Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

1-30-1996

Compiling Evaluable Functions in the Gödel Programming Language

David Shapiro
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Programming Languages and Compilers Commons
Let us know how access to this document benefits you.

Recommended Citation
Shapiro, David, "Compiling Evaluable Functions in the Gödel Programming Language" (1996). *Dissertations and Theses.* Paper 5101.
https://doi.org/10.15760/etd.6977

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of David Shapiro for the Master of Science in Computer Science were presented January 30, 1996, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Sergio Antoy, Chair

James Hein

Michael Driscoll
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

John McHugh, Chair
Department of Computer Science

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by ______________ on 26 Feb 1996

ABSTRACT

An abstract of the thesis of David Shapiro for the Master of Science in Computer Science presented January 30, 1996.

Title: Compiling Evaluable Functions in the Gödel Programming Language

We present an extension of the Gödel logic programming language code generator which compiles user-defined functions. These functions may be used as arguments in predicate or goal clauses. They are defined in extended Gödel as rewrite rules. A translation scheme is introduced to convert function definitions into predicate clauses for compilation. This translation scheme and the compilation of functional arguments both employ *leftmost-innermost narrowing*. As function declarations are indistinguishable from constructor declarations, a function detection method is implemented.

The ultimate goal of this research is the implementation of extended Gödel using *needed narrowing*. The work presented here is an intermediate step in creating a functional-logic language which expands the expressiveness of logic programming and streamlines its execution.

COMPILING EVALUABLE FUNCTIONS IN THE GÖDEL PROGRAMMING LANGUAGE

by

DAVID SHAPIRO

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1996

ACKNOWLEDGMENTS

Sergio Antoy, for overall guidance and answering my questions, and answering my questions, and answering my questions, over and over and over again.

Janet Vorvick, for a parser well done, a compiler well presaged, and both well explained.

Table of Contents

1 Introduction 1
2 Background 3
  2.1 Logic Programming 3
  2.2 Functional Programming 9
  2.3 Narrowing 11
3 Gödel 18
  3.1 Introduction to Gödel 19
  3.2 Extended Gödel 20
4 Compiler Design 23
  4.1 Evaluation of functional arguments 24
  4.2 Rule translation 25
  4.3 Function detection 27
5 Implementation Overview 28
  5.1 Parsed Program Structure 30
  5.2 Build Process 35
  5.3 Rule translation by tau 37
6 Implementation Details 39
  6.1 Compiler invocation 39
  6.2 Predicate compilation 40
  6.3 Rule compilation 42
  6.4 Function detection 46
  6.5 Building constraints 47
  6.6 Making 50
  6.7 Loading 51
7 Conclusion 52
  7.1 Feasibility of functional-logic programming language 52
  7.2 Narrowing 53
  7.3 Compiler Design 54
  7.4 Versions 55
8 Other functional-logic programming languages 61
  8.1 ALF 61
  8.2 K-LEAF 64
Bibliography 67
Appendices 69

1 Introduction

Logic programming offers the programmer the opportunity to create programs using the powerful, familiar tool of mathematical logic. A programmer can translate a set of logic axioms directly into a computer source program. The programmer need not be concerned with control details of program execution, but only with the correct statement of that program.

Unfortunately, logic programming languages do not include the ability to compute using functions. Functions are a natural way to express many relations. Their unavailability in logic programming languages leads, at the least, to less clarity and expressiveness in logic programs.

A combined functional-logic programming language could offer such increased clarity and expressiveness. What is more, depending on the implementation of function evaluation, other gains may be achieved. Programs may run more efficiently. Programs which would otherwise be non-terminating can be successfully executed to termination. Thus, a functional-logic programming language is a very desirable goal.

This thesis represents part of an attempt [ASV] to integrate functional and logic programming using *needed narrowing*. Specifically, I have extended the Gödel logic programming language code generator to accept and evaluate terms which contain functions. Of course, it is impossible to generate compiled code without any input.

Janet Vorvick [JV] has modified the front-end of the Gödel compiler to parse functions defined as rewrite rules in an extended Gödel source program. The parsed Gödel program output by her parser provides the input for my code generator. Taken together, our work represents an integrated functional-logic programming language.

I have implemented *leftmost-innermost narrowing* to evaluate functional terms. *Narrowing* is a functional computational method compatible with logic programming languages. It is the most popular computation method in use today for the integration of functional and logic languages [Hanus].

Narrowing requires that functions be defined in a manner known as rewrite rules. Rather than narrowing terms directly, my implementation of narrowing requires the translation of rewrite rules into clauses. Thus, one major addition I have made to the Gödel compiler is the inclusion of a rule-to-clause translator for rewrite rule compilation.

The other major changes I have made concern the detection and evaluation of functional arguments within clauses and goals. I have greatly expanded Gödel's primitive pre-defined function evaluation system to permit the evaluation of user-defined functions. I have added a narrowing mechanism for evaluating functional arguments during clause and goal compilation.

Due to the way functions are declared in this extension of Gödel, they are indistinguishable from constructors in the parsed code. I have implemented a method of function detection for user-defined functions.

Technically speaking, the Gödel compiler comprises the front-end parser and the back-end code generator. I will refer to the front-end as the parser and to my back-end code generator as the compiler. This corresponds to the division made in the Gödel source code.

The structure of this paper is as follows. Chapter 2 explains some theoretical underpinnings--logic and functional programming and narrowing. Chapter 3 introduces both the standard and extended Gödel logic programming languages. Chapter 4 discusses the compiler design changes and additions necessitated by extended Gödel. Chapter 5 examines the implementation of extended Gödel from a broad perspective, while Chapter 6 does so in finer detail. Chapter 7 draws some conclusions from the work done. Chapter 8 looks briefly at two other functional-logic programming languages.

2 Background

2.1 Logic Programming

This section introduces basic logic programming terminology used throughout the paper. For a full treatment of logic programming, see [Art]. It concludes with a motivation for choosing narrowing as the computational method for integrating functions into a logic programming language.

"A logic program is a set of axioms...defining relations between objects. A computation of a logic program is a deduction of consequences of the program." [Art, p. 9]. The classic logic programming example of a relation is the family relations program

```
Father(Abraham, Isaac).
Father(Abraham, Ishmael).

Male(Isaac).
Male(Abraham).
Male(Ishmael).
```

The first statement states that Abraham is the father of Isaac, the second that he is Ishmael's father. The last three state that they're all men.

Another example is

```
Plus(2, 3, 5).
```

A more complex relation may be of the form

```
Son(x, y) <- Father(y, x) & Male(x).
```

which means that x is the son of y if y is the father of x and x is a male.

The relations *Father, Son* and *Plus* are known as *predicates*. They, and all terms comprising a name and arguments, are known as *functors*. "A functor is characterized by its *name*...and its *arity* or number of arguments. Constants are

considered functors of arity 0.... A functor *f* of arity *n* is denoted *f/n*. Functors with the same name but different arities are distinct." [Art p. 27]. Accordingly, the predicates we have defined above are denoted by the functors *Father/2, Son/2* and *Plus/3*.

The above relation statements are known as *clauses*. The *head* of the clause is an indivisible term known as an atom. *Plus(2, 3, 5)* and *Son(x, y)* are heads of their respective clauses. A clause consisting of a head only is also referred to as a *fact*. The *Plus* clause is a fact. The *body* of a clause is an optional, possibly complex series of atoms appearing to the right of the <- arrow. A logic program consists of one or more of these clauses.

A logic program executes through the evaluation of *goals*. A goal resembles a clause head or body, but is a query which is answered positively or negatively by examination of the program clauses during execution. For example, the goal

```
Father(Abraham, Isaac)
```

would be answered positively, or *succeed,* whereas the goal

```
Father(Abraham, Sarah)
```

would be answered negatively, or *fail.*

Another type of goal is an existential query. Given the facts above, we expect the goal

```
Son(w, Abraham)
```

to succeed with answers

```
w = Isaac.
w = Ishmael.
```

whereas we expect the goal

```
Son(w, Isaac)
```

to fail.

Goals are solved using *resolution*. Goals are repeatedly replaced by sub-goals until each sub-goal can be solved. Replacement of goals by sub-goals is accomplished through *unification*. Two terms are unifiable if performing substitutions on variables of one or both of the terms makes the terms identical. To resolve a goal, we attempt to unify the goal with the head of a program clause, resolving any sub-goals in the clause body as necessary. For example, the goal

```
Son(w, Abraham)
```

binds *y* in *Son/2* to *Abraham*. The first body clause of *Son/2* becomes

```
Father(Abraham, w).
```

which becomes our first sub-goal. If we substitute *Isaac* for *w*, we can unify this sub-goal with the program fact

```
Father(Abraham, Isaac).
```

The second body atom of *Son/2* now becomes

```
Male(Isaac).
```

due to the substitution, and our goal

```
Son(w, Abraham)
```

succeeds as

```
Son(Isaac, Abraham).
```

Note that we need not stop evaluation there. We may also attempt the substitution of *Ishmael* for *w*, and our goal also succeeds with this value. This *backtracking* mechanism is a key element of logic programming, and enables us to find multiple correct solutions to a query, if they exist.

It also enables us to continue evaluation after discarding incorrect solutions. For example, suppose our program contains the fact

```
Father(Abraham, Deborah).    %  Forgive the blasphemy
```

When executing the query

```
Son(w, Abraham).
```

our newest fact will allow *x* in the

```
Father(y, x)
```

subgoal of *Son/2* to succeed as

```
Father(Abraham, Deborah).
```

But

```
Male(Deborah)
```

will fail, causing Deborah to be rejected as a son of Abraham. Nevertheless, due to backtracking, our computation can continue and successfully return Isaac and Ishmael as sons of Abraham.

Note further that we are not guaranteed in which order our solutions will be returned to us. *Deborah* may be attempted before or after the boys. Nor is the order of body formula evaluation guaranteed. The program may evaluate the *Male* sub-clause before the *Father* sub-clause. In these two ways, a logic program is *non-*

deterministic. That is, the declaration of a program does not indicate the order of execution of clauses or sub-clauses.

*   *   *   *

A logic program states what to do, but not how to do it. Otherwise said, a logic program states the logic, but not the control, of the program [Escher, p. 3]. This implies, as previously noted, that the execution of a logic program is non-deterministic; i.e., the order of statement evaluation at execution time is unknown. Consequently, it is quite possible that some statement variables will be uninstantiated when that statement is executed.

Consider, for example, a trivial example where the predicates *add1* and *add2* mean, respectively, add one and add two to the first argument to obtain the second:

```
add2(x, z) <- add1(y, z) & add1(x, y).
```

If the body sub-clauses are evaluated in left-to-right order when solving a goal like *add2(1, w), y* will be uninstantiated at the first call to *add1*.

Evaluation with partial information, the ability to proceed with program execution in the face of uninstantiated predicate variables [Escher p. i], is a requirement of a viable logic programming language. Narrowing is a functional computational model which proceeds, via unification, when faced with partially-instantiated terms. The concept of narrowing is thus quite compatible with the logic programming paradigm.

2.2 Functional Programming

In this section, several advantages reaped by adding a functional component to a logic programming language are discussed. The lack of a mechanism in logic programming to handle functions is noted, a lack we will fill using narrowing.

Many relations are most naturally expressed as functions. For example, the *Father* relation expressed above as a predicate may also be expressed functionally as

```
Father(Isaac) --> Abraham.
Father(Ishmael) --> Abraham.
```

In fact, this method of expression is clearer than its logic counterpart in denoting exactly who is the father and who is the son. Arithmetic operations are also much more easily understood as functional operations; e.g., it makes more sense to think of 2 + 3 returning the value 5 than it does to think of a *Plus* predicate with arguments 2, 3, and 5. Naturalness of expression is one of the main goals of logic programming. The addition of functions to logic language aids in that goal.

There are other implementation-dependent benefits to adding a functional component to logic programming. As opposed to predicates, functions expressed as rewrite rules do not necessarily require *choice points*. That is, a compiler may demand that rewrite rules defining a function be mutually exclusive, or it may so interpret them. In either case, there will be only one rule which can be chosen for execution, given ground arguments. This implies there will be no time-consuming backtracking during function evaluation.

For example, if we are trying to find the father of Isaac using the predicate version of *Father,* we have to create a choice point when selecting Abraham as the father to account for the possible existence (?!) of another fact defining a second father for Isaac. If we use the functional version of *Father,* we immediately have the unique result

Father(Isaac) = Abraham

and no choice point need be created for future backtracking.

Functions also afford the possibility of implementing *lazy evaluation.* An expression is not evaluated until it is actually needed. One benefit of this is to increase the number of programs that successfully terminate.

For example, consider the following functions from [Antoy] for printing the first *n* prime numbers:

```
             primes --> sieve(ints_from(2))
       sieve([A|B]) --> [A|sieve(filter(B,A))]
    filter([A|B],C) --> if factor(C,A) then filter(B,C)
                                       else [A|filter(B,C)]
       ints_from(A) --> [A|ints_from(succ(A))]
      show(0,[A|B]) --> []
show(succ(A),[B|C]) --> [B|show(A,C)]
```

A call such as *show(50, primes)* will never finish if eagerly evaluated, since *primes* is a non-terminating function. If lazily evaluated, this call can succeed, since *show* can terminate after it has received fifty prime numbers from *primes*.

Integrating functions into a logic programming language may be a great idea, but logic languages are equipped neither to compile function definitions nor to evaluate functional predicate arguments. A mechanism must be installed in the language for this purpose. Narrowing is the mechanism we will use for this purpose.

2.3 Narrowing

In order to implement narrowing, functions must be defined as rewrite rules. Here we define *rewrite rules* (also referred to simply as *rules*), as well as *terms* and *substitutions*. We then show how narrowing consists of: 1) applying a substitution to a term; and 2) applying a rewrite rule to that substituted-for term.

Terms

A *symbol* is an arbitrary item. We typically enumerate a set of symbols which we will find useful. For example, the set of symbols

S = {Zero, Succ, Plus, True, False, &, | , <}

will be useful for defining some basic arithmetic and boolean operations.

A *sort* defines the type of a symbol. For our example, *natural* and *boolean* are likely sorts.

A *typing function,* denoted *tf,* associates a symbol with a non-empty list of sorts. We type our set of symbols as follows:

tf(Zero) = natural
tf(Succ) = natural, natural
tf(Plus) = natural, natural, natural
tf(True) = boolean
tf(False) = boolean
tf(&) = boolean, boolean, boolean
tf(|) = boolean, boolean, boolean
tf(<) = natural, natural, boolean

As we can see, each symbol resembles a function whose output type is the last element in its type list and whose inputs types are all the preceding elements.

A *term* is an expression made up of validly typed symbols. Some terms from our example are

Succ(Zero)
Plus(Zero, Zero)
True & (Zero < Zero)

and some expressions which are not terms are

Plus(Zero)
True | Zero.

Substitutions

Symbols for unknown values are called *variables*. In the following examples, we will denote them as *w, x, y,* or *z*. They take no arguments, but assume the sort of the value for which they stand.

A *substitution* maps the elements of a finite set of variables to a set of terms, which can themselves contain variables. For example, if we apply the substitution

{ x |--> Zero, y |--> Succ(z) }

to the term

Plus(z, Succ(Plus(x, y)))

we obtain

Plus(z, Succ(Plus(Zero, Succ(z)))).

Rewrites

A *rewrite system* is a set *R* of pairs of terms, called *rewrite rules,* with the following requirements: if *l* --> *r* is in *R,* then *l* and *r* have the same sort, *l* is not a variable, and each variable of *r* is a variable of *l.* The set *R* for our example might contain, in part, these rules

Plus(Zero, x) --> x	R1
Plus(Succ(x), y) --> Succ(Plus(x, y))	R2
True & x --> x	R3
False & x --> False	R4

If a substitution applied to the left side of a rewrite rule yields some subterm of a term, the *rewrite* operation consists of replacing that subterm with the result of applying the same substitution to the right side of the rule. For example, if we have the term

Plus(Succ(Zero), Succ(Zero))

then the substitution

{ x |--> Zero, y |--> Succ(Zero) }

applied to the left side of R2 yields our entire term. Thus, we may rewrite the whole term by applying the substitution to the right side of R2 to get

Succ(Plus(Zero, Succ(Zero)))

We may continue rewriting by applying the substitution

{ x |--> Succ(Zero) }

to R1 to yield the subterm

Plus(Zero, Succ(Zero))

Again, we rewrite the whole term by applying the substitution to the right side of R1, yielding

Succ(Succ(Zero))

In sum, we have applied two consecutive rewrites to simplify

Plus(Succ(Zero), Succ(Zero))

to

Succ(Succ(Zero))

A broader view reveals that we have used rewriting to compute

$1 + 1 = 2.$

As a second example, the term

Zero < Plus(Zero, Zero)

rewrites to

Zero < Zero

by applying the substitution

{ x |--> Zero }

to R1.

Narrowing

Narrowing is rewriting with a more generalized substitution method used. Rewriting can be seen as employing a pattern-matching type of substitution, where the substitutions are one-way. That is, variables in the rule are replaced by symbols in the subterm. Narrowing employs unification, which, as explained previously, is a two-way substitution. Variables in the rule may be replaced by symbols in the subterm, or vice versa--symbols in the subterm may be replaced by rule variables. For example, the term

Plus(w, z)

may be narrowed by applying the substitution

{ w |--> Zero, x |--> z }

to *w* of the term (the subterm and term are one and the same in this case) and *x* of R1 to yield the unified term

Plus(Zero, z)

Applying rewrite rule R1 yields

z.

As a second example, the term

Plus(w, z)

unifies with the left side of R2 as

Plus(Succ(Zero), Zero)

when the substitution

{ w |--> Succ(Zero), z |--> Zero, x |--> Zero, y |--> Zero }

is applied to w and z of the (sub)term and x and y of R2. Completing the narrowing operation by applying rewrite rule R2 yields

Succ(Plus(Zero, Zero))

Further rewriting is possible using R1 to yield

Succ(Zero)

Function evaluation using narrowing

We will use narrowing to simplify functional terms which we encounter as we process the arguments of a predicate clause. Specifically, we use repeated rewrites, and the more general narrowing process when we are unable to rewrite, until we've rewritten and narrowed the functional subterm out of existence. We will be left with variables and constructors (unevaluable, irreducible functor terms).

As an example, let's define a predicate to test whether a number is equal to one:

IsOne(Succ(Zero)).

Now let's attempt to solve the goal

IsOne(Plus(Succ(Zero), x))

Using rule R2, we rewrite the expression to

IsOne(Succ(Plus(Zero,x)))

Then we use rule R1 to rewrite this as

IsOne(Succ(x))

As we see in the example, we have applied repeated rewrites to the function call until it is reduced to constructor terms. This is as simplified a term as we can hope for.

Where does narrowing enter into the picture? Consider an example where we are solving the following goal:

IsOne(Plus(w, z))

In this case, there is no substitution we can apply to the variables on the left side of any of our rewrite rules which will produce the subterm *Plus(w, z)*. In order to continue the computation, we must narrow by applying a unifying substitution to a rule and the term. We can choose to unify with R1 with the substitution

{ w |--> Zero, x |--> z }

and then apply the rule to yield

IsOne(z).

Or we can unify with R2 using the substitution

{ x |--> Zero, w |--> Succ(Zero), y |--> z }

yielding

IsOne(Succ(Plus(Zero, z)))

which can be rewritten using R1 to

IsOne(Succ(z))

In essence, we have made a 'guess' as to an appropriate value for *w* that allows us to continue rewriting. In practice, we will make the *most general unification,* i.e., a unification from which all other unifications can be derived.

What if we 'guess' wrong? If a choice point exists, we backtrack and try a different unifier. Otherwise, we fail. The ability to backtrack is another distinguishing factor between narrowing and rewriting. The latter does not require backtracking.

Partially-instantiated terms are a fact of life in logic programming. The allure of narrowing resides in its ability to carry out computations in the face of such terms.

3 Gödel

Rather than create a functional-logic programming language from scratch, we have chosen to integrate functions into an existing logic programming language. This integration is a much more tractable project than the creation of an entirely new language. The obvious drawbacks are the potential constraints contained in that language and the obstacles it presents when altering it.

We chose the Gödel logic programming language as our starting point. Gödel has many desirable features that make it a sensible choice. It is publicly and freely distributed along with its compiler's source code. It has good documentation. With the exception of input/output, it is nearly free of non-declarative predicates.

The Gödel compiler, implemented in Prolog, run as an application above the underlying Prolog system (SICStus in our case). It compiles a Gödel source file into Prolog. The Prolog system then executes the Prolog code.

Except where noted, all the logic programming language examples included in this paper thus far are valid Gödel fragments.

In the following sections, we introduce first the Gödel language as it presently exists, and then our extension to it.

3.1 Introduction to Gödel

A brief description of the Gödel language follows. (For more details, see [Gödel].) A Gödel source program is contained in one or more modules. A module declares all the symbols of the language of that module. A module may *import* other modules in order to avail itself of their symbols.

Symbols begin with upper-case letters and variables begin with lower-case letters. Every symbol is declared as one of six categories: BASE, CONSTRUCTOR, CONSTANT, FUNCTION, PROPOSITION, or PREDICATE.

BASE declarations enumerate the types (sorts) of the module language. Constants of any type are declared under the CONSTANT category. Predicates are declared using the PREDICATE category. The name of the predicate is listed followed by the type of each argument. One or more clauses defining the predicate may appear anywhere in the module following the predicate's declaration. PROPOSITION declares a proposition; i.e., a predicate with no arguments.

Unfortunately for our discussion, the meanings of the CONSTRUCTOR and FUNCTION categories are **not** the same as the meanings we have given them throughout this paper. What we have heretofore referred to as a constructor is declared in Gödel under the FUNCTION category. What we have been calling a function--an evaluable functor term--barely exists in Gödel. Gödel contains a few pre-defined evaluable functions--the basic arithmetic operations, and some operations on strings.

We will continue to refer to functions as functions, or evaluable functions, and to constructors as constructors. When we wish to refer to the Gödel meaning, we will either preface it with the word *Gödel* or denote it using upper-case letters.

A CONSTRUCTOR is declared, along with its arity, to construct new types from a BASE. For example, if we have declared a BASE *Day* and a CONSTRUCTOR *List* of arity one, then the types of the module language are: *Day, List(Day), List(List(Day)),* etc. [Gödel, p. 17].

Gödel is polymorphic. A single FUNCTION or PREDICATE may be declared to accept different types as its arguments. For example, the declaration

```
PREDICATE Append : List(a) * List(a) * List(a).
```

declares *Append* to be a predicate which accepts three list arguments. The type of elements contained within the list may vary from predicate call to predicate call.

3.2 Extended Gödel

Our extension to Gödel consists of allowing users to define functions. These functions may be employed as arguments to predicates in definitions or goals.

Our extension to integrate functions into Gödel requires one syntax change, the addition of the reserved, binary, infix operator symbol =>. A function is declared identically to a constructor as the category FUNCTION. Its definition is accomplished similarly to a predicate definition, but with the left arrow <- being replaced by =>. So, our *Father* predicate could be declared and defined in standard Gödel as

```
BASE Guy.
CONSTANT Abraham, Isaac, Ishmael : Guy.
PREDICATE Father : Guy * Guy.
Father(Abraham, Isaac).
Father(Abraham, Ishmael).
```

Or, *Father* could be declared and defined as a function in extended Gödel as

```
BASE Guy.
CONSTANT Abraham, Isaac, Ishmael : Guy.
FUNCTION Father : Guy -> Guy.
Father(Isaac) => Abraham.
Father(Ishmael) => Abraham.
```

Note that a rewrite rule may have a conditional clause, much like a predicate has a body. For example, the rule

```
Plus(x, y) => y <- IsZero(x).
```

can be understood as

If IsZero(x), then Plus(x, y) => y.

Rule definitions must observe certain restrictions to insure that extended Gödel possesses desirable qualities such as confluence and consistency. They must be left-linear [Klop]. That is, variables may not be repeated on the left side. They must be constructor-based. In other words, the outermost left side symbol must be a function, and no functions may appear in left side subterms. Extra variables, those appearing in the right side or condition but not in the left side of a rule, must satisfy several technical conditions discussed in [SMI]. For an extensive discussion of the restrictions on extended Gödel rewrite rules, see [JV].

Here is a simple example module, first in standard Gödel and then in extended Gödel:

```
MODULE     Nat.
BASE       Nat.
CONSTANT   Zero.
FUNCTION   Succ: Nat -> Nat.
PREDICATE  Plus: Nat * Nat * Nat;
           IsZero: Nat.
IsZero(Zero).
Plus(Zero, x, x).
Plus(Succ(x), y, Succ(z)) <- Plus(x, y, z).
```

```
MODULE     Nat.
BASE       Nat.
CONSTANT   Zero.
FUNCTION   Succ: Nat -> Nat;
           + : yFx(510): Nat * Nat -> Nat.
PREDICATE  IsZero: Nat.
IsZero(Zero).
Zero + x => x.
Succ(x) + y => Succ(x + y).
```

Whereas *Plus* must be declared as a predicate in standard Gödel, + is declared as an infix function in extended Gödel, which allows for a more natural mode of expression. For example, to test whether the sum of two numbers is zero using the standard Gödel module, we would have to write the goal

```
Plus(x, y, z) & IsZero(z).
```

Extended Gödel allows us the more natural expression

```
IsZero(x + y).
```

4 Compiler Design

Extending Gödel to include functions required additions or alterations to the Gödel compiler in the areas of rule translation, function detection, and function evaluation. The design decisions underlying these additions and alterations are discussed below. (In the following discussion, I have adopted some of the terminology used within the Gödel compiler. Thus, when I refer to a 'predicate' or 'statement', I am really referring to a clause which defines a predicate. Likewise, I refer to a 'rule' to mean a formula which defines a function. Thus, when I write about predicate (or statement) or rule compilation, I am referring to the compilation of a clause or formula which defines a predicate or function.)

Recall that the Gödel compiler compiles Gödel source code into Prolog source code, then relies on the Prolog compiler for compilation and evaluation of predicates and goals. Implicit in the Prolog compilation process is the unification of goals with

clause heads using substitution, backtracking upon failure to unify; i.e., all the processes we have mentioned in our discussion of computation using narrowing.

The extended Gödel compiler takes advantage of these Prolog facilities. In effect, all we need do is re-package functions in a form palatable to the Prolog compiler. That done, the standard Gödel compilation process passes them to the Prolog compiler, which performs the bulk of the narrowing process.

This re-packaging is accomplished via a simple implementation of narrowing. This implementation is leftmost-innermost, not needed narrowing. Consequently, it does not permit lazy evaluation. Its overriding virtue is simplicity, a major concern in this first functional-logic implementation.

4.1 Evaluation of functional arguments

When a functional expression is encountered as an argument of a predicate, rule, or goal during compilation, it must be evaluated by narrowing. The standard Gödel compiler provides a narrowing-like implementation for its pre-defined arithmetic and string functions. The technique converts the functional argument into predicate form for subsequent standard Gödel compilation. We extend this technique to user-defined functions. The extended technique conforms to the method expounded in [vEY, p 281] for reducing terms and goals.

4.2 Rule translation

Since the narrowing process for evaluating functional expressions appearing as arguments to predicates has converted the functional argument into predicate form, we must do the same to the function definition. Otherwise, the Prolog compiler will be unable to find a unifying clause head during evaluation of the functional argument.

Recall that user-defined functions are denoted in extended Gödel source programs as rewrite rules. Accordingly, we have created a rule translation scheme which translates the rule into predicate form. The former rule (now a predicate) is then compiled via the standard Gödel compilation method.

The rule translation scheme is consistent with that demonstrated in [vEY]. It translates all rules of a constructor-based rewrite system; that is, rules of the form

f(t1,...,tk) --> t

where *f* is a function and no functions appear in arguments *t1,...,tk.*

tau algorithm

The **tau** algorithm [AFM] is the basis of the rule translation scheme. It translates a function into a predicate by flattening the rewrite rule representing the function and

recursively flattening any functional expressions found among the arguments within the rule. On its initial invocation, **tau** processes the whole rewrite rule. Recursive calls to **tau** handle arguments of the rule, which may be variables, constructor terms, or function terms.

In order to understand the **tau** algorithm, it is necessary to introduce some special notation, *dot* and *bar*. "If *f* is a function whose range is a set of non-null strings, then *dotf(x)* is the last element of *f(x)*, and *barf(x)* is *f(x)* without its last element." [AFM]. Using a comma to denote the separation of string elements, we have *f(x)* = *dotf(x), barf(x)*. We will use *dottau* and *bartau* to define **tau** algorithmically.

First consider how **tau** processes a rewrite rule argument. In the simplest case, where the argument is a variable, **tau** returns that variable. That is,

tau(X) = X, if X is a variable.

Thus, *bartau* of a variable is empty and *dottau* of a variable is the variable itself.

If the argument is a constructor term, **tau** returns a list of elements consisting of *bartau* of each constructor term argument and the constructor term itself with new arguments *dottau* of each of its old arguments:

tau(c(t1,...,tk)) = bartau(t1),...,bartau(tk), c(dottau(t1),...,dottau(tk))
where c is a constructor.

bartau is empty for constructor term arguments which are variables, so nothing is extracted from the constructor term. *dottau* of a variable constructor term argument is the variable itself, so the constructor term argument remains unchanged. If the

constructor term argument is itself a constructor term, it produces an empty *bartau* and itself as *dottau,* with the proviso that its arguments will be recursively processed. If the constructor term argument is a functional term, it will be extracted and processed further, and a simple, new variable left in its place, thus accomplishing flattening.

A functional term argument must itself be given a predicate 'look'. This means an extra variable (denoted as *T* below) must be created and every functional term argument processed. Thus:

tau(f(t1,...,tk)) = bartau(t1),...,bartau(tk), f(dottau(t1),...,dottau(tk),T), T
where f is a function.

Now consider how **tau** behaves when first invoked. **tau** invoked on a rewrite rule returns one of two possible values. If the right side of the rule--that is, the value of the function--is a simple variable or constructor term, it becomes the new variable in the predicate clause being created. This predicate clause will have only a head. Otherwise, the right side must be further processed, and the predicate clause will have a head and a body:

tau(f(t1,...,tk) --> t) = f(t1,...,tk, dottau(t))
if bartau(t) is null.

tau(f(t1,...,tk) --> t) = f(t1,...,tk, dottau(t)) :- bartau(t)
otherwise.

4.3 Function detection

Since the Gödel parser does not differentiate between functions and constructors, functional expressions used as predicate arguments must be detected by the compiler in order to be compiled correctly. The standard Gödel compiler detects pre-defined functions such as arithmetic functions. But this ability needs to be greatly augmented in order to handle user-defined functions.

Function detection is accomplished by furnishing a list of names of rules, both local and imported, to any predicate which needs to do such detection. This list is contained in a file created by the parser. It includes the names of all rules declared in the local module and in modules imported by the local module. The names are the parsed name structures of the rules. They include the module name and arity of the rule, thereby guaranteeing uniqueness.

5 Implementation Overview

The parser hands the compiler a fully parsed Gödel program. The extended Gödel parser also creates a rule file, *<module_name>.ef,* which contains a list of names of rewrite rules.

The standard Gödel compiler extracts a body of code containing predicate clauses from this parsed program. It then processes the clauses one-by-one, compiling them into equivalent Prolog clauses and writing them out to a Prolog code file, *<module_name>.pl,* as a side effect. This Prolog file is further processed to create a compiled Prolog file, *<module_name>.ql,* used by the Gödel loader to quickly load the module. In addition, a language, or symbol table file, *<module_name>.lng,* is created.

The extended compiler also extracts the body of code from the parsed program. The code contains predicate clauses, among which are the rewrite rules parsed as predicates. The compiler separates the predicate clauses and the rewrite rules into two structures known as the statement code and the rule code.

Predicate compilation does not differ from standard Gödel to extended Gödel. The rules are compiled analogously to the predicate clauses, with one major exception: each rule is converted from a functional form into a predicate form which can be compiled in standard Gödel fashion into a Prolog clause.

The compiler comprises rule, statement (predicate), and goal compilation. Goal compilation follows the statement compilation path; it will not be described separately.

The compiler's work consists of two main functions. The first is to dissect the complex parsed program structure to obtain the statement or rule to be compiled. The second is to compile (or build, as it is termed by the Gödel compiler) the rule,

statement, or goal into a Prolog predicate or goal. The build process includes rule translation and function detection and evaluation. The narrowing process which we have implemented is initiated during the build process.

5.1 Parsed Program Structure

The parsed program is a complex bundle of multiply-nested structures. The compiler must unravel this complex structure and extract the desired parts in order to accomplish its tasks. Since the parsed program is a Prolog object, the structures are not typed. They do, however, have a functor-like composition consisting of a name and arguments. The name contains three parts. The first part indicates the meta-type of the object. The second part is the name of the structure. The third part is a letter-number combination. The letter *F* stands for FUNCTION, which is best understood in the Gödel sense; that is, as a constructor. The number indicates the arity of the structure.

Program

The program handed to the Gödel compiler by the parser is structured as follows:

ProgDefs.Program.F4(module name, module definition tree, language tree, code).

Its four arguments are: the name of the main module in string form; the module structure (the main module and imported modules) in tree form (all trees in the program are AVL trees); the language (symbol table) of the program in tree form; the program code in tree form.

Code

The program code is the program structure of most interest to the compiler:

ProgDefs.Code.F2(integer, code tree).

The first argument is an integer, used as a version number, which is not significant for our purpose. The second argument is the code itself in tree form.

Code tree

AVLTrees.Node.F5(left tree -- predicates and rules,
predicate name (the name => indicates a rule),
list of predicate definitions,
balance state of tree,
right tree -- predicates and rules
)

Each node of the tree contains, in standard Gödel, a list of predicate definitions. The list contains all predicate definitions for a given predicate name. Each element of the list is itself a list containing all the definitions for a given predicate name and arity. In extended Gödel, the node may alternatively contain all the definitions for the

predicate => ; i.e., a list of all rewrite rule definitions for the module undergoing compilation.

Predicate Definition

ProgDefs.PredDef.F4(arity, definition list, import delays, export delays)

The structure of the predicate definition is as follows. Its first argument is its arity. Its second argument is a list of the predicate clauses comprising its definitions. The third and fourth arguments are delay declarations which affect when the predicate is evaluated.

If the predicate definition contains rewrite rules, its arity is always two, its two arguments being the left and right sides of the rule. Delay declarations are not allowed in rule definitions, so they will always be empty, or a compile error will occur.

Predicate

MetaDefs. <-.*F2*(head, body)

The individual elements of the predicate definition list are identified as *MetaDefs.* <-.*F2*. Each one is one predicate clause. The two arguments represent the head and the

body of a predicate. The predicate head is an atom identified as *MetaDefs.Atom.F2,* and the body is labeled a term, *MetaDefs.Term.F2*. The head is a two-part structure consisting of the predicate name and a list of terms representing its arguments. The body term is a list of terms comprising the body. It is referred to by the compiler as a formula.

If the predicate clause is a rewrite rule definition, its predicate head name always is = >. The head contains two arguments, the left hand side and the right hand side of the rule. The name of the first argument is the actual rule name. The arguments of the first argument are the arguments of the left hand side of the rule. The second head argument is the right hand side of the rule. If the rule has a conditional clause, it is represented as the clause body.

Name

The standard Gödel symbol name is a four-part structure:

> *MetaDefs.Name.F4*(module name, symbol name, symbol type, symbol arity)

Its arguments are: the name of the module in which the symbol is declared; the name of the symbol itself; the type of the symbol; its arity. The type is identified as *MetaDefs.<X>.C0*. *X* is *Predicate* for a predicate, *Function* for a Gödel FUNCTION or rewrite rule, *Constant* for a constant, etc. The only symbol which does not use this name form is a variable. A variable is identified as

MetaDefs. Var.F2, where the first argument is the variable identifier (e.g., 'x'), and the second argument an integer index for generating unique variables.

To illustrate the use of these structures, here is an example of a parsed rule:

Example rule: `Plus(Succ(x), y) => Succ(Plus(x, y)).`

```
MetaDefs.<-.F2(                                    % predicate
  MetaDefs.Atom.F2(                                % head
    MetaDefs.Name.F4(",
                    "=>,                           % rule
                      MetaDefs.Predicate.C0.,
                      2),
    [ MetaDefs.Term.F2(                            % rule lhs
      MetaDefs.Name.F4("Nov7,
                      "Plus,                       % rule name
                        MetaDefs.Function.C0,
                        2),
      [ MetaDefs.Term.F2(                          % rule args
          MetaDefs.Name.F4("Nov7,
                          "Succ,                   % 1st arg
                            MetaDefs.Function.C0,
                            1),
          [ MetaDefs.Var.F2("x, 0)]),              % 1st arg's
                                                       arg
        MetaDefs.Var.F2("y, 0)]),                  % 2nd arg
      MetaDefs.Term.F2(                            % rule rhs
        MetaDefs.Name.F4("Nov7,
                        "Succ,                     % rhs name
                          MetaDefs.Function.C0,
                          1),
        [ MetaDefs.Term.F2(                        % rhs args
            MetaDefs.Name.F4("Nov7,
                            "Plus,                 % arg name
                              MetaDefs.Function.C0,
                              2),
            [ MetaDefs.Var.F2("x, 0),              % arg's 1st
                                                       arg
              MetaDefs.Var.F2("y, 0) ])])]),       % arg's 2nd
                                                       arg
    MetaDefs.Empty.C0)                           % no condition
```

To clarify the predicate structure of a rule as shown above: a rule such as

```
Plus(Succ(x), y) => Succ(Plus(x, y)).
```

is parsed as the bodyless predicate clause

=> (Plus(Succ(x), y), Succ(Plus(x, y)).

To get to the rule itself, it is necessary to extract the predicate => head arguments, which represent the left and right hand sides of the rule.

If the rule contains a condition; for example

```
Plus(x, y) => y <- IsZero(x).
```

It is parsed as the predicate

=> (Plus(x, y), y) <- IsZero(x).

where the parsed condition becomes the predicate body.

5.2 Build Process

Once the compiler has extracted a rule or predicate from its enclosing structure, the essence of the compile process is to convert the rule or predicate to a Prolog predicate clause. The compiler terms this process the build process. Each rule or predicate is converted to a flat form; i.e., changed from a complex to a single-level structure which can be used as a Prolog identifier. The rule or predicate name is converted from a Gödel structure to a flat form. Then each term (argument) of the rule or predicate is built.

How this is done depends on the type of the term. Variables are added to a variable dictionary, so that they will be recognized if they re-occur in the clause. The values of primitive system types such as integers, floats, and strings are extracted from their Gödel structural representation; e.g., the integer 1 is represented '1'. Constants are also converted from the Gödel structure to a flattened form. Finally, terms which are functors (functions or constructors) are built recursively: the functor name is flattened, and each argument is built.

Here is an example of the build process, using the predicate clause

```
IsZero(Zero).
```

from the example module *Iz* shown previously. The parsed Gödel predicate clause structure extracted by the compiler for input into the build process is

```
MetaDefs.Atom.F2(
 MetaDefs.Name.F4("Iz, "IsZero, MetaDefs.Predicate.C0,
                                                    1),
   [MetaDefs.CTerm.F1(
     MetaDefs.Name.F4("Iz, "Zero, MetaDefs.Constant.C0,
                                                    0))]).
```

The built predicate clause looks like this:

'Iz.IsZero.P1'('Iz.Zero.C0')

Each symbol has been given a flat form. The meta-type of the symbol (e.g., *MetaDefs.Atom.F2* or *MetaDefs.Name.F4*) has been discarded. The symbol's flat Prolog identifier contains three parts: the module name, the symbol name, and a one-letter, one-number combination in which the letter represents the symbol type (P for predicate and C for constant in this example), and the number represents the symbol arity. For example, the Gödel structure for the symbol *Zero*:

```
MetaDefs.CTerm.F1(
  MetaDefs.Name.F4("Iz, "Zero, MetaDefs.Constant.C0, 0))
```

is converted to the Prolog:

Iz.Zero.C0.

The head and the body of a predicate clause are built separately and then combined into a Prolog clause. Any functions in the head and/or body are transformed into what are known as constraints. Constraints also undergo a build process and are then inserted into the Prolog clause as part of the body. The constraint-building process is described later in detail.

5.3 Rule translation by tau

Rule building is the same as for predicates, except that the rule must first be translated into predicate form before being built. This is accomplished by the **tau** predicate. **tau** translates rewrite rules into predicate clauses to begin the narrowing process that is finished by the Prolog compiler. Conceptually, **tau** works by flattening. The rule representing a function definition is converted into predicate form by adding an extra variable argument to the function's argument list. This new argument receives the value that results from applying the function to its original arguments. Functional expressions found as arguments in the right hand side of the rewrite rule are lifted from the argument list and replaced by the extra argument.

They are placed back down, in predicate form, as part of the body of the clause being created.

Let's look at some examples. Consider the rewrite rule

```
Plus(Succ(x), y) => Succ(Plus(x,y))
```

Since *Plus* is binary, a third, extra argument is created for it. Now calls to *Plus* will be of the form

Plus(x, y, z)

where *z* takes on the value of the function call *Plus(x, y)*.

Calls to the function *Plus* must be removed from any argument lists it appears in--in this case, the argument list of *Succ* on the right hand side of the rule--and placed back down in predicate form as part of the predicate body. So the *Plus* predicate takes the form

Plus(Succ(x), y, n) <- Plus(x, y, z) & n = Succ(z).

But *Succ* is a constructor, hence unevaluable, so there is no reason to represent it with the extra variable *n*. In practice, the above intermediate step never occurs. Instead, we represent *Plus* directly as:

Plus(Succ(x), y, (Succ(z)) <- Plus(x, y, z).

Consider, for a second example, the rewrite rule for summing a list of numbers

```
Sumlist([a|b]) => Plus(a, Sumlist(b)).
```

Create a second argument, *z,* for *Sumlist.* Lift *Sumlist* from the argument list of *Plus* and place it back down in predicate form in the predicate body. Since *Plus* is also a function, create a third argument for it. The result is

Sumlist([a|b], z) <- Sumlist(b, n) & Plus(a, n, z).

6 Implementation Details

6.1 Compiler invocation

The predicates **compile_program** and **compile_program_aux** are called to invoke the compiler. The program received from the parser is the four-part structure containing the module name and the tree structures representing the module structure, the language (symbol table), and the code. The standard Gödel compiler code contains only predicate definitions, whereas the extended Gödel code also contains all rewrite rules.

The compiler at this point also dumps the language tree into a language file for use during program loading. Here we also read the rule list from the rule list file.

Just prior to initiating the compilation of the statements and rules, the two are separated by a call to **separate_statements_and_rules**, a new predicate. This predicate steps through the program code examining each node to determine whether it contains a rule or a statement (predicate). The two are distinguished by the predicate name, which in the case of a rule is =>. Two new structures, one containing only rule code, and one containing only statement code, are created.

These two structures are handed to **general_compile_module**, which calls a series of predicates which effect the compilation of each predicate clause and rewrite rule.

6.2 Predicate compilation

The series of predicate calls responsible for predicate compilation is essentially unchanged from standard Gödel to new. **general_compile_module** calls **outer_compile_module,** a new predicate created to split predicate compilation from rule compilation, which in turn calls the newly-named **p_compile_module** (formerly **compile_module**) to compile the predicates. This predicate calls **compile_predicates** to extract all the statements for one predicate and pass them to **compile_statement_list,** which extracts one statement and passes it to **compile_statement.**

compile_statement is the workhorse predicate. It compiles each predicate statement into Prolog which is written to the Prolog code file. Each Gödel statement is divided into a head and a body, which are processed separately. The predicate **build_head** is called to convert the predicate name and arguments found in the head into Prolog names. Each argument must be checked to see whether or not it contains functions, which must be converted into constraints. After this is done, each argument is built as described previously. The built head is returned along with a (possibly empty) list of constraints. The latter are built into Prolog by **build_constraints** and its auxiliary

predicates. These constraints will be merged with the built body when the final Prolog clause is created.

Analogous processing is done on the body by **compile_formula**. **compile_formula,** like **build_head,** checks for functions in the body terms' arguments and replaces them with constraints. It then builds all term names and arguments into Prolog equivalents and combines them into a Prolog predicate formula. **build_constraints** is called, and any built constraints are concatenated with the built predicate formula into a comma-separated sequence of goals which is the Prolog clause body. If the original Gödel body is a compound clause, each clause functor is compiled separately by **compile_formula**, and the results are sequenced.

When the head, head constraints, and body have all been built, they are combined into a Prolog clause as follows:

BuiltHead :- BuiltConstraints, BuiltBody.

The built constraints and/or body may be empty. Alternatively, either or both may be comma-separated sequences.

As an example of a built predicate, consider this predicate from the *Sumlist* module which sums a list of integers:

```
PREDICATE SumList : List(Integer) * Integer.
SumList([], 0).
SumList([x|xs], x + y) <- SumList(xs, y).
```

The pre-defined function + is converted to a constraint and prepended to the body. The built Prolog equivalent clauses are

```
'Sumlist.SumList.P2'([], 0).
'Sumlist.SumList.P2'([A|B], C) :-
        'Integers':plus(A, D, C),
        'Sumlist.SumList.P2'(B, D).
```

6.3 Rule compilation

The rule compilation path, modeled after predicate compilation, is entirely new code. **outer_compile_module** calls **r_compile_module**, handing to it the tree structure which contains the module's rule code. **r_compile_module** hands **compile_rules** a node containing a list of *PredDef* structures defining the => predicate. **compile_rules** extracts an element of this list. **compile_rules** passes it to **compile_rule_list,** which in turn extracts one rule from the list and passes it to **compile_rule.** The rule is extracted from the head arguments of the => predicate. The optional rule conditional clause is extracted from the => predicate body. The rule and condition are passed as separate arguments to **compile_rule.**

compile_rule performs the actual rule compilation. It is very similar to **compile_statement**, with one notable difference and one huge difference.

Compilation of conditional clauses

The first difference between **compile_statement** and **compile_rule** is the compilation of conditional clauses. Rewrite rules may contain a conditional clause. For example, in the conditional rewrite rule

```
Plus(x, y) => y <- x = 0.
```

if x has been instantiated to *0,* then *Plus(x, y)* rewrites to y.

In the general conditional rewrite rule

```
X => Y <- Z.
```

Z , the conditional clause, may be any legitimate Gödel clause, although confluence of rules can only be guaranteed when the conditional clause contains only equations. This conditional rewrite rule is compiled as follows: X and Y are treated as the left and right hand sides, respectively, of the rewrite rule. Z is compiled separately by **compile_formula,** just as would be done for the body of a Gödel predicate. The output of **compile_formula** is then prepended to the sequence of compiled right hand side terms to form the complete Prolog predicate body.

Consider, for example, the recursive rule for factorial from the *SFact* module:

```
FUNCTION Fact : Nat -> Nat.
Fact(x) => Mul(x, Fact(y)) <- x = Succ(y).
```

The condition $x = Succ(y)$ is prepended to the body of the built Prolog clause. The *Fact(y)* argument to the multiplication function and *Mul* itself are also processed as described in the next section, yielding the built Prolog clause

```
'SFact.Fact.F2'(A, B) :-
        A = 'SFact.Succ.F1'(C),
        SFact.Fact.F2'(C, D),
        SFact.Mul.F3'(A, D, B).
```

tau implementation details

The major distinction between **compile_rule** and **compile_statement** is the former's use of **tau** to translate rules into predicate form. **tau** is implemented as follows. If **tau** is called on a variable or a non-evaluable constant, it will return that variable or non-evaluable constant unchanged as a one-element list.

tau called on a constructor term invokes **tau_loop** on the constructor term's arguments. **tau_loop** calls **tau** recursively on each of the arguments and returns the processed arguments in a list which becomes the new argument list for the constructor.

If **tau** is working on a functional term, it calls **tau_loop** to construct a new argument list. Then, the function's arity is increased, and a new variable appended to the new argument list to give the function the predicate 'look'. **tau** checks the rule list to detect a function. Thus, it only detects user-defined functions. Pre-defined functions are treated like constructors.

During its work, **tau_loop** has employed **tau** recursively to flatten the arguments. This means that the new argument list contains no functional terms. Any such terms have been lifted from inside the argument list and placed down in predicate form in

the body of the clause being created. The whole sequence of new predicates and new 'predicate-style' functions is now returned by **tau.**

tau called on a CONSTANT function acts similarly to **tau** called on a function. The main difference is that the constant function has no arguments on which to recurse.

In its initial invocation, **tau** called on a rewrite rule, the left hand and right hand sides of the rewrite rule are passed into **tau** as a two-element list. **tau** is called recursively on the right hand side of the rule, resulting in one of the cases described above. This recursive call returns, first of all, the new variable which is the new argument placed in the left hand side of the rewrite rule, now become the clause head. The recursive call also returns a list of statements. This list is converted to a conjunctive clause which becomes the clause body.

Completion of rule compilation

When **tau** has finished its processing, the result looks comparable to a Gödel statement. Since **tau** treats pre-defined functions (e.g., +) like constructors, it places these functions in the head arguments of the clause it returns. Consequently, when **build_head** is called on the translated left hand side, it may return constraints, and **build_constraints** must be called.

Subsequently, **compile_rule** completes its work similarly to **compile_statement.**

6.4 Function detection

The decision to use the list of rule names to detect functions leads to the necessity of passing the rule list all over the compiler code. It is passed throughout the predicates responsible for the compilation of predicates, rules and goals. It is also passed throughout Gödel's top-level command-execution sequence to enable program makes and loading of the rule list for goal compilation. This passing of the rule list throughout the compiler is fairly primitive, but straightforward to implement.

compile_program passes the rule list through all the predicates responsible for compilation. For rule compilation, the rule list used by **tau** to differentiate between functions and constructors. Elsewhere, it is used by the building predicates. These predicates include **build_head**, where it is used for detecting functions within a clause head or rewrite rule left hand side; **compile_formula**, where it is used for detecting functions within a clause body or rewrite rule right hand side or condition; **build_constraints,** where it is used for detecting functions found as arguments to another function.

The method used to detect user-defined functions follows from the method used for pre-defined functions. The predicate **replace_evaluatable** is called by every predicate that needs to do function detection. **replace_evaluatable** checks every functor argument. If the argument is not a Gödel FUNCTION or CONSTANT, the argument is not modified. If an argument is declared as a Gödel FUNCTION or

CONSTANT, **evaluatable_functor** is called to determine whether it is evaluable (a function) or not. A FUNCTION is evaluable under both standard and extended Gödel if it is a member of one of the following modules: Integers, Rationals, Sets, Strings. Standard Gödel provides hard-coded Prolog predicates to replace functions of these modules. In extended Gödel, **evaluatable_functor** also checks the rule list to see whether the FUNCTION or CONSTANT in question is a member of the list. If it is a member, it is evaluable.

Once **replace_evaluatable** has determined it is dealing with an evaluable function, it creates a new variable which is returned to the caller as the built term. This variable represents the 'result' of the function-soon-to-be-predicate. It also creates a pair consisting of the variable and the function. This pair is called a constraint and is also returned to the calling predicate.

If the FUNCTION or CONSTANT is not evaluable, it is returned untouched as the evaluated term, and any arguments are recursively checked for evaluable functions by **replace_evaluatable.**

6.5 Building constraints

When a functional argument is detected during compilation, it is reported out as a constraint pair as described above. This constraint must be built into a Prolog clause just as is done to the predicate.

The narrowing technique used to build constraints is a simple, recursive flattening technique similar in concept to that performed by **tau**. Essentially, the functional argument is made to look like a predicate. This is done by adding one more argument to it. This extra argument represents the value of the erstwhile function. Then the functional argument is removed from the argument list in which it is embedded and promoted to become one more predicate in the clause. It is replaced in the argument list by the extra argument.

For example, the clause

```
IsOne(Plus(Succ(Zero), Zero)).
```

where *Plus* is a function is flattened to

Plus(Succ(Zero), Zero, a) & IsOne(a).

where *a* is the extra argument and *Plus* is no longer a function, but is now a predicate.

The flattening of functional terms is done recursively, in case one of the function's arguments is itself a function.

build_constraints and several auxiliary predicates of varying arities accomplish constraint building. **build_constraints** extracts the first constraint from a list of constraints and breaks the variable/function constraint pair into separate arguments which it hands to **build_constraints_aux. build_constraints** then calls itself recursively on any remaining constraints in the list.

build_constraints_aux breaks the evaluable function structure into a functor name and arguments and passes these components to **build_constraints_aux** of a higher arity. In the case of pre-defined binary Gödel functions (e.g., +, -; ++ for strings; etc.), there is a hard-coded **build_constraints_aux** clause for it. All of these clauses are alike. **replace_evaluatable** is called on each of the arguments to detect nested functions. Both evaluated arguments are then built. The new argument, the variable obtained from the constraint pair, is also built. These three arguments form the argument list for the Prolog clause being created. The new Prolog functor name is hard-coded in the clause (e.g., *plus* for +, *minus* for -, etc.).

If **replace_evaluatable** detected any nested functions and returned them as constraints, **build_constraints** is recursively called to build them. The original call to **build_constraints** sequences the clause it creates with any created recursively to form the final clause.

User-defined evaluable FUNCTIONs are handled by a new **build_constraints_aux** clause. Its operation is very similar to the already-existing clauses. In contrast to those clauses, however, the new clause does not know the arity of the function it will be processing, so it calls **build_term** on an argument list, rather than on each of two arguments separately. Also, the predicate **make_flat_name** is called to create a new Prolog predicate name which reflects the increased arity of the former function.

User-defined functions defined as CONSTANT are also handled by a new **build_constraints_aux** clause. Since a constant function has no arguments, the extra

constraint variable is built and becomes the sole argument to the built constraint. The arity of the constant function is incremented to one.

6.6 Making

Making a multi-module program necessitates access not only to the rewrite rules defined within the module being compiled, but also to the rules in modules imported by the module being compiled, since local predicates may have imported functions among their arguments. The rule list contains the names of all needed rules, both local and imported. Loading the rule list file at the beginning of the make procedure allows all modules to avail themselves of the rule list.

A program consisting of multiple modules is created using the Gödel *make,* rather than the Gödel *compile* command. This invokes the predicate **make_program**. In standard Gödel, this predicate breaks the code tree out of the program and hands it, along with the whole program structure, to **make_program_aux.** **make_program_aux** parallels the actions of **compile_program** on the head of the code tree, extracting the module language and module definition from the program, and then calling **compile_program_aux** to compile the module. **make_program_aux** subsequently calls itself recursively on the left and right sub-trees of the code in order to compile the code of the imported modules.

As mentioned above, the problem faced in extended Gödel when making a multiple module program is one of having access to a list of all rules. This is done by loading the rule list, which does contain all the needed rules, at the beginning of the make process and passing it to **make_program_aux**, which passes it to the module compile process via **compile_program_aux**, and to subsequent modules via recursive calls to itself.

6.7 Loading

The command-processing sequence of Gödel is a loop of reading, parsing, interpreting and executing the command entered, then returning to the top of the loop for the next command. In particular, the command-processing sequence must keep track of which program, if any, is loaded, and which symbol table (language), if any, is being accessed. These arguments are passed down through the entire sequence of command-processing predicates.

In order to enable the detection of functions during the compilation of goals under extended Gödel, the rule list for an extended Gödel program must be loaded at the same time the program and its language are loaded. Subsequently, the command-processing sequence must keep track of the loaded rule list.

This is accomplished as for the language and the program. **top_loop** is initially called with an empty rule list argument. It will be filled by an eventual *load*

command, as explained below. **top_loop** calls **next_command**, which calls **process_command**, which calls **build_command**, which calls **command_execution**, which finally calls the predicate which will execute the desired command. The rule list is passed down through each predicate. If the command does not involve loading a new module, the rule list is passed unchanged all the way back up to **next_command**. If the command involves loading, the *load* procedure will modify the rule list. The modified rule list will be returned up to **next_command**, which will re-start the command-processing sequence with a call to **top_loop** using the modified rule list.

load_cmd and two auxiliary predicates are responsible for loading. **load_cmd** passes the parsed program to the initial invocation of **load_cmd_aux.** This predicate loads the language, Prolog, and compiled Prolog files into memory. It also builds a list of imported modules which will be recursively loaded using **load_cmd_aux2**, which calls **load_cmd_aux** to load the first of a list of imported modules, and then calls itself recursively to load the rest of the list of modules. The extended Gödel **load-_cmd** predicate also loads the rule list file into memory for subsequent use by goal evaluation.

7 Conclusion

7.1 Feasibility of functional-logic programming language

This thesis demonstrates the feasibility of creating an integrated functional-logic programming language. The language is usable. Given the slightness of the syntax change to accommodate functions, a knowledgeable Gödel programmer can learn in it minimal time. The addition of functional compilation to the Gödel compiler was accomplished in a reasonable amount of time.

7.2 Narrowing

Leftmost-innermost narrowing is a simplistic narrowing implementation, and deficient in comparison to needed narrowing. Since it converts functions into predicates, the code it generates is no different than that which would be generated were the functions written as standard Gödel predicates. It precludes the use of lazy evaluation, which, as previously shown, allows the execution of many programs which do not terminate when arguments are eagerly evaluated. The main benefit of this functional-logic implementation is the expressiveness gained from the use of functional arguments.

Nevertheless, this implementation creates a code framework in which to create a functional-logic Gödel which employs needed narrowing. The isolation of rules and the detection of function calls, as well as the creation of predicates to implement

narrowing, have all been done. While it remains to be seen how much help this will be to the implementor of needed narrowing, it certainly should serve as a guide.

7.3 Compiler Design

Function Evaluation

The method for function evaluation is a generalization of the method already used by Gödel for its pre-defined functions. It is a straightforward implementation of leftmost-innermost narrowing. No other method was considered.

Rule translation

The **tau** predicate is a concise translation algorithm to flatten rules into predicates. Its implementation is terse, yet not intuitive nor easy to comprehend. On the other hand, given the flattening technique we are using to narrow functional arguments, this type of translation is required for rules. A different narrowing implementation might avoid the need for rule translation as done in this thesis.

Function detection

As stated before, passing a list of rules throughout the code is a primitive yet simple and effective way of facilitating function detection. That the parser furnishes a complete list of local and imported rules only increases its simplicity.

There are alternative methods of function detection. The most obvious would be to declare user-defined functions differently from constructors; i.e., not in the FUNCTION category. (Or better yet, re-name CONSTRUCTOR category items, move standard Gödel FUNCTION items to the CONSTRUCTOR category, and reserve the FUNCTION category for functions.) From a programming language point of view, this makes eminent sense. It would obviate the need for function detection by the compiler. However, we deemed it desirable to make as few changes as possible to the standard Gödel syntax. (For further discussion on this syntax change, see [JV].)

Another alternative would be to flag evaluable functions in the language file. This would avoid creation of the rule list file. But it would entail altering the structure of the language, which has already been created by code generation time. Nor does this address the question of how to use the language file for detection purposes. The rule information contained therein would still have to be passed throughout the code.

7.4 Versions

Version 1

Two versions of the extended Gödel parser and compiler were implemented. The version described in this paper is the second implementation. The ways in which the first version differs are described below.

In the first version, the parsed structure of a rule differs from that of a predicate. More precisely, the two structures are analogous, but a rule definition structure is labeled a *RuleDef,* whereas a predicate definition structure is labeled a *PredDef.* As a result, several parsed structures are different:

Code tree

```
AVLTrees.Node.F5(  left tree -- predicates and rules,
                   predicate or rule name in string form,
                   list of predicate or rule definitions,
                   balance state of tree,
                   right tree -- predicates and rules
                 )
```

In extended Gödel version 1, a node containing a list of rewrite rules defines one function only.

Rule Definition

ProgDefs.RuleDef.F4(arity, definition list, import delays, export delays)

The rule definition exists only in version 1. Other than its name, it is identical in structure to a predicate definition.

Rule

MetaDefs.=>.F2(left hand side, right hand side)

The version 1 compiler contains a rule formula, *MetaDefs.=>.F2.* The two arguments represent the left and right hand sides of a rule. The structure of the rule and predicate clauses are almost identical, the only difference being that predicate heads are atoms identified as *MetaDefs.Atom.F2,* whereas the left hand side of a rule is labeled a term, *MetaDefs.Term.F2.* The two-part left side structure consists of the rule name and a list of terms representing its arguments.

The version 1 parser does not create a rule list file. The version 1 compiler saves the local module rule code by writing it to a rule code file, *<module_name>.rc.* This is done after the parsed code has been separated into statement code and rule code. An imported module's rule code may be accessed by reading in its rule code file. The rule code is used instead of the rule list to detect functions. A new predicate, **function_has_rules**, is called by **evaluatable_functor** to search through the rule code for a rule matching the term being tested for evaluability.

In version 1, **r_compile_module** hands **compile_rules** a node containing a list of *RuleDef,* not *PredDef,* structures. The series of predicate calls down to **compile_rule** is as for version 2. However, the structure passed to **compile_rule** is a single rule structure, declared as *MetaDefs.=>.F2.* No extraction of the rule from

the head arguments as described for version 2 need take place. This rule structure is passed by **compile_rule** to **tau**.

compile_rule's call to **build_constraints** to build pre-defined functions does not exist due to a version 1 parsing problem with numerical types. The numerical types *Integer*, *Rational*, and *Float* are the types for which Gödel pre-defined functions exist; i.e., the types for which constraints exist. In version 1, they are all parsed as type *Num*, for reasons which remain a mystery to this day. No operations (constraints) are defined for type *Num*. Consequently, no numerical operations can be used in rules in version 1.

As stated before, making a multi-module program necessitates access not only to the rewrite rules defined within the module being compiled, but also to those in modules imported by the module being compiled, since local predicates may have imported functions among their arguments. The version 1 solution is to join the rule code of the module undergoing compilation with that of modules which have already been compiled during the make and pass that throughout the make process.

This is done by passing the rule code of already-compiled modules into each invocation of **make_program_aux**, and by adding the local module's rule code to it for subsequent recursive invocations. **make_program** calls **make_program_aux** initially with an empty tree for the rule code. **make_program_aux**, like **compile_program**, calls **separate_statements_and_rules** to divide the code into its two parts. The newly-separated rule code is joined with the existing rule code tree (which is empty during the first pass through **make_program_aux**). Both the local

rule code and the combined rule code are passed to **compile_program_aux**, and through it, to the series of predicates responsible for statement and rule compilation. Recursive calls to **make_program_aux** will be passed the combined rule code, and join it with the rule code of the next module to be compiled, and so on.

A problem created by this solution is that a multi-module program must be *made*; its component modules cannot be separately compiled. In the latter case, modules will not have access to the rule code of imported modules. Detection of imported functions will fail. This is a bug.

At the time of goal compilation, the rule code retained during program compilation is no longer in memory. It must be re-obtained during the load process or goals containing functional arguments will be incorrectly compiled. In version 1, the rule code is obtained by reading in all rule code files from the local and imported modules. An empty tree for the rule code is passed to the initial invocation of **load_cmd_aux. load_cmd_aux** reads into memory the rule code of the module being loaded, and then joins this rule code with the rule code passed into this predicate, which represents the already-read rule code. It calls **load_cmd_aux2** and passes the augmented rule code to the latter predicate, which in turn adds to the loaded rule code during its execution and returns the again-augmented rule code to its caller.

Version 1 does not accept CONSTANT functions.

Comparison between versions

In both versions, rule code and statement code are separated at the top level of the compiler. While this seemed an obvious step in version 1, since rules and predicates are different structures, it was not a necessity in version 2, since rules are parsed as predicates. Rules could conceivably be sent down the same compilation path as predicates. However, since rules require extra processing before they can be compiled, it is desirable to separate them from standard predicates before compilation. Several advantages are reaped from this decision. First, the rule-processing code created in version 1 can be re-used in version 2 with slight modification. Second, the standard Gödel predicate-processing code may be used unchanged in both version 1 and version 2.

Separating the rule and statement code is comparable in version 1 and version 2. Version 1 keys on the *RuleDef* structure to distinguish a rule, while version 2 keys on the predicate name =>.

The fact that rules are parsed as predicates in version 2 makes dissecting the rule structure prior to building somewhat more difficult than in version 1. In addition, all rules in version 2 are mingled in one *PredDef* structure. In version 1, each *RuleDef* contains all rules for one function only. This does not have an effect on this compiler, but may have an impact on future implementations.

Version 2 function detection employs the list of rule names, which is a much smaller structure than the parsed rule code which is used in version 1. This undoubtedly

saves memory. Detection may be faster with the smaller version 2 list, although this has not been tested.

The most significant advantage of version 2 over version 1 is the existence of the rule list file, which contains the names of all local and imported rules, prior to compilation. Loading the rule list from the rule list file is a one-step process. Obtaining the rule code for version 1 module making and loading is a much more complex process. In addition, there exists the version 1 *make* bug when compiling modules separately, as mentioned previously.

In addition, version 2 permits the definition of CONSTANT functions, whereas version 1 does not.

8 Other functional-logic programming languages

Two other currently existing functional-logic languages are briefly examined as a contrast to extended Gödel. The languages are ALF and K-LEAF (Kernel-LEAF).

8.1 ALF

ALF (Algebraic Logic Functional programming language) [ALF] combines a clause-based logic programming language with functions. Its overall design is in many ways similar to our implementation of extended Gödel. It employs leftmost-innermost narrowing to evaluate functional expressions. It attempts to simplify terms as much as possible through rewriting before narrowing is applied. Like the Prolog system underlying Gödel, it uses resolution to solve goals. Also like Gödel and the underlying Prolog, it uses a backtracking strategy to evaluate queries.

ALF uses a module system, like Gödel, which allows both the importation of other modules and the exportation of the local module. ALF has a renaming feature, not found in Gödel, which allows imported objects to be renamed in order to prevent clashes with local object names. For example, if importing a function *f* from module *m* when a function *f* already exists locally, the declaration

```
use m with fm for f.
```

associates the name *fm* with the function *f* imported from module *m*.

The program structure of ALF is slightly different from Gödel in that the goals to be proved are included in the main module, as opposed to the Gödel interactive method. The execution methods also differ. ALF is not based on Prolog. Rather, it compiles programs into instructions for an abstract machine based on the Warren Abstract Machine which is currently implemented by an emulator written in C.

ALF avoids the major syntactic pitfall encountered when extending Gödel; that is, the conflict between constructors and functions. ALF constructors are declared along with the type. Functions are declared separately. For example,

```
datatype nat = { 0 ; s(nat) }.
func + : nat, nat -> nat infixleft 500.
```

defines two constructors of type *nat* and one operation on *nat*.

Like Gödel, ALF supports polymorphic parameters. For example, a type *stack* can be defined to take elements of any sort.

Semantic restrictions placed on ALF evince some similarities and some differences with Gödel. Rewrite rules, as in Gödel, must be confluent and terminating. Unlike Gödel, ALF does not allow functional arguments in the head of a predicate. Like Gödel, ALF permits extra-variables (variables appearing in the right side or condition but not the left side of a rule), but disregards the confluence of the system if they exist.

In general, ALF's computation method is like this implementation of extended Gödel. But ALF goes a bit further by trying to evaluate terms which include partial functions (functions not reducible on all ground terms). Such functions are evaluated using *innermost reflection.* This is essentially a lazy evaluation method, where evaluation of the partial function is avoided unless necessary to the evaluation of the whole term.

ALF also permits specifying that certain equations should be used only for narrowing or only for rewriting. For example, the declaration

```
N * 0 = 0.
0 * N = 0 onlyrewrite.
```

allows for the computation of terms where *N* is unevaluable, while avoiding re-computation of the same value in the case that a term can unify with both rewrite rules.

8.2 K-LEAF

The salient features of the K-LEAF [KLEAF] logic and functional programming language are: 1) the use of flattening rather than narrowing; 2) its ability to handle non-terminating functions; 3) its acceptance of weakly orthogonal (or weakly non-ambiguous) rewrite systems.

Its computation technique is to flatten K-LEAF programs into an intermediate language called Flat-LEAF, and then apply resolution to the resulting Flat-LEAF program. The concept of flattening and resolving is similar to that used in this implementation of extended Gödel, save the lack of an explicit intermediate language in Gödel.

Where K-LEAF's computation technique differs from Gödel is in its handling of partially-defined functions and non-strict functions, those functions which terminate on input from non-terminating functions. In order to handle such functions, K-LEAF

uses an outermost computation method, similar to lazy evaluation of functions. K-LEAF first attempts to resolve goals essential for computation. It delays evaluation of non-terminating functions until absolutely necessary.

For example, a function

```
nats(x) = cons(x, nats(s(x)))
```

which generates a list of the natural numbers beginning from *x* is a non-terminating function. The function

```
first(cons(x, l)) = x
```

is defined as a non-strict function which returns the first element of a list. To solve the goal

first(nats(2))

K-LEAF first tries to evaluate the outermost function, *first.* To do so, it must evaluate *nats,* yielding

first(cons(2, nats(3))).

K-LEAF now attempts once again to compute the value of *first,* and succeeds with the value *2.* At this point, the overall computation is finished, even though the evaluation of *nats* has barely begun.

K-LEAF has some of the usual restrictions on function definitions--left linearity and constructor-based functions. In addition, it allows for *weak non-ambiguity* in its rewrite rules. A term may unify with more than one rule left side so long as confluence is maintained.

That is, two rules may have similar left-side structures such that applying some substitution to both left sides unifies them. The resulting apparent ambiguity is not a

problem if applying the substitution to the rules' right sides (known as a *critical pair*) makes them equal. The existence of such *trivial* critical pairs in a rewrite system renders it weakly orthogonal.

For example, the rewrite system

$$N + 0 = N$$
$$0 + N = N$$

is weakly non-ambiguous, since $0 + 0$ unifies with both rules' left sides, but rewrites to the same value regardless of which rule is applied.

K-LEAF, like ALF, is implemented on the Warren Abstract Machine.

Bibliography

[AEH] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proceedings 21st ACM Symposium on Principles of Programming Languages,* pages 268-279, Portland, OR, 1994.

[AFM] S. Antoy, P. Forcher, and M. Molfino. Specification-based code generation. In *Proceedings of the 23rd Annual Hawaii International Conference on Systems Sciences,* pages 165-173, Kailua-Kona, 1990.

[ALF] M. Hanus and A. Schwab. ALF User's Manual. F. B. Informatik, University of Dortmund, 1991.

[ASV] S. Antoy, D. Shapiro, and J. Vorvick, Gödel with user-defined evaluable functions. *Visions for the Future of Logic Programming,* pages 37-46, Portland, OR, December 1995.

[Antoy] S. Antoy. Lazy Evaluation in Logic. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming,* 3rd International Symposium Proceedings, pages 371-382, Passau, Germany, August, 1991.

[Art] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 2nd edition, 1994.

[Bowers] A. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, University of Bristol, Dept. of Computer Science, 1992.

[Curry] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: a truly functional logic language. *Visions for the Future of Logic Programming,* pages 95-107, Portland, OR, December 1995.

[DJ] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics,* chapter 6, pages 243-320. North Holland, Amsterdam, 1990.

[DP] N. Dershowitz and D. Plaisted. Equational programming. In J. Hayes, D. Mitchie, and J. Richards, editors, *Machine Intelligence 11,* chapter 2, pages 21-56. Clarendon Press, Oxford, 1988.

[Escher] J. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, University of Bristol, Dept. of Computer Science, 1995.

[Gödel] P. Hill and J. Lloyd. *The Gödel Programming Language.* MIT Press, 1993.

[Hanus] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming,* 19&20:583-628, 1994.

[KLEAF] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language, *The Journal of Computer and System Sciences,* 42:139-185, 1991.

[Klop] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II,* pages 1-112. Oxford University Press, 1992.

[SMI] T. Suzuki, A. Middledorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *RTA '95,* pages 179-193, 1995. *LNCS* 914.

[vEY] M. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming,* 4:265-288, 1987.

Appendices

Appendix A

Differences between version 1 and version 2

Issue	Version 1	Version 2
code	contains *RuleDefs* and *PredDefs*	contains *PredDefs* only
basis for rule and predicate differentiation	*RuleDef* structure	*PredDef* name =>
where rule and predicate separation occurs	compiler top level	compiler top level
predicate structure	<-(head, body)	<-(head, body)
rule structure	=>(lhs term, rhs term)	<-(headAtom(=>,[lhs,rhs]), condition)
rule file	*.rc* contains parsed rule code--created by compiler	*.ef* contains list of rule name--created by parser
detector	function_has_rules(f, RuleCode)	member(f, RuleList)
rule structure passed to **tau**	=>(lhs, rhs)	[lhs, rhs]
tau's handling of pre-defined functions	not applicable	handled as constructors. caller must call **build_constraints**
function detection in make	join rule code module-by-module. Doesn't permit single module compile of multi-module program	use rule list from rule list file
function detection in load	read rule code module-by-module	read rule list from rule list file
CONSTANT functions	no	yes

Appendix B

Sample extended Gödel module

This module demonstrates the clarity of expression gained by adding a functional component to Gödel. It contains functional declarations and definitions for many of the Gödel built-in List predicates. Most of these predicates are more naturally conceived of as functions, as shown here.

Where a functional implementation is not possible, the reason is noted.

```
EXPORT          Mylists.

IMPORT          Integers.

CONSTRUCTOR     MyList/1.

CONSTANT  MyNil : MyList(a).

% Constructor Cons
FUNCTION  MyCons : a * MyList(a) -> MyList(a).

% Member cannot be implemented as a function due to lack of Boolean type.

% Permutation implemented using functions
PREDICATE MyPermutation : MyList(a) * MyList(a).

FUNCTION  MyAppend : MyList(a) * MyList(a) -> MyList(a).

% Delete one occurrence of the first argument from the second argument.
FUNCTION  MyDelete : a * MyList(a) -> MyList(a).

FUNCTION  MyReverse : MyList(a) -> MyList(a).
```

```
% Return the first N elements of a list.
FUNCTION   MyPrefix : MyList(a) * Integer -> MyList(a).

% Return the last N elements of a list.
FUNCTION   MySuffix : MyList(a) * Integer -> MyList(a).

% FUNCTION MyLength : MyList(a) -> Integer
% Length function cannot be implemented because it is not transparent
% [Gödel, p. 21]; i.e., every parameter in the domain types must appear in the
% range type. 'a' does not appear in range.

% Sorted cannot be implemented as a function due to lack of Boolean type.

FUNCTION   MySort : MyList(Integer) -> MyList(Integer).

FUNCTION   MyMerge : MyList(Integer) * MyList(Integer) -> MyList(Integer).

LOCAL            MyLists.

% Permutation predicate implemented with functions
MyPermutation(MyNil, MyNil).
MyPermutation(xs, MyCons(z, zs)) <- MyPermutation(MyDelete(z, xs), zs).

% Append function
MyAppend(MyNil, ys) => ys.
MyAppend(MyCons(x,xs), ys) => MyCons(x, MyAppend(xs, ys)).

% Delete function
MyDelete(_, MyNil) => MyNil.
MyDelete(x, MyCons(x, xs)) => xs.
MyDelete(x, MyCons(y, ys)) => MyCons(y, MyDelete(x, ys)).

% Reverse function
MyReverse(MyNil) => MyNil.
MyReverse(MyCons(x, xs)) => MyRev(MyCons(x, xs), MyNil).

% A niftier reverse implementation
FUNCTION   MyRev : MyList(a) * MyList(a) -> MyList(a).
MyRev(MyNil, ys) => ys.
MyRev(MyCons(x, xs), ys) => MyRev(xs, MyCons(x, ys)).

% Prefix function
```

```
MyPrefix(MyNil, _) => MyNil.
MyPrefix(_,0) => MyNil.
MyPrefix(MyCons(x, xs), n) => MyCons(x, MyPrefix(xs, n - 1)).

% Suffix function
% Inefficient?, but effortless implementation
MySuffix(xs, n) => MyReverse(MyPrefix(MyReverse(xs), n)).

% Sort function
MySort(MyNil) => MyNil.
MySort(MyCons(x, xs)) => MyInsert(x, MySort(xs)).

% Insert function
FUNCTION  MyInsert : Integer * MyList(Integer) -> MyList(Integer).
MyInsert(x, MyNil) => MyCons(x, MyNil).
MyInsert(x, MyCons(y, ys)) => MyCons(x, MyCons(y, ys)) <- x =< y.
MyInsert(x, MyCons(y, ys)) => MyCons(y, MyInsert(x, ys)) <- x > y.

% Merge function
MyMerge(xs, MyNil) => xs.
MyMerge(MyNil, ys) => ys.
MyMerge(MyCons(x, xs), MyCons(y, ys)) =>
        MyCons(x, MyMerge(xs, MyCons(y, ys))) <- x =< y.
MyMerge(MyCons(x, xs), MyCons(y, ys)) =>
        MyCons(y, MyMerge(MyCons(x, xs), ys)) <- x > y.
```

Appendix C

Predicate Call Sequences

The following diagrams illustrate the series of predicate calls and data flow of the major sections of the extended Gödel compiler. Not all predicates in each sequence are shown. Nor are all arguments of each predicate included. The intent of the diagrams is to facilitate understanding, rather than provide completeness.

Key

Struct

ExplodedStruct — Struct and ExplodedStruct are the same structure

Struct

SameStruct — Struct and SameStruct are the same structure

Pred(Arg, Arg,...) — Predicate call showing arguments of interest

Pred

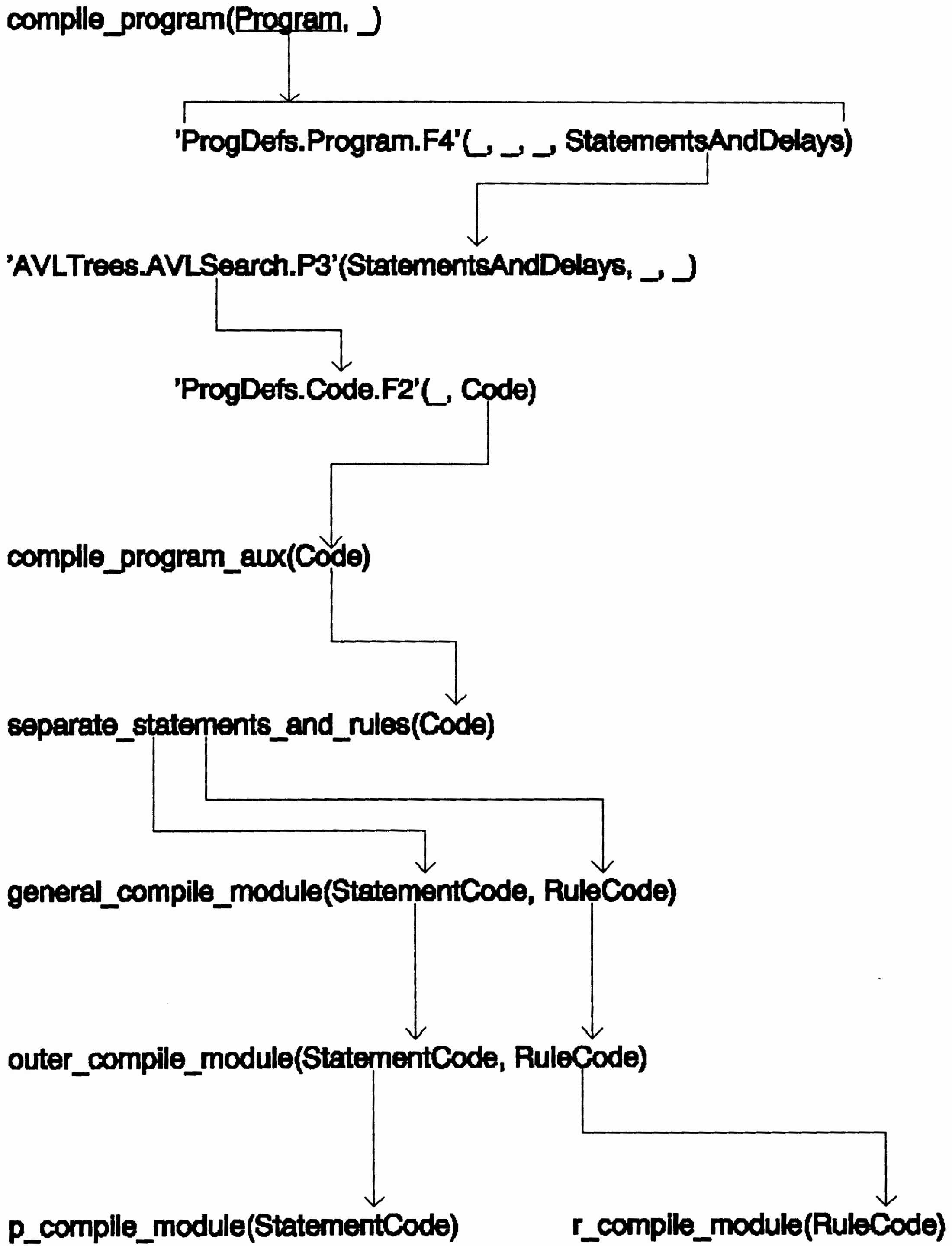
Struct — Call to Pred instantiates Struct

Pred

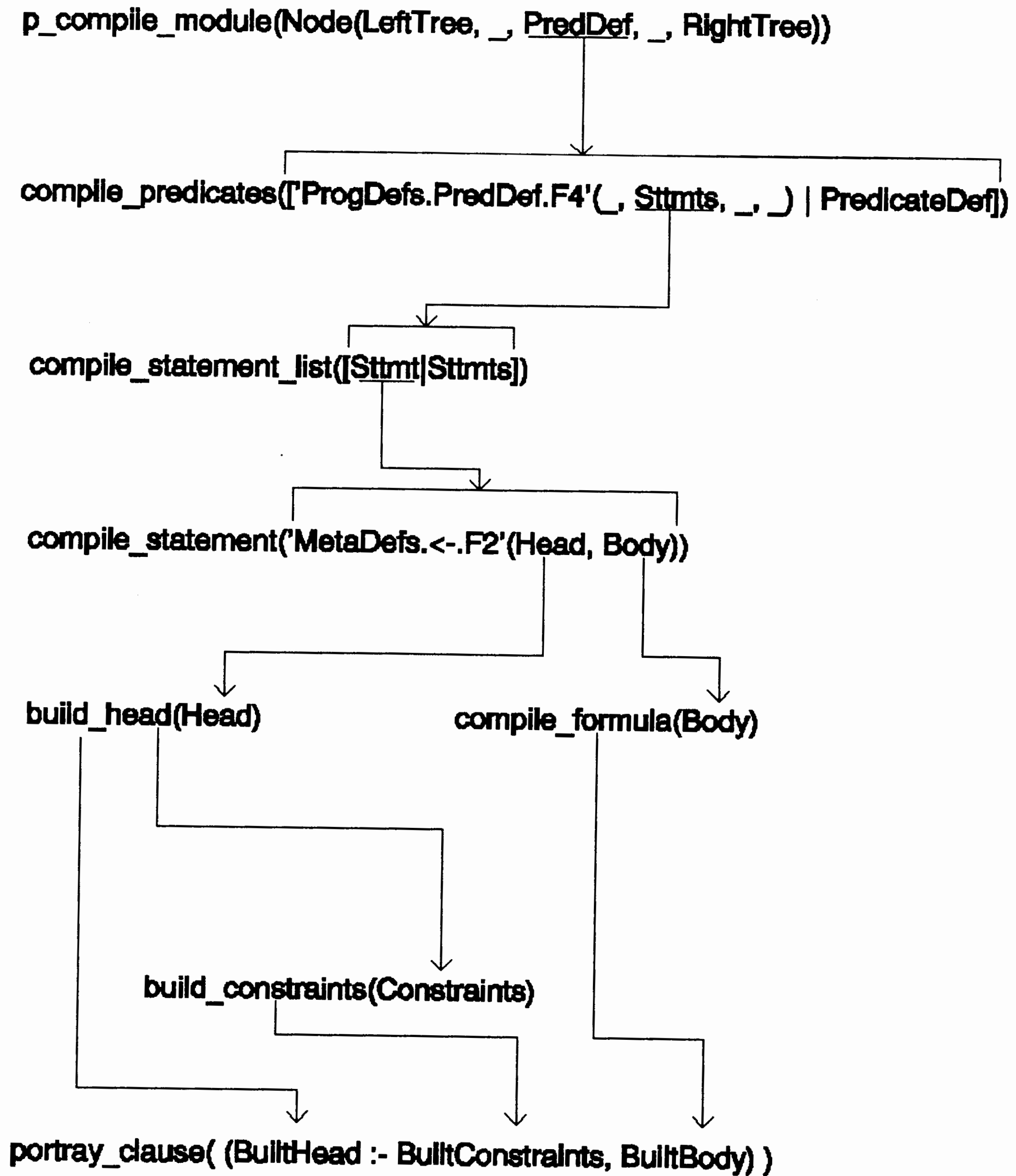
NextPred — Pred calls NextPred

ItalicizedStruct — ItalicizedStruct is uninstantiated at time of call

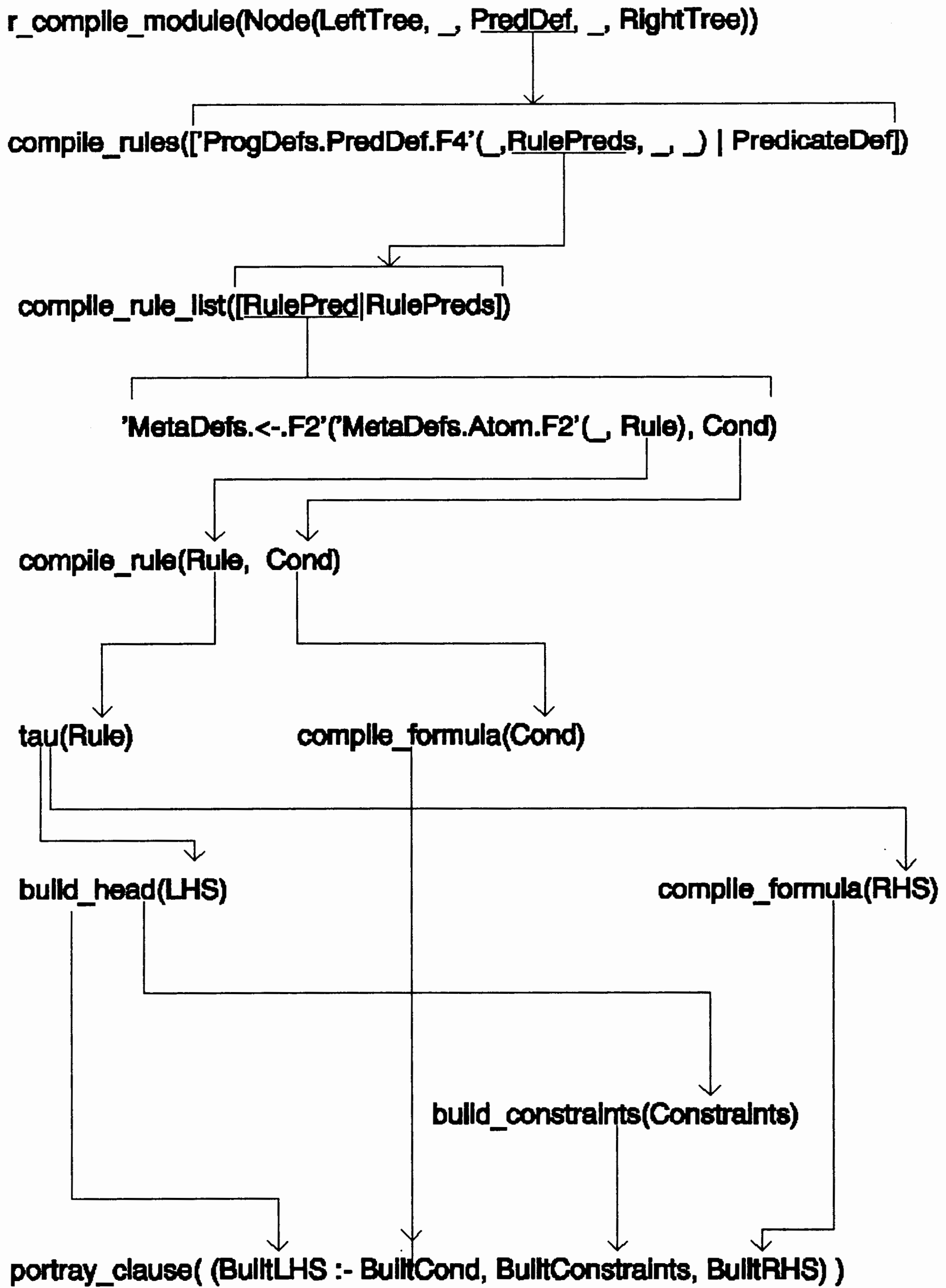
Conditional path



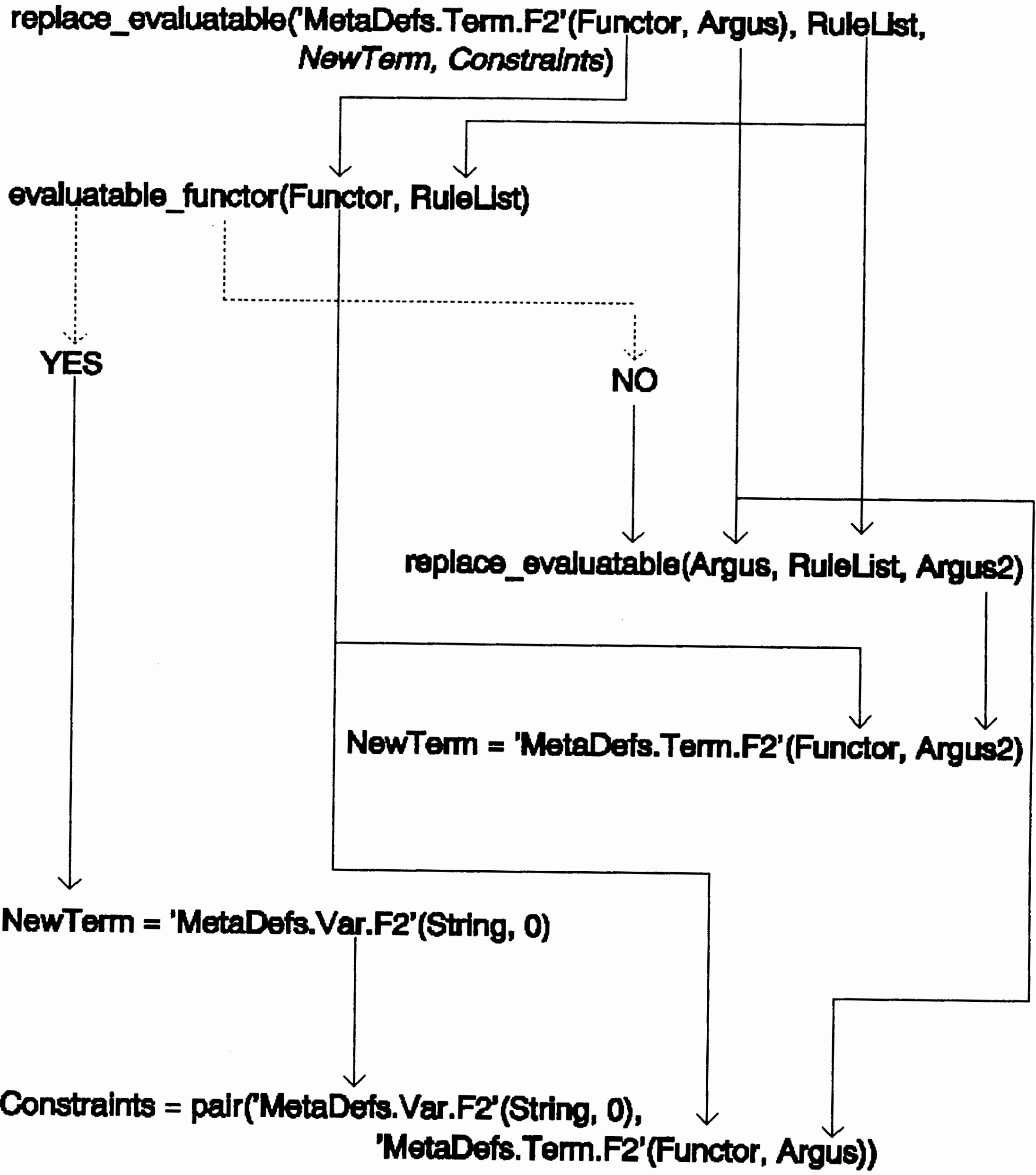
Compiler Invocation



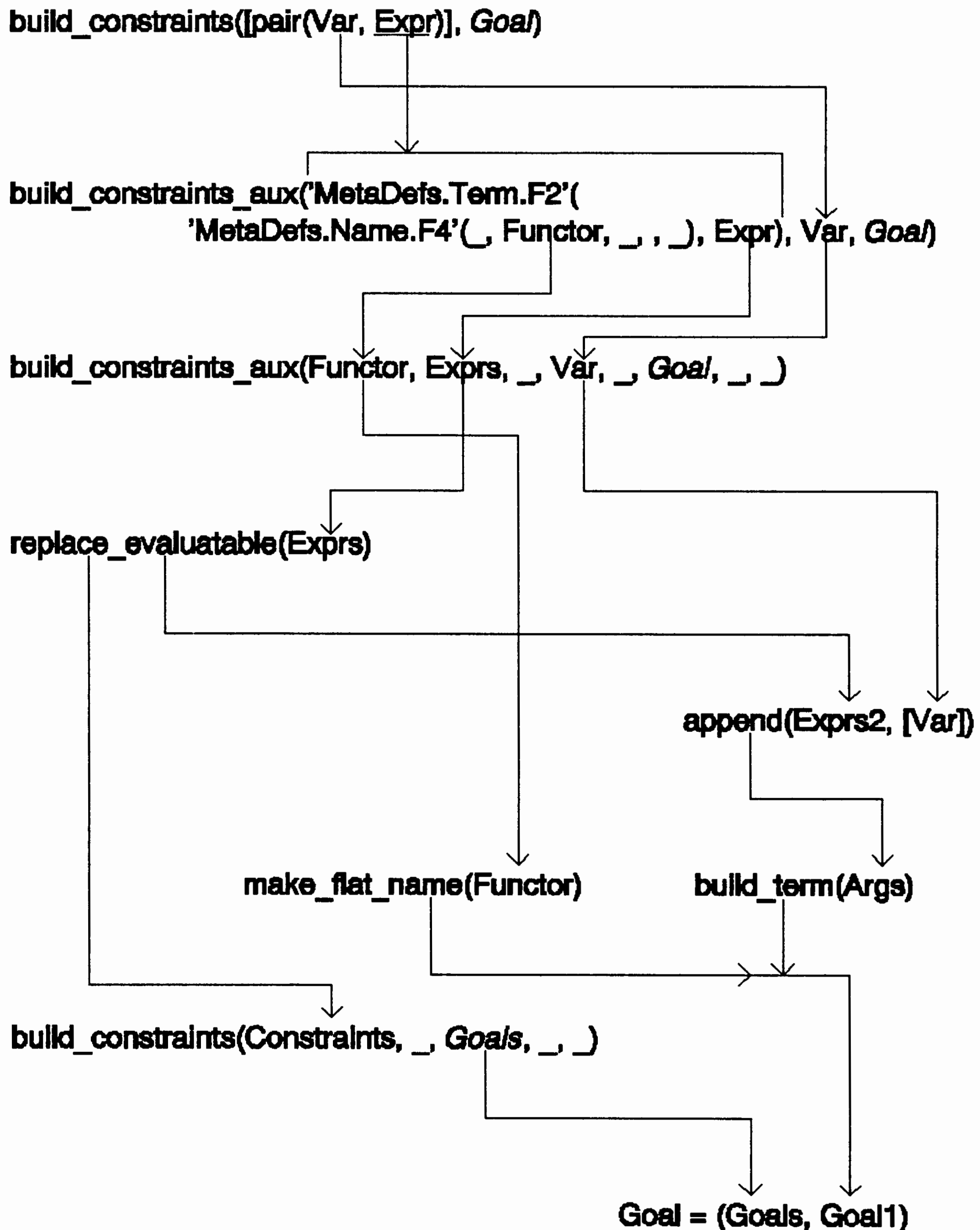
Predicate Compilation



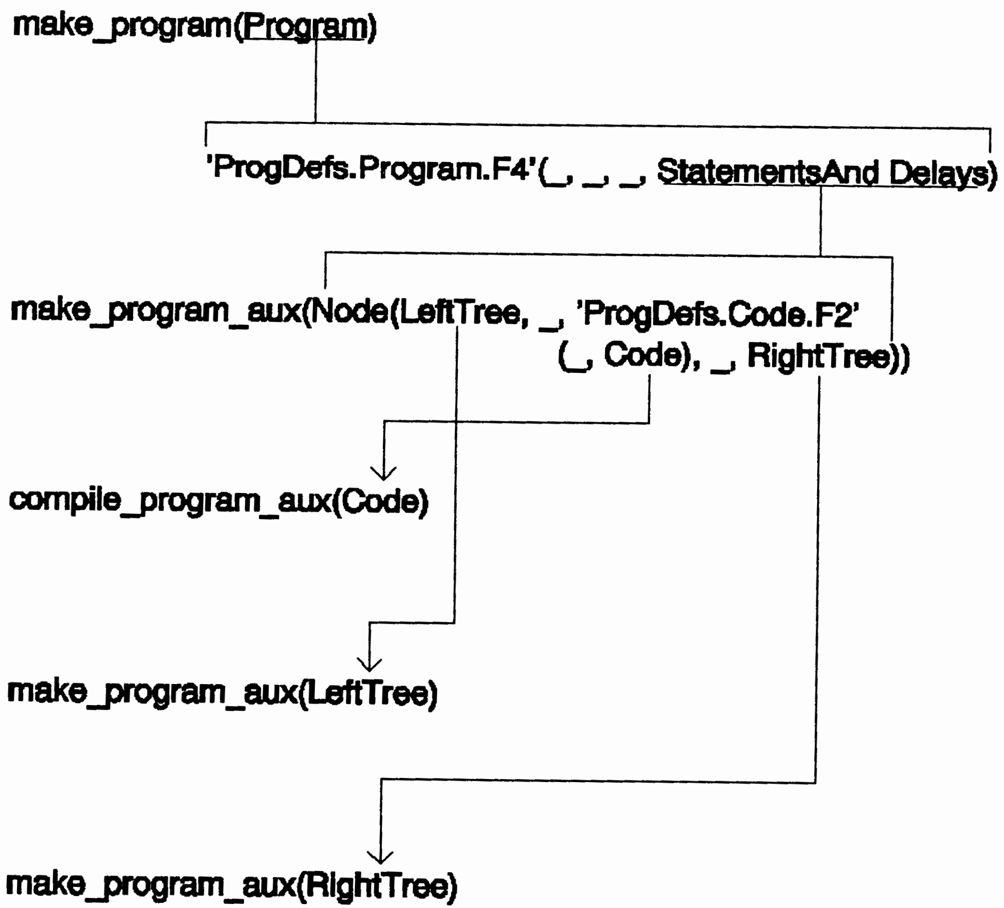
Rule Compilation



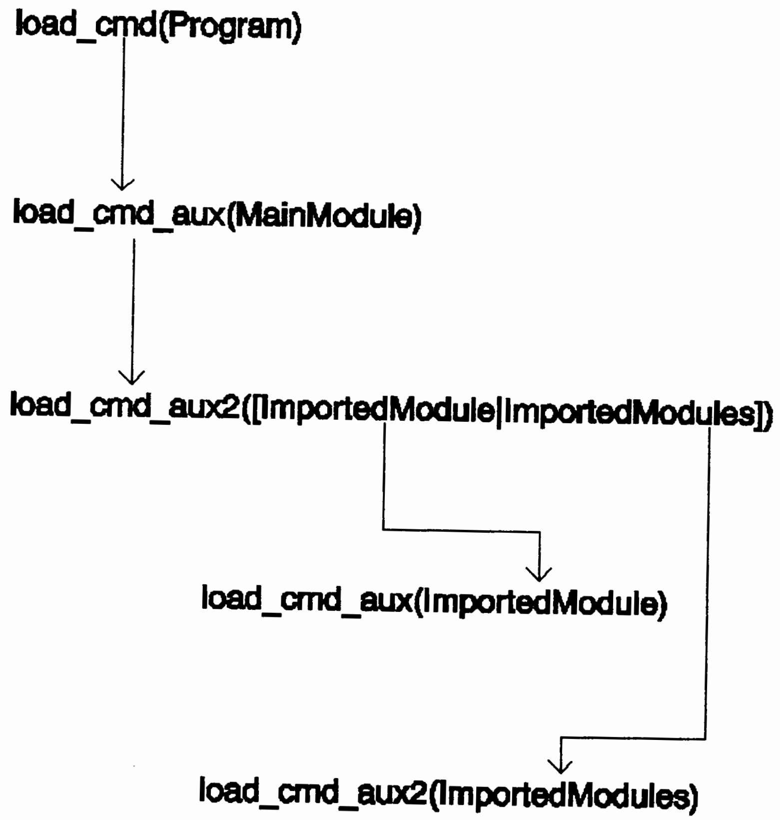
Function Detection



Constraint Building



Make



Load