LEDA: A Blending of Imperative and Relational Programming*

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
budd@cs.orst.edu

September 20, 1994

Abstract

This paper describes features of a new strongly typed, compiled programming language, called LEDA. LEDA attempts to combine aspects of both imperative (value-oriented) and logical (relation-oriented) styles of programming. Logical behavior is introduced by means of a new programming structure, called a *relation*. Relations have similarities to functions and procedures, but are distinct from both. Several examples are presented illustrating how the combination of features from the two paradigms are mutually beneficial. Finally a short overview of the implementation is given.

1 Introduction

The field of programming languages has undergone a transformation in the last several years, from considering programming languages each as representing just a particular syntax and perhaps an emphasis on a particular data structure, to recognizing the existence of diverse philosophies or outlooks in languages. These different approaches are called language *paradigms*, a term which emphasizes that the important distinguishing characteristic is one of "world-view" and not simply syntax.

Different paradigms arise because problems in disparate areas require diverse approaches to solution, and even in a single area contrasting techniques are often advantageous. Thus individual paradigms emphasize different aspects of both problem solving and programming. The imperative paradigm emphasizes variables and values, the functional paradigm concentrates on functional composition and the development of higher-order functions (functions using functions as arguments), the object-oriented paradigm places more importance on behavior than on values, and the relational

*To appear in *IEEE Software*

or logical paradigm deals with the relations between objects, and not necessarily the objects themselves. Clearly all of these techniques are useful at different times, and indeed in any sufficiently complex task a combination of approaches may be called for. Thus the field of "multi-paradigm" programming language research has developed [Hai86].

Combining languages is more than just connecting syntaxes, however. A language must have some internal consistency, the parts must work easily together. A natural blending is one that combines the best features of two or more paradigms and allows them to work easily together, as well as achieving the benefits of either individually. The development of such natural blendings has proved difficult. The present work is one more experiment in this vein. The programming language LEDA, described in the next section, is a strongly typed, compiled language that attempts to combine features of both the imperative (value or state-oriented) and logical (relational-oriented) styles of programming.

2 The language LEDA

We will not provide a detailed description of either the syntax or the semantics of the programming language LEDA; interested readers are referred to some of our other papers [Bud89a, Bud89b, Bud89c]. Instead we will simply describe those features necessary for understanding the examples we will subsequently present.

The imperative portion of LEDA is derived from the ALGOL tradition of languages such as Simula, Pascal or Modula. One notable difference is that functions are first class types, and function execution is terminated using an explicit `return` or `suspend` statement (the latter to be discussed shortly). Data types supported in LEDA include the standard types of integer, real and boolean, enumerated types, subprogram types, and object-oriented class types [Bud89a]. All types must possess some value used to indicate "undefined"; the state of not yet having been assigned a value. All variables are undefined prior to their first use, and the attempted use of an undefined value in a computation will result in a run-time error. A system-provided predicate `defined()` can be used to determine if a particular variable has been defined yet or not.

In most imperative languages, such as Pascal or C, there are two different mechanisms for structuring procedural abstractions; namely the procedure and the function. The first is used as a statement, whereas the second is an expression. Both can take values as arguments, and can modify external values through either global variables or through parameters which are passed by reference (pointer parameters in C)[1]. Relational programming is provided in LEDA by means of a third type of procedural abstraction, the `relation`. A relation is neither a procedure nor a function, although in imperative code a call on a relation can be made as if it were a procedure, and in relational code it is often thought of as a function of type `boolean`. Like a function or a procedure, a relation is defined with an argument list, most typically invoked using call-by-reference (or `var`) parameters. This allows information to flow both into and out through parameters.

The body of a relation consists of Prolog-style rules. (Those unfamiliar with Prolog might want to consult a reference for that language, such as [StS86]). As in Prolog, these can be thought of as

[1] This is not to say this is good programming practice. Functions which modify global variables or change values via parameters indiscriminately are often difficult to understand.

either representing facts or rules of inference. For example, the relation shown in Figure 1 can be thought of as a database of genealogical information. A fact, such as

child(helen, leda, zeus).

can be viewed as asserting that helen is a child of leda and zeus. Each of the separate lines of the body is an individual *rule*. The collection of rules constitutes all the knowledge in the system about the relation child. Note that names is an enumerated datatype that includes leda, zeus, helen, and so on.

A rule of inference describes how new relational information can be derived from existing relations. For example, the relation shown in Figure 2 can be thought of as describing how to determine if two individuals possess the "mother" relationship; the first is mother to the second if we have recorded in the child relationship that there is some value for the variable dad such that child(kid, mom, dad) is true. In addition to simply checking the veracity of facts, rules of inference can also generate values for which the relation is satisfied through a process known as *unification*.

Rules in the body of a relation can have multiple *clauses* that must each be satisfied for the relation to hold. For example, if we assume a relation female has been defined which indicates the sex of each person in the names enumerated datatype, then the relation daughter can be given as shown in Figure 3. This can be read as asserting that girl is a daughter of parent if and only if girl is female and parent is the mother of girl, or parent is the father of girl. As we will see in section 3.1, the body of a relation can also include arbitrary imperative style code surrounded by begin end pairs.

Within imperative code a relation is invoked as if it were a procedure. As in conventional procedures, unification can cause information to flow out across var parameters. For example, if mom is a variable of type names, the statement:

```
mother(mom, aeneas);
```

would assign mom the value aphrodite. Parameters are not specifically defined as input or output parameters in relations, however, and so can be used in both manners[2]. For example the statement:

```
mother(aphrodite, son);
```

would assign the variable son the value aeneas. A call on a relation, with some arguments potentially bound to constant values, is known as a *query*[3].

Of course, it is frequently the case that queries can be satisfied in a number of ways. For example, if we ask for a child of zeus we see, examining the database in Figure 1, that there are multiple answers as both helen and hercules are children of zeus. But only one value can be assigned to a variable at any one time. The solution provided in LEDA is to use *backtracking*. If a query has multiple solutions one response (the first, for example helen) will be returned. If subsequently any other relation turns out to be unsatisfiable, it will be said to *fail*. Failure will invoke backtracking,

[2] This is not strictly true, as information can only flow out through a parameter if it has been declared using the var specifier. Parameters not declared using the var modifier are always call-by-value, and are thus used only for input to a subprocedure.

[3] It is natural to wonder why a relation is treated as a procedure in imperative code, and not as a boolean function. The reason is that such a function could return *true* but, as we will see when we discuss the backtracking mechanism, will never return *false*, instead returning control to the last point at which there was an alternative.

which will restore the system to the most recent point where there was a choice of multiple solutions. This time the second solution will be returned. This will continue until a satisfactory alternative is found or all choices exhausted.

Failure, and its associated backtracking, can be forced from within imperative code by executing the special statement fail. As we will see in sections 3.1 and 3.2, multiple answers can also be produced in imperative code through the use of the suspend form of procedure return.

Functions, including relations, are first class values in LEDA. Thus, for example, relations can be passed as arguments. The code shown in Figure 4 illustrates a relation called closure that takes another relation as an argument. The relation closure is satisfied if the transitive closure of the argument relation is satisfied[4]. Passing "mother" as an argument to closure would capture the notion of arbitrary levels of "motherness" (maternal grandmother, maternal greatgrandmother, and so on). Used in this manner functional arguments are similar to functors in Prolog [StS86]. Further discussion of the use of functional arguments in conjunction with the object-oriented features of LEDA can be found in [Bud89c].

3 Blending Imperative and Relational Programming

Both the imperative and relational styles of programming benefit from the abilities provided by the other. The imperative style benefits from the succinct representation of relational information, from the bidirectional dataflow opportunities presented by unification, and from the unique control flow regime imposed by backtracking. The relational style benefits from a more direct way of dealing with negation and with numbers, two concepts that are difficult in pure logic programming.

The next two sections will illustrate, by means of several examples, how the two styles of programming are mutually beneficial, and can be usefully blended together. The third section shows how the mechanisms of choice points and backtracking can be utilized in imperative programs even in situations having nothing to do with relations. In languages which try to combine features from two or more different domains, what is often "lost" is a difficult-to-quantify sense of consistency, simplicity, coherence or clarity. We would like to argue that we have not fallen prey to these problems, but we appreciate the fact that this judgement is subjective, and that simplicity and coherence is very much in the eye of the beholder.

3.1 Using Imperative Statements in Relations

In LEDA arbitrary imperative style code can be used between rules in relations, by surrounding the code with begin end pairs. This can be used, for example, to trace the flow of execution in a relation as the various rules are invoked by the backtracking mechanism (Figure 5). As execution moves from one rule to the next any intervening imperative code will be executed. The values of any variables can be used within the imperative code, although the values of variables set by the relations will be undone by the backtracking mechanism before the imperative code is executed.

Suppose, for example, we try to discover the name of a grandchild of leda using the query grandchild(leda, kid), where kid is a variable of the appropriate type. First the imperative code will be executed, and the output from the statement writeln('test father-father descent') produced. Then

[4] Just with conventional recursion, the programmer must be careful to avoid infinite loops here.

an attempt will be made to find a value Z such that father(leda, Z) is true. This will fail, of course, since leda is father to no one. So the second writeln will be executed, and the second rule attempted. Again this will fail for the same reason. The output of the third writeln will be printed, and an attempt will be made to find a value Z such that mother(leda, Z) is true. This time this succeeds with Z bound to helen. However the second clause of the third rule then fails, since helen is father to no one. It is only on the forth and final rule that we discover that leda is grandmother to hermione.

Within a relational rule an imperative function can be invoked as long as the function returns a boolean value. Returning false from the function is treated the same as the failure of a clause, and invokes backtracking. This use of imperative functions in rules does not affect the meaning of the function when used in imperative contexts, where it is simply a boolean function. The imperative function notsame is used by the brother relation in Figure 6 to insure that an individual is not considered to be a brother to themselves. Again there are two ways that two individuals can be brothers; by sharing the same father or the same mother.

Consider a query brother(pollux, X) for example. First an attempt will be made to find the father of pollux, which is zeus. Next an attempt will be made to bind the variable X to another child of zeus, such as helen. Then a test will be made to make sure helen is male. This will fail, which will invoke backtracking to find another child of zeus. The next candidate in our database is pollux. Pollux is male, and thus the function notsame will be invoked with the arguments pollux and pollux. These match, and thus the notsame procedure returns false. This is interpreted as failure, and thus yet another attempt is made to find a third child of zeus. This time the candidate is hercules, who turns out to be both male and not the same as pollux. Thus the relation succeeds with the variable X bound to hercules. Should a subsequent relation fail and backtracking return to this point, the second clause of the relation will be tried, which will result in the variable X being set to castor (since they share a common mother, namely leda).

In this example the imperative procedure is not used to generate new values, but merely to filter out previously generated values and invoke backtracking if necessary. Here it is assumed the values of the variables X and Y will be set, if they were not set already when the relation brother was invoked, by the two invocations of the father relation (or, if that is unsuccessful, by the mother relations). Using imperative procedures is frequently useful when, as in this example, inequality is involved, since inequality is a difficult notion to handle in pure logic programming [StS86].

In Prolog two undefined (unbound) variables can be unified, so that if either variable is subsequently assigned a value this will be reflected in the values of both variables. LEDA does not support this level of unification, since to do so would require multiple levels of indirection for variables, inconsistent with the use of reference parameters in the imperative portion of the language and the simple implementation to be described in a subsequent section[5]. It can, however, support the weaker requirement that if either value is defined and the other undefined, the undefined value is set equal to the defined value. This is illustrated by the imperative procedure shown in Figure 7, which also shows use of the built-in predicate defined. The function same first tests to see if the first argument is defined. If it is, it then tests the second argument to see if it is also defined. If they are both defined it returns true if they are the same, and false otherwise. If either value is not defined the system procedure assign is used to assign the defined value to the undefined variable. The assign procedure is used instead of simply an assignment statement as it establishes the requirements necessary for

[5] For a good survey of unification and its implementation, see Knight [Kni89].

the assignment to be "undone" as a consequence of backtracking. This will be discussed more in Section 4. Finally, if both variables are undefined the procedure returns false.

We can extend this technique in order to define "relations" of a sort that are not easy to define using Prolog-style rules. The function recip in Figure 8, for example, maintains the property that X is the reciprocal of Y, assigning a value to either if one is set. The recip function returns false if both values are undefined, or if they are both defined but the relation is not satisfied. The function can be used either in a purely imperative setting, or as a clause within a relation.

3.2 Using Relations in Imperative Programs

The last section showed how imperative programming techniques are frequently useful from within relations. The converse is also true, namely that relations are often useful in the middle of otherwise imperative programs. We will illustrate this by means of several examples. To understand these, a small amount of information about the implementation is necessary. A more detailed explanation of the implementation will be given in section 4.

Whenever the system is faced with multiple solutions to a query, a record called a *choice point* is placed on the activation record stack. The choice point records all the information necessary to restore the state of the program to the point where the choice was made. Any time a failure occurs the most recent choice point is used to reset the system, and another alternative is tried.

In our first example the failure and backtracking mechanism is used to simulate a simple catch / throw control flow, similar to that of Lisp [Ste84], or to the Setjmp/Longjmp facility of C. This is shown in Figure 9. An imperative procedure calls the relation catch, which succeeds and unifies the parameter with the value true, but leaves a choice point on the activation record stack. Execution then continues in the imperative program. At some later point, perhaps even deep in a series of procedure calls, a throw can be made back to the catch routine simply by failing. In an imperative program this can be accomplished by executing the statement fail. Following execution of the fail statement control is returned to the last point of choice, which in this case was in the catch relation. The next alternative is attempted, which reassigns the parameter X the value false. Control is then once more "returned" from the relation, despite the fact that we have already once returned from this very same invocation. This time, however, the value of the actual parameter v is different, and thus we follow a different execution path.

If we use a global variable to transmit information from the sender to the receiver, we can extend this technique to build a general exception handling mechanism. Consider the code shown in Figure 10, which illustrates the coding of a *stack* module which uses exceptions to signal underflow and overflow. The enumerated type exception lists the types of exceptions, and the global variable stackexcept will hold values returned by the exception handler associated with a stack code. The relation handler is used to set down a choice point. When first invoked, it returns the value noerror. If subsequently a procedure, say the *push* routine, wishes to raise an exception it does so by calling the procedure raise. The raise procedure signals the exception, setting the global variable stackexcept to the appropriate value and returning control to the relation handler[6]. Handler uses the current value of stackexcept to assign a value to the reference parameter X before returning once again. This time, since the value of the variable except will have changed, the exception handler code will be

[6] To be precise it returns control to the most recent choice point, which we are here assuming to be the handler.

executed. The nature of the exception can be determined by examining the local variable except.

3.3 Backtracking in Imperative Programs

Examples in the last section illustrated how the fail statement could be used in imperative code to initiate backtracking to alternatives established by previous invocations of relations. It is possible to create choice points in purely imperative code. This is accomplished by means of a suspend statement, which acts like a return, but in addition places a choice point on the activation record stack. A subsequent failure will then cause execution to be continued with the statement immediately following the suspend.

Through the use of the suspend and fail statements, the mechanisms of choice points and backtracking are valuable in imperative programs independently of their utility in implementing relations. For example, they can be used to create *generators*. A generator is an expression that is capable of producing more than one result, producing values one at a time on demand. The concept was popularized in the language Icon [GrG83]. LEDA differs from Icon in being strongly typed and compiled, instead of interpreted, and in lacking the rich set of data structures provided in Icon. Nevertheless, the technique of programming with generators exploited by Icon programs can often be adapted to LEDA code. The function shown in Figure 11, for example, is an exact translation from a function shown on page 118 of [GrG83]. When given a value n, the function fibseq produces the first n Fibonacci numbers. It initially returns the first Fibonacci number, namely 1. If a subsequent failure is encountered, control is returned to the point following the suspend statement and the next number is returned. This continues until all n numbers have been generated, at which point the function fails.

The utility of generators is greatly increased through the use of the every statement. The preface every before a statement bounds backtracking within the statement, and forces the exploration of all possible alternatives. This is most useful if two or more generators are used within the statement, where it will force the exploration of all possible combinations of values. For example, suppose we have a generator primes which produces all prime numbers less than some specified value (the development of this is left to the reader); the following statement will print those values which are both Fibonacci numbers and prime numbers.

```
every
   begin
      i := fib(100);
      if i = primes(100) then
         writeln('number both prime and fib ',i);
   end;
```

As we will note in the next section, debugging programs written using generators and relations can sometimes be made difficult because of the backtracking mechanism returning control to an unanticipated earlier point in the program. We are currently investigating syntactic devices that would give the programmer greater control over the flow of execution in a program following failure.

4 Implementation

LEDA is implemented as a compiler producing 68000 assembly language. The details of code generation for the imperative portion of the language are conventional, and will not be discussed. Information on the implementation of first class functions can be found in [Bud89b]. In this paper we will describe only the implementation of the logical features of the language.

An invocation of a relation is treated similarly to an invocation of a function or procedure; an activation record containing space for the parameters (or pointers to them, in the case of var parameters), local variables, the dynamic chain and return information is created. The activation record register (fp in our notation), always points to the current activation record. A stack pointer (sp) points to the current top of the activation record stack. Code is then generated which sets the values of all local variables to the appropriate "undefined" value for the type of the variable.

When the program is faced with the possibility of several alternative answers to a query, either implicitly in the case of relations or explicitly in the case of a suspend statement in an imperative procedure, a special record is created and pushed onto the activation record stack. This record is called a *choice point*. The choice point record maintains sufficient information necessary to restart the system should another alternative be necessary. As with activation records, a register (cp) always points to the most recent choice point. Figure 12 shows a pseudo-code representation of the code generated for the child relation of Figure 1, and illustrates the code used to create a choice point.

The information stored in a choice point consists of the current value of the activation record register, the address of the previous choice point, a instruction address where execution will continue should backtracking be required, and enough information to "undo" the effect of certain assignment statements during the process of backtracking. This latter is accomplished by maintaining another register pointing to a separate data structure, called the *trail*. The trail is a stack of tag-address pairs, where the tag indicates the type of object assigned and the address of the location for the variable. Each choice point records the current location in the trail, indicated by the value of the trail register (tp). As part of the backtracking process, all values marked in the trail above this location are set once more to "undefined"[7]. Figure 13 gives a typical snapshot of the activation record stack, showing the presence of several activation records and choice points, as well as the trail.

If a choice point is to be used to restore the state of a program, it is important that the choice point not be removed from the activation record stack. This is accomplished by a simple change to the procedure epilogue, namely that the stack pointer is never decremented lower than the current choice point. (This can be determined simply by comparing the frame pointer and the choice point registers). If returning from a procedure call would result in this happening, the activation record for the procedure is simply left on the stack. This is true for both imperative procedures and relations. Note that this places the responsibility for removing arguments on the called routine and not, as is often done, on the caller.

Choice points will accumulate on the stack until they are used. Subtle difficulties in programming

[7] In general, backtracking requires that variables be reset to their old values. When used to construct relations, it will always be the case that the previous value of a variable is "undefined". This has the advantage that only the addresses of the altered locations need be remembered. We have emphasized in this paper, however, that backtracking can be used in contexts other than relations; whether a more general facility which remembers values as well as addresses is necessary in these circumstances is a debatable question.

using generators and backtracking can arize as a result of control flowing through unanticipated choice points. As we will mention in the conclusions section, we are currently working on developing techniques to give the programmer more complete control over the creation of choice points.

To implement backtracking is then simply a matter of restoring the values of the registers stored in the most recent choice point, undoing the effects of any assignments recorded in the trail, decrementing the stack to eliminate the current choice point (which may free activation records which were not previously released because they were below the choice point), and branching to the code location given in the choice point record.

Given this background, the code generated for relations can now be described. This code is inspired by the WAM (Warren Abstract Machine) for Prolog [MaW88], although we produce assembly language directly instead of virtual machine instructions.

We first consider the code generated when the head of a relational rule (the portion to the left of the :- symbol, or the entire rule in the case of a fact), contains constants. This occurs, for example, in the child relation of Figure 1. Here the generated code tests the value of the associated variables, assigning or branching to the failure routine as the situation may warrant. A pseudo-code description of the initial part of the code for this relation is shown in Figure 12.

Code generated for assign places an entry of the correct type in the trail stack before modifying the given location (as shown in Figure 12). A suspend(expression) statement generates code similar to the following:

```
        makeChoicePoint L1
        return expression
L1:
```

The every statement is similar, only it fails following the given statement, thereby forcing the statement to run through all possible alternatives before continuing. That is, code generated for every statement is as follows:

```
        makeChoicePoint L1
        statement
        fail
L1:
```

Invocations of relations in the right hand portion of relational rules are treated exactly like procedure calls, since the backtracking mechanism will automatically take care of control flow. Calls on functions (which must be declared to be of type boolean) are made into relations by surrounding them with a conditional tied to an explicit fail, as in:

```
if not func(X,Y) then fail
```

All other aspects of the implementation are straightforward and need not be discussed.

5 Conclusions

Once multiple programming paradigms have been identified, the integration of several paradigms into a single language is an appealing objective, and we are certainly not the first to try to combine Prolog

style relations with another language. Several researchers, such as Lindstrom [Lin85], and Darlington [DFP86], have attempted to add various features of Prolog to functional languages. Others, such as Rumbaugh [Rum87], and Korth [Kor86], target object-oriented languages. Languages such as PLEASE [Ter87] combine imperative and relational features, but are not compiled.

A language is more than just a collection of features, however, and thus the blending of two or more paradigms into a single language is a complicated undertaking. It is rather easy to interface features of two different languages; but often the result is like two separate worlds joined by a narrow opening. To combine two separate programming styles into one uniform linguistic whole in a manner that they flow naturally together and derive mutual benefit from each other is a subtle process; and we are not at all convinced yet that we have achieved this.

In our mind, the significant aspects of LEDA are as follows:

- The language is strongly typed and compiled. Although the implementation is not sufficiently developed for actual timings to be meaningful, the fact that the system is compiled rather than interpreted should mean the imperative portions will execute as fast as most compiled languages, and the relational aspects should be comparable to other compiled WAM-based systems.

- Relations are provided as a third type of subprocedure, similar in many ways to both functions and procedures. Relations can be used both in logical and imperative code, and imperative code can be used inside of relations.

- The two paradigms are mutually beneficial, each providing features advantageous to the other, so that the whole of the combination is more than merely the sum of the parts. For example, the mechanisms of choice points and backtracking necessary to support relations have utility quite apart from this use.

The development of truly multiparadigm programming languages opens up new possibilities for the computing community:

- They allow us to more easily compare and contrast solutions to problems presented in various forms for such features as succinctness, clarity, and efficiency.

- They allow students and programmers to become proficient in a variety of programming styles, and perhaps better understand the importance and appropriate use of techniques from different paradigms, while still working in only a single language.

- Perhaps most interesting, they provide for the possibility of software developed in a truly multiparadigm manner. In any sufficiently complex software system it is perhaps inevitable that different parts of the system might be most appropriately addressed with different techniques. In a compiler, for example, the parser might be best described using logical relations, while the symbol table could be implemented as an object, and optimizing transformations on an intermediate form described in a functional manner. All of these styles are possible in LEDA.

We emphasize that our experiences thus far are promising, but only beginning. The language itself is also evolving. In addition to adding more complex data structures, we are also investigating ways of controlling backtracking. This appears to be important if we are to fully utilize the generator

style of programming described in Section 3.3. Interestingly, the problem here is different from that of Prolog. The primary mechanism used to control backtracking in Prolog is the *cut*, which discards recent choice points. An equally necessary function is the ability to ignore early choice points created prior to entering some context; so that failure can be handled gracefully rather than passing control to some far flung location. We are currently investigating various alternatives to the every statement. One approach is similar to the drive statement of Cg [Bud82]. An alternative proposal is more like the notion of continuation in Scheme [SpF89]. There is also a question concerning whether simply copying the Prolog syntax for relational rules is a good idea, or whether an alternative form that uses explicit logical connectives (and, or and not) might be more efficient (since it would allow clauses to be left factored out of rules, for example) and natural.

Acknowledgements

Clearly many of the ideas in LEDA were adapted from earlier programming languages, notably Pascal, Prolog and Icon. I am indebted to the designers of these languages for providing me with such good material to work with. I am also grateful to David S. Warren for explaining to me the working of the David H.D. Warren Abstract (Prolog) Machine. Chris Fraser, Frank Griswold, Ralph Griswold, and the seven unnamed referees made detailed comments on earlier drafts of this paper.

References

[Bud82] Budd, T. A. An Implementation of Generators in C. *Computer Languages*, Vol 7(2): 69-88 (1982).

[Bud89a] Budd, T. A., Data Structures in LEDA, Oregon State University, Technical Report 89-60-17, August 1989.

[Bud89b] Budd, T. A., Low Cost First Class Functions, Oregon State University, Technical Report 89-60-12, June 1989. *submitted for publication.*

[Bud89c] Budd, T. A., Functional Programming in an Object Oriented Language, Oregon State University, Technical Report 89-60-16, August 1989. *submitted for publication.*

[DFP86] Darlington, J., Field, A.J. and Pull, H. *The Unification of Function and Logic Languages*, in Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall, Englewood Cliffs, New Jersy, 1986.

[GrG83] Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, New Jersy, 1983.

[Hai86] Hailpern, B. Multiparadigm Languages and Environments. *IEEE Software*, 3(1):6-9 (January 1986).

[Kni89] Knight, K. Unification: A Multidisciplinary Survey, *ACM Computing Surveys*, 21(1): 93-124 (March 1989).

[Kor86] Korth, H. F. Extending the Scope of Relational Languages, *IEEE Software*, 3(1):10-28 (January 1986).

[Lin85] Lindstrom, G. Functional Programming and the Logical Variable, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp 266-279 (January 1985).

[MaW88] Maier, D. and Warren, D.S. *Computing with Logic: Logic Programming with Prolog* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.

[Rum87] Rumbaugh, J. Relations as Semantic Constructs in an Object Oriented Language, *Sigplan Notices*, Vol 22(12):466-481 (December 1987).

[SpF89] Springer, G. and Friedman, D. P., *Scheme and the Art of Programming*, MIT Press, Cambridge, Mass. 1989.

[Ste84] Steele Jr., Guy L. *Common Lisp*, Digital Press, Bedford Massachusetts, 1984.

[StS86] Sterling, L. and Shapiro, E. *The Art of Prolog*, MIT Press, Cambridge, Mass. 1986.

[Ter87] Terwilliger, Robert B. PLEASE: A Language Combining Imperative and Logic Programming,

```
relation child( var name, mother, father : names );
begin
    child(helen, leda, zeus).
    child(hermione, helen, menelaus).
    child(castor, leda, tyndareus).
    child(pollux, leda, zeus).
    child(aeneas, aphrodite, anchises).
    child(telemachus, penelope, odysseus).
    child(hercules, alcmene, zeus).
end;
```

Figure 1: A Relation consisting only of Facts

```
relation mother( var mom, kid : names );
var dad : names;
begin
   mother(mom, kid) :- child(kid, mom, dad).
end
```

Figure 2: A Relation Implementing a Rule of Inference

```
relation daughter( var girl, parent : names);
begin
   daughter(girl, parent) :- female(girl), mother(parent, girl).
   daughter(girl, parent) :- female(girl), father(parent, girl).
end
```

Figure 3: A Relation with Multiple Clauses

```
type
     binaryNameFun : relation ( var names, var names );

relation closure ( F : binaryNameFun, var A, B : names );
var C : names;
begin
     closure(F, A, B) :- F(A, B).
     closure(F, A, B) :- F(A, C) , closure(F, C, B).
end
```

Figure 4: A Relation used as an Argument in another Relation

```
relation grandChild(var X, Y : names);
var Z : names;
begin
    begin writeln('test father-father descent'); end;
    grandChild(X,Y) :- father(X,Z), father(Z,Y).
    begin writeln('test father-mother descent'); end;
    grandChild(X,Y) :- father(X,Z), mother(Z,Y).
    begin writeln('test mother-father descent'); end;
    grandChild(X,Y) :- mother(X,Z), father(Z,Y).
    begin writeln('test mother-mother descent'); end;
    grandChild(X,Y) :- mother(X,Z), mother(Z,Y).
end;
```

Figure 5: Tracing Execution with a Relation

```
function notsame( X, Y : names) : boolean;
begin
     return X <> Y;
end;

relation brother(var X, Y : names);
var Z : names;
begin
     brother(X,Y) :- father(Z, X), father(Z, Y), male(Y), notsame(X,Y).
     brother(X,Y) :- mother(Z, X), mother(Z, Y), male(Y), notsame(X,Y).
end;
```

Figure 6: Using an Imperative Function within a Relational Rule

```
function same(X, Y : names) : boolean;
begin
   if defined(X) then
     if defined(Y) then return X = Y;
     else assign(Y, X);
   else if defined(Y) then assign(X, Y);
   else return false;
   return true;
end;
```

Figure 7: Simplified Unification in an Imperative Function

```
function recip( var X, Y : real ) : boolean;
var Z : real;
begin
   if defined(X) then begin
     Z := 1 / X;
     if defined(Y) and Y <> Z then return false;
     else assign(Y, Z);
     end
   else if defined(Y) then assign(X, 1 / Y);
   else return false;
   return true;
end;
```

Figure 8: An Imperative Function Used to Define Values

```
relation catch(var X : boolean);
begin
      catch(true).
      catch(false).
end;

begin
      catch(v);
      if v then
            ... normal execution
      else
            ... process caught execution
      ...
      fail; ...
end;
```

Figure 9: A simple catch - throw mechanism

```
type
    exception : (noerror, underflow, overflow);
var
    stackexcept : exception;

relation handler( var X : exception );
begin
    handler(noerror);
    handler(stackexcept);
end;

procedure raise ( X : exception );
begin
    stackexcept := X;
    fail;
end;

...
handler(except);
if except == noerror then
    normal execution
else
    handle raised exception
```

Figure 10: An Exception Handler built using Relations

```
function fibseq( n : integer) : integer;
var count, i, j, k : integer;
begin
    i := 1;
    suspend(i);
    j := 1;
    suspend(j);
    for count := 3 to n do begin
        k := i + j;
        suspend(k);
        i := j;
        j := k;
    end;
    fail;
end;
```

Figure 11: Using Choice Points to Implement a Generator

```
L1:       makeChoicePoint L2
          if not defined(name) then
                assign(name, helen)
          else if name <> helen then fail
          if not defined(mother) then
            assign(mother, leda)
          else if mother <> leda then fail
          if not defined(father) then
            assign(father, zeus)
          else if father <> zeus then fail
          goto return
L2:       makeChoicePoint L3
          ...
return:   if fp > cp then
            sp := fp - argument size
          goto return address
```

Figure 12: Code Generated for the Child Relation

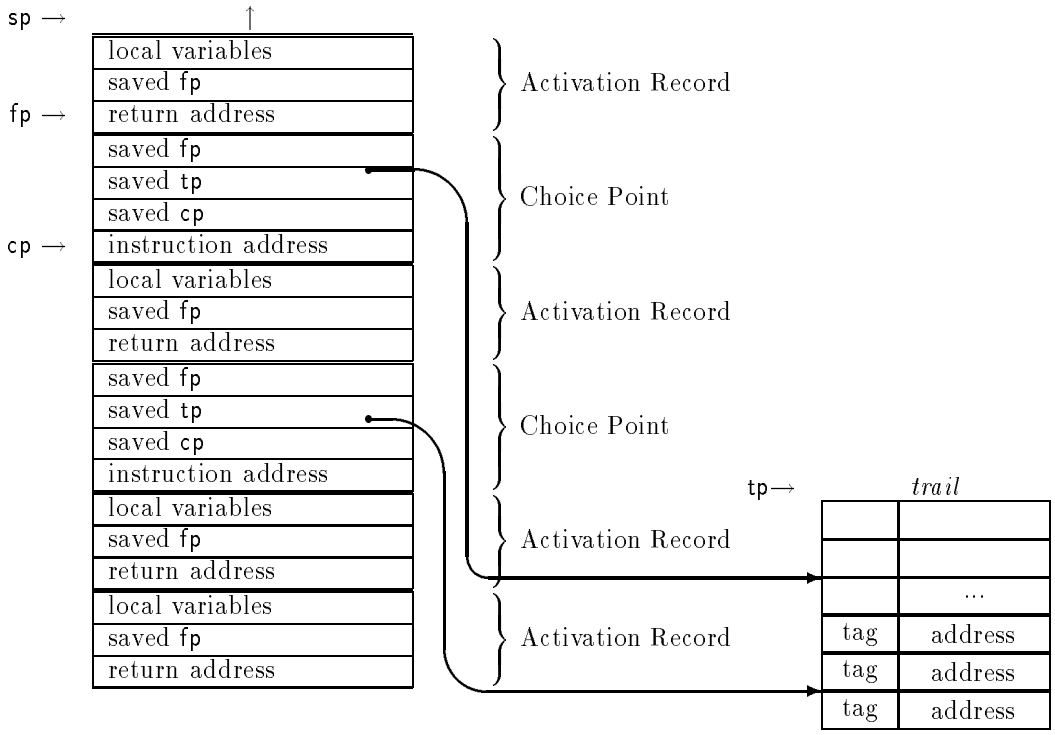

Figure 13: A Snapshot of the Activation Record Stack