Zephyr: Toward true compiler-based programming in Prolog

by Y. Asakawa
H. Komatsu
H. Etoh
Y. Hama
K. Maruyama

Prolog is widely used in prototyping, especially in artificial intelligence, but it has yet to gain widespread acceptance in application development. We think that the problems in this area result from the programming style enforced in existing Prolog systems. Zephyr is a new Prolog system refined and enhanced to help solve such problems. It allows users to do modular programming by always using a compiler instead of an interpreter. In this paper, we describe the unique features of Zephyr which make this possible, focusing especially on *package*, *metafunctions*, and *tables*, and the implementation of the system on OS/2.®

Introduction

With the inception of the fifth-generation computer project in Japan and the adoption of logic programming as its basis, Prolog has become famous as a promising AI programming language. Many research-based and commercially based Prolog systems now exist and are widely used for the prototyping of searching, database querying, simulation, design, planning, expert systems, compiler writing, natural-language processing, and so on. On the other hand, there are few really practical applications written in Prolog. The many possible reasons for this gap may be summarized as follows:

- Failure to meet application requirements such as execution performance and memory size.
- Failure to develop applications themselves because of software engineering problems.

Our research motivation is that in terms of both language and system, a programming style that is effective for conventional programming languages, which we call *true compiler-based programming*, helps solve the difficulties found in developing practical applications.

True compiler-based programming supports program modularity, the sharing and reuse not only of source code but also of object code, separate compilation, executable module generation, and debugging in compiled code. These are not fully realized in many current Prolog systems.

To realize true compiler-based programming, the language itself must also be refined and extended in the following aspects: the introduction of a package system to

provide name spaces for predicates, the use of interface definitions, the restriction of program modification, the introduction of closure for metaprogramming, the elimination of commands for the interpreter and debugger from the actual language part, and so on.

Since Prolog has many unique features that distinguish it from conventional programming languages, the key to supporting true compiler-based programming in Prolog is to determine how to support these unique features, especially *metafunctions* and *symbols* as data.

The first section of this paper analyzes problems in current Prolog systems. The next section describes the approach adopted in the Zephyr Prolog system. The following two sections deal with details from the viewpoint of language features and an implementation of Zephyr Prolog on Operating System/2® (OS/2®). The final section gives the results of our preliminary evaluation.

Problem analysis

• *In meeting application requirements*

In most practical applications, there are many requirements and restrictions which are not always considered in prototype systems: 1) execution speed, 2) working memory size, 3) application code size, 4) openness to other systems, and 5) hiding of source code. Let us consider these points with regard to Prolog.

1. *Execution speed.* After efficient compilation techniques were introduced by Warren [1], compilers became indispensable components for obtaining good performance in Prolog systems. Much research based on his work has been done, but often the code still fails to clear the performance requirement and is then rewritten in a conventional language, such as C. This means that Prolog systems must be open to other languages to allow performance-critical parts of applications to be rewritten in conventional languages, even in compiler systems. Of course, compilers should also have more effective optimizing techniques.
2. *Working memory size.* Unique features in Prolog, such as *unification, backtracking, "type-less" variables*, and *single assignment*, provide elegant and powerful ways of handling complex data structures dynamically. To support these features, programs written in Prolog require more memory than those in conventional languages. Optimization, done primarily in compilation, can reduce the size. For example, tail recursion optimization (TRO) [2] and clause indexing [1] eliminate redundant stack frames very quickly; thus, much more memory can be retrieved by "garbage collection" (GC) and used in further execution. Many existing Prolog systems do not allow an application to be separated from the language processor itself; that is, an application always includes the interpreter, debugger, and compiler, even though only the run-time library is necessary. In addition, Prolog possesses metafunctions such as **call**. A naive implementation always requires an interpreter at execution time. As a result, applications require more memory to run. To get good performance, especially on the segmented architecture typical of microprocessor-based machines, it is effective to restrict the maximum size of one memory block (stack, heap, symbol table, etc.) to the segment size, even if there is a large surplus of memory. However, this prevents many applications which use much memory from running.
3. *Application code size.* For commercially based applications, the maintenance cost is frequently related to the application code size. If an application includes a language processor, this raises the cost.
4. *Openness to other systems.* In addition to performance, openness is very important for integration and communication with other systems, such as existing databases and window systems, and for taking advantage of existing code written in conventional programming languages such as C and FORTRAN. The method of communication and code utilization depends on the individual system. For example, it sometimes involves issuing subtasks or processes, but at other times it is necessary to link object libraries.
5. *Source-code hiding.* It is desirable to hide valuable source programs for applications, especially in the case of commercial applications. That is true in Prolog because the knowledge used in an application may be described at a much higher level, but it is not possible in an interpreter system. Even in a compiler system it is not possible if the system supports only on-memory compilation and not the functions required to save and load compiled code.

• *In developing applications themselves*

To build a large application, the program must be divided into small pieces which can be developed and tested step by step, shared among many people, and reused. This is true whatever language is used. In Prolog, unfortunately, neither the language nor the systems encourage this. For example,

1. In the *de facto* standard Prolog, the so-called Edinburgh Prolog [3, 4], there is no certain way to avoid accidental conflicts of predicate names between separately developed programs. These occur frequently in the development of large systems and disturb that development. It is possible to avoid name conflicts only by using a compiler with special directives, but even this will fail if metafunctions such as **call** and **assert** are used.

2. In most Prolog systems, the result of compilation depends on the environment in which it is done [3, 5–7], because of implementation requirements such as easy maintenance of a symbol table and a predicate table that are required not only at compilation time but also at execution time. As a result, object code is generated in a format that does not allow it to be used in different environments. Programmers are therefore unable to share their programs as compiled code, to reuse them, to build their own object libraries, or to enhance existing libraries. All they can do is to keep and exchange source programs, and recompile them in each environment. (However, they may wish to hide the source code, as mentioned previously.)
3. The programming scenario most widely used in Prolog is as follows:

   - Build a source program using a text editor.
   - Start a Prolog session.
   - Consult utility programs.
   - Consult a user program.
   - Debug using the trace function of an interpreter.
   - Invoke a text editor and fix bugs.
   - Reconsult the program.
   - (Repeat this cycle).

   To use a compiler in this cycle, the programmer must always invoke a compiler during or after (re)consult, which takes much extra time. Moreover, there is no way to debug compiled programs in most systems. Therefore, many Prolog users are discouraged from using a compiler. However, debugging a big program by using an interpreter is exhausting because it takes a long time to reach a bug.
4. Problems related to memory size, which are mentioned in application requirements, are more critical in application development because the use of an interpreter for debugging requires more memory than normal execution.

Approaches

True compiler-based programming, which we adopted to make Prolog more useful in the area of application development, needs careful consideration of both the system architecture and the language design, which are mutually affected.

- *Architecture and implementation*

The basic features which a compiler-based system should have, and which are pursued in our work, are as follows:

- *Separate compilation.* This is essential for step-by-step development and sharing and reuse of objects.
- *Environment-independent compilation.* This means that object code does not depend on the environment in which it is generated. With this feature, objects can be shared and reused in different environments.
- *Generation of object files in the format regarded as the standard on the operating system.* This feature is indispensable in the use of libraries provided as objects and the use of objects implemented in Prolog in other systems; it will increase the chance of Prolog being used in practical applications.
- *Generation of executable modules in the format regarded as the standard on the operating system.* This enables an application to run by itself.
- *Optimization.* To allow for optimization without sacrificing portability, we adopted an abstract machine, called the Abstract Prolog Machine (APM), for an internal phase of compilation. APM is based on, but provides many extensions to, Warren's Abstract Machine (WAM) [1], which allows it to support Zephyr's new features for practical use. Optimization is done in two stages:
  - Prolog-to-APM-code translation. Well-known optimization techniques, such as TRO, clause indexing, and get/put elimination are applied.
  - APM-code-to-native-code translation. By control and data flow analysis on lower-level code, register allocation is effectively done on a real machine.
- *Efficient garbage collection.* Since the memory used for a failure computation is reclaimed automatically during backtracking, garbage collection in Prolog is not as essential as in Lisp. Even so, there are programs that are necessarily executed in forward-loop style and therefore require garbage collection. If they are actual applications, it should not look as if they have stopped during garbage collection. To avoid this, we have applied to Prolog the technique known as generation-scavenging GC [8], which is described in the section below on implementation in OS/2.
- *Fast compilation.* One of the disadvantages of using a compiler is that it takes much longer to remake code after modification than with an interpreter, because of the compilation itself and linking. As with linking, there is less chance of improving performance, because the standard linker must be used in order to generate standard modules. Consequently, we must try fast compilation to compensate for the disadvantage.
- *Detecting all errors and possible erroneous parts.* One of the advantages of using the compiler for development is that it always does program analysis and then detects and informs the programmer of errors and erroneous parts. For Prolog, the most common errors include simple syntax errors and references to undefined predicates. The compiler attempts to find all of these errors.
- *Debugging support.* Because we separate the debugging function from the language proper, we must support

```
package(PackageName [,PackageType] ).
    Declarations...
    Clauses...
endpackage(PackageName).
```

Figure 1

Style of a package.

```
import( [predicate,] [PackageName : ] Predicate).
export( [predicate,] Predicate).
```

Figure 2

Style of import and export declarations.

separate debuggers. We are working on this in several ways, using meta-level debugging known as algorithmic debugging [9] and source-code debugging (CodeView™, etc.).

We have kept in mind that there can be no restrictions on memory usage and that the system must be portable.

- *Language*

In designing the language, we adopted as its base the *de facto* standard, Edinburgh Prolog, refined it, and added many extensions such as the following:

- Introduction of *package* as a predicate name space and as a compilation unit.
- Refinement and enhancement of metafunctions.
- Introduction of *table* for persistent data.
- Introduction of new data types, such as character, string, big integer, float, ratio, complex, and infinite term.
- Introduction of new control mechanisms, such as freeze, on_backtrack, and catch-and-throw.
- Support of SQL and external language interfaces.
- Separation of interpreter/debugger commands from the language proper.

In the next section, we describe important new language features, focusing especially on the first three of the above because they are most closely related to compiler-based programming.

Language extensions

- *Packages*

Modularity is one of the most important features required in large-application development. This means that separately developed programs cannot affect the behavior of other programs accidentally or by design without accessing them via interfaces open to others. *Package* is introduced to realize this modularity in Prolog.

In a sense, Prolog is a modular programming language because of the unique features of variables. The scope of a variable is limited to the clause in which it occurs, and variables cannot be overwritten after they are assigned values by unification. A collection of clauses whose head name and arity are the same is called a *predicate*. A predicate is the smallest functional unit in Prolog and corresponds to a function or a subroutine in conventional languages. But predicates in Prolog tend to be used much more than functions in conventional languages to produce equivalent results, because a predicate is necessary in order to perform a simple operation such as a condition check, and another predicate is used to keep global values. Unfortunately, there is no way of controlling the scope of predicate names in standard Prolog, and it is impossible to remove the possibility of predicate-name conflict with certainty.

In Zephyr, a package provides a name space for predicates and tables. (From now on in this subsection, we discuss only predicates for simplicity.) Because predicates in a package are hidden from other packages, their names never conflict with those of others. Explicit declarations are used to control the visibility of predicates. For example, predicates which are to be used from other packages should be declared as "exported" in the package where they are defined, and declared as "imported" in the package where they are used.

As for *symbols*, we have not introduced any way of controlling visibility. Thus, symbols are always unifiable if, they have the same name, even when they occur in different packages. MPROLOG [10] and BIM PROLOG [11] have the capability to control visibility of symbols. As long as metafunctions are not used, a symbol simply expresses the name itself and is not used to bind something with it, as in Lisp [12]. Thus, in such a case, there are no serious problems related to the scope of symbols. Issues related to metafunctions are discussed in the next section.

In Zephyr, a complete program consists of one or more packages. Each package consists of predicates made up of clauses and declarations. **Figure 1** shows the style of a package. Each package name should be a symbol, and the package type should be either *open* or *closed*. If the type

is not given, it is assumed to be *closed*. The meaning of the package type is described, together with metafunctions. In a complete program, one and only one predicate whose name is main should be exported, and this predicate becomes an entry point.

There are three kinds of declaration: *import/export declaration*, *table declaration*, and *predicate declaration*. The last two are discussed later.

Import and export declarations are used to control the visibility of predicates. The style is shown in **Figure 2**. *Predicate* in the declaration is specified by the most general head pattern. For example, a predicate whose name is sort and whose arity is 2 can be specified as

```
sort(_,_)
sort(In,Out)
...
```

Notice that in an import declaration, a predicate can be declared with a package name that specifies the package which exports the predicate. This is important in the following senses:

- In Zephyr, more than one predicate with the same name and arity may be exported, and each is treated as independent. By specifying packages in import declarations, we can avoid ambiguity of import and accidental name conflict among exported predicates.
- It allows the system to look for specified packages, for example during linking, even if they are not explicitly given.

It is also possible to import more than one predicate whose name and arity are the same in a package. To avoid ambiguity in calling predicates, we can specify a package for each goal using the infix operator : as follows:

```
quickSort : sort([1,3,2],X)
bubbleSort : sort([1,3,2],Y).
```

This is also used to refer to built-in predicates redefined by the user. In Zephyr, built-in predicates are treated as if they were defined in a package called builtin and imported implicitly. It is possible to define a predicate whose name and arity are the same as those of some built-in predicate. In this case, the definition overrides the imported built-in predicate, but it is still possible to use the built-in predicate by specifying package builtin. For example, the following clause defines a new write(X):

```
write(X) :- builtin : write(X), nl.
```

A more complex and complete example is given in **Figure 3**.

In the example, predicate p(X) is defined in two packages and both are exported. To import and use them at the same time without ambiguity, package qualification

```
package(main).
export(main).
import(foo:p(X)).
import(bar:p(X)).
main:- foo:p(X), bar:p(X), write(X), fail.
endpackage(main).

package(foo).
export(p(X)).
p(X) :- r(X).
r(a).
r(b).
endpackage(foo).

package(bar).
export(p(X)).
p(X) :- r(X).
r(b).
r(c).
endpackage(bar).
```

Figure 3

Example of a program.

is required in import declarations and predicate calls. Predicate r(X) is also defined in two packages. These are not exported, however, and do not affect each other. The symbol b occurs in package foo and in package bar, and these occurrences are unifiable. Thus, as a result of execution, symbol b is written.

To summarize, packages and explicit specification of interfaces in Zephyr provide an effective method of ensuring the modularity of programs. In addition, they allow the system, that is, compiler and linker, to detect errors resulting from simple mistakes.

- *Metafunctions*

Prolog has unique features, which we call *metafunctions*, that treat data as programs and programs as data, as in Lisp. These features are realized by special built-in predicates such as **call**, **clause**, **assert**, and **retract** (which we call *metapredicates*). Standard Prolog has a single name space for a program, or set of predicates, and there is no ambiguity in mapping between symbols and predicates.

Approaches to defining metafunctions for Prolog with multiple name spaces depend on the characteristics of visibility control. If a symbol has limited scope, as in MPROLOG [10] and BIM PROLOG [11], there is also no

```
package(main).
export(main).
import(foo:call(X)).
import(bar:call(X)).
main:- X=father(F,tom), p(X), write(F), fail.
p(X) :- foo:call(X).
p(X) :- bar:call(X).
endpackage(main).

package(foo).
export(call(X)).
father(jack,tom).
endpackage(foo).

package(bar).
export(call(X)).
father(jim,tom).
endpackage(bar).
```

Figure 4

Example of call.

ambiguity in mapping between symbols and predicates. Therefore, metafunctions can be a natural extension of those for a single name space. On the other hand, in Zephyr, there are multiple name spaces only for predicates. Thus, we need a new rule for mapping between symbols and predicates.

The former approach provides simple and flexible metafunctions. Such flexibility, however, would lose the reliability and modularity of a program. Instead, we apply visibility control also to metafunctions: Each name space has its own metafunctions, which map a symbol to a predicate in that name space.

Our solution is to treat metapredicates as if they were defined in each package. This means that a metapredicate in a package works only for those predicates in that package. To use a metapredicate from another package, the metapredicate should be exported and then imported into the package where it is to be used. **Figure 4** shows an example of metapredicates. In the example, call(X) is defined implicitly in the packages foo and bar, and is exported from both. In package main, two call(X) are imported and executed with an argument term father(F,tom). The argument is interpreted in both the foo and bar packages. As a result, jack and jim are written.

A possible approach might have been to extend each metapredicate so that it would take an extra argument to specify the package in which a term is interpreted. We did not take this approach because the argument specifying a package is given dynamically, and thus does not allow a package to be compiled in a completely static way. That is, the compiler must always generate a predicate table for future possible use of metapredicates, but this table is hardly ever used. Moreover, all the predicates in any package become visible when **call** is used. This destroys the modularity which we are trying to introduce by means of *packages*. Furthermore, source programs cannot be hidden because the built-in predicate **clause** is able to get all the clauses as terms.

The way in which we have introduced metapredicates is rather static, but it ensures modularity, even if metapredicates are used, because only the package which exports a given metapredicate can be accessed by that metapredicate. However, this becomes a restriction on programming *generic* predicates. Suppose that we are writing a predicate to do sorting, which takes a comparison operator as its argument. In our approach, the comparison operator or **call** defined in the same package must be imported explicitly, so we cannot write such a sorting program as a library. Therefore, we introduce a new data type called *closure*, which combines a term and the name space of the package in which the closure is created, and makes it possible to interpret the term in the package later.

There are two kinds of closure: *goal closure* and *function closure*. The difference between them is that the latter takes real arguments when it is executed. Goal closure is created by a built-in predicate

goal(*GoalTerm, GoalClosure*)

and executed by

call_code(*GoalClosure*).

Function closure is created by

closure(*ArgVars, GoalTerm, FunctionClosure*)

and executed by

apply(*FunctionClosure, Parameters*).

The first argument of the built-in predicate **closure** is a list of variables that occur in *GoalTerm* and is unified with the second argument of **apply**, *Parameters*, before really starting execution of the closure.

The idea of closure is not new; it exists in Lisp [12]. But even though many recent Prolog systems [5, 6, 10, 11, 13, 14] have methods to define modules that are similar to those of our package, none of them have methods such as closure.

Figure 5 is an example of goal closure. Predicate countSolutions/2* counts how many times the given goal succeeds. In this case, goal closure for nqueen(8,S)

*In this paper, X/N denotes a predicate whose name is X and whose arity is N.

created in package main is passed and executed in the package countSolutions. However, the execution is done as in package main and is not affected by clauses defined in countSolutions.

Figure 6 is an example of function closure. genericQuickSort/3 is an alternative version of quick sort, but it takes a function closure as its third argument, which is used as a comparison operator. Comparison is done by **apply** with real arguments. Notice that execution of function closure may be done many times with different arguments. This is the primary difference from goal closure. In the example, the comparison operator is @>; therefore, comparison is done in descending order on any terms. The result is [f(1),a,1].

Both countSolutions and genericQuickSort can be written and compiled without the programmer knowing which package will use them, and these two packages can thus be used as libraries.

```
package(countSolutions).
export(countSolutions(X,Y)).
import(counter:new(X)).
import(counter:inc(X)).
import(counter:get(X,Y)).

countSolutions(Code,N) :-
        new(Counter),
        do(Code,Counter),
        get(Counter,N).

do(Code,Counter) :-
        call_code(Code),
        inc(Counter),
        fail.
do(_,_).
endpackage(count_solutions).

package(main).
export(main).
import(countSolutions(X,Y)).
main :-
        goal(nqueen(8,S),Code),
        countSolutions(Code,N),
        write(N), nl.
nqueen(X,Y) :-
        ...
endpackage(main).
```

Figure 5

Example of goal closure.

```
package(genericQuickSsort).
export(genericQuickSort(List,SortedList,CompareOp)).

genericQuickSort(L,S,Op) :- gqsort(L,S,[],Op).

gqsort([X|Xs],S,T2,Op) :-
        partition(Xs,X,Littles, Bigs,Op),
        gqsort(Littles,S,T1,Op),
        gqsort(Bigs,Bs,T2,Op),
        T1 = [X|Bs].
gqsort([],X,X,_).

partition([X|Xs],Y,[X|Ls],Bs,Op) :-
        apply(Op,[X,Y]), !, partition(Xs,Y,Ls,Bs,Op).
partition([X|Xs],Y,Ls,[X|Bs],Op) :-
        partition(Xs,Y,Ls,Bs,Op).
partition([],_,[],[],_).

endpackage(genericQuickSort).

package(useGenericQuickSort).
export(main).
import(genericQuickSort(In,Out,Op)).
main :-
        closure([X,Y], X@>Y, C),
        genericQuickSort([a,f(1),1],S,C),
        writen(S), nl.
endpackage(useGenericQuickSort).
```

Figure 6

Example of generic quick sort.

Closure is a very important mechanism on a compiler-based system. By using closure, we can write subroutines which take goals as their arguments without losing generality. In fact, in our implementation, closure is used internally to support some built-in predicates, such as **setof** and **bagof**.

Another important aspect of metafunctions is their ability to modify programs dynamically during execution. The modification is done by using the built-in predicates **assert** and **retract**. However, the unlimited use of **assert** and **retract** does not coincide with our general approach of making it possible to compile statically and discard source information as much as possible.

According to our study, the uses of **assert** and **retract** can be categorized as follows:

- Building a program interactively in an interpreter.
- Maintaining global data.
- Actually modifying a program during execution.

declare([predicate,] *Predicate*, dynamic).

Figure 7

Style of predicate declaration.

```
package(fibonacci).
export(fib(X,Y)).
declare(fib(X,Y),dynamic).
fib(0,1).
fib(1,1).
fib(X,Y) :-
    X > 1,
    X1 is X - 1, fib(X1,Y1),
    X2 is X - 2, fib(X2,Y2),
    Y is Y1+Y2,
    asserta(fib(X,Y)).
endpackage(fibonacci).
```

Figure 8

Example of self-modification program.

The first use is a function of the programming environment rather than being a feature of the language. Therefore, we ignore it here. The second use is more common: Because a Prolog variable is localized to a clause, Prolog programmers tend to use **assert** and **retract** for this purpose. In Zephyr, as a result of the introduction of a new data type *table*, this usage is eliminated (see the next section for details). We have found that the third case still remains, and decided to support it in a slightly restricted way.

The approach was that any predicates modified or created during execution should be declared as *dynamic* in the style shown in **Figure 7**, and that **assert** and **retract** work only for them. Like **call, assert** and **retract** do not affect dynamic predicates defined in other packages. Since all dynamic predicates should be declared, a compiler can detect errors such as undefined predicate calls in the same way as for a completely static package. Predicates not declared as dynamic can be compiled in the static way.

Figure 8 is an example of a program modification that calculates Fibonacci numbers. Once a Fibonacci number is calculated, it is stored as a fact by modifying the program, and the next time it is referred to in the recursive loop, it is used without being calculated again. Because fib/2 will be modified, it is declared as dynamic and the modification is done by **asserta**, which is a variety of **assert** and always inserts a fact on top of the predicate.

From the viewpoint of the implementation of Zephyr, most compiled programs can run without source terms or predicate tables. However, once **call** appears in a source package, a predicate table of the package is required for its execution. In addition, there is a possibility that the built-in predicate **clause** will be invoked during execution of the **call**, which requires all source terms. Because this is a very rare case, we have restricted the default behavior here so that **clause** via **call** does not work. To make it work, we must declare the package to be *open* by using the optional argument of the package statement. Of course, if **clause** appears in the source explicitly, a predicate table and source terms must be generated.

- *Tables*

A *table* resembles somewhat a variable or an array in conventional programming languages, and is used to retain terms without losing them during backtracking.

In most Prolog systems, this is done by using **assert** and **retract**, but it has many drawbacks:

- It disturbs the static compilation of programs.
- Its effect on program behavior is difficult to understand.
- The modularity of data depends on that of predicates.
- It requires maintenance of the rule base, which is a time-consuming task in Prolog, thus making it difficult to obtain good performance.

We know that using global data is not consistent with the clear semantics of Prolog, but in real applications, it is indispensable. We have therefore introduced *table* to support global data clearly and safely, as a replacement for the inappropriate use of **assert** and **retract**.

To introduce it, we took the same kind of approach as for (*dynamic*) predicates:

- All tables should be declared.
- The scope of a table is limited to the package in which it is generated.
- *Import* and *export* declarations are necessary to control visibility.

Figure 9 shows how tables are declared. *Table* in the declarations should be the most general term and should specify the name of the table and the dimension. A zero-dimensional table is specified by a symbol and is treated as a simple variable. For example

declare(table, *Table*).

import(table, [*PackageName* :] *Table*).

export(table, *Table*).

Figure 9

Style of table declarations.

```
declare(table,wordCounter(W,A)).
declare(table,flag).
```

declare a two-dimensional table wordCounter and a variable flag.

Accessing and changing a value in a table are performed by the built-in predicates

get_table(*Pattern, Value*);
put_table(*Pattern, Value*).

Pattern should be a ground term which matches one of the tables declared in the package. On the other hand, *Value* can take any Prolog term. The location where *Value* is kept is uniquely identified by *Pattern*. In other words, the whole of *Pattern* works as an index for a table. **put_table** places *Value* at the location identified by *Pattern*, and the previous value is discarded. **get_table** unifies *Value* with the term found at the location *Pattern*. If nothing is written at the location, the process ends in failure.

Figure 10 is a package to maintain counters, which are used as in Figure 5. The one-dimensional table counter is used to keep real values of counters that are identified by symbols generated by gensym. Since the table is not exported, it is not accessed directly from other packages.

In terms of implementation, a table is exactly the same as a hash table. The hash value of *Pattern* is used to identify the location quickly, and in the same amount of time for all cases. The pair of *Pattern* and *Value* is stored at the location. The size of the table is automatically changed according to the number of entries actually used.

Since we have also introduced static interfaces for tables, we need a new mechanism similar to *closure* for predicates. We call it a *table descriptor*. A table descriptor becomes necessary when we write a general package which works on any tables—for instance, sorting the elements of a given table. A table descriptor is created by the built-in predicate

table_descriptor(*Table, Descriptor*)

and can be passed to other packages. To access the table using the descriptor, use

```
package(counter).
export(new(Name)).
export(inc(Name)).
export(get(Name,Value)).
% declare put, dec, ...
import(gensym:gensym(X)).
declare(table,counter(Name)).
new(Name) :-
        gensym(Name),
        put_table(counter(Name),0).
inc(Name) :-
        get_table(counter(Name),N),
        N1 is N+1,
        put_table(counter(Name),N1).
get(Name,Value) :-
        get_table(counter(Name),Value).
% define put, dec, ...
endpackage(main).
```

Figure 10

Example of table.

Table 1 New data types in Zephyr Prolog.

Type	*Example*
Big integer	12345678901234567890
Character	#'a
Floating number	1.2e-3
Rational number	#ratio(1,3) (= 1/3)
Complex number	#complex(1,2) (= 1 + 2i)
String	$Zephyr Prolog$
Infinite term	f(#inf(1)) (= f(f(f(. . .))))

get_table(*Descriptor, IndexList, Value*);
put_table(*Descriptor, IndexList, Value*).

In this case, an access pattern is dynamically created from *Descriptor* and *IndexList*.

- *Other extensions*

Data types

Many new data types that are useful for practical applications but are not found in Edinburgh Prolog are introduced in Zephyr Prolog. The introduction is influenced by Common Lisp [12] and recent Prolog systems [5, 6, 14]. New data types introduced in Zephyr Prolog are listed in **Table 1**.

One of the most important is the character data type. Because Edinburgh Prolog does not have a character data type, programmers in Edinburgh Prolog must write character code directly and debug it with character code. The resulting programs are hard to read and not portable among different character code sets. Introduction of the character data type helps solve these problems. Problems related to character code are also found in *string syntax sugar*, which is an alternative notation of a list of character codes. Ideally, the string syntax sugar of Edinburgh Prolog should be converted into a list of characters, for instance, from ABC to [#'A, #'B, #'C]. However, to maintain compatibility with Edinburgh Prolog, this translation is not used.

Controls

The following new control mechanisms are introduced. They are sometimes useful for improving performance and simplifying programs. However, it is hard to define them in pure Prolog, or rather, it is hard to define them so that they work efficiently in pure Prolog [15].

- **freeze**(*Var, Goal*) suspends execution of *Goal* until *Var* is bound to a nonvariable term. This is the same as the mechanism introduced in Prolog-II [16].
- **on_backtrack**(*Goal*) suspends execution of *Goal* until backtracking to an earlier point occurs. This effect cannot be eliminated by **cut**.
- **catch**(*Goal*) executes *Goal*. It fails immediately when **throw** is executed during its execution. A combination of **catch** and **throw** allows a global exit from the execution of a goal.

C-interface

C-interface, introduced in Zephyr Prolog, is a natural extension of the import and export of predicates. To use a C function in a Zephyr package, declare it in the following way:

import(c, *PredicatePattern*).

A goal that matches the second argument is treated as a C function call. If the C function returns TRUE, the goal succeeds; otherwise it fails.

To use a Zephyr predicate from C, declare it in the following way:

export(c, *PredicatePattern*).

The predicate can be considered as a function for returning TRUE or FALSE according to the SUCCESS or FAILURE of predicate execution. Note that even though the predicate succeeds with choice points, all of them are removed because the predicate is treated as a C function.

Since both C and Prolog allow recursive calls, we have placed no restriction on nested calls between C and Prolog.

Parameter passing between C and Prolog is done by using APM registers, not the C stack, and automatic data conversion is not supported. This means that C programmers must know APM architecture. The reasons we adopted this strategy are as follows:

- If the C stack contains Prolog data such as structures and lists, garbage collection cannot work correctly and may make dangling references, because the C stack is outside the control of Prolog memory management.
- There is no way to map any structure data of Prolog to C data without keeping tag information on C data. If tag information is used, there is no meaningful difference from using the original Prolog data structure directly.
- Automated data conversion, especially for structure data, causes extensive overhead. In most cases, this makes the use of C meaningless.

Implementation on OS/2

In this section, we describe the implementation of Zephyr on OS/2, focusing particularly on the realization of environment-independent separate compilation that generates standard objects.

Originally, we initiated our project on VM/370 and on top of VM/Prolog [13]. At that time, all of the Zephyr compiler was written in Zephyr itself. By bootstrapping the compiler, we examined our approaches. There was an increasing demand for Prolog on personal computers and workstations. The biggest problem with DOS environments is that they do not allow more than 640 KB of memory without additional memory-management tools. But on OS/2 up to 16 MB of memory is available, which allows many Prolog applications to run on personal computers, thereby making Prolog familiar to a much wider range of users. This is why we implemented Zephyr on OS/2.

The OS/2 version includes a compiler, a symbolic linker, libraries, preprocessors, and a debugging interpreter, and runs on the OS/2 1.X family. Zephyr thus runs in the 286 mode even on a 386 machine. This degrades performance, because APM is a 32-bit machine supporting large memory, and the implementation of Zephyr on the OS/2 1.X family is something like an emulation of a 32-bit machine on a 16-bit machine.

For the implementation, we used the following restrictions:

- The maximum code size of a package is 64 KB.
- The maximum size of one item of structure data is 64 KB.

These help improve performance on a 286 machine by minimizing the occurrence of segment switching. We provide separate compilation, and Prolog allows a nested data structure, so those restrictions are not serious.

- *Compile and link*

To create an executable module in Zephyr, the following process is needed:

- Preprocess if necessary (to use DCG or Prolog-style SQL).
- Compile each package.
- Link all objects with Zephyr and C libraries.

Figure 11 shows this process, which is very similar to that in conventional languages such as C and FORTRAN, but is not common among existing Prolog systems.

The compiler compiles a package and generates an object file and a special file called a *symbolic information file*. Internally, as shown in **Figure 12**, this compilation consists of two phases: generating an assembly code file and generating an object file. The latter is done by invoking IBM Macro Assembler/2™ (MASM/2).

A symbolic information file contains the following information:

- Package name.
- Symbols that are used in the package: Each symbol is expressed by a name and an internal identifier.

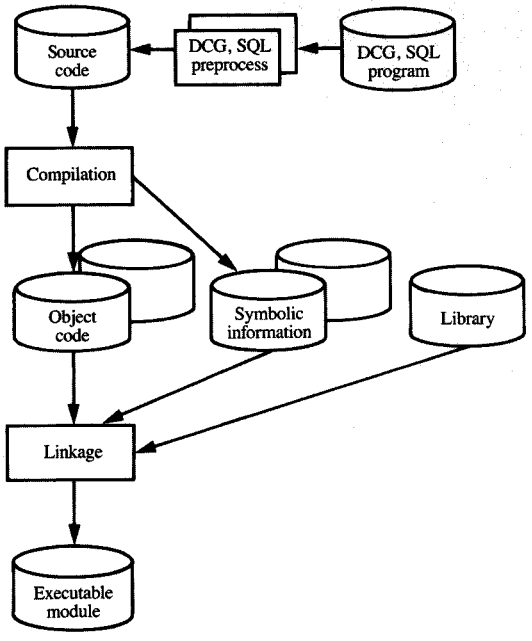

Figure 11

Compilation and linkage flow.

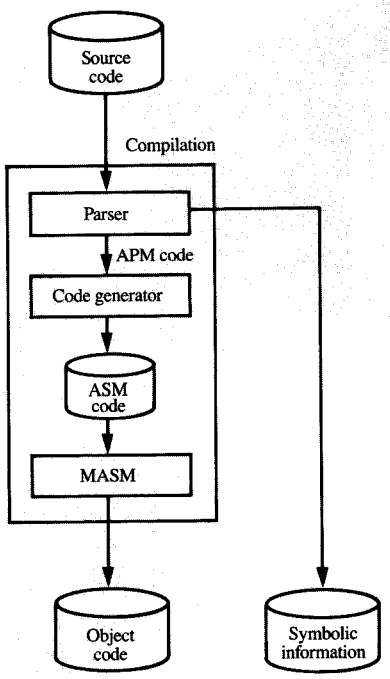

Figure 12

Compilation internal flow.

- Name, arity, and internal identifier for every predicate that is exported or imported, and additional package names for imports if they exist.
- Predicate table if metapredicates are used: This consists of predicate entries for all predicates, each with a name, arity, and internal identifier.
- Clauses represented as terms if the package is open.

This information is necessary to support environment-independent separate compilation and must be handled separately from standard objects. Symbols in Prolog are needed at execution time, and, as mentioned in the section on packages, symbols in Zephyr should be uniquely identified among separately compiled packages. In addition, exported predicates are identified by two-dimensional name spaces indexed by package names and predicate names. Unfortunately, the name space management and naming convention (that is, the characters that are allowed in a name and the maximum length of the name) are completely different in Prolog and conventional systems, requiring a special linker. However, because we still wish to retain the capability to link Prolog objects with objects in other systems, we separate a

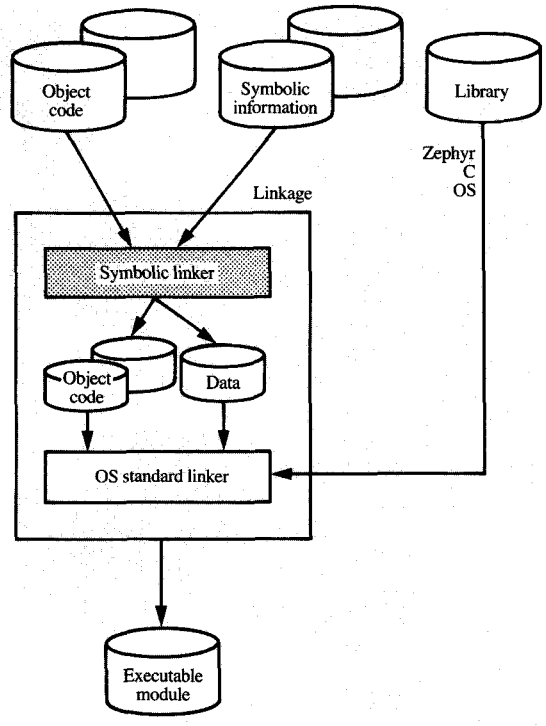

Figure 13

Linkage internal flow.

Zephyr object into a standard object file and a symbolic information file.

A symbolic linker consists of two components, a front-end linker and IBM Linker/2 (LINK/2), as shown in **Figure 13**. It accepts several Zephyr objects with symbolic information files, objects of other languages such as C, and libraries. First, the front-end linker collects those Zephyr objects which are provided explicitly by the user and any Zephyr objects which are contained in Zephyr built-in libraries and used in user packages. Then, it

- Serializes all the symbols in every complete program and generates a symbol table. Internal identifiers are used to modify object files. For example, during compilation, a symbol may be referred to by the label S1, but the address is unresolved at the time. At link time there may be many labels S1 for different symbols. As a result of serialization, the same symbols become identified by the same labels, and thus the label S1 in the package may be changed to, for example, S27. This change should be reflected in the object file. That is, the label S1 in the object should be changed to the label S27.
- Resolves import and export in every complete program. Internal identifiers are used to modify object files. For example, assume that father/2, whose internal identifier is P2, is exported from package foo and that the predicate is imported in another package, bar, with internal identifier P5. At first, all exported predicates are serialized like symbols. Assume that father/2 in foo has the identifier P15 as a result. Next, by scanning exports, it is found that father/2 is exported with identifier P15. Therefore, the identifier P5 in bar is changed to P15. This serialization and change of identifiers is reflected in object files.
- Collects predicate tables and source terms if they exist.
- Finally, generates an assembly code file which contains all the data generated in the above processes and invokes MASM/2.

After the front-end link process is finished, LINK/2 is invoked with all modified Zephyr objects, the data object, objects of other languages, Zephyr run-time routines, and C libraries (these are needed because most Zephyr run-time routines are written in C for portability). The generated module will run by itself.

- *Code generation*

The code-generation part of compilation consists primarily of two parts: Zephyr-to-APM-code translation, and APM-code-to-native-code translation. APM, adopted as an interim step, provides an efficient and portable way to map Zephyr to conventional CPU architectures, and plays a central role. Two translation phases are described below, following a discussion of new and important features of APM.

APM

Once the idea of using an abstract machine for Prolog compilation was exploited by Warren [1], it became a popular way to implement a Prolog compiler, and his abstract machine is now famous as the Warren Abstract Machine (WAM). But the original WAM focuses only on the pure part of Prolog and gives little consideration to full support of the Prolog language and to practical aspects. To adopt the idea of an abstract machine for Zephyr, we introduced many refinements and enhancements into WAM.

Memory organization A heap is added to the environment stack, choice point stack, global stack (called *heap* in WAM), trail stack, and work stacks. It is used to maintain global data such as symbols, tables, and predicate tables. The trail stack is used not only to keep variables to be undone during backtracking but also to keep goals to be executed during backtracking.

Data types The size of a data cell is 32 bits, and to distinguish symbols from constants, its tag field occupies three bits (two bits in WAM). The basic types expressed by the 3-bit tag are variable, list, structure, constant, symbol, frozen variable, table, and code. A constant data item may be an integer, character, or system constant, and has extra bits to distinguish it from others. The code is a pair, comprising an address in APM code and its arguments, and is used to express closure. Floating numbers, big integers, strings, complex numbers, and ratio numbers are expressed as structure data with special function operators.

Symbols and related instructions Because the original WAM was designed without regard to separate compilation, we need some refinements in this respect to design APM starting from WAM. In WAM, there is no distinction between integers and symbols; both are treated as constant data. It is supposed that all symbols already have identifiers at compilation time and that the identifiers will never be changed in future execution or in another compilation. In Zephyr, because of separate compilation, the identifiers should be given at link time, and during compilation symbols are treated as being of pointer data type, not constant type. All of the pointer data for one symbol refer to the same address, which contains the name of the symbol. Pointer data are mapped to internal identifiers during generation of an object file and a symbolic information file.

To distinguish these two types, we introduced new APM instructions such as get_pointer, put_pointer, unify_pointer, and select_pointer. The first three are simply counterparts of get_constant, etc., and select_pointer is an indexing instruction on pointers. For efficient indexing, a hash table is used. In WAM, symbol identifiers are uniquely given at compilation time and can be used directly as hash values. In APM, real identifiers are not given at compilation time, and the pointers cannot be used as hash values. To make it possible to build a hash table at compilation time, we use a unique hash function for every compilation, and a hash value is kept at the location pointed to by the pointer data. During the execution of an indexing instruction, the hash value is indirectly looked up from the pointer, and the location of its entry is calculated from the hash value and the hash table size. This process differs from that of constant data, and therefore we have introduced new indexing instructions such as select_pointer.

Table The tables are stored in the heap. Each table entry consists of the hash value of the index term, the index term, and the value term. The hash value of structure data is calculated by adding the hash values of each argument term, function operator, and arity. By keeping the hash values of index terms, it is possible to detect a mismatch quickly when two terms are mapped to the same entry but have different hash values, and to resize a hash table efficiently when the table overflows.

Index terms and value terms are also stored in the heap as continuous memory blocks, with the use of offsets for addressing instead of pointers. This allows efficient garbage collection in the heap area.

Garbage collection We use incremental garbage collection (GC), known as generation-scavenging GC in Smalltalk [17]. Conventional GC starts when the whole global stack is exhausted, and scans all stacks. Incremental GC starts when a small area, which is called a GC window, is exhausted, and only scans the global stack within the GC window. We placed the GC window at the end of the global stack to reduce the overhead of copying data during GC, and adopted a 64KB GC window on a 286 machine to reduce the occurrence of segment switching during GC. According to our study [8], newly created cells tend to contain a large proportion of garbage, and because it focuses on them, GC takes less time. In addition, because incremental GC is done within a small memory area, it causes fewer page faults and cache misses, improving total system performance. On the other hand, it has some drawbacks: It does not collect garbage outside the GC window, and it may be invoked in some cases for which conventional GC would not occur.

Figure 14 illustrates how incremental GC works on the global stack. (1) shows the state in which the global stack is growing. If backtracking occurs beyond the GC window, the state will be as shown in (2). Some data cells of the old part are discarded, and the top of the global stack is set to the bottom of the GC window. (3) shows the state in which a GC window overflows and the garbage collector is invoked. The garbage collector scans the contents of the GC window and stack frames created after the previous GC. Living cells are then moved to the old part of the global stack. Finally, the top of the global stack is set to the bottom of the GC window, as shown in (4).

Additional instructions APM has additional instructions to support new features of Zephyr and to improve compiler optimization. Examples are

- Support for the **cut** operation.
- Explicit variable initialization for safe GC.
- Explicit stack overflow checking.
- Creation of closure.
- Escape to underlying system.
- Tag checking.
- Support for new controls such as **catch** and **throw**, **freeze**, and **on_backtrack**.
- Instructions for read-mode-specific and write-mode-specific operations.

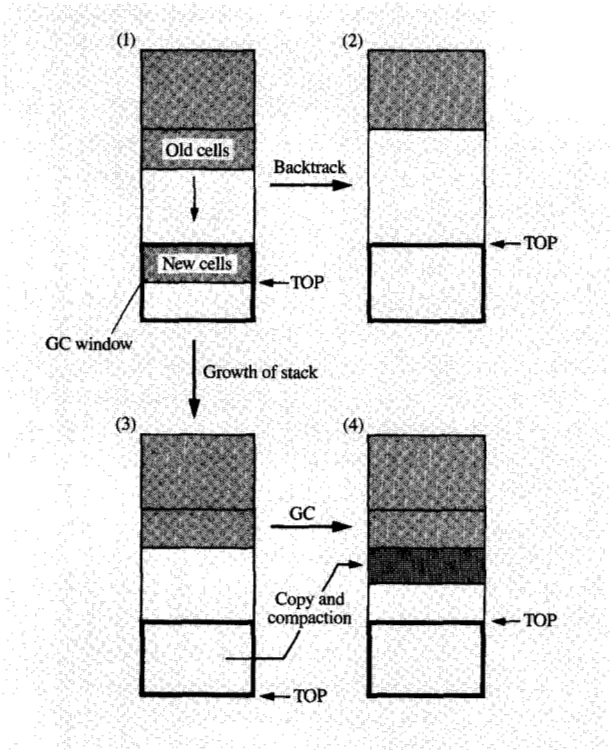

Figure 14

Incremental garbage collection.

Zephyr to APM

Compilation from Zephyr to APM is similar to WAM-based compilation; that is, it performs trail recursion optimization (TRO), GET/PUT elimination, indexing, and management of registers/stacks.

In addition, it performs the following new operations:

- GC checking. Since APM instructions do not check global stack overflow, the compiler evaluates the possible maximum size of growth of the global stack in an execution block and generates code to check explicitly whether GC may be needed. This code is located just before the first access to the global stack in the block.
- Delay stack allocation. Stack allocation is a heavy task in Prolog, so the code is located just before the first access to the environment stack in a clause.
- Permanent variable initialization. For safe GC, all permanent variables (variables allocated to environment stacks) must be initialized. The compiler generates initialization code for permanent variables not referred to in the first execution block of a clause, and places it just before the first call instruction.
- Freeze checking. Code that checks whether variables bound to some delayed goals are bound to nonvariables is generated between the clause head code and the clause body code.
- Built-in compilation. Built-in predicates are compiled in one of the following ways:
  - Run-time calls. Most built-ins, such as read(_) and write(_), are compiled in this way.
  - **call** to predicates predefined in Zephyr Prolog itself, that is, libraries [bagof(_,_,_), etc.].
  - Expansion into several clauses (;, ->, etc.).
  - In-line APM code [var(X), etc.].
- Error detection. The compiler detects all errors related to syntax and undefined predicate calls, and does not stop compilation at the first error.
- Generation of a symbolic information file.
- Generation of basic block information. Basic block information is additional information for code generation, containing
  - Conditions on variables such as value, type, dereference, and mode.
  - Reference count.
  - Visibility; i.e., local (accessed only in the package), exported (accessed from other packages), or imported (having access to other packages).

This information helps the code generator to create efficient and compact native code. Our previous work [18, 19] and recent research [20, 21] show that the first one is particularly helpful. However, the current version of the Zephyr compiler does not fully utilize this.

APM to native

Since there is still a semantic gap between APM and conventional CPUs but, on the other hand, a need for efficient and portable code generation, we have introduced Zephyr Intermediate Language (ZIL), which is an assembly language for generalized CPUs with tag-handling capability. Thus, the code generation consists of two phases: APM-to-ZIL and ZIL-to-native code.

In translation from APM to ZIL, the code generator uses basic block information to eliminate redundant ZIL code generation. For example, native translation generates tag-testing code and operation code for each possible tag. If the possible types are known, the tag-testing code may be simplified, and operation code for some tags can be eliminated.

In translation from ZIL to native code, the generator does

- Control flow optimization, such as basic block straightening and elimination of multiple jumps.
- Register allocation using register coloring [22].

However, for a 286 machine, register allocation is not effective, since a 286 has few available general-purpose

registers. Thus, register allocation is inhibited. In addition, to reduce the amount of segment switching, which takes a lot of time on a 286, the generator does

- Separation of the internal entry point and external entry point of an exported predicate. This makes it possible to generate **near jump** for calling a predicate defined in the same package, even though it is exported.
- Generation of frequently used run-time routines as package-local subroutines.
- No spreading of structure data across a segment boundary.

- *Libraries*

On OS/2, the Zephyr library includes

- A dynamic link library (DLL) version of Zephyr run-time code.
- A static version of Zephyr run-time code.
- Built-in/utility objects written in Zephyr.
- DLL for SQL support.
- A stream-I/O-based window server.

One of the most important drawbacks of compiler-based programming is that it takes a long time to remake a module and execute it again after program modification. Separate compilation minimizes the size of the part that must be recompiled. Thus, the compilation time is not a serious problem. On the other hand, the linking time is significant because libraries must be searched. The use of DLLs can provide a partial solution on OS/2. In a DLL, most address references are resolved; when linking with user programs, it is only necessary to resolve function entries, and the linkage can be achieved in a fairly short time. During execution using DLLs, however, it takes more time because it is necessary to load and maintain the DLLs. We therefore provide two types of run-time library: a DLL for the development cycle and a static library for final applications.

- *Preprocessors*

Currently, two preprocessors are supported: a DCG preprocessor and an SQL preprocessor. The DCG preprocessor is used to convert a package which includes DCG rules into a normal Zephyr package, which is compiled.

The SQL preprocessor provides an elegant and efficient way of issuing queries to an SQL database managed by the OS/2 database manager. The program is written like a normal Zephyr package, except that it includes *import database declarations* instead of import predicate declarations. The execution is optimized to minimize the number of SQL queries by a *lazy execution mechanism*. This will be discussed further in a future paper.

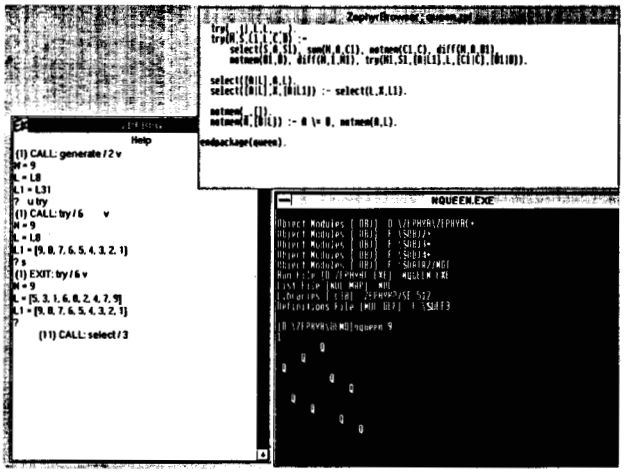

Figure 15

Sample screen showing debugging.

Both DCG and SQL preprocessors are written in Zephyr itself, and run like conventional applications. For example, to use the DCG processor, it is acceptable to type

```
[C:\ZEPHYR]dcg logic.dcg logic.zpl
```

on the command line. The preprocessed file logic.zpl will be generated from logic.dcg.

- *Debugging*

Zephyr Prolog provides two kinds of debugging facility: a debugging interpreter and debugging in compiled code.

The debugging interpreter is an ongoing prototype that integrates interactive execution, box model tracing [23], algorithmic debugging [9], and type inference based on abstract interpretation. The essential part of the system is described in another paper [24].

Debugging in compiled code is done by compiling packages with the DEBUG option. During execution of packages compiled with the DEBUG option, functions can be used that do the following:

- Monitor the **call** and **exit** gates of the box model.
- Execute one step or N steps.
- Skip to specified predicate.
- Skip the current goal.
- Display variable bindings of the clause under execution with the original variable names.
- Browse the source program being executed.
- Locate the cursor of the browser automatically on the clause being executed.

Figure 15 shows one screen using the debugging functions. As shown in the figure, each browsing source program and

Table 2 Results of rough measurement on an IBM PS/55 (20-MHz 386) by using the naive reverse benchmark.

	Static	*DLL*
Compile time	17 s	17 s
Link time	66 s	9 s
Module size	340.5 KB	18.6 KB + DLL
Execution speed	14.5 KLIPS*	13.4 KLIPS*

*LIPS = logical inference per second.

the dialog for debugging are done in windows separate from the application window.

These debugging facilities are realized by

- Considering all variables as permanent variables. This is necessary in order to be able to show all variable bindings at any time.
- Inserting a call of a debugging routine before the first goal of each clause. The debugging routine creates a frame that contains the depth of calls and mapping of variable cells to their names.
- Generating APM **nop** instructions before and after APM **call** instructions. These **nop** instructions are considered as calls to a routine that provides debug functions. One parameter of **nop**, nop number, is used to obtain goal information.
- Generating a debugging information file, which contains
  - Mapping of nop numbers to goal information, which contains the name, arity, and source position of the goal.
  - Mapping of clause number to clause information, which contains the source file name, predicate name, predicate arity, source position, number of variables, number of goals, and the size of the frame for debugging.

Preliminary evaluation

It is difficult to judge whether a language system is well designed. The only effective way of doing so is to use the system extensively in numerous large applications, which could take years. But while implementing Zephyr, we became convinced that our approach in Zephyr (that is, compiler-based programming) is effective in application development because Zephyr was used to implement many of its own components. For example, most built-in predicates are written in C, but currently there are 20 packages written in Zephyr as built-ins or utilities, including metapredicates (such as **setof**, **bagof**), list-processing predicates (such as **append**, **reverse**, **member**), tracing, SQL support, sorting, record handling, DCG support, and stream-based window support. They are provided as compiled objects, and there is no need to recompile them for use in applications, thanks to the environment-independent separate compilation. Moreover, the user can easily extend the library. In fact, the number of utilities is still increasing through the addition of packages which have generality. The SQL preprocessor, the DCG processor, and the debugging interpreter are written in Zephyr. The debugging interpreter is one of the biggest applications, consisting of 14 packages with more than 6000 lines.

Although performance is not the focus of this paper, we give the results of a rough measurement in **Table 2**. The measurements were obtained by the well-known naive reverse benchmark program (one package, 63 lines in our case) on OS/2 1.1, running on the PS/55 Model 5570, which is a Japanese version of the PS/2® with an 80386®, 20-MHz CPU. Since we had difficulty in measuring the exact CPU time on OS/2, it was measured by the response time base when there was no other user session.

The execution speed seems to be a little slow in comparison with that of Arity Prolog [5] running on DOS, but we know that this comes from supporting a large memory and many new data types, and from run-time routines written in C.

We can test a small program within a minute using a DLL, but in a large application the link time will be a problem. In fact, it takes more than one minute simply to link all the objects of the debugging interpreter.

Conclusion and future work

In this paper we have shown that compiler-based programming, which is normal in conventional programming languages, can solve many of the difficulties found in large-application development in Prolog. To do this, we have refined and extended the *de facto* standard, Edinburgh Prolog, and have implemented a compiler system on OS/2. The system was carefully designed to be suitable for application development. In fact, the debugging interpreter, the SQL interface program, some preprocessors, and some of the built-in predicates are written in Zephyr itself. Several tasks remain as future work, such as full implementation of source-code debugging, an external language interface, and optimization based on type inference. Porting our system to other operating environments will be valuable and will provide necessary evidence of portability. Finally, we would like our work to have a positive impact on the standardization of Prolog.

Acknowledgments

We would like to thank Toshiaki Kurokawa for his useful advice on starting our project, Naoyuki Tamura for working with us on the design, Mitsuru Ohba for managing our work, and Yoshio Tozawa for encouraging us to write this paper.

References

1. D. H. D. Warren, "An Abstract Prolog Instruction Set," *Technical Note 309*, Artificial Intelligence Center, SRI International, Stanford, CA, 1983.
2. D. H. D. Warren, "An Improved Prolog Implementation Which Optimizes Tail Recursion," *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary, 1980, pp. 1–11.
3. D. L. Brown, *DECsystem-10 PROLOG USER'S MANUAL*, University of Edinburgh, Scotland, 1981.
4. W. F. Clocksin and C. S. Mellish, *Programming in Prolog* (Third, Revised and Extended Edition), Springer-Verlag, New York, 1987.
5. *Arity/Prolog Compiler and Interpreter, Language Manual*, Arity Corp., Concord, MA, 1987.
6. *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Palo Alto, CA, 1986.
7. *TURBO PROLOG 2.0 Reference Guide, User's Guide*, Borland International, Scotts Valley, CA, 1988.
8. H. Touati and T. Hama, "A Light-Weight Prolog Garbage Collector," *Proceedings of the International Conference on Fifth-Generation Computer Systems*, Institute of New Generation Computer Technology, Tokyo, 1988, pp. 922–930.
9. E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, MA, 1983.
10. *MPROLOG LANGUAGE REFERENCE Release 2.1*, Logicware International, Mississauga, Ont., Canada, 1985.
11. *BIM_PROLOG MANUAL*, B. I. Moyle Associates, Inc., Minneapolis, MN, 1986.
12. G. L. Steele, Jr., *COMMON LISP Reference Manual*, Digital Press, Cambridge, MA, 1984.
13. *VM/Programming in Logic*, Order No. SH20-6541, 1985; available through IBM branch offices.
14. *The SB-Prolog System, Version 2.2. A User Manual*, University of Arizona, Tucson, 1987.
15. M. Carlsson, "Freeze, Indexing, and Other Implementation Issues in the WAM," *Proceedings of the 4th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 40–58.
16. A. Colmerauer, *Prolog-II Reference Manual and Theoretical Model*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, Marseilles, France, 1982.
17. D. Ungar, "The Design and Evaluation of a High Performance Smalltalk System," *ACM Distinguished Dissertations*, MIT Press, Cambridge, MA, 1987.
18. T. Kurokawa, N. Tamura, Y. Asakawa, and H. Komatsu, "A Very Fast Prolog Compiler on Multiple Architecture," *Proceedings of the 1986 ACM/IEEE Fall Joint Computer Conference*, Dallas, pp. 656–662.
19. N. Tamura, "Knowledge-Based Optimization in Prolog Compiler," *Proceedings of the 1986 ACM/IEEE Fall Joint Computer Conference*, Dallas, pp. 237–240.
20. T. Hickey and S. Mudambi, "Global Compilation of Prolog," *J. Logic Program.* **7**, 193–230 (1989).
21. A. Taylor, "Removal of Dereferencing and Trailing in Prolog Compilation," *Proceedings of the Sixth International Conference on Logic Programming*, Cambridge, MA, 1989.
22. G. J. Chaitin, M. A. Auslander, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Lang.* **6**, 47–57 (1981).
23. L. Byrd, "Understanding the Control Flow of Prolog Programs," *Proceedings of the Logic Programming Workshop*, Debrecen, Hungary, 1980.
24. T. Hama, "Prolog Type System and Its Application to Algorithmic Debugging," *Proceedings of Software Symposium '90*, Japan Society for Software Science and Technology (Tokyo), Kyoto, Japan, 1990, pp. 126–132.

Received November 7, 1990; accepted for publication February 26, 1992

Yasuo Asakawa *IBM Japan, Ltd., 1623–14 Shimotsuruma, Yamato-shi, Kanagawa-ken 242, Japan (JL05032 at YMTVM1, asakawa@trlvm1.vnet.ibm.com).* Mr. Asakawa is a Staff Software Engineer at the IBM Yamato Laboratory. He received his B.E. and M.E. degrees in computer science from the Tokyo Institute of Technology in 1982 and 1984, respectively. After graduation, he joined the IBM Tokyo Research Laboratory in 1984, and worked on two Prolog compiler projects. Mr. Asakawa's current interests are in intelligent personal information systems based on artificial intelligence and multimedia technology. He is currently working in Multimedia Development at the Yamato Laboratory.

Hideaki Komatsu *IBM Research Division, Tokyo Research Laboratory, 5–11, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan (KOMATSU at TRLVM, komatsu@trl.vnet.ibm.com).* Mr. Komatsu received his B.A. and M.A. degrees in electrical engineering from Waseda University in 1983 and 1985, respectively. He has been a Research Staff Member at the IBM Tokyo Research Laboratory since 1985. His research interests include fine-grain parallel architecture, compiler optimization techniques for instruction-level parallelism (code scheduling and register allocation), and compiler optimization techniques for coarse-grain parallelism (massively parallel). Mr. Komatsu is currently a member of the advanced compiler group at the Tokyo Research Laboratory.

Hiroaki Etoh *IBM Research Division, Tokyo Research Laboratory, 5–11, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan (ETOH at TRLVM, etoh@trl.vnet.ibm.com).* Mr. Etoh is a Research Staff Member at the Tokyo Research Laboratory. He received his B.E. and M.E. degrees from the Tokyo Institute of Technology in 1983 and 1985, respectively. After graduation, he joined the IBM Tokyo Research Laboratory in 1985, working on the Zephyr Prolog System. Mr. Etoh is currently working on GUI building systems. His main interests are in simulation and optimization in scheduling systems.

Toshiyuki Hama *IBM Research Division, Tokyo Research Laboratory, 5–11, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan (HAMA at TRLVM).* Mr. Hama received the B.S. degree in electrical engineering in 1984 and the M.S. degree in electronic engineering in 1986, both from Tokyo University. He joined IBM Japan in 1986 as an Associate Research Staff Member.

He worked on the Zephyr Prolog project for four years, with responsibility for designing and implementing an abstract Prolog machine. Mr. Hama's interests include artificial intelligence, logic programming, and nonmonotonic reasoning.

Keiichi Maruyama *IBM Research Division, Tokyo Research Laboratory, 5–11, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan (MARUYAMK at TRLVM).* Mr. Maruyama is a Research Staff Member at the Tokyo Research Laboratory. He received his B.E. and M.E. degrees from the University of Tokyo in 1985 and 1987, respectively. After graduation, he joined the IBM Tokyo Research Laboratory in 1987, working on the Zephyr Prolog System. Mr. Maruyama is currently working on scheduling systems. His main interests are in knowledge-based systems and programming languages.