies considered here; if transformation is the only method used in a transformation system, verification is not required. An advantage of transformation over verification is that the correctness of a transformation needs to be proved only once; in this sense, transformations are equivalent to proof rules. This advantage may not be of practical use, however, if the number of transformations in the library is large. The following are desirable properties of a transformation system.

1. The system automatically selects appropriate transformations from a library.
2. Each transformation has applicability conditions and, if these are satisfied, the transformation preserves the correctness of the program.
3. The transformations cover a wide spectrum: from specification level to machine code level.
4. If distinct transformation sequences of a single program lead to irreducible programs P and Q then P = Q.

5. If P can be transformed to Q then Q is in some sense "better" than P.

The fourth property is called the Church-Rosser property, by analogy with reduction in the lambda calculus, and the fifth is called monotonicity.

The Church-Rosser property implies a canonical representation for programs. This may not be achievable or even desirable. Monotonicity may also be unattainable. It is certainly possible to imagine situations in which a program might get "worse" before getting "better." The advantage of monotonicity is that it prevents an automated system from looping; it is of less importance in an interactive system.

1.1.6 Notation

A programming language should not restrict programmers. Ideally the programmer should enjoy the freedom of the mathematician, introducing new notation at appropriate places. It is easier to provide syntactic freedom than semantic freedom. One approach, used in languages such as Mary2, allows the user to introduce new syntax. Another approach, of which LISP and Prolog are the principal examples, is to provide tools for language processing and hence to simplify the task of embedding new languages into a host language.

1.1.7 Summary of Requirements

Our view is that there are three factors that should influence the design of a programming environment: a programming methodology, a programming language, and the selection of software tools that constitute the environment. Our model for methodology is "structured growth" [Sandewall 1978], derived from experience with LISP, coupled with source transformation. Our language is designed for this methodology. Our principal tools, an interactive editor, type checker, interpreter, and incremental compiler, are intended to support the effective development of correct and efficient programs.

1.2 Dee

Dee is a generic term denoting the family of languages around which this thesis is organized. The individual languages are called L_1, L_2,... where L_n is a "higher level" language than L_{n+1}.

Dee is a family of programming languages intended to meet some of the requirements stated or implied in the foregoing discussion. The novelty of Dee lies not so much in any one of its features as in their combination. The important features of Dee include the following.

1. The languages of Dee are applicative languages with referential transparency and without side-effects.
2. Dee has a simple kernel language but provides powerful abstraction mechanisms.
3. Large programs can be built from generic modules, thus minimizing the need for re-programming for a new application.
4. The language processors use type inference. Dee programs are type-safe but need not contain type declarations.
5. Dee is both a programming language and its own metalanguage. In Dee, as in LISP, programs may be manipulated as data by other programs.
6. Dee has a simple syntax but provides parsing directives that enable appropriate notation to be designed by the programmer.

1.2.1 Origins of Dee

Dee, like LISP and several other programming languages, has its origins in the lambda calculus. It also borrows heavily from LISP.

Lambda Calculus

The lambda calculus [Church 1941] [Barendregt 1984] provides a useful foundation for a programming language for several reasons. First, functions in the lambda calculus are defined by evaluation rules rather than by graphs. Second, there is a close relationship between bound lambda variables and the "formal parameters" of programming languages. Third, the typed and untyped lambda calculi provide theories for typed and untyped languages. Finally, the lambda calculus can describe all computable functions.

LISP was the first programming language to be based explicitly on the lambda calculus. Other, more recent languages based on lambda calculus include ISWIM [Landin 1965], GEDANKEN [Reynolds 1970], and Asp [Nordstrom 1984].

LISP

The success of LISP seems to be due not so much to its origins in the lambda calculus as to its other important innovations: automatic storage management, simple syntax, a single data structure, and the equivalence of program and data.

Despite its advantages and popularity, LISP is inadequate as a vehicle for exploring modern techniques of program development. LISP reveals its own early history in features such as a mixture of dynamic and static scoping, awkward treatment of higher order functions, and non-applicative features introduced to simplify programming and to improve the efficiency of interpretation. The tricks required to implement LISP efficiently show through as semantic anomalies. As examples of these anomalies, consider the dual nature of nil, which is both an atom and a list, the use of cons to construct pairs as well as lists, and the semantics of the various equality tests. Recent dialects of LISP have addressed some of these problems. SCHEME [Steele 1975], for example, uses static scoping and provides full support for higher order functions in a consistent way.

The price paid for the flexibility of representation in LISP is lack of security. The data structures correspond to types, and a data template with its selectors and constructors corresponds to an abstract data type. This abstract data type, however, exists only in the mind of the programmer, and the language provides no mechanims for ensuring consistency of access or detecting the effect of a change in implementation. It is evident to the careful reader of a LISP program that types are being used implicitly and in fact can be inferred by a simple algorithm. Inference was introduced in ML and has subsequently been used in languages such as HOPE [Burstall 1980] and Miranda [Turner 1983].

Comparison of Dee and LISP

An important similarity between LISP and Dee is that both languages treat functions intensionally. Functions in LISP and Dee are represented by S-expressions and can be processed as data. Some of the more recent functional languages treat functions extensionally, as independently existing objects.

Dee is purely applicative. Unlike LISP, in which it is possible to "escape" from purely functional programming by using SETQ, RPLACx, and property lists, Dee employs functional languages. Some programs that can be written easily in LISP are harder to write in Dee. They may also be less efficient. On the other hand, we claim that program development and transformation are easier in Dee, and a transformed Dee program may be more efficient than a LISP version of the same program. Finally, the semantics of Dee are simpler than the semantics of LISP.

Dee programming "style" is based on LISP. Dee does not permit some of the freedoms of LISP that are exploited in AI programs. For example, Dee expressions do not have side-effects. Dee is based on the observations that a functional style promotes clarity, that symbolic processing should not be more difficult than logical or numerical processing, and that a program should not be confused by low-level considerations such as pointer manipulation or storage management. The type system of Dee goes beyond that of LISP by providing type inference, type checking, generic functions, and type constructors.

Dee is strongly typed. Compile-time type-checking enables efficient code to be generated and errors to be detected before execution.

An important advantage of LISP is the close relationship between the internal and external representations of programs. The kernel language of Dee has this property, but the surface language does not. The syntax of Dee is such that Dee programs are much less readable than LISP programs when written in Cambridge prefix notation.

LISP uses dynamic scoping whereas Dee uses static scoping. Static scoping makes the interpreter less efficient but simplifies the construction of the Dee compiler.

LISP uses call-by-value whereas L_1, the top-level language of Dee, uses call-by-name. The practical effect of this is that some programs that do not terminate in LISP do terminate in Dee. Structures that are conceptually infinite can be created although only finite portions of them can actually be processed.

Dee provides more facilities for structuring data than LISP.

1.2.2 Overview of Dee

The kernel language of Dee is a language called L_2. L_3 is a machine representation of L_2 and for most purposes we ignore the distinction between L_2 and L_3. They are related in the same way that S-expressions and M-expressions are related in theoretical discussions of LISP [Allen 1978, p. 236].

The language L_2 has pleasant theoretical properties closely related to the lambda calculus. It has, however, two important disadvantages. First, because of its rudimentary syntax, it is difficult to write useful programs in it. Second, a naive implementation of L_2 would be inefficient.

The Dee system builds on L_2 in two directions. The high-level language L_1 can be translated in a relatively straightforward manner into L_2. The low-level language L_4 is an imperative language produced by compiling L_2.

We consider L_2 to be the "canonical" level of the system. In principle, any language with a denotational semantics could be used as L_1. In practice, we anticipate the use of "dialects" of L_1 tailored to specific applications. Similarly, there are a number of ways of implementing L_2, including interpreting it, which renders L_4 unnecessary. The languages L_1 and L_2 play a larger role in this thesis than L_4 because translation from L_2 to L_4 is currently based on well-known methods.

We hope to achieve two things from the implementation of a Dee system. First, Dee may turn out to be a useful environment for program development, supporting L_1 as its principal language. Second, and of greater importance if Dee is viewed as a research project, a Dee system will provide a framework for experimenting with techniques of current interest, such as programming by transformation. We are also investigating the feasibility of using Dee as a development environment for Brouwer [Boom 1982].

1.3 Organization of the Thesis

Various aspects of Dee and its implementation are discussed in subsequent chapters. The description has a "middle-out" form. First, in Chapter 2, we describe the kernel language, L_2. This description is brief because L_2 is typical of simple, functional languages.

In Chapter 3 we show how the kernel language is extended into a usable programming language, L_1. The use of two languages in this way is innovative and, we suggest, essential for the kind of programming environment that we propose. A language rich enough to express program transformations comfortably is not suitable as a target language in a transformation system.

Chapter 4 is a detailed discussion of the type system of L_1. The type system is based on the polymorphic type system of ML [Milner 1978] but also provides generic functions and coercion. The type system is not entirely novel because others have used a similar approach [Burstall 1980] [Letschert 1984]. The emphasis in ML and these systems, however, seems to be on type inference prior to interpretation. Another important application of type inference, exploited in Dee, is the use of type information to generate better compiled code than is possible within a conventional LISP-like system.

We claim that Dee's type system provides greater flexibility than others without compromising security.

Our concern in Chapter 5 is with implementation issues such as parsing, interpreting, and compiling Dee programs. The flexibility of the type system combined with the modular structure of Dee programs presents unusual problems for the compiler. In a conventional modular system, changes to a module affect only its clients; thus the flow of information in "trickle-down" recompilation is uni-directional. If a Dee module imports generic functions, changing it may require recompiling parts of the modules that it calls. We propose a new technique for incremental compilation that allows for this possibility.

In Chapter 6, we address the design of an environment for the development of Dee programs. Most existing programming environments belong to one of two classes. Either they are based on a compiled, typed language such as Ada [Ichbiah 1983] or Pascal, or they are based on an untyped, interpreted language such as APL or LISP. The first kind of environment is secure, at least insofar as the type system permits, but the unit of interaction is large: a small change necessitates the recompilation of an entire module. The second kind of environment permits a much smaller unit of interaction -- a single expression can be evaluated -- but typically provides much less security. The motivation for the design of Dee was the goal of combining the advantages of both kinds of environment by providing a fine level of interaction with the security of a robust type system.

In this thesis, we are proposing a programming language and a programming environment. The language has not been completely implemented and the present environment is rudimentary. In Chapter 5 we provide a precise account of what has been implemented and what remains to be done.

"A mechanician, or a mechanical workman, is he, whose skill is, without knowledge of mathematical demonstration, perfectly to work and finish any sensible work, by the mathematician principal or derivative, demonstrated or demonstrable."

2 The Kernel Language

The "kernel language" of Dee, L_2, is an untyped, applicative language. The language L_2 is similar to LISP in that it is based on the lambda calculus and there is a close relationship between the internal and external representations of programs. The internal representation of L_2 is called L_3. For most purposes, the distinction between L_2 and L_3 is unimportant and we use "L_2" to mean both L_2 and L_3. The language L_2 plays the role of an assembly language in a conventional programming environment. Programs in L_2 are mechanically generated from L_1 programs and consequently they do not contain type errors. The language L_2 is therefore a suitable medium for program transformation.

A conventional assembly language is not suitable for transformation, other than trivial kinds of transformation such as peephole optimization, because too many implementation decisions have already been made by the time it is created. In Dee, these decisions are deferred until L_3 is compiled into L_4. The language L_2 provides a useful intermediate level at which expressions are referentially transparent and a wide variety of significant optimizing transformations are available. Programs in L_2 are representable as typed objects in L_1 and so transformation schemata can be written in L_1.

We describe L_2 in three ways. First we give the syntax, then an informal description of the semantics, and finally a denotational semantics.

2.1 Syntax

A program in L_2 is an expression composed of one or more primitive objects or sub-expressions. An expression is a constant, an identifier, an abstraction, an application, a local definition, or a conditional expression.

Constants

The following objects are constants: members of the set {false,true} of truth values; members of the set {...,-1,0,1,...} of integers; and members of the set of primitive functions. In a complete implementation of Dee, there would be other kinds of constant, such as characters, strings, and tokens, but the constants given suffice for the purpose of this exposition.

Identifiers

Each identifier in a program abbreviates an expression. An identifier has a scope, and within that scope it is bound to a value. In L_2, identifiers are used as formal parameters of functions and in local definitions as abbreviations for expressions. When a function is invoked, the value of each argument is bound to the corresponding formal parameter.

Abstractions

If $X_1, X_2, \ldots, X_n$ are identifiers and E is an expression, then

$$(\text{LAMBDA } (X_1\ X_2\ \ldots\ X_n)\ E) \tag{2.1}$$

is an expression whose value is a function. In general, E will contain free variables, including some of $X_1, \ldots, X_n$.

Application

If $E_0, E_1, \ldots, E_n$ are expressions, then

$$(E_0\ E_1\ \ldots\ E_n) \tag{2.2}$$

is an expression.

Local Definitions

If $X_1, X_2, \ldots, X_n$ are identifiers and $E_0, E_1, \ldots, E_n$ are expressions, then

$$(\mathrm{LET}\ (X_1\ X_2\ \ldots\ X_n)\ (E_1\ E_2\ \ldots\ E_n)\ E_0) \tag{2.3}$$

and

$$(\mathrm{LETREC}\ (X_1\ X_2\ \ldots\ X_n)\ (E_1\ E_2\ \ldots\ E_n)\ E_0) \tag{2.4}$$

are expressions.

Conditional

If E_0, E_1, and E_2 are expressions, then

$$(\mathrm{IF}\ E_0\ E_1\ E_2) \tag{2.5}$$

is an expression.

The conditional is included for convenience. It is not necessary, because we could follow the approach of the lambda calculus, defining the conditional as follows.

$$(\mathrm{IF}\ E_0\ E_1\ E_2) = (E_0\ E_1\ E_2)$$
$$\text{true} = (\mathrm{LAMBDA}\ (X\ Y)\ X)$$
$$\text{false} = (\mathrm{LAMBDA}\ (X\ Y)\ Y)$$

All L_2 expressions can be constructed by repeated application of these rules.

2.2 Informal Semantics

The evaluation of expressions in L_2 is based on the reduction rules of the lambda calculus. There are additional rules for constants, which are not part of the pure lambda calculus. We do not consider the possibility of errors occurring during the execution of an L_2 program. This is because L_2 plays the role of an assembly language in the system. Programs in L_2 are mechanically generated, and any required error-checking code is supplied by the compiler.

The evaluation of a constant yields the value of the constant.

The evaluation of an identifier yields the value to which the identifier is currently bound. (An unbound identifier would be an error but, by hypothesis, this cannot occur.)

The evaluation of the abstraction (2.1) yields an object called a closure. The closure is a triple consisting of the parameters $X_1,\ldots,X_n$, the expression E, and the current bindings of the free variables of E. We write this closure as

(LET U (LAMBDA (X_1 ... X_n) E)),

in which U contains the bindings. It is an important feature of L_2 (and of any other language that provides full support for high order functions) that the evaluation of an abstraction yields a value that can be bound to a variable. Expression (2.1) may be applied to arguments $E_1,\ldots,E_n$, in which case E will be evaluated with X_i bound to E_i, $i = 1,2,\ldots,n$.

The evaluation of the application (2.2) starts with the evaluation of E_0. If the value obtained is a primitive function, f, the arguments $E_1,\ldots,E_n$ are evaluated and f is applied to them. Otherwise, the value of E_0 must be a closure and the value of (2.2) is obtained by evaluating E. During the evaluation of E, each X_i is evaluated and the result is bound to the corresponding E_i. The bindings of the other free variables of E are taken from U.

The forms LET and LETREC were introduced by Landin [Landin 1965]. The evaluation of (2.3) proceeds as follows: first, the expressions $E_1,\ldots,E_n$ are evaluated; a new environment is created in which X_i is bound to E_i for i = 1,2,...,n; and, finally, the expression E is evaluated in this environment.

The evaluation of (2.4) is similar except that $E_1,\ldots,E_n$ are evaluated in a dummy environment that already contains the identifiers $X_1,\ldots,X_n$. The variables $X_1,\ldots,X_n$ cannot be evaluated in the dummy environment, however, so $E_1,\ldots,E_n$ must be function definitions. This construction allows recursive and mutually recursive functions to be defined but it does not provide immediate recursion. For example, (2.6) does not terminate in L_2.

```
        (LETREC (S) ((CONS 0 S)) (CAR S))                          (2.6)
```

All occurrences of LET and LETREC can be removed from an L_2 program by applying the following reductions. We write E ==> E' to denote the reduction of E to E'. FIX is the fixpoint combinator.

```
(LET (X1 ... Xn) (E1 ... En) E0) ==>
    (LAMBDA (X1 ... Xn) E0)(E1 ... En)
(LETREC (X1) (E1) E0) ==>
    (LAMBDA (X1) E0)(FIX (LAMBDA (X1) E1))
(LETREC (X1 ... Xn) (E1 ... En) E0) ==>
    (LETREC (X1)
              (LETREC (X2 ... Xn) (E2 ... En) E1)
              (LETREC (X2 ... Xn) (E2 ... En) E0) )
```

Although these reductions do not change the values of the expressions, they discard information that is of use to an interpreter or compiler. Moreover, it is impractical to compile expressions containing FIX. For these reasons, we consider LET and LETREC to be irreducible forms in L_2.

The evaluation of the conditional expression (2.5) commences with the evaluation of E_0. If E_0 yields true, the value of (2.5) is found by evaluating E_1. If E_0 yields false, the value of (2.5) is obtained by evaluating E_2.

2.3 Formal Semantics

By design, L_2 has simple semantics. The important points to note are that the semantics does not assign a meaning to an incorrect program and that it permits applicative order ("call by value") evaluation. We describe L_2 by means of a standard denotational semantics.

We use upper case letters for syntactic objects and domains and lower case letters for semantic objects and domains. Domains have three-letter names and typical members have one letter names.

We require the concept of an environment for the semantics. An environment is a partial function from identifiers (syntactic objects) to denotable expressions (semantic objects). Let ids(u) be the set of identifiers bound by an environment u. We denote the sum of environments u_1 and u_2 by u_1+u_2, where $\text{ids}(u_1+u_2)$ is the union of $\text{ids}(u_1)$ and $\text{ids}(u_2)$. If an identifier is bound in both components of a sum, its binding from the left component is used. The notation [I->e] denotes the environment in which the identifier I is bound to the expression e. Since an environment is a function, we have [I->e](I) = e and, by the rule above:

$$([I\text{->}e_1] + [I\text{->}e_2])(I) = e_1.$$

Each row in a domain table contains three entries. The first, a single letter, is a typical element of the domain. The same letter, usually decorated with subscripts, is used in the semantic equations. The second entry is the name of the domain and the third is an informal description of the domain.

In L_2, any value that can be yielded by an expression can be bound to an identifier. This fact is expressed by the domain equation

$$\text{den} = \text{exp}.$$

The domain den is therefore redundant, but we have retained it in the semantics for clarity.

The semantic function K maps the syntactic object C, denoting a constant, to the semantic object which is the representation of the constant. We do not specify the details of K.

The semantic function M maps an expression and an environment to a value in the semantic domain exp. Values in exp are expressions in a lambda calculus extended with constants. All syntactic objects -- that is, values in CON, IDE, EXP, and DEF -- are enclosed in braces {...} in the defining equations.

The function STRICT is used to convey the meaning of a call-by-value implementation. STRICT is defined by equation (2.7).

$$\text{STRICT}(F)(X_1 \ldots X_n) = \text{if } \perp \in \{X_1,\ldots,X_n\} \text{ then } \perp \text{ else } F(X_1,\ldots,X_n). \tag{2.7}$$

The effect of including STRICT in the semantic equation for abstraction is that the arguments of an application must have a normal form. This corresponds to call-by-value in an implementation. If STRICT is omitted, we can show that if E_1 is converted to E_2 using the reduction rules of the lambda calculus (with obvious syntactic changes) then in all environments [Stoy 1977, pp. 158-67]

$$M\{E_1\} = M\{E_2\}$$

Syntactic Domains

C	CON	constants
I	IDE	identifiers
E	EXP	expressions
D	DEF	definitions

Semantic Domains

c	con	constant values
e	exp	expressible values
f	fun	functions
d	den	denotable values
u	env	environments

Domain Equations

exp = con + fun

fun = den -> exp

den = exp

env = IDE -> den

Semantic Functions

K : CON -> con

M : EXP -> env -> exp

U : DEF -> env -> env

M{C}u = K{C}

M{I}u = u{I}

M{(LAMBDA (I_1 ... I_n) E)}u =

  STRICT(lambda(x_1,x_2,...,x_n).M{E}([I_1->x_1]+...+[I_n->x_n]+u))

M{(E_0 E_1 ... E_n)}u = (M{E_0}u)(M{E_1}u,...,M{E_n}u)

M{(LET D E)}u = M{E}(U{D}+u)

M{(LETREC D E)}u = M{E}fix(lambda v.U{D}v+u)

M{(IF E_0 E_1 E_2)}u = if M{E_0}u = true then M{E_1}u else M{E_2}u

U{I = E}u = [I->M{E}u]

U{D_0, D_1}u = U{D_0}u + U{D_1}u

"And our two sciences, remaining pure, and absolute, in their proper terms, and in their own matter: to have, and allow, only such demonstrations, as are plain, certain, universal, and of eternal verity."

3 Extending the Kernel

The language L_1 is an expressive, high-level programming language. With the addition of user-defined types, it attains the expressive power of a typical, high-level, imperative language. The expressions of L_1, however, are easily translated into L_2, a language with simple semantics. Furthermore, the manipulation of L_2 programs by L_1 programs is straightforward because of the simple internal representation of L_2 programs. Thus L_1 is a language that enables us to reason about programs and to formalize our reasoning.

The language L_1 is strongly typed, but the discussion of types is deferred until Chapter 4. Consequently, this chapter makes use of a small number of undefined concepts. Some features of L_1 are not relevant to this thesis and are mentioned briefly or not at all.

A program in L_1 consists of expressions and directives. We discuss expressions first.

3.1 Expressions

An expression in L_1 can be evaluated. The evaluation yields an equivalent expression in irreducible form or an error, or it fails to terminate. Whereas L_2 has LISP-like syntax, L_1 has a syntax that is closer to the syntax of the ALGOL family of languages.

3.1.1 Constants

Corresponding to each of the primitive types of Dee, there is a set of constants that evaluate to the values that they denote. In the examples we use values of the type bool = {false,true} and of the type int = {...,-1,0,1,...}. The full language also contains denotations for characters, strings, symbols, and other constants. All constants have the property that both their type and their value can be inferred from their representations. In L_1, primitive functions are identifiers with global bindings, although in L_2 they are constants. This difference is necessary because primitive functions can be redefined and overloaded in L_1 whereas, in the interests of efficient implementation, they must be constants in L_2.

3.1.2 Identifiers

Identifiers play a conventional role in L_1. An identifier is bound when it occurs in a formal parameter position or as the subject of a local definition. Other uses of an identifier must occur within the scope of a binding. If an identifier is overloaded, several bindings may be visible from certain parts of the program.

Any token that is not recognized by the parser as a special symbol is assumed to be an identifier. In addition to conventional identifiers, the following tokens would be recognized as identifiers in L_1:

```
+   ##   !   *SPEC-NAME*
```

3.1.3 Abstraction

If e is an expression and $x_1,...,x_n$ are identifiers, then

$$[x_1,...,x_n] \text{ -> } e \qquad (3.1)$$

is an expression denoting a function. (3.1) corresponds to the L_2 expression

(LAMBDA (X_1 ... X_n) E) (3.2)

The occurrence of x_i in (3.1) is a binding occurrence of x_i for $i = 1,\dots,n$. The scope of x_i is the expression e.

Abstractions are anonymous functions with free variables bound in the context of the definition ("lexical scoping"). In L_1 it is always permissible to replace, for example, the successor function by the expression

[n] -> n + 1.

The arrow ("->") may be omitted, although we use it consistently in this thesis. Thus we can also write the successor function in the form

[n] n + 1.

This notation is used in Combinatory Logic, where it is called "bracket abstraction". It was also used by Church in early versions of the lambda calculus [Seldin 1985].

3.1.4 Application

If $f,e_1,\dots,e_n$ are expressions, then

$f(e_1,\dots,e_n)$ (3.3)

is an expression denoting the application of the function f to the arguments $e_1,\dots,e_n$. The expression (3.3) corresponds to the L_2 expression

(F E_1 ... E_n)

3.1.5 Local Definitions

If $x_1,\dots,x_n$ are identifiers and e, $e_1,\dots,e_n$ are expresssions, then

let $x_1=e_1,\dots,x_n=e_n$ in e (3.4)

and

letrec $x_1=e_1,\dots,x_n=e_n$ in e (3.5) are expressions denoting the value of the expression e in the environment determined by the definitions $x_1=e_1,\ldots,x_n=e_n$. A let or letrec expression binds the identifiers that occur in it. These expressions correspond to the L_2 expressions

(LET (X_1 ... X_n) (E_1 ... E_n) E)

and

(LETREC (X_1 ... X_n) (E_1 ... E_n) E)

The expressions

e where $x_1=e_1,\ldots, x_n=e_n$ end (3.6)

and

e whererec $x_1=e_1,\ldots, x_n=e_n$ end (3.7)

are equivalent to (3.4) and (3.5) respectiviely.

3.1.6 Conditionals

If e_0, e_1, and e_2 are expressions, then

if e_0 then e_1 else e_2 (3.8)

is an expression with the natural meaning. This expression may also be written in the abbreviated form

e_0 ? e_1 | e_2. (3.9)

Both expressions correspond to the L_2 expression

(IF E_0 E_1 E_2).

Conditional expressions are used in the definitions of recursive functions such as the familiar factorial function:

letrec fac = [n] -> n = 0 ? 1 | n * fac(n-1)

3.2 Patterns

L_1 has a collection of functions called **constructors** that are used to create objects which have a particular type and internal structure. Corresponding to the constructors are **recognizers** and **selectors** [McCarthy 1960]. A recognizer for a type T is a function IS-T such that IS-T(X) is true iff X is of type T. A selector is used to extract a particular field from a compound object.

A **pattern** is an expression composed of constructors and identifiers. A pattern may occur at any place in an expression where an identifier is required in a binding context. For example, suppose that we have defined type RAT of rational numbers with constructor MAKE-RAT, recognizer IS-RAT, and selectors NUM and DEN. The function

```
[r] -> let n = NUM(r); d = DEN(r) in n/d                    (3.10)
```

can be written using a pattern in the following form:

```
[MAKE-RAT(n,d)] -> n/d.                                     (3.11)
```

We can replace MAKE-RAT by the user-defined infix operator % (Section 3.3.2), obtaining

```
[n%d] -> n/d.                                               (3.12)
```

Patterns do not introduce any new semantics. (3.12) is transformed first into (3.10) and then to the L_2 expression

```
(LAMBDA (R) (LET (N D) ((NUM R) (DEN R)) (DIV N D))).      (3.13)
```

All occurrences of selectors in a program can be replaced by introducing appropriate patterns. It is never necessary to define selectors in L_1 programs.

3.2.1 The Case Expression

The expression

$$\textbf{case } e \textbf{ of } p_1 : e_1;\ p_2 : e_2;\ \ldots\ p_n : e_n;\ \textbf{else } e_0$$

contains patterns $p_1,\ldots,p_n$. If the expression e is such that it could have been constructed by p_i, then e matches p_i, and the value of the case expression is e_i evaluated with the identifiers in p_i bound to the corresponding components of e. If e does not match any pattern, the value of the case expression is e_0.

Case expressions are frequently used to discriminate values of union types. Suppose that we have defined the type NUMBER as the union of int and RAT with constructors

```
INT-NUM: int -> NUMBER
```

and

```
RAT-NUM: RAT -> NUMBER.
```

The following expression returns the integer component of a value of type NUMBER:

```
case u of
    INT-NUM(i) : i;
    RAT-NUM(n%d) : n/d;
    else error('Incorrect type').
```

This corresponds to the following $\mathcal{L}_2$ expression:

```
(IF (IS-INT U)
    (INT U)
    (IF (IS-RAT U)
        (LET (N D)
             ((NUM (RAT U))
              (DEN (RAT U)) )
             (DIV N D) )
        (ERROR "Incorrect type") ) ).
```

3.3 Directives

In addition to expressions, an L_1 program may contain **directives** that affect the environment in which expressions are evaluated. Directives fall into three classes: **parser directives** control the translation of L_1 into the internal tree representation; **environment directives** define partial functions from tokens to values; and **global definitions** introduce new objects into the global environment. In the first two cases, the purpose and implementation of a directive is similar to defining a property in a LISP system. The difference in Dee is that properties are static and must be established during initialization. An L_1 expression can access a property value but cannot alter it.

3.3.1 Scope of Directives

When a directive is processed, the environment changes. Expressions following the directive are evaluated in the new environment. The effect of this rule is that, within a module, declarations must be appropriately ordered, and during an interactive session, directives become effective as soon as they have been entered.

3.3.2 Parser Directives

The Dee parser recognizes the special tokens of (3.14). All other tokens are treated as identifiers.

( ) [ ] -> , ; : ? | (3.14)

The parser directive

let x = infix(f,m,n); (3.15)

defines a binary infix operator, x, with left precedence m and right precedence n. The effect of (3.15) is that the infix expression

$e_1 \times e_2$

is transformed to

$f(e_1,e_2)$.

The expression

$e_1\ x_1\ e_2\ x_2\ e_3$,

in which x_1 and x_2 are infix operators with left precedence $lp(x_i)$ and right precedence $rp(x_i)$, will be parsed as

$e_1\ x_1\ (e_2\ x_2\ e_3)$

if $rp(x_1) < lp(x_2)$ and otherwise as

$(e_1\ x_1\ e_2)\ x_2\ e_3$.

3.3.3 Global Definitions

The forms

let $x_1=a_1$, ..., $x_n=a_n$; (3.16)

and

letrec $x_1=a_1$, ..., $x_n=a_n$; (3.17) introduce new values $x_1,\ldots,x_n$ into the global environment. The definitions (3.16) and (3.17) have the same effect as

let $x_1=a_1, \ldots, x_n=a_n$ in e

and

letrec $x_1=a_1, \ldots, x_n=a_n$ in e

if we think of "e" as being the rest of the current module or the rest of the interactive session.

3.4 Input and Output

Input and output facilities are provided by streams [Landin 1965]. A stream is a lazily evaluated infinite list.

An input stream appears to the program as a list of characters ("character directed input") or as a list of tokens ("token directed input"). The type tok is a standard type: its values are valid Dee tokens. The lexical analyzer splits the incoming character string into tokens and passes these to the program. By default, the rules used by the lexical analyzer are the same as those used for L_1. The lexical analyzer is table-driven, however, and its tables can be modified by directives.

An output stream may consist of characters, tokens, or arbitrary data structures. In Dee, as in LISP, there is a default external representation for every data type: the generic function SHOW constructs the external representation of any object given to it. Problem-specific external representations can be obtained by overloading SHOW.

3.5 Modules

A group of declarations may be collected together in a **module**. The effect of the directive

```
    import M
```

in which M is a module name, is similar to the effect of processing the declarations within M. The difference is that a module does not necessarily reveal all of the objects declared within it. The following module introduces into the environment the type RAT of rational numbers.

```
module rational
  type RAT = int * int;
  let MAKE-RAT = constructor(RAT);
  let % = infix(MAKE-RAT,20,30);
  let PLUS = [a % b, c % d] -> a * d + c * b % b * d,
      TIMES = [a % b, c % d] -> a * c % b * d,
      SHOW = [a % b] -> format(a/g,"%",b/g)
        where g = GCD(a,b);
  export %, PLUS, TIMES, SHOW
end rational
```

The type RAT is represented by the Cartesian product of two ints. The function MAKE-RAT, defined using the system function "constructor", constructs a rational number from two integers. A client of the module constructs a rational number using the binary infix operator %. The expression 4%6, for example, evaluates to the rational <4,6>. This constructor is used as a pattern in the definitions of the functions PLUS and TIMES. These functions overload the primitive integer functions PLUS and TIMES. Since the infix operators + and * are associated with the names

"PLUS" and "TIMES" in the global environment, they can be used to add and multiply rationals as well.

The function SHOW displays or prints a rational number. When displaying a value of a user-defined type, the system uses an applicable definition of SHOW if one exists, otherwise it uses a default representation. The ability of the system to display an object of any type, whether or not the programmer has defined a formatting function for it, is extremely useful during program development.

The names exported by this module are declared in an export directive. These names provide the only means by which a client can construct and use rational numbers.

The abbreviation of MAKE-RAT to % illustrates a defect of L_1 that has yet to be solved. We would like to define a constructor for RAT that yields a rational in lowest terms. For example, we use the function GCD (greatest common denominator) in (3.18).

```
    MAKE-LOW = [m,n] -> MAKE-RAT(m/g,n/g)                    (3.18)
        where g = GCD(m,n).
```

We can define % to be the infix form of MAKE-LOW, but if we do, n%d is not a pattern and could not be used in (3.12).

Identifiers imported from a module may be renamed by the importer. For example, we could write

```
    import rational[%,ADD,MUL,SHOW]
```

The identifiers ADD and MUL become local synonyms for PLUS and TIMES. This avoids the need for qualified names, but it is not clear whether this is a better device in practice.

3.6 Error Handling

In the current version of L_1, errors are handled in an unsophisticated way. If a partial primitive function receives an argument outside its domain, the program terminates with an error message. This mechanism is used to handle division by zero, for example. The programmer can use the same mechanism by passing a string to the primitive function ERROR. When ERROR(S) is evaluated, the program terminates and returns the value of S.

We expect that later versions of Dee will incorporate a mechanism that allows errors detected at a low level to be trapped at a higher level without terminating the program. The effects of a mechanism of this kind on the usefulness of Dee, and the details of its implementation, have not yet been explored.

3.7 Syntactic Variants

It is relatively straightforward to design dialects of L_1 that are suitable for other applications. For some purposes, for example, the notation

$$f(x) = e \tag{3.18}$$

for function definition is preferable to

$$f = [x] \mathbin{->} e \tag{3.19}$$

This is a simple syntactic transformation. Note, however, that the notation of (3.19) is still useful for anonymous functions.

For transformations of the type introduced by Burstall and Darlington [Burstall 1977], the definition of a function as a set of equations is useful. For example, a function that computes the length of a list might be written:

$$\begin{aligned}&\text{len}(\text{nil}) = 0 \\ &\text{len}(x.y) = 1 + \text{len}(y)\end{aligned} \tag{3.20}$$

The difficulty in transforming equations of this kind into L_1 is the introduction of approriate recognizers and inverse functions. For example, (3.20) should be transformed to

```
len = [s] -> null(s) ? 0 | ...
```

rather than

```
len = [s] -> s = nil ? 0 ... .
```

An equation of the form

$$f(g(x)) = \dots \qquad (3.21)$$

corresponds to a definition of the form

$$f = [x] \text{ -> } \dots\ g^{-1}(x)\ \dots\ .$$

Thus (3.21) is only acceptable if g^{-1} can be derived by the transformation system.

We have also considered the practicality of a syntax for L_1 in which all expressions are constructed from prefix operators, infix operators, and operands. Abstractions and applications, for example, would have the forms

$$x_1,\dots,x_n \text{ +> } e$$

and

$$f \text{ @ } e_1,\dots,e_n$$

respectively. (The symbol "+>", borrowed from category theory, was suggested for this purpose by H. Boom [Boom 1984].) A preliminary investigation suggests that the resulting dialect is less readable. Either many levels of precedence would be required, as in ALGOL 60 [Naur 1963] or C [Harbison 1984], or all operators would have the same precedence, with evaluation proceeding from left to right, as in Mary2, or from right to left, as in APL.

3.8 Equality in Dee

The equality predicate in L_1 is represented by the polymorphic primitive function EQUAL. Atomic objects are "EQUAL" if they have the same bit representation, whether or not they have the same address. Structured objects are "EQUAL" if they have the same type, and hence the same structure, and have "EQUAL" leaves.

This definition of equality contrasts with typical LISP systems, which provide several equality predicates. For example, COMMON LISP [Steele 1984] has EQ, EQL, EQUAL, and EQUALP. The various predicates correspond to different implementations of the test for equality. The choice between them is often based on the programmer's knowledge of the implementation. For example, if (EQ X Y) yields T, then (EQUAL X Y) certainly yields T because EQ compares addresses. The converse is not true because objects may have different addresses but the same value.

The Dee definition is a compromise between convenience and precision. It is convenient because it simplifies programming and transformation. The intended meaning of equality in Dee is equality of abstract values. Thus "equal" in Dee corresponds most closely to "EQUAL" in COMMON LISP. The expression EQUAL(X,Y) in Dee will fail to terminate (or will cause stack overflow) if X and Y are isomorphic, cyclic structures. This is not a serious problem because the user cannot create cyclic structures in a purely applicative language. Nevertheless, the run-time environment of Dee contains cyclic structures, and primitive functions provide access to them.

A more serious problem is equality of functions. Since equality of functions is undecidable in principle, the best solution is to define the function "equal" in such a way that no two functions are equal. Dee ducks this issue by implementing "equal" in the obvious way: atomic values are "equal" if they have the same value; structures are "equal" if they are isomorphic and have "equal" leaves. Consequently, if the evaluation of EQUAL(F,G) yields true, then F and G are extensionally equal (F(X) = G(X) for all x), but if the evaluation of EQUAL(F,G) yields false, F and G may or may not be extensionally equal. For example, if we have defined

```
        f = [n] -> n * n - 1
        g = [m] -> m * m - 1
        h = [n] -> (n + 1) * (n - 1)
```

then no two of f, g, and h are EQUAL.

"In mathematical reasoning, a probable argument, is nothing regarded: nor yet the testimony of sense, any wit. credited: but only a perfect demonstration, of truths certain, necessary, and invincible: universally and necessarily concluded: is allowed as sufficient for an argument exactly and purely mathematical."

4 Types

In strongly-typed languages, type declarations are required by the compiler. In untyped languages, types are present but implicit. In Dee, type declarations are permitted but not required. Thus a type declaration in a Dee program is a form of assertion. The compiler checks that type information provided by the programmer is consistent with the information that it has inferred, and it may exploit additional precision provided by the programmer. In this chapter, we describe the type structure and the type inference algorithm of Dee.

4.1 The Role of Types

Four important attributes of a computational object are its name, type, location, and value. An object possesses different combinations of attributes at different times, depending on both the language and the current form of the program. In a typed, compiled language, each object possesses a name and a type in the source code, a location in the compiled code, and a value during execution of the program. In an untyped, interpreted language, each object possesses a name in the source code, a type, a location, and a value during interpretation. There are, of course, many variations of these simple paradigms; the important issue is the time at which names are bound to their attributes.

It is a useful rule of thumb that early binding provides efficiency and that late binding provides flexibility. The binding of a variable to an address, for example, occurs at different times in different languages. In machine code, it is determined at coding time; in assembly language or FORTRAN, it is determined at linkage time; in ALGOL 60, it is determined at block activation time; and in LISP, it is determined when SETQ is evaluated.

Types and values are handled in different ways. An object always has a type, and the question is whether the type is part of the representation of the object. In a typed, compiled language, the representation of an object is an entry in the symbol table: the type is part of the representation. In the compiled code, the type no longer appears explicitly because the compiler has ensured that there are no type errors. In an untyped language, type errors can be detected only if objects carry their types around with them.

4.1.1 Requirements

The primary requirements for the type discipline of Dee are that type errors should be detected by the system and that type declarations should be optional. Type errors may be detected during interpretation, compilation, or execution of compiled code. For convenience, errors should be detected at the earliest possible opportunity. An additional reason for preferring compile-time error detection is the overhead incurred by postponing error checking until run-time.

Other desirable features that a type discipline might provide are polymorphic functions, generic functions, and coercion. A polymorphic function accepts arguments of an unlimited number of different types: for example, LISP's CAR selects the first component of a list of objects of any type. A generic function is actually several functions that share a name: we say that the name of a generic function is overloaded. For example, in many languages, "+" is a generic function that may be used to add both integers and floating point numbers. The distinction is important for implementation because the same code can be used for every instance of a polymorphic function, but different code is required for instances of a generic function, the appropriate code depending on the types of the arguments.

Coercion is the implicit conversion of a value from one type to another. Coercion may require a change of representation, as when int is coerced to real, or it may not, as in char to byte.

The requirements may be summarized as follows.

1. All type errors should be detected during interpretation or compilation.
2. Type declarations should be permitted but not required.
3. The type discipline should provide polymorphic functions, generic functions, and coercion.

4.1.2 Meeting the Requirements

When we read a program written in an untyped language, such as LISP or APL, we attempt to deduce the types that the programmer intended but did not state explicitly. We do this because type information helps us to understand the program. This intuitive form of type inference was formalized by Hindley for Combinatory Logic and hence for the lambda calculus [Hindley 1969]. The key result is that if type annotations are removed from a well-formed term in the lambda calculus, the principal type of the expression can be inferred from the untyped expression. This result was first applied to a programming language by Milner [Milner 1978]. Since Milner published his paper, other type disciplines have been proposed [Coppo 1980] [Reynolds 1981] [MacQueen 1982] [Holmstrom 1983] [Coppo 1983] [Meertens 1983] [Letschert 1984]. We describe the type discipline of Dee and then compare it to other disciplines.

The requirements of Dee can be met by an algorithm that can: (1) infer the type of an expression; (2) check that, if declared types are present, they are compatible with inferred types; (3) resolve overloading; and (4) insert appropriate coercions.

Type checking is performed during the translation from L_1 to L_2. Each object in an L_1 program has a type that is either explicit or inferred by the system. Objects in an L_2 program are known to have correct types, but the types are not represented explicitly.

4.2 Preliminaries

In this section, we provide a foundation for the discussion of types by defining a sublanguage of L_1, a term language for types, and the concepts necessary for type inference.

4.2.1 A Sublanguage for the Discussion of Types

For the discussion of types, we consider a language that is a subset of L_1. Applying the type discipline to the full language requires a considerable amount of detailed work but offers no new insights. The main differences between the full language and the sublanguage are that functions have exactly one argument and that forms that are merely sugared versions of other forms are omitted. The conditional expression

$$\textbf{if } E_0 \textbf{ then } E_1 \textbf{ else } E_2$$

is correctly typed and has type T if E_0 has type **bool** and the types of E_1 and E_2 are both T. We use concatenation to denote application, writing fx for f(x). Some of the examples use features of the full language such as infix binary operators.

An expression in the sublanguage has one of the following forms.

1. A constant.
2. A variable.
3. An abstraction, [X] -> E.
4. An application, E_1E_2.
5. A let form, **let** X = E_1 **in** E_2.

For the purposes of type inference, we assume that expressions of the form

$$\textbf{letrec } X = E_1 \textbf{ in } E_2$$

have been transformed to

$$\textbf{let } X = \text{FIX}([X] \rightarrow E_1) \textbf{ in } E_2.$$

4.2.2 The Type Sublanguage

The type sublanguage consists of terms constructed according to the following rules.

1. A primitive type is a term.
2. A type variable is a term.
3. If T is a term, (u)T is a term.
4. If T is a term, #(T) is a term.
5. If T_1 and T_2 are terms, $(T_1) \text{->} (T_2)$ is a term.
6. If $T_1,T_2,\ldots,T_n$ are terms, $(T_1) + (T_2) + \ldots + (T_n)$ is a term.
7. If $T_1,T_2,\ldots,T_n$ are terms, $(T_1) * (T_2) * \ldots * (T_n)$ is a term.
8. If $T_1,T_2,\ldots,T_n$ are disjoint terms, (T_1) & (T_2) & ... & (T_n) is a term.

We assume the existence of the primitive types bool, with values {false,true}, and int, with values {...,-1,0,1,...}. We use u,v,... to denote type variables and $T,T_1,T_2,\ldots$ to denote terms in the type language. In the term (u)T, (u) is a quantifier that binds occurrences of the type variable u.

#(T) is the type whose values are sequences (or tuples) of values of type T. It is introduced as a convenience in order to avoid the necessity of a recursive type definition for an intuitively simple concept.

$(T_1)\text{->}(T_2)$ is the type of functions from T_1 to T_2. $(T_1)+(T_2)+\ldots+(T_n)$ is the discriminated union of the types $T_1,T_2,\ldots,T_n$. $(T_1)*(T_2)*\ldots*(T_n)$ is the Cartesian product of the types $T_1,T_2,\ldots,T_n$.

(T_1)&(T_2)&...&(T_n) is the conjunction of the types $T_1,T_2,\ldots,T_n$. The symbol "&" and the name "type conjunction" were introduced by Coppo [Coppo 1983]. Our type conjunction is slightly different from Coppo's, however, because in our system the type (T_1)&(T_2) may not exist.

Intuitively, type conjunctions correspond to overloaded names. If we use PLUS to add both integers and rationals, its type is

    (int*int->int) & (RAT*RAT->RAT)

In Dee, a type conjunction cannot be formed if the components of the conjunction have values in common. The purpose of this rule is to ensure that it is always possible to resolve an overloaded name at compile-time.

The operators + and * are associative. We assume an order of precedence amongst the unary operator # and the operators ->, +, *, and *: # (highest), *, +, ->, & (lowest). When there is no ambiguity, we omit parentheses. For example, we write **int**->**bool** rather than (**int**)->(**bool**).

If T is a term containing free and bound variables, then refresh(T) is a new type term which is the same as T except that new variables are substituted for the bound variables. For example, v' is new in

    refresh((v)u*v) ==> (v')u*v'.

4.2.3 Explicit Type Declaration

Types may be declared explicitly in Dee programs in two ways. First, a new type may be introduced by a declaration of the form

    type type-id = type-term;

For example:

    type RAT = int * int;

The left side of this definition is a type identifier, and the right side is a term in the type sublanguage that specifies the representation of the type. As a notational convention, we write primitive types in lower-case bold letters and user-defined types in upper-case letters.

Recursive types are introduced by typerec. Values of the type CHARTREE are binary trees with a character at each node. Nil is an implicit niladic constructor for every recursive type. For example:

    typerec CHARTREE = char * CHARTREE * CHARTREE;

An identifier may be given a type wherever it occurs in a binding context. There are three binding contexts. The identifier I is bound in each of the expressions

[I] -> E,

let I = E_1 in E_2,

letrec I = E_1 in E_2.

There are other binding occurrences in L_1 because pattern matching may occur, but they all reduce to one of the forms above. In any of these contexts, the identifier I may be replaced by I:T, where T is a term in the type sublanguage.

4.2.4 Substitutions and Unification

A substitution is a list of pairs. Each pair consists of a type variable and a type term. For example, the substitution

[u=#int, v=bool] (4.1)

denotes the substitution that replaces u by #int and v by bool. "I" denotes the empty substitution, equivalent to the identity function.

A substitution, S, may be applied to a term E; we write S(E) or SE. Application consists of replacing occurrences of the variables in the term by the terms specified by the substitution. For example, applying the substitution (4.1) to the term u*v yields the term #int*bool. Applying the empty substitution I to a term leaves the term unchanged.

Two substitutions, S_1 and S_2, may be composed. We use juxtaposition to denote composition. For any substitutions, S_1 and S_2, and any term, E:

$$(S_1S_2)(E) = S_1(S_2(E)).$$

A unifier for terms E_1 and E_2 is a substitution U such that

$$U(E_1) = U(E_2).$$

For example, the terms #int->v and u->bool are unified by the substitution (4.7) because

[u=#int, v=bool](#int -> v) = #int -> bool

and

[u=#int, v=bool](u -> bool) = #int -> bool.

4.2.5 Bases

A basis is a set of bindings of identifiers to types. A basis is analogous to the symbol table of a conventional interpreter, which is a list of bindings of identifiers to values. The basis

{X:u, F:int->bool}

represents the information that X has type u and F has type int->bool.

The types of standard identifiers are passed to a top-down type inference procedure in a special basis called the standard basis. The standard basis has the form

{..., CAR:#u->u, ..., PLUS:int*int->int, ...}

A basis can also be regarded as a partial function from identifiers to types. If

$$B = \{X_1:T_1,\ldots,X_n:T_n\},$$

then $B(X_i) = T_i$. If X does not occur in B, then B(X) fails.

There are two ways of adding a binding, X:T, to a basis B. The first, denoted by B++X:T, corresponds to conventional scope rules. The second, denoted by B&&X:T, allows variables to be overloaded. The semantics of these operations are defined by the following rules.

1. The extension B' = B++X:T' can always be formed and B'(X) = T', whether or not B contains a binding for X.
2. If B does not contain a binding for X, then B' = B&&X:T' can be formed and B'(X) = T'.
3. If B contains a binding X:T and T&T' is a well-formed type, then B' = B&&X:T' can be formed and B'(X) = T&T'.
4. If B contains a binding X:T and T&T' is not a well-formed type, then B' = B&&X:T' cannot be formed.

4.2.6 A Partial Order for Types

The type of an object tells us something about its value. A type that allows a small number of values provides more information than a type that allows a large number of values. We formalize this observation by defining a partial order on types. The partial order is useful because it makes precise the idea of finding more information about a type during analysis of the program, and it permits the introduction of coercion in a systematic way.

There are a number of situations in which the conversion of a value from one type to another is always a safe operation. If a conversion of this kind is introduced implicitly by the system, it is called a **coercion.** Coercion from a type, T, to a type, T', is safe if the set of values of T is a subset of the set of values of T'.

Suppose that in a certain context a value of type T' is required and a value of type T is supplied. If the substitution is acceptable, we write $T \leq T'$.

If we consider types as sets of values, then $T \leq T'$ says that T is a subset of T'. The relation $\leq$ is clearly reflexive and transitive. If $T \leq T'$ and $T' \leq T$, then the types T and T' may be identified. Thus $\leq$ is a partial order. The rules for determining the situations in which $T \leq T'$ are given below. In rules 4, 5, and 6, assume that $T_1 \leq T_1'$ and $T_2 \leq T_2'$.

1. We define the set of conversions allowed between primitive types. There are no coercions between int and bool, but in a full implementation of Dee there might be conversions such as char to string.
2. If T is any type and u is a type variable, then $T \leq u$.
3. $T_1 \& T_2 \leq T_1$ and $T_1 \& T_2 \leq T_2$.
4. $T_1' \text{->} T_2 \leq T_1 \text{->} T_2'$.
5. $T_1 * T_2 \leq T_1' * T_2'$.
6. $T_1 * T_2 \leq T_1$ and $T_1 * T_2 \leq T_2$.
7. $T_1 + T_2 \leq T_1' + T_2'$.
8. $T_1 \leq T_1 + T_2$ and $T_2 \leq T_1 + T_2$.

Rule 6 says that a Cartesian product may be coerced to a product with fewer components, and Rule 8 says that a value may be coerced to a union that contains the type of the value. Rule 8 is used in ALGOL 68; rule 6 is its dual. Both rules are discussed by Reynolds [Reynolds 1981].

4.3 Type Inference

The type discipline of Dee consists of axioms and inference rules that enable us to assign a type to some of the expressions in the language. The result of applying the inference rules to an expression may be a simple type, a type scheme, or failure. The effectiveness of a type inference algorithm depends on an appropriate choice of rules.

For example, the constant 3 has a simple type, int. The type of [X] -> X is the type scheme, (u)u->u. Some expressions, such as XX, may have a type according to one set of rules but not according to another set. Other expressions, such as

([X]->XX)([X]->XX)

may not have a type under any reasonable set of inference rules.

The following properties are desirable in a type discipline.

1. The discipline should be sound: if a type T is assigned to an expression E, the evaluation of E does in fact yield a value of type T.
2. The discipline should be complete: if E_1 and E_2 are expressions with the same denotation, the same types should be assigned to them.
3. It should be possible to infer types by an efficient algorithm.
4. The language should be referentially transparent.

Although the precise theoretical limitations are apparently unknown, it seems to be difficult to design a type discipline that satisfies all of the requirements. Thus a practical system must compromise [Leivant 1983].

Well-typed Expressions

Let E be an expression, B be a basis, and T be a type. The statement B|E:T means that E is a well-typed expression with type T, and B is a basis that assigns a type to each free variable of E. If E is well-typed, has type T, and has no free variables, we write |E:T.

The following list contains the axioms (R1 and R2) and inference rules (R3 through R8) used in the type system of Dee.

R1 If K is a constant of type T and B is a basis, then B|K:T.

R2 If X is a variable and B(X) = T, then B|X:T.

R3 If B|E:T_1&...&T_n, then B|E:T_i for i = 1,...,n.

R4 If B|T_1:E, ..., and B|T_n:E, and T_1&...&T_n is a well-formed type, then B|E:T_1&...&T_n.

R5 If B|E:T_1 and $T_1 \leq T_2$, then B|E:T_2.

R6 If (B++X:T_1)|E:T_2, then B|([X]->E):T_1->T_2.

R7 If B|E_1:T_1->T_2 and B|E_2:T_1, then B|(E_1E_2):T_2.

R8 If B|E_1:T_1 and (B&&X:T_1)|E_2:T_2, then B|(let X = E_1 in E_2):T_2.

4.4 Type Inference Algorithms

Type inference algorithms are most naturally defined by recursive decomposition of expressions. An algorithm is a case expression with one arm for each of the basic expression forms. Type inference algorithms can be conveniently classified by the parameters that are passed in recursive calls [Leivant 1983]. If a basis for the free variables of the expression is passed, we call the algorithm "top-down". If the basis is not passed, we call the algorithm "bottom-up".

We can apply all of the type inference rules except one in a straightforward way. Rule R7, for application, is the interesting case. When we have $B|E_1:T_1\text{->}T_2$ and $B|E_2:T_1$, we apply the rule trivially to obtain $B|E_1E_2:T_2$. More often, however, the domain type of E_1 is not identical to the argument type, and we must match the types.

In the general case, we must infer the type of E_1E_2 from

$$E_1:T_{11}\text{->}T_{12} \,\&\, \dots \,\&\, T_{m1}\text{->}T_{m2}$$

and

$$E_2:T'_{11} \,\&\, \dots \,\&\, T'_{n1}.$$

The algorithm must find all values of i and j for which there is a substitution, S, such that

$$S(T_{i1}) = S(T'_{j1}).$$

For each S, we have $E_1E_2:S(T_{i2})$. In general, there will be several such types, and thus the type of the application is itself conjunctive.

Since several matches may occur at each node of the expression tree, type inference requires exponential time in the general case. Thus the method has two disadvantages. First, it is likely to be impractically slow. Second, it does not uniquely determine the type of a function at the point of its application.

These disadvantages are avoided in various ways in existing languages. Most languages do not support generic functions at all, and the problems do not arise. In Ada, a function may be generic, but its arguments must be simple types determined at compile time. In Mary2, parameters and arguments are matched one at a time without backtracking. Thus the exponential complexity of the general algorithm is avoided at the cost of imposing a discipline on the user. This discipline is apparently acceptable in practice.

The restriction that we make in Dee is that the type of an application may not be conjunctive. The effects of this restriction are: (1) the time required for type inference is reasonable in most cases, although the worst case is still exponential in the size of the expression; and (2) the type of a generic function is instantiated at the point of application, which provides useful information to the compiler.

This restriction does not reduce the language to the level of, say, Ada. Polymorphic functions can be passed to functions, and functions can yield both polymorphic and generic functions as their values.

There is another interesting feature of the type inference rules. When a variable is bound by abstraction (R6) or local definition (R8), the basis is extended by the addition of the bound variable and its type. R6 uses "++" to extend the basis and R8 uses "&&". This requires explanation because, formally, the expression

$$\textbf{let}\ X = E_1\ \text{in}\ E_2 \tag{4.2}$$

is equivalent to

$$([X]\text{->}E_2)(E_1) \tag{4.3}$$

and it would appear that the type of (4.3) could be inferred from rules R6 and R7. This observation is correct, and it is the route that Coppo follows [Coppo 1980]. It has the important theoretical advantage that an expression is well-formed iff it is strongly normalizable. The disadvantage for practical purposes is that there cannot be an algorithm for type inference.

There are two aspects of the special treatment of local definitions. The first is the so-called "let anomaly" and it applies to Milner's Algorithm W; the second is particular to Dee. In both systems, the expression

$$[I]\ \text{->}\ II \tag{4.4}$$

cannot be typed because the terms u and u->u, in which u occurs free, cannot be unified. Consequently,

([I] -> II)([X] -> X) (4.5)

cannot be typed either. Expression (4.6) can be typed, however, because I receives the type scheme (u)u->u which can be instantiated to either (v)v->v or (w)(w->w)->(w->w).

let I = [X] -> X in II (4.6)

The second aspect, particular to Dee, of the treatment of let is the use of "&&" to extend the basis. The effect of this is that a let binding may overload an identifier but a lambda binding cannot. The motivation for this distinction, as in Milner's algorithm, is practical. The overloading of lambda variables is undesirable because the value of a function may be affected by textually remote definitions.

4.4.1 Top-down Type Inference

Algorithm FT ("find types") is an extended version of Milner's Algorithm W [Milner 1978]. It is passed an expression, E; a substitution list, U; and a basis, B. In contrast to Milner's algorithm, which returns a unique type for an expression, Algorithm FT may return a conjunctive type. The equation

<S,T> = FT(B, E)

means "the result of applying FT to a basis, B, and an expression, E, is a substitution S and a type T". The equation is valid if FT succeeds.

In the description below, there are two operations that may cause failure. The first is the function MATCH used in Case 4. If T_1 and T_2 are terms and MATCH(T_1,T_2) succeeds, giving a substitution U, then $UT_1 = UT_2$. If there is no such substitution, types cannot be matched, and the algorithm fails. The second possible cause of failure is basis extension in Case 5: evaluating B&&X:T' will cause failure if B contains an entry X:T that cannot be overloaded by X:T'.

We describe the type inference algorithm by cases on the structure of E. I is the identity substitution.

```
1. The expression is a constant, K, of type T:
        return <I, T>.
2. The expression is a variable, X, and B(X) = T:
        return <I, refresh(T)>.
3. The expression is an abstraction [X] -> E:
        let <S,$T_2$> = FT(B++X:$T_1$, E) where $T_1$ is a new type variable;
        return <S, S$T_1$->$T_2$>.
4. The expression is an application, $E_1E_2$:
        let <$S_1$,$T_1$> = FT(B, $E_1$);
        let <$S_2$,$T_2$> = FT(B, $E_2$);
        let U = MATCH($S_2T_1$, $T_2$->$T_3$) where $T_3$ is a new type variable;
        return <U$S_2S_1$, U$T_3$>.
5. The expression is a local definition, let X = $E_1$ in $E_2$:
        let <$S_1$,$T_1$> = FT(B, $E_1$);
        let <$S_2$,$T_2$> = FT(B&&X:$S_1T_1$, $E_2$);
        return <$S_2S_1$, $T_2$>.
```

4.4.2 Soundness of the Type Inference Algorithm

Theorem

If <S,T> = FT(B,E), then SB|E:T.

Proof

The proof is by induction on the structure of E. The first two cases constitute the basis of the induction.

Suppose the expression is a constant, K, of type T. Then from the algorithm, <I,T> = FT(B, K) and it follows immediately that B|K:T.

Suppose that the expression is a variable, X, and B(X) = T. Then, from the algorithm, <I,T> = FT(B, X) and it follows immediately that B|X:T.

Suppose that the expression is an abstraction, [X]->E. Then

$$\langle S, ST_1 \text{->} T_2\rangle = FT(B, [X]\text{->}E)$$

and, from the algorithm,

$$\langle S, T_2\rangle = FT(B\text{++}X{:}T_1, E).$$

By the induction hypothesis, it follows that

$$S(B\text{++}X{:}T_1)|E{:}T_2 \quad \text{and so} \quad SB\text{++}X{:}ST_1|E{:}T_2.$$

Consequently, by rule R6,

$$SB|([X]\text{->}E){:}ST_1\text{->}T_2.$$

Suppose that the expression is an application, E_1E_2. From the algorithm, we have

$$\langle S_1, T_1\rangle = FT(B, E_1) \quad \text{and} \quad \langle S_2, T_2\rangle = FT(S_1B, E_2).$$

By the induction hypothesis, we can deduce

$$S_1B|E_1{:}T_1 \text{ and } S_2S_1B|E_2{:}T_2.$$

Also, since $US_2T_1 = U(T_2\text{->}T_3)$, we have

$$\langle US_2S_1, UT_3\rangle = FT(B, E_1E_2).$$

Consequently,

$$US_2S_1B|E_1{:}US_2T_1$$

and hence

$$US_2S_1B|E_1{:}U(T_2\text{->}T_3) \qquad (4.7)$$

and

$$US_2S_1B|E_2:UT_2. \tag{4.8}$$

Finally, applying rule R7 to (4.7) and (4.8), we have

$$US_2S_1B|E_1E_2:UT_3.$$

Suppose that the expression is a local definition, let $X = E_1$ in E_2. From the algorithm we have

$$\langle S_1,T_1\rangle = FT(B, E)$$

and hence, by the induction hypothesis,

$$S_1B|E_1:T_1$$

which we may rewrite as

$$S_2S_1B|E_1:S_2T_1. \tag{4.9}$$

Also from the algorithm,

$$\langle S_2,T_2\rangle = FT(B\&\&X:S_1T_1, E_2)$$

and, by the induction hypothesis,

$$S_2(B\&\&X:S_1T_1)|E_2:T_2$$

or

$$S_2B\&\&X:S_2S_1T_1|E_2:T_2. \tag{4.10}$$

Hence, applying rule R8 to (4.9) and (4.10), we have

$$S_2S_1B|(\text{let } X = E_1 \text{ in } E_2):T_2.$$ Q.E.D.

The proof is similar to Milner's proof of Algorithm W. The difference between the algorithms is confined to the methods for basis extension and type inference for applications (Case 4). In both instances, the function MATCH replaces unification. This does not affect the soundness of the algorithm and MATCH does not even appear in the proof. It affects the completeness of the algorithm, however, because matching may succeed where unification fails.

4.4.3 Completeness of the Type Inference Algorithm

The converse of the preceding theorem is not valid: there are well-typed expressions whose types cannot be inferred from the algorithm. For example, as explained above, the algorithm fails for expressions (4.4) and (4.5).

4.4.4 Type Matching

MATCH is a relation on type expressions. In general, given two type expressions, T_1 and T_2, MATCH returns a set of substitutions:

$$\mathrm{MATCH}(T_1,T_2) \Longrightarrow \{S_1,\dots,S_n\}.$$

Each substitution, S_i, satisfies the following condition: if $S_i = I$, then $T_1 \leq T_2$; otherwise there is a type T_i such that $T_i = S_i T_1 = S_i T_2$.

The set of substitutions may be empty, in which case the types cannot be matched. It may contain one substitution, in which case the match is unique. It may contain more than one substitution, in which case the match is ambiguous. The following examples illustrate these possibilities.

```
MATCH(int,bool) ==> {},
MATCH(bool,bool) ==> {I},
MATCH(bool&int,int) ==> {I},
MATCH(u->int,bool->v) ==> {[u=bool,v=int]},
MATCH(int->int&bool->bool,u->u) ==> {[u=int], [u=bool]}.
```

It is important to distinguish between the first example, in which the match fails, and the second and third examples, in which the match succeeds with the identity substitution. The third example corresponds to the choice of type int for an expression with the overloaded type int&bool. The fourth example illustrates a match between two polymorphic functions, of which the first has a polymorphic parameter type and the second has a polymorphic result type. The last example shows what happens when a generic type is matched with a polymorphic type: two substitutions are possible.

The detailed description of MATCH follows. We assume that MATCH is called with type expressions T_1 and T_2, and we describe the algorithm by cases.

We use some special notation in the algorithm. A type expression is atomic if it is a primitive type or a type variable. If T_1 is a conjunctive type, it is assumed to be $T_{11}\&\ldots\&T_{1m}$. If T_2 is a conjunctive type, it is assumed to be $T_{21}\&\ldots\&T_{2n}$. The union of the sets MATCH(T_{11},T_2), ..., MATCH(T_{1m},T_2) is written [MATCH(T_{1i},T_2), i=1,...,m]. The union of the sets MATCH(T_1,T_{21}), ..., MATCH(T_1,T_{2n}) is written [MATCH(T_1,T_{2j}), j=1,...,n]. The symbol "U" in Case 8 denotes set union.

The tests must be performed in the order given. We describe only successful matches. If the conditions are not satisfied, the match fails. The partial order, $\leq$, is required only for primitive types; its values for composite types are computed by the algorithm. We do not give rules for union and product types because they are obvious extensions of the rules given.

```
1.  $T_1$ is a type variable:
      return MATCH-VAR($T_1$,$T_2$).
2.  $T_2$ is a type variable:
      return MATCH-VAR($T_2$,$T_1$).
3.  $T_1$ and $T_2$ are both atomic:
      if $T_1 \leq T_2$
        then (3.1) return {I},
        else (3.2) return {}.
4.  $T_1$ is atomic:
      if $T_2$ is conjunctive
        then (4.1) return [MATCH($T_1$,$T_{2j}$), $j = 1,...,m$],
        else (4.2) return {}.
5.  $T_2$ is atomic:
      if $T_1$ is conjunctive
        then (5.1) return [MATCH($T_{1i}$,$T_2$), $i = 1,...,m$],
        else (5.2) return {}.
6.  $T_1$ is conjunctive:
      if $T_2$ is conjunctive
        then (6.1) return [MATCH($T_{1i}$,$T_{2j}$), $i = 1,...,m$, $j = 1,...,n$],
        else (6.2) return [MATCH($T_{1i}$,$T_2$), $i = 1,...,m$].
7.  $T_2$ is conjunctive:
      return [MATCH($T_1$,$T_{2j}$), $j = 1,...,n$].
8.  $T_1 = T_{11}$ -> $T_{12}$ and $T_2 = T_{21}$ -> $T_{22}$:
      RES := {};
      for each R in MATCH($T_{21}$,$T_{11}$) do
        RES := RES U MATCH(R$T_{12}$,R$T_{22}$);
      return RES.
```

The algorithm MATCH-VAR is described below. It is called with two arguments: v, a type variable, and T, a type expression.

1. If v occurs in T, return {}.
2. If $T = T_1\&...\&T_n$, return {[$v=T_1$],..., [$v=T_n$]}.
3. Otherwise, return {[v=T]}.

In order to establish the correctness of MATCH, we need a precise statement of what it does.

Theorem

If S is a member of the set of substitutions returned by MATCH(T_1,T_2), then there is a type T such that $T = ST_1 = ST_2$. If MATCH(T_1,T_2) returns the empty set, no such type exists.

Proof

The proof is by structural induction on the type terms. There are four kinds of type term: primitive type, type variable, conjunctive type, and function type, and hence sixteen cases. For each case, we must prove that the algorithm identifies the case correctly, returns the appropriate substitution if it exists, and fails otherwise. The proof, which we omit, is straightforward but long.

Basis Extension Revisited

If B is a basis for which $B(X) = T_1$, the basis $b' = B\&\&X{:}T_2$ exists iff MATCH(T_1,T_2) fails. Intuitively, B' cannot be formed if there are any values that could have both of the types T_1 and T_2. As we have seen, this is precisely the condition that MATCH detects.

4.4.5 Bottom-up Type Inference

Leivant introduced a new type algorithm and surveyed various methods of polymorphic type inference [Leivant 1983]. His Algorithm V performs bottom-up type inference: given an expression E, V returns a multi-basis B and a type T. If <B,T> = V(E), then B|E:T. (A multi-basis differs from a basis in that there may be multiple entries for a particular identifier.) Algorithm V has several advantages over Milner's Algorithm W, and an extended version of it was used in an early version of Dee.

The advantage of Algorithm V is that it is conceptually simpler than Algorithm W because it separates the tasks of inferring types and applying a type discipline. It works well if standard functions are treated as constants with known types.

In Dee, standard functions can be overloaded. Thus standard functions are not constants and must be treated as free variables by Algorithm V. The multi-basis returned by the algorithm tends to be large even in quite simple cases because it contains an entry for each occurrence of each standard function. For this reason, we have found Algorithm V to be unsuitable for Dee.

4.4.6 . Other Approaches

Coppo introduced the idea of type conjunction [Coppo 1980]. Type conjunction in Dee is close enough to Coppo's that we use the same terminology and notation, but his method does not require generic type variables. The theoretical advantage of Coppo's type discipline is that an expression can be typed iff it is strongly normalizable. The disadvantage for practical purposes is that type inference is undecidable. By retaining generic type variables in Dee, we can guarantee that MATCH terminates, but only at the expense of not being able to assign a type to some normalizable expressions.

The language HOPE [Burstall 1980] provides type inference for variables with simple types but requires type declarations for functions. Polymorphic functions are provided and can be defined by the programmer.

Holmstrom defines a high level polymorphic language and a lower level monomorphic language [Holmstrom 1983]. The higher level language contains macros that expand into expressions in the lower level language. This approach avoids the "let anomaly" because the lower level language is monomorphic. The expansion of II leads to II', in which I and I' are distinct functions (cf. (4.4)). We can compare these languages to L_1 and L_2 in Dee. The difference is that polymorphism is retained in L_2 for compatibility between interpretation and compilation.

The type system of B is elegant and powerful. The system performs type inference incrementally, and the algorithm is sound and complete [Meertens 1983]. The language B, however, does not provide higher-order functions.

The type system of Letschert is similar to ours [Letschert 1984]. Letschert's system has a different approach to type conjunction, although the motivation is similar. The type inference algorithm is bottom-up rather than top-down. This increases the opportunities for parallelism but, as we have noted, may yield results in a form less suitable for compilation.

4.5 Examples

We relax the restrictions that we have employed for the formal development. The examples make use of functions with more than one argument. The standard basis contains the names "+" and "*" used in Dee to denote the functions that add and multiply integers.

Suppose that we have introduced a new type, RAT, for rational numbers, with constructor MAKE-RAT. We can extend "+" to add rationals by defining

```
+ = [U,V] -> let MAKE-RAT(NU,DU) = U,                    (4.11)
                 MAKE-RAT(NV,DV) = V
             in MAKE-RAT(NU*DV + NV*DU, DU*DV).
```

The type-inference algorithm assigns the type RAT*RAT->RAT to the function being defined. The "+" within the body of the definition has type int*int->int. We express the type of "+" within the scope of this definition as a conjunctive type:

```
int*int->int & RAT*RAT->RAT.                              (4.12)
```

Within the scope of definition (4.11), an expression of the form X+Y will be accepted if X and Y are either both integers or both rationals.

We can overload "+" still further. The defined function adds corresponding components of two lists. It could be used, for example, in a module that provides facilities for polynomial manipulation. PF and PR stand for "first of P" and "rest of P"; similarly for Q. The symbol "." denotes infix CONS.

```
+ = [P,Q] -> case P of                                    (4.13)
                NIL() : Q;
                PF.PR : case Q of
                           NIL() : P;
                           QF.QR : PF+QF.PR+QR
```

The type inference algorithm determines that the "+" in PR+QR is a recursive invocation of the function being defined. The "+" in PF+PQ, on the other hand, is not a recursive invocation; it has the generic type given in (4.12). The type of the function defined in (4.13) is

#int*#int->#int & #RAT*#RAT->#RAT

and the type of "+" within the scope of this definition is

int*int->int & RAT*RAT->RAT

& #int*#int->#int & #RAT*#RAT->#RAT.

A program that makes extensive use of overloading and contains no type declarations may be unreadable. Type inference and overloading are powerful tools that can be misused. We do not believe, however, that a language designer can prevent users from writing bad programs. We hope to ameliorate the problem of unreadability in two ways. First, although Dee does not require type declarations, it does permit them. Thus a conscientious programmer can provide as many type declarations as necessary to make a program intelligible to other programmers. Second, since the compiler has to infer types anyway, it can include the inferred types in an annotated listing of the program.

It is evident that the compiler will make extensive use of both space and time. Global program analysis is required for efficient code generation, and the number of names visible within a scope is larger than in a conventional system. If overloading is used extensively, type lists become long and time is spent checking inappropriate combinations. We do not know yet whether this will turn out to be a serious problem in practice, but in the meantime we are investigating ways of improving the efficiency of the type inference algorithm.

4.6 Extensions of the Type System

We have specifically excluded some type constructs from Dee. In this section, we briefly describe some of these constructs and explain their omission.

4.6.1 Subtypes

A type T_1 is a subtype of a type T_2 if every value of T_1 is also a value of T_2. In Pascal, for example, the type 1..100 is a subtype of the type <u>integer</u>. In the presence of subtypes it is not possible to infer the type of a constant by examining its value.

In its present form, Dee does not allow subtypes. This restriction is not necessary, however, in the presence of coercion. We can assign to each constant the "smallest" type possible and coerce it to a "larger" type when necessary. As a practical restriction, the compiler should use only types defined within the current scope; it should not invent new types of its own. For example, in the presence of types 0..5, 10..15, and **int**, the type of 4+14 should be **int** rather than 10..20.

4.6.2 Types as Parameters

In Dee, a type variable, like any other, can be abstracted, and in this sense Dee provides types as parameters. The processing of parametric types, however, is completed before the program is interpreted or compiled. No computation with types is performed at run-time.

4.6.3 Parametrized Types

A type may be parametrized with another parameter or with a constant. For example, the bounds of an array type can be considered as parametrizing the array type. As mentioned above, Dee allows a type to have type parameters but not numeric parameters. A consequence of this is that all Dee structures are either fixed in size or conceptually infinite. This restriction makes some styles of programming inefficient but simplifies the semantics of the language and, in many cases, the task of writing programs.

4.6.4 Enumerated Types

There is really no reason for excluding enumerated types in Dee, and they may eventually be added to the language. There were problems in Pascal with the scope of the identifiers of an enumerated type [Welsh 1977] but these seem to have been resolved in more recent languages such as Euclid [Lampson 1977] and Ada.

"Thus, in sundry ways you may furnish yourself with such strange and profitable matter: which, long has been wished for. And though it be naturally done and mechanically, yet it has a good demonstration mathematical."

5 Implementation

Experience with LISP systems has shown that an interactive environment should provide facilities for both interpreting and compiling programs. In general, interpreted functions and compiled functions should be able to call one another.

Several conditions must be satisfied if interpretation and compilation are to coexist. The most important of these is that the interpreted code and the compiled code must both execute within the same run-time environment. In particular, the system must ensure that a function is not applied to an argument of the wrong type. In an interpreted environment, functions usually check the type of their arguments before using them. In a compiled environment, types are usually checked at compile-time and not during execution. Our decision to maintain a run-time environment precludes the use of graph-reduction methods of compilation, at least during program development.

LISP contains a number of generic functions, such as CONS, that can be used by the interpreter or the compiler without type checking. For a function such as PLUS, which requires integer arguments, a naive compiler can simply invoke the interpreter's PLUS, thereby performing a run-time type-check. A more sophisticated compiler may be able to recognize that an argument must be of a particular type and to generate in-line code that is more efficient. For example, ADD1 always returns an integer and so TIMES can be compiled in-line in the expression (TIMES 5 (ADD1 N)). Modern LISP compilers use an ad hoc form of type inference to identify such situations and produce numeric code comparable to that of a good FORTRAN compiler [Fateman 1973].

In this chapter, we describe techniques required for a complete implementation of Dee. Since a complete implementation does not yet exist, however, we indicate the status of the existing implementation at appropriate places in the text.

5.1 Dee Processing

The interpretation or compilation of an L_1 program requires several phases of analysis. We describe these as if they follow one another, but in practice a certain amount of overlapping is both possible and desirable. The phases are summarized in the following list and described fully in the remaining sections of this chapter.

Parse

The result of parsing an L_1 program is a tree with LISP-like structure.

Apply basic transformations

Special forms of L_1 and expressions containing patterns are converted to L_2 forms.

Perform type inference

Each variable is tagged with its principal type.

Label generic functions

Each applied function now has a unique type and the interpreter or compiler can effect the appropriate instantiation.

Apply user transformations

Transformations coded by the user may be applied at this point, either interactively or automatically.

Perform call-by-value analysis

The default method of passing parameters is "by name". For efficiency, this is converted to call-by-value wherever possible.

Select representations

The default representation of a value is a pointer to its location, as in LISP. For some applications, such as numerical computation, the value itself provides faster access.

Apply final transformations

These transformations simplify code generation.

Generate code

Three steps involve program transformations. The division of transformations is important because transformations must be applied at appropriate levels. We anticipate that in practice, user transformations will comprise several levels. When the fourth step, labelling generic functions, has been completed, the program can be interpreted.

5.1.1 Parsing

The result of parsing a program is a directed acyclic graph that is represented by a suitable data structure. One of the non-terminals of the grammar is "infix-expression": an expression that contains operands, prefix operators, and infix operators. The parser is required to process infix operator directives by modifying its tables dynamically to ensure that occurrences of newly defined operators are parsed correctly. The easiest way to satisfy this requirement is to write an ad hoc recursive descent parser with a procedure that parses infix expressions according to the latest available information. This is the method used in the current implementation of Dee. The parser was constructed by an LL(1) parser generator.

There are advantages to using an LALR(1) or LR(1) parser generator [Aho 1977]. In principle, the behaviour of an LR(1) parser can be modified by parser directives just as the behaviour of an LL(1) parser can be modified. In practice, however, the relation between LR(1) parsing tables and the behaviour of the parser is much more complicated than the relation between the recursive procedures of an LL(1) parser and the behaviour of the parser. Consequently, we have not attempted to construct an LR(1) parser for Dee.

The Dee parser uses LISP-like property lists to represent the relation between operators and function names. An operator is either unary prefix or binary infix. The effect of the directive

- = infix(DIFF,100,105) (5.1)

is to give the identifier "-" the properties BINOP, NAME = DIFF, LEFTPREC = 100, and RIGHTPREC = 105. This enables the parser to translate an expression such as

a - b - c (5.2)

to either

DIFF(DIFF(a,b),c) (5.3)

or, in practice, directly to the L_3 form

(DIFF (DIFF A B) C). (5.4)

The directive (5.1) also assigns the property OP = "-" to the symbol DIFF, thereby enabling the pretty-printer to reconstruct the form (5.2).

It is not possible to associate an operator with several functions, or vice versa. Prefix and infix operators are syntactic devices that are unrelated to the semantics of polymorphism and overloading.

The combination of a simple grammar with provision for user-defined operators has been used in functional languages such as HOPE. Other languages allow the user more or less to design a language suited to a particular problem domain. Mary2, for example, has a general method for defining operator syntax but all operators have the same precedence. R-Maple [Voda 1983] provides a simple and elegant method for arbitrary syntactic extension based on the relationship between patterns and trees.

5.1.2 Basic Transformations

Several simple transformations are applied to an L_1 program to reduce it to an L_2 program. Expressions that contain patterns are replaced by equivalent expressions that contain selectors, recognizers, and new variables. Case expressions are transformed to IF expressions. An important basic transformation corresponds to beta-reduction in the lambda calculus. The general rule for beta-reduction is, in the notation of Chapter 4:

([x] -> e)(a) ==> [x=a]e. (5.5)

Beta-reduction is not usually applied in this form because it is not likely to improve the efficiency of the prorgam. In general, x will occur more than once in e and the substitution will increase the amount of computation required. Special cases of beta-reduction, however, subsume a number of conventional compiler optimizations.

We can sometimes omit a parameter and the corresponding argument from an application. The following transformation is valid if x_i does not occur in e.

$$([x_1,\ldots,x_i,\ldots,x_n] \text{ -> } e)(a_1,\ldots,a_i,\ldots,a_n) \tag{5.6}$$
$$\text{==> } ([x_1,\ldots\ \ldots,x_n] \text{ -> } e)(a_1,\ldots\ \ldots,a_n).$$

If x_i occurs only once in e, it is reasonable to make the substitution at compile-time:

$$([x_1,\ldots,x_i,\ldots,x_n] \text{ -> } \ldots x_i \ldots)(a_1,\ldots,a_i,\ldots,a_n) \tag{5.7}$$
$$\text{==> } ([x_1,\ldots\ \ldots,x_n] \text{ -> } \ldots a_i \ldots)(a_1,\ldots\ \ldots,a_n).$$

Transformations (5.6) and (5.7) look somewhat similar. The difference between them is that in (5.6), x_i does not occur in e, and in (5.7), x_i occurs exactly once in e. Finally, as a result of these tranformations, it may happen that we eliminate all of the parameters and arguments from an application. In this case, we can apply the transformation

$$([] \text{ -> } e) \text{ ==> } e. \tag{5.8}$$

An application $f(a_1,\ldots,a_n)$ may be replaced by its value if the value of each argument is known at compile time. This is true even if f is a user-defined function. The conditional expression p=>x|y can be simplified to x or y if p can be evaluated at compile-time.

These transformations are familiar and have been used in recent LISP compilers [Brooks 1982]. The language L_1, however, provides a particularly favorable environment for employing them because it has a small number of special forms, normal order semantics, and no side-effects. The transformations described are performed by the current implementation of Dee.

5.1.3 Type Inference

The system maintains a current environment. This is a data structure containing the name, type, and value of each accessible object. (These objects are usually, but not necessarily, functions.) Initially, the current environment contains entries for the standard functions, including operators for the primitive types and polymorphic operators for the basic data structures. When a declaration is processed, an entry for the new object is added to the current environment.

When a new expression is encountered, it is subjected to type inference. During type inference, each occurrence of each identifier in the expression is replaced by a pair consisting of the identifier and its type. If an identifier is overloaded, it may be associated with several different types in a single expression. Using this decorated form of the expression, the interpreter or compiler can select appropriate instantiations of overloaded identifiers and appropriate representations for data.

The current implementation does not attempt incremental type inference. If an expression is changed, the type inference algorithm is applied to the entire new expression.

Type inference may fail. As we have seen in Chapter 4, there are two possible causes of failure. After examining f(x), the system may complain that "$f{:}T_1$ cannot be applied to $x{:}T_2$". This error occurs either because f is not a function or because x is not in its domain. The other cause of error is an invalid attempt to overload an identifier. This occurs during type inference of

```
let x = a ...
```

if the new type of x, inferred from the type of a, conflicts with the type or types of x in the surrounding environment.

5.1.4 User Transformations

At this stage, the program has call-by-name semantics and is referentially transparent. This is an appropriate point at which to apply problem-specific transformations.

The L_2 program is represented by a data structure that can be expressed in L_1. Transformations can be expressed in L_1 by case expressions in which the case labels are patterns. Thus L_1 is an approporiate metalanguage for coding transformations, and we do not need a further level of notation.

5.1.5 Call-by-value Analysis

The language L_1 has call-by-name (CBN) semantics whereas L_2 has call-by-value (CBV) semantics. This discrepancy must be resolved during compilation. It is performed as the last step before code generation because this allows all other transformations to assume normal order evaluation.

Mycroft has formulated a theory and a practical technique for converting from call-by-need to CBV [Mycroft 1980]. This suffices for Dee because call-by-need is merely an optimization of CBN. It is unnecessary to discuss the theory in detail here, but we mention the basic idea. The important point is that a program may terminate under CBN but not under CBV. The reason for this is that arguments may be evaluated under CBV even if their values do not contribute to the final value of the expression. If an argument of an application necessarily terminates, the argument can be passed by value. Conversely, if a parameter of a function is necessarily evaluated when the function is invoked, the corresponding argument can be passed by value. We can assume that primitive functions terminate and, with the exception of a few special forms such as if, that they evaluate all their arguments. With this information, most calls by name can be safely converted to CBV. Calls that cannot be converted to CBV are implemented by passing closures.

5.1.6 Representation Selection

Type inference provides most but not all of the information required for the choice of an appropriate representation. Suppose that we have obtained the types n:int and y:#int by type inference in the expression

CONS(2 * (n + 1), y).

Clearly, the appropriate representation for n is an integer value. The operators * and + can be applied to integers, yielding an integer result. Type inference instantiates CONS to have type int*#int->int. This is misleading, however, because the polymorphic function CONS requires its first argument to be a pointer to a value, rather than the value itself. The code for 2*(n+1) must therefore be followed by code to create a cell for that value and to return a pointer to it.

The foregoing example illustrates the general case. Type inference can be relied upon for the choice of representation except when a polymorphic function is instantiated, in which case the arguments and result must be pointers. This restriction does not apply to generic functions because the appropriate generic instantiation has already been chosen by the time that code is generated.

5.1.7 Final Transformations

Our approach is similar to that used in the RABBIT compiler for SCHEME [Steele 1978] and subsequently in other LISP compilers [Brooks 1982]. There are a number of simple transformations that can be applied to lambda expressions to improve the quality of the generated code. The correctness of these transformations is assured by lambda calculus semantics.

As a simple example of a transformation that can be carried out at this stage, we consider the expression

```
if P & Q                                                    (5.9)
   then A
   else B.
```

Since P and Q have no side-effects, we can short-circuit the evaluation of P&Q. Expression (5.9) transforms to:

```
if P                                                        (5.10)
   then
     if Q then A else B
   else B.
```

The duplicated evaluation of B wastes space, especially if B is a large expression. In practice, we would not generate (5.10) but would transform (5.9) directly to (5.11).

```
let X = B in                                                (5.11)
   if P
     then if Q then A else X
     else X.
```

From (5.11), we can generate the machine code that we would expect from a compiler. The important point is that the transformations are applied to the source code rather than during code generation. Source-level transformations may interact and improve the resulting code in ways that would not be feasible if they were postponed until code-generation.

5.1.8 Code Generation

We can generate code either for a conventional von Neumann machine or for a machine with a novel architecture. There have been a number of proposals for graph-reduction machines for the execution of functional languages. An important advantage of graph reduction is that normal order reduction is achieved by default [Turner 1979]. Graph-reduction machines have also been proposed by Augustsson and Johnsson [Augustsson 1982] [Johnsson 1983]. Current graph-reduction compilers are slow, and it is not easy to see how interpreted and compiled functions could call one another in an environment based on graph reduction. Very few graph-reduction machines have actually been built, and initially it would be necessary write a simulator for an abstract machine and generate code for it.

The current Dee compiler generates machine code for a conventional processor. This enables intepreted and compiled functions to coexist and allows the system to use the same run-time environment for each.

5.2 Pretty-printing

As far as possible, parsing should be a minimal transformation that can be reversed. The reverse transformation is called "pretty-printing", "unparsing", or "back-translation". In LISP, pretty-printing is straightforward because there is a close relation between S-expressions and their external representation. In Dee, it is not always possible to restore all of the syntactic sugar in exactly the way that it was given. For example, the expressions

```
let x = a in e
```

and

e where x = a

have the same internal representation. The pretty-printer will use one form or the other for this expression, depending on options selected by the programmer.

The parser and pretty-printer use the same tables for processing infix expressions. This ensures that the pretty-printer inserts parentheses only when necessary and always uses the most recently defined infix operators.

5.3 The Run-time Environment

The key to designing a system in which the user can choose freely between interpretation and compilation is the run-time environment. Accordingly, we describe the environment before we discuss interpretation and compilation.

The oldest and simplest form of environment for a LISP-like language is an association list, or a-list. Each component of the list contains a name and a value. When a new scope is entered, local bindings are pushed onto the front of the a-list. When the interpreter needs the current binding of a variable, it searches the a-list from the front. Thus local bindings hide other bindings of the same variable in enclosing scopes, in accordance with normal scope rules.

In its original implementation, the a-list environment provides dynamic scoping: non-local variables are evaluated in the environment of the caller. We can use the same data structure for lexical scoping by modifying the interpreter so that the evaluation of an abstraction yields a closure containing the current environment. When a closure is applied, its body is evaluated in the environment of the closure.

This form of environment allows a function to yield a function as a value. We can define higher-order functions such as COMP, which returns the functional composition of its arguments, and TWICE, which returns a function that applies a given function twice.

```
let COMP = [f,g] -> [x] -> f(g(x)),
    TWICE = [f] -> [x] -> f(f(x));
```

For example, the application of TWICE to SUCC yields a function that adds 2 to its argument.

```
TWICE(SUCC) ==> [x] -> SUCC(SUCC(x))
```

The admission of higher-order functions into the language precludes the use of a stack for the environment because, in general, the environment is a tree.

We can make a further simple refinement to the structure of the environment. Instead of representing each local scope by a list of name/value pairs, we represent it by a list of identifiers and a list of values [Henderson 1980] [Wise 1982]. Consider the expression

```
let x = a, y = b in                                    (5.12)
  let x = c, z = d in e.
```

With an a-list environment, the environment of e is

```
((x.c) (z.d) (x.a) (y.b)).                             (5.13)
```

With the modifed structure, the environment of e is

```
( ((x z).(c d)) ((x y).(a b)) ).                       (5.14)
```

Some of the values in the environment are closures, the result of evaluating abstractions. The environment of a closure is represented by a pointer to a component of the current environment. In most cases, this pointer points to the enclosing scope — the next component of the environment list. The environment of a recursive function or a set of mutually recursive functions is the current scope, so in this case the environment contains cycles.

If we use the form of (5.14), the position of a value in the environment can be specified by two integers corresponding to its positions in the "outer" and "inner" lists respectively. The first integer corresponds to the static nesting level of the variable and the second to its "offset" in the current "frame". (The terminology indicates the close relationship between this data structure and the stack of a traditional block-structured language.) These integers are lexical **addresses**, and they can be generated by a compiler.

Another feature of the form (5.14) is that application of user-defined functions is efficient. The application $f(x_1,\ldots,x_n)$ is represented by (F.L), in which L is the argument list. If f is a user-defined function, its evaluation yields a closure (P E U) in which P is the formal parameter list, E is the body of the function, and U is the environment in which the function is defined. The value of the application is obtained by evaluating E in the environment ((P.L).U).

Names are unnecessary when compiled code is being executed, and they are easily omitted from the data structure. If we compiled (5.12), the run-time environment during the execution of the code for e would be

((nil.(c d)) (nil.(a b))). (5.15)

On the other hand, it may be useful to have names in the environment during debugging. In "debug mode", the compiler can generate the environment of (5.14).

We can refine this data structure in two stages to improve efficiency of access to values. First, we can replace the variable and value lists by arrays of pointers (known to LISP programmers as "hunks"). With this refinement, it is still feasible for the compiler and the interpreter to use the same run-time environment.

The second stage is to replace the array of pointers to values by the values themselves. With this refinement, the value list has the same structure as an activation record in the conventional implementation of a language of the ALGOL family. This data structure has two disadvantages: it is difficult to interpret programs, and it is difficult to implement polymorphic functions.

The current version of Dee uses neither of these refinements. The run-time environment is a linked list, as in (5.15).

The elaboration of a Dee program has two phases. First, an environment for the evaluation of expressions is established. Second, one or more expressions are evaluated in this environment.

Suppose that the first phase is performed by an interpreter. When a function definition is encountered, the interpreter constructs a closure containing pointers to the global environment and to the expression that represents the body of the function. If, on the other hand, the declarations are compiled, the compiler constructs a closure containing a pointer to the compiled code for the body of the function.

During the second phase of elaboration, control switches between the interpreter, evaluating uncompiled functions, and compiled code, evaluating compiled functions. Execution becomes faster as more functions are compiled.

It is clear that L_1 cannot be interpreted directly. Certain initial transformations, followed by type-inference, are required. In many cases, however, a relatively small amount of reprocessing is actually needed. For example, the user will typically change a function definition or transformation and re-evaluate only that function or a small section of the code around it.

5.4 Modules

In the foregoing sections, we have discussed the processing of a program as if the program were a monolithic entity. But in general this is not the case because Dee programs consist of modules.

In Dee, the unit of interpretation is arbitrarily small -- any expression may be interpreted -- but the unit of compilation is the module. The requirement that a module may be compiled separately places restrictions on the analysis that the compiler can perform and on the efficiency of the compiled code. We discuss first the basic issues of incremental compilation and then the more interesting issues of interfaces and the strategy for recompiling several modules when only one has changed.

Both the semantics and implementation of a module can be described in terms of the effect of importing a module into the current environment. If E is an expression and M is a module that exports a collection, D, of declarations, then the effect of

```
import M; E
```

is the same as the effect of

```
let D in E.
```

The Dee user interacts with the system at the level of modules. (This is described in greater detail in Chapter 6.) When a module is "opened", the system must restore or create the environment for that module. This may be done in any of several ways, depending on the history of the module.

Suppose that the module is opened for the first time in interpretive mode. Each of its declarations is interpreted, and an environment is created in which expressions within the module can be evaluated interactively. The processing of declarations is quite rapid because very little work is actually done: types are inferred and checked, and closures are built from abstractions. If the module is now closed, the environment is retained as one of its attributes.

If the interpreter encounters an import directive, it looks for the environment attribute of the imported module. If none exists, it reports this fact to the user. (Alternatively, it could call itself recursively to construct the required environment. This is potentially a slow operation, however, and should be performed only when explicitly requested.)

The compiler operates in a similar fashion, creating a "compiled" environment for the module instead of an "interpreted" environment. When it encounters an import directive, it looks for the compiled environment of the imported module, but it will accept the interpreted environment.

This simple scheme is complicated by various factors. The most important and obvious problem, common to all modular languages, is ensuring the consistency of modules. We must investigate the circumstances under which recompiling a module will affect other modules. A second problem, one that arises only in languages that provide generic features, is the compilation of a module that exports generic functions. Dee provides two solutions to problems of this kind. The first solution requires only local analysis and can be performed incrementally. The second requires global analysis but produces efficient object code.

5.4.1 Recompilation

If a module M_1 imports objects from another module M_2, directly or indirectly, we write "M_1 is a client of M_2", or "M_2 is a server of M_1". A module cannot be a client or server of itself.

When a module is altered and recompiled, its clients and servers may also have to be recompiled. In other languages that provide modules or a similar feature, altering a module usually requires recompiling its clients. Recompilation is necessary because the changes will have altered relative addresses in relocatable code: without recompilation, errors would occur during linking.

The circumstances in a Dee system are different for two reasons. First, a Dee module exports an environment rather than relocatable code. The environment is accessed by lexical addresses that provide an extra level of indirection; this obviates the need to recompile a client simply because code sizes have changed. (Recompiling would still be required if the values of exported constants were changed, but in Dee these also go in the environment.) On the other hand, the use of generic functions in Dee may require the recompilation of a server.

Suppose that module M is changed, but its exported interface does not change; that is, the same identifiers are exported and they have the same types as before. In this case, the clients do not have to be recompiled. If the names or types of exported identifiers were changed, recompilation of the clients is pointless because type conflicts would be inevitable. In this case, the system should mark the clients "to be changed" and inform the user.

Next, we consider generic functions. At the source level, a generic function can be imported. In compiled code, however, one or more instantiations of the generic function are linked to the client module. When a client module is altered and recompiled, the server may be called upon to provide different instantiations of the generic functions. Thus altering a module may require recompilation of its servers.

In general, a module that exports generic functions has to maintain separate code for each instantiation. When it is requested to provide a new instantiation, it merely creates new code. Old code is not destroyed because it may be referenced by other clients.

In contrast to other systems, in which changes flow from servers to clients, changes in Dee flow from clients to servers. The natural way to compile an entire Dee program is top-down, first compiling modules without clients. When a module is changed, code may be added to its servers.

The method used by Dee speeds up program development by minimizing recompilation, but the code generated is less efficient because there is more indirection. It is possible in principle to use global analysis during compilation, but this is likely to take considerably longer than incremental compilation because multiple passes would be required to resolve overloads. Global analysis would be useful at the end of development to produce a production quality program.

There was an early version of Dee with modules. The current implementation, which was developed for the purpose of experimenting with the type system, does not support modules.

5.5 An Architecture for Dee

The description of a machine designed specifically for executing Dee programs is beyond the scope of this thesis. In any case, Dee is not sufficiently different from other functional languages to require a special architecture: any of the machines proposed for functional languages could execute Dee programs.

It is of some interest, however, to consider the kind of enhancement or modification to a conventional von Neumann architecture that would improve the performance of Dee. Dee programs, like programs written in other functional languages, spend a large amount of time looking for their data. Their performance could be improved by additional addressing modes in the instruction set. For example, an addressing mode that permitted an arbitrary number of indirections (that is, traversing a linked list), followed by an address offset, would enable any operand to be accessed with one instruction. The same addressing mode would also be useful for data access in any block-structured imperative language. Imperative programs, however, typically have fewer levels of procedure nesting than functional programs.

"But herein great care I have, least length of sundry proofs, might make you deme, that either I did misdoubt your zealous mind to virtuous school: or else mistruct your able wits, by some, to guess much more."

6 A Programming Environment

A programming environment provides a programmer, or a team of programmers, with facilities for the development of software. A programming environment is effective if it facilitates the rapid development of correct software. In contrast to early programming environments, which provided minimal tools for developing programs in several languages, several recent programming environments are dedicated to providing a high level of support for a single language.

Dee was designed to be used in a particular kind of programming environment. We outlined some of the requirements of such an environment in Chapter 1. In this chapter, we give a detailed description of the proposed Dee programming environment and discuss the relationship between this environment and the languages L_1 and L_2.

The word "environment" is used in this thesis in two quite different senses. In Chapters 2 through 5, we used "environment" to denote an internal data structure containing bindings required for the evaluation of expressions. In this chapter, we use "environment" to denote a collection of hardware and software that assists a programmer during the development of a program. The intended use should be clear from the context.

6.1 Requirements

We can derive requirements for an effective programming environment by considering the characteristics of earlier programming languages that made them amenable to software development. The characteristics that we consider significant are enumerated below.

Rapid prototyping

It must be possible to obtain a working version of a program with minimal attention to detail. Rapid prototyping enables fundamental design and specification errors to be detected before a major investment in software development has been made.

Modularity

Separating the internal mechanism of a portion of the program from its external interface provides a powerful abstraction mechanism.

Interaction

The traditional edit/compile/test cycle is too slow for modern software development. Separate compilation of modules helps but may be insufficient. The environment must provide mechanisms for entering and testing code rapidly. This requires an interpreter and/or an incremental compiler.

Type checking

Thorough type-checking is useful both for detecting errors and generating efficient code.

Representation

The programming environment should permit flexibility in the choice of representation and it should allow programmers to refine representations during program development.

Semantics

Explicit semantic principles are useful to both the designer of a language and its users. We can reason about programs only if the language has a robust semantic foundation. Languages with expressive power can be constructed from a small set of appropriate semantic principles and well-chosen syntactic sugar.

Efficiency

Implementations of applicative languages are often inefficient. We do not subscribe to the view that cheaper and faster hardware will make efficiency considerations irrelevant.

Conciseness

Language designers must compromise between verbosity and hieroglyphics. As a general trend, however, we should expect programs to become shorter rather than longer.

These requirements are not independent. Some of them tend to go together: for example, postponing choice of representation is an important aid to rapid prototyping. Others tend to be incompatible, at least if we base our experience on existing languages. For example, modularity and strong typing are associated with compiled, non-interactive languages, whereas interaction and rapid prototyping are associated with interpreted, untyped languages.

Nor are the requirements particularly novel. LISP fulfills many of them. Most implementations of LISP, however, are not applicative, use dynamic scoping, and have call-by-value semantics. LISP programs, other than those restricted to an applicative subset of LISP, are not amenable to source-to-source transformations and are difficult to maintain. APL is suitable for prototyping. A number of recent languages, including Ada, Mesa [Geschke 1977], and Modula-2 [Wirth 1983], provide facilities for creating program modules that can be separately compiled.

6.2 Meeting the Requirements

Dee is an experimental system, and we envisage an adaptable environment in which different techniques of program development can be explored. In designing Dee, we considered all of the requirements listed above for a programming environment.

Applicative

Dee is a purely applicative language. There is no assignment statement, and the value of an expression is determined by the environment in which it is evaluated.

Type checking and type inference

Dee accepts, but does not require, type declarations. The principal type of each object is inferred by the compiler.

Abstraction

The principle of abstraction in Dee is simple and powerful: any name can be abstracted from an expression, turning the expression into a function of which the name is a parameter. This abstraction mechanism provides polymorphic and generic functions, higher-order functions, and abstract data types.

Call-by-name semantics

An applicative computation that terminates under call-by-value (applicative order) semantics also terminates under call-by-name (normal order) semantics. The converse is not true. Some program transformations are valid under call-by-name semantics or call-by-value semantics but not both.

Modules

Dee programs consist of modules that can import the environments of other modules and can export selected names to other modules. This contrasts with the "flat" name-space of early implementations of APL and LISP.

Multi-level

Dee encourages separation of concern by providing several notations, each appropriate to a particular level. Programs can be manipulated by the user at all levels.

Extensibility

Dee has extensible syntax. Extensible languages were popular for a time but went out of favor because syntactic extensions are not useful without semantic power. We believe that the semantics of Dee, based on function application, are sufficiently powerful to provide a basis for a variety of useful syntactic extensions.

Compilation

Even with today's fast and cheap hardware, high-level interpreted languages make heavy demands on both time and space. Any system intended for software development must be capable of translating programs into object code of reasonable quality.

In addition to the "large" operations of editing, interpreting, and compiling, the environment should also provide "small" services. For example, the user should be able to obtain reasonably rapid responses to questions such as the following. Who calls F? By whom is F called? What is the type of E? How often is F called during the evaluation of E? How often was transformation T applied during the compilation of E? What transformations can be applied to E?

6.3 Programming Methodologies

A programming environment must support various kinds of programming activities. These activities include rapid prototyping, systematic development of large programs, program transformation, and the ability to re-use substantial amounts of software.

6.3.1 Rapid Prototyping

The term "rapid prototyping" is used by engineers and software engineers to denote an interim product that simulates some of the characteristics of a proposed final product [Squires 1982]. A prototype program, for example, might have a simplified user interface and low time and space efficiency yet be able to provide potential users with useful "hands on" experience. Traditional programming languages are not suitable for this kind of program development because they tend to be oriented towards either prototyping or production programs but not both. Consequently, prototype programs are often written in a language other than the language of the final product [Gomaa 1981].

6.3.2 Structured Growth

A related methodology, in which development takes place within a single language, has been given the name "structured growth" [Sandewall 1978]. Starting from a simple program with a clean structure, a complex program is built by small, cautious modifications [Kernighan 1976]. The dangers of this approach are obvious: the original, simple program may evolve into an unintelligible monster. Effective use of this method requires maturity and experience in the programmer.

6.3.3 Transformation

The transformational method also starts with a simple program and alters it [Burstall 1977]. There are several important differences between this and the preceding methods. Transformations are intended to alter the efficiency of the program but not its semantics. An important criterion for the acceptibility of a transformation rule is that it does not alter the meaning of a program.

6.3.4 Program Manipulation

The methods described above have a common feature: a program changes form during its development. This raises the problem of how to represent the program while it is being changed.

The traditional solution to this problem is to represent the program by its source text. This method has many disadvantages. The most important disadvantage, which underlies all the others, is that the programming environment can provide only minimal support. This approach typically leads to an edit/compile/test environment in which an omitted semicolon may be expensive in both programmer time and machine cycles. This kind of annoyance is ameliorated slightly by syntax-driven editors and modular programming languages, but the dependence on program text remains.

The success of LISP-based programming environments is due in large part to the close relationship between the external and internal representations of a LISP program. Most attempts to build LISP-like environments for other languages have been unsuccessful. This is because the need for translation between surface language and internal structure pervades every part of the system [Sandewall 1978].

Significant advances in methodologies based on program manipulation will require radical innovations in the notation that we use for writing programs [Backus 1979] [Meertens 1984].

6.4 The Dee Programming Environment

A Dee system consists of a collection of modules. This collection constitutes a database and presents the usual problems of database maintenance, in particular the necessity for maintaining consistency. The modules represent "information" that can be examined and altered, as in a conventional database, but can also be executed because they are programs. The modules also constitute an interface between the user and the system. A user interacts with the system by "opening" a module. When a module is open, the objects defined within it may be examined, altered, executed, and compiled. Objects imported by the module may be examined and executed but not altered or compiled.

6.4.1 Rapid Prototyping

Rapid prototyping is supported by a complete set of type constructors and facilities for both functional and data abstraction. Programs are concise because most declarations can be omitted during the early stages of development. The ability to define prefix and infix operators allows the use of a notation appropriate to the problem domain. Interpretation permits functions to be tested as soon as they have been written. The language provides simple default external reprsentations for objects of all types, including user-defined types. Thus it is not necessary, in the early stages of program development, to write parsing and pretty-printing functions for new data types.

6.4.2 Transformation

Transformation capabilities are obtained by using L_1 as a metalanguage for L_2. The type of an L_2 program is provided as a standard data type of L_1. A generic mapping function, with functional parameters, maps one L_2 program into another. A transformation consists of a recognizer that identifies a particular kind of program and an instantiation of the generic mapping function with appropriate arguments.

Dee does not contain a built-in theory of program correctness. Thus programs are not automatically verified beyond the usual level of consistency required by a compiler. The ability to represent L_2 programs as L_1 data, however, makes it practicable to construct a theory in L_1 and apply it to L_2. (This is analogous to the manipulation of theorems of PPLAMBDA in ML [Gordon 1978b].)

6.4.3 Structured Growth

The modes of interaction in Dee are closely related to the modular structure of Dee programs. At a given point in the interaction, we want extended access rights to one module and limited access rights to others that depend on it or on which it depends.

When we initially activate Dee, the visible environment contains only standard constants and identifiers. To extend this environment, we "open" a module. Opening a module gives us access to the source text of the module and enables us to evaluate functions defined within it. We can change the text of a function defined in the opened module and test the revised function. We can also evaluate functions imported from other modules by the opened module. We do not, however, have access to the source text of these external functions, and we cannot alter them.

This method of interaction encourages the correct use of modules by enforcing "information hiding", but it is too restrictive for practical use. We must be able to open several modules simultaneously. With suitable hardware, the "window" paradigm might be an appropriate choice. Each module has its own window, and different windows can be provided with different acces rights.

When we "leave" a module, we may or may not want to preserve the changes, if any, that we have made to it. This choice can be made in the same way as in text editing, where we have the choice of "exiting" with any changes saved or "quitting" without saving changes.

Consideration of these requirements shows that a module must be a fairly complicated data structure. For reasonable performance, we require that the source text, the internal (L_3) form, and possibly the compiled code be components of an active module. We would like these three representations of the module to be consistent, but we may not want to incurr the overhead of recompiling affected modules after a small change. The type system of Dee creates a further burden because the system may not be able to compile a module until it has examined the requirements of its clients.

6.4.4 Maintaining Consistency

Maintaining the consistency of a large number of modules is difficult. The ideal solution would be to check the consistency of every change as it is made. We assume that, athough it might be feasible for small programs, this is in general impractical. For a practical implementation of Dee, we propose the following levels of consistency checking. The proposal is intended to provide a compromise between an unattainable ideal and a system that would provide no basis for confidence.

1. When a new object is introduced into the current environment, or an existing object is changed, the syntax of the new object is checked during parsing.
2. At any time, the user can request that an object be type-checked. This check will reveal inconsistencies in the number or type of arguments of a function and the misuse of generic functions.
3. When an expression is evaluated interpretively before its type has been checked, a type-check is performed.
4. When a module is compiled, the compiler checks for both internal consistency and consistency with the interface of imported modules. If compiling changes the exported interface of the compiled module, client modules are recompiled automatically or marked to indicate that recompilation is required.

5. When a collection of modules is compiled, the compiler attempts to optimize the generated code by using type information. Actual values, for example, rather than pointers to values, may be passed between modules.

In items 1 through 4, an inconsistency is reported, but the system takes no action beyond issuing the report. In item 5, the compiler attempts to remove the inconsistency by recompiling. For example, if the user attempts to interpret

a:T+b:T (6.1)

in an environment in which "+" is undefined for type T, the type checker reports an error. Compiling (6.1), however, would succeed if "+" were a generic function that can be instantiated for the type T.

6.5 Dee and the Operating System

An important consideration in the design of a programming environment is the number of existing operating system utilities that the environment should incorporate. A simple environment makes full use of the file managemant, text editing, assembling, and linking capabilities of the underlying operating system. An integrated environment incorporates all of these capabilities as components of itself.

A simple environment is easy to implement but has several disadvantages. For the user, the disadvantage is the need to switch between "modes" -- system mode, edit mode, program development mode, and so on. For the environment, the disadvantage is that it has no control over or even knowledge of what the user does outside it. Maintaining consistency is infeasible because, for instance, the user may alter a source file but not reload it. These disadvantages can be overcome partially by using an elaborate system of time-stamps.

An integrated environment has a number of advantages but requires considerably more implementation effort. The principal advantage to the user is that the environment presents a single, consistent interface, and therefore it can prevent meaningless or destructive acts by the user.

A compromise can be achieved if the underlying operating system provides adequate facilities. For example, it must be able to support concurrent tasks and allow a single user to switch between them. With such a system, source text editing can be interleaved with program development in a reasonably secure way [Sandewall 1978].

6.5.1 A Built-in Editor

One of the decisions that an implementor has to make is whether the user has access to the source text of a module after it has been presented to the system for the first time. The alternative is to provide access only to the pretty-printed form of the internal code. Some programmers feel that their personal layouts are sacrosanct. Nevertheless, there are advantages for both the user and the system in maintaining code in internal form only. The amount of time spent in parsing is reduced because a declaration or definition is re-parsed only when it is changed. Since the program exists only in its internal form, there is no need to maintain consistency between internal and external versions. This approach requires an editor that maintains a textual representation of an object on the terminal screen and an internal representation of the object in memory. The representations must be consistent while editing is in progress. Many LISP systems incorporate such editors.

6.5.2 A Built-in Assembler and Linker

The Dee compiler can be written in L_1 and is an essential part of the system. Ideally, its output would be a block of relocatable machine code that could be treated by the system like any other dynamic data object. Writing such a compiler requires a large amount of effort and is probably not justified for an experimental system. A simpler approach is to write a compiler that generates assembly code and to use the standard assembler and linker. This is the method used in the current implementation of Dee; it requires that the entire system be relinked when new compiled functions are installed.

"For, whereas, it is so ample and wonderful, that, an whole year long, one might find fruitful matter therein, to speak of: and also in practice, is a treasure endless: yet will I glance over it, with words very few."

7 Further Work and Conclusion

In this chapter, we outline directions for further research and bring the thesis to a conclusion.

7.1 Directions for Further Research

Dee extends the type inference system of ML in a practical way without sacrificing its useful theoretical properties. There are several ways in which the type system might be extended. It should be relatively straightforward to incorporate subtypes, because subtypes can be accomodated within the existing partial ordering of types. A more interesting problem is whether type inference can guide the introduction of the referencing and dereferencing operations needed during code generation. In this case, coercion cannot be controlled by the partial ordering. We would also like to investigate the use of parametrized types if they can be introduced without undue complexity. A deeper problem with the current type system is that it is syntactic. A semantic model for the type sublanguage is desirable [MacQueen 1982].

Dee was designed to facilitate program transformation. When the implementation is complete, we expect to use Dee to develop transformation techniques. The possibility of improving the performance of applicative programs by altering data structures in situ is of particular interest.

The semantics of modules in Dee are not entirely satisfactory. The view of modules as an abstraction of bindings seems fruitful, but its deeper consequences are not yet clear. We intend to investigate the semantics of bindings with the goal of making bindings "first-class citizens". This will lay the foundation for both parametrized modules and modules as parameters.

7.2 Conclusion

The description of a complete Dee system in Chapter 6 is speculative compared to the preceding chapters because a complete system has not yet been implemented. Nevertheless, some of the advantages of Dee over other languages for program development are apparent. These advantages include: data abstraction, multiple levels of representation, type inference, polymorphic and generic functions, and coercion.

The importance of Dee, however, is that it is a step in the direction of a new kind of programming environment in which the computing system provides support and assistance at all stages of the programming process. Ultimately, the environment will provide for the programmer the services of a librarian, diarist, manager, apprentice, and programming expert.

References

Aho, A. and Ullman, J. Principles of Compiler Design. Addison-Wesley, 1977.

Allen, J. The Anatomy of LISP. McGraw-Hill, 1978.

Ashcroft, E. and Wadge, W. R for semantics, ACM Transactions on Programming Languages and Systems, 4 (1982), 283-94.

Augustsson, L. Functional Compiler Status Report #1. Programming Methodology Group, Chalmers University of Technology, Goteborg, 1982.

Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM, 21 (1979), 613-41.

Barendregt, H. The Lambda Calculus: its Syntax and Semantics, revised edition. North-Holland, 1984.

Bauer, F., et al. Description of the Wide Spectrum Language CIP-L. Institut fur Informatik, Technische Universitat Munchen, 1983.

Berkling, K. Transformations in a Reduction System. In Symposium on Functional Languages and Computer Architecture (Aspenas). Programming Methodology Group, Chalmers University of Technology, 1981, 64-71.

Boom, H. Programming in the language of types. In CIPS Session '82 Proceedings (Saskatoon). CIPS, 1982, 109-16.

Boom, H. Private communication, 1984.

Brooks, R., Gabriel, R., and Steele, G. An optimizing compiler for lexically scoped LISP. In Proceedings of the SIGPLAN '82 Symposium on Compiler Construction (Boston, Massachusetts). ACM, 1982, 261-75.

Burstall, R. and Darlington, J. A transformation system for developing recursive programs, Jour. ACM, 24 (1977), 44-67.

Burstall, R., MacQueen, D., and Sannella, D. HOPE: An experimental applicative language. In Conference Record of the LISP Conference (Stanford, California). The LISP Company, 1980, 136-43.

Church, A. The Calculi of Lambda Conversion. Princeton University Press, 1941.

Coppo, M. An extended polymorphic type system for applicative languages. In Mathematical Foundations of Computer Science 1980, (Dembinski, Ed.). Lecture Notes in Computer Science 88, Springer-Verlag, 1980.

Coppo, M. On the Semantics of Polymorphism, Acta Informatica, 20 (1983), 159-70.

Dee, J. The Mathematicall Praeface to the Elements of Geometrie of Euclid of Megara, London, 1570. Reprinted with an Introduction by Allen G. Debus, Science History Publications, 1975.

Dewar, R., Schonberg, E., and Schwartz, J. Higher Level Programming: Introduction to the Use of the Set-Theoretic Programming Language SETL. Courant Institute of Mathematical Sciences, New York University, 1981.

Fateman, R. Reply to an Editorial, ACM SIGSAM Bulletin, 25 (1973), 9-11.

Geschke, C., Morris, J., and Satterwaite, E. Early experience with Mesa, Comm. ACM, 20 (1977), 540-53.

Goguen, J. Parametrized Programming, IEEE Transactions on Software Engineering, SE-10 (1984), 528-543.

Goldberg, A. and Robson, D. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.

Gomaa, H. and Scott, D. Prototyping as a tool in the specification of user requirements. In **Proceedings of the 5th International Conference on Software Engineering** (San Diego, California). IEEE, 1981, 333-42.

Gordon, M., et al. A metalanguage for interactive proof in LCF. In **Conference Record of the Fifth Annual Symposium on the Principles of Programming Languages** (Tucson, Arizona). ACM, 1978, 119-30. (Referenced as [Gordon 1978a].)

Gordon, M., Milner, A., and Wadsworth, C. **Edinburgh LCF. Lecture Notes in Computer Science 78**, Springer-Verlag, 1978. (Referenced as [Gordon 1978b].)

Griswold, R., Poage, J., and Polonsky, I. **The SNOBOL4 Programming Language**, second edition. Prentice-Hall, 1971.

Grogono, P. and Alagar, V. An Environment for High~~-level~~ Program Development. In **Programmiersprachen und Programmentwicklung** (Ammann, Ed.). **Informatik-Fachberichte** 77, Springer-Verlag, 1984, 144-55.

Harbison, S. and Steele, G. **A C Reference Manual.** Prentice-Hall, ~~1984.~~

Henderson, P. **Functional Programming Application and Implementation.** Prentice-Hall International, 1980.

Hindley, J.R. The principal type schema of an object in Combinatory Logic, **Trans. Am. Math. Soc.**, 146 (1969), 22-60.

Hoare, C. Recursive data structures, **Int. J. Comp. Inf. Sc.**, 4 (1975), 105-32.

Holmstrom, S. **A Flexible Type System.** Report 83.04R, Programming Methodology Group, Chalmers Institute of Technology, Goteborg, 1983.

Ichbiah, J., et al. **Military Standard Ada Programming Language.** ANSI/MIL-STD-1815A, U.S. Department of Defense, 1983.

Iverson, K. Notation as a tool of thought, Comm. ACM, 23 (1980), 444-65.

Jensen, K. and Wirth, N. PASCAL User Manual and Report, second edition. Springer-Verlag, 1978.

Johnsson, T. The G-Machine: an abstract machine for graph reduction. In Declarative Programming Workshop (University College, London). Programming Methodology Group, Chalmers Institute of Technology, 1983, 1-20.

Kernighan, B. and Plauger, P. Software Tools. Addison-Wesley, 1976.

Kowalski, R. Algorithm = logic + control, Comm. ACM, 7 (1979), 424-36.

Lampson, B. Report on the programming language Euclid, SIGPLAN Notices, 12, 2 (1977), 1-77.

Landin, P. A correspondence between ALGOL 60 and Church's lambda notation, Comm. ACM, 8 (1965), 89-101.

Landin, P. The next 700 programming languages, Comm. ACM, 9 (1966), 157-66.

Leivant, D. Polymorphic type inference. In Conference Record of the Tenth Annual Symposium on Principles of Programming Languages (Austin, Texas). ACM, 1983, 88-98.

Letschert, T. Type Inference in the Presence of Overloading, Polymorphism, and Coercions. In Programmiersprachen und Programmentwicklung (Ammann, Ed.). Informatik-Fachberichte 77, Springer-Verlag, 1984, 58-70.

Martin-Lof, P. Constructive Mathematics and Computer Programming. Paper read at the 6th International Conference for Logic, Methodology and Philosophy of Science, Hannover, August 1979.

McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Part I, Comm. ACM, 3 (1960), 184-95.

McCarthy, J. A basis for a mathematical theory of computation. In Proceedings of IFIP Congress (Munich). North-Holland, 1962, 21-8.

MacLennan, B. Values and objects in programming languages, SIGPLAN Notices, 17, 12 (1982), 70-79.

MacQueen, D. and Sethi, R. A Semantic Model of Types ~~for~~ Applicative Languages. In Conference Record of the 1982 ACM Symposium on LISP and Functional Programming (Pittsburgh, Pennsylvania). ACM, 1982, 243-52.

Meertens, L. Issues in the design of a beginner's programming language. In Algorithmic Languages (deBakker and van Vliet, Eds.). North-Holland, 1981, 167-84.

Meertens, L. Incremental polymorphic typechecking in B. In Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages (Austin, Texas). ACM, 1983, 265-75.

Meertens, L. Algorithmics: towards programming as a mathematical activity. In Mathematics and Computer Science, CWI Monographs, Vol. I (deBakker, Hazewinkel, and Lenstra, Eds.). North-Holland, 1984.

Milner, R. A theory of type polymorphism in programming, Jour. of Computer and Systems Sciences, 17 (1978), 348-375.

Mycroft, A. The theory and practice of transforming call-by-need into call-by-value. In International Symposium on Programming (Paris), (Robinet, Ed.). Lecture Notes in Computer Science 83, Springer-Verlag, 1980, 269-81.

Naur, P., et al. Revised Report on the algorithmic language ALGOL 60, Comm. ACM, 6 (1963), 1-17.

Nordstrom, B. Description of a Simple Programming Language. Report 1, Programming Methodology Group, Chalmers University of Technology, Goteborg, 1984.

Parnas, D. Information distribution aspects of design methodology. In Proceedings IFIP Congress (Ljubljana). North-Holland, 1971.

Rain, M. Mary2 Language Reference Manual, Version G. Penobscot Research Center, Deer Isle, Maine, 1980.

Reynolds, J. GEDANKEN - A simple typeless language based on the principle of completeness and the reference principle, Comm. ACM, 13 (1970), 308-19.

Reynolds, J. The Essence of ALGOL. In Algorithmic Languages (deBakker and van Vliet, Eds.). North-Holland, 1981, 345-72.

Sandewall, E. Programming in the interactive environment: the LISP experience, ACM Comp. Surv., 10, 1 (1978), 35-71.

Scott, D. Logic and programming languages, Comm. ACM, 9 (1977), 634-41.

Seldin, J. private communication, 1985.

Squires, S. (Chairman). Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop, ACM Software Engineering Notes, 7 (1982).

Steele, G. and Sussman, G. SCHEME: An interpreter for the extended lambda calculus. Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1975.

Steele, G. RABBIT: A Compiler for SCHEME. Technical Report 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1978.

Steele, G. COMMON LISP: The Language. Digital Press, 1984.

Stoy, J. Denotational Semantics. MIT Press, 1977.

Strachey, C. Towards a Formal Semantics. In Formal Language Description Languages for Computer Programming (Steel, Ed.). North-Holland, 1966, 198-220.

Turner, D. SASL Language Manual. University of St. Andrews, 1976.

Turner, D. A new implementation technique for applicative languages, Software: Practice and Experience, 9 (1979), 31-49.

Turner, D. The semantic elegance of applicative languages. In Proceedings of the 1981 Conference on Functional Languages and Computer Architectures (Portsmouth, New Hampshire). ACM, 1981, 85-92.

Turner, D. Private communication, 1983.

Vessey, I. and Weber, R. Research on structured programming: an empiricist's evaluation, IEEE Transactions on Software Engineering, SE-10 (1984), 397-407.

Voda, P. R-Maple: a concurrent programming language based on predicate logic. Part I: syntax and computation. TR 83-9, Department of Computer Science, University of British Columbia, 1983.

Voda, P. A view of programming languages as symbiosis of meaning and control. Technical Report 84-9, Department of Computer Science, University of British Columbia, 1984. (Referenced as [Voda 1984a].)

Voda, P. and Yu, B. RF-Maple: a logic programming language with functions, types, and concurrency. In Fifth Generation Computer Systems (Tokyo). Institute for New Generation Computer Technology, 1984.

Welsh, J., Sneeringer, W., and Hoare, C. Ambiguities and insecurities in Pascal, Software: Practice and Experience, 7 (1977), 685-96.

van Wijngaarden, A., et al. Revised report on the algorithmic language ALGOL 68, Acta Informatica, 5 (1975), 1-236.

Wirth, N. Program Development by Stepwise Refinement, Comm. ACM, 14 (1971), 221-7.

Wirth, N. Programming in Modula-2, corrected edition. Springer-Verlag, 1983.

Wise, D. Interpreters for Functional Programming. In Functional Programming and its Applications (Darlington, Henderson, and Turner, Eds.). Cambridge University Press, 1982.