10-31-1995

Evaluable Functions in the Godel Programming Language: Parsing and Representing Rewrite Rules

Janet Vorvick
Portland State University

THESIS APPROVAL

The abstract and thesis of Janet Vorvick for the Master of Science in Computer Science were presented October 31, 1995, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Sergio Antoy, Chair

[redacted]
James Hein

[redacted]
Andrew Tolmach

[redacted]
Michael A. Driscoll
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL: [redacted]
John McHugh, Chair
Department of Computer Science

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 12 Dec 1995

ABSTRACT

An abstract of the thesis of Janet Vorvick for the Master of Science in Computer Science presented October 31, 1995.

Title: Evaluable Functions in the Gödel Programming Language: Parsing and Representing Rewrite Rules

The integration of a functional component into a logic language extends the expressive power of the language. One logic language which would benefit from such an extension is Gödel, a prototypical language at the leading edge of the research in logic programming. We present a modification of the Gödel parser which enables the parsing of evaluable functions in Gödel. As the first part of an extended Gödel, the parser produces output similar to the output from the original Gödel parser, ensuring that Gödel modules are properly handled by the extended-Gödel parser. Parser output is structured to simplify, as much as possible, the future task of creating an extended compiler implementing evaluation of functions using narrowing.

We describe the structure of the original Gödel parser, the objects produced by it, the modifications made for the implementation of the extended Gödel and the motivation for those modifications. The ultimate goal of this research is production of

a functional component for Gödel which evaluates user-defined functions with *needed* narrowing, a strategy which is sound, complete, and optimal for *inductively sequential* rewrite systems.

EVALUABLE FUNCTIONS IN THE GÖDEL PROGRAMMING LANGUAGE: PARSING AND REPRESENTING REWRITE RULES

by

JANET VORVICK

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

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1995

Contents

Acknowledgments

I'd like to thank Sergio Antoy for his encouragement and help. Also, my thanks to the creators of the Gödel language, especially John Lloyd and Tony Bowers who answered many questions. Thanks to the members of the thesis committee for their participation.

Chapter 1

Background

The logic programming paradigm and the functional programming paradigm share a number of appealing features. Both allow a programmer to formalize a problem and use the formalization as a program from which to compute a solution—which is to say, both are declarative programming paradigms. However, the formal systems underlying the two are different. Functional programming is based on the λ-calculus and logic programming is based on the first-order predicate calculus.

Some appealing features of functional programming are not shared by logic programming. In particular, we are concerned with functions. Though the first-order predicate calculus can accommodate function symbols, they play the role of data constructors. A term (a function symbol together with it arguments) is never equal to some other (distinct) term. There is a lack of consensus about the vocabulary for distinguishing symbols which stand for evaluable functions from those which stand for data constructors. Often the word *function* means *evaluable function*

and *constructor* means *data constructor*. However, in the Gödel language [7], a function is never evaluable. Thus every function symbol is playing the part of a data constructor. Also, the term *constructor* is used in Gödel to mean a symbol which is the name of a type and which does not have arity zero. (Those of arity zero are *bases*.) For this reason, we will use the phrases *evaluable function* and *data constructor* here.

1.1 Evaluable Functions

The introduction of a functional component to the Gödel language allows the user to define evaluable functions as part of the formalization of the problem at hand. In some domains, programmers are accustomed to using functions when formalizing a problem. Consider, for example, computing the circumference of a circle given the length of its radius. In logic programming (in the absence of evaluable functions) even this simple computation requires several statements:

$$\begin{aligned} \text{Circumference(rad, answer)} \quad &\leftarrow \quad \text{Multiply(2, 3.14, ans1)} \\ &\wedge \quad \text{Multiply(ans1, rad, answer)} \end{aligned}$$

when the same ideas expressed with evaluable functions are much more pithy and appealing:

$$\text{Circumference(rad)} = \text{Multiply(2, Multiply(3.14, rad))}.$$

Because programming without evaluable functions presents these

troubles, the Gödel language provides evaluable functions for the built-in types Integers, Rationals, Floats and Sets. It is interesting to note that no evaluable functions are provided for Lists, Strings, Numbers, or Programs. There are a number of other system modules which have predicates only.

Gödel does not provide a mechanism by which the user can define her own evaluable functions. Providing this facility and supporting the evaluation of functions in an efficient manner is the goal of the research of our group under the direction of Sergio Antoy. In addition to some administrative and communications tasks, my part of the research has been the parser.

1.2 Narrowing

The language designer, having decided to include evaluable functions in a logic language, is faced with the questions, "What will be the meaning of a term?" and "How will we compute that meaning?" In Gödel, the meaning of a term is based on an equality theory that includes information about functions [3]. The same is true of the extended Gödel, except that it provides a mechanism for the user to extend the equality theory that is built into Gödel. In the abstract, equality is an equivalence relation which partitions the set of all terms and which places a term in an equivalence class that includes its meaning.

In actuality, it is not possible to generate the equivalence classes for the set of all terms of a program because there are (almost always) an infinite number of terms. Thus the answer to the question "How will we compute the meaning of a term?" differs dramatically from the description of the meaning of a term. In the implementation of Gödel upon which the extension was built, the basic method of dealing with a term whose root is an evaluable function involves processing it using a predicate. Loosely speaking, a function call in a goal is changed by Gödel into a subgoal which finds the 'answer'. An 'answer' is some specific term whose value is the same as the original term's value. Since Gödel is implemented in Prolog, a function call in a Gödel goal is changed into a Prolog predicate call.

The extended Gödel processes a term whose root is an evaluable function in the same manner, except that the subgoal which finds the answer may execute code generated from user-provided rewrite rules. There are sufficient restrictions on the rewrite systems accepted by the extension to ensure that a term has a unique normal form. This means that a function called on ground terms gives just one answer.

Computing the normal form of a term requires some evaluation mechanism. Term rewriting can accomplish function evaluation, but would not serve to integrate the extension into Gödel in a tidy manner because term rewriting cannot handle logic variables.

One can rewrite a term only if it can be made to correspond to one of the rewrite rules by pattern matching. For example, consider this rewrite system defining addition on the natural numbers represented in unary form:

Plus(Zero, x)	$\Rightarrow$	x	rule 1
Plus(Succ(x), y)	$\Rightarrow$	Succ(Plus(x, y))	rule 2

If one wishes to compute an answer to the equation

$$\text{Plus(Succ(Zero), Zero)} = z$$

The left-hand side can be rewritten until the value of z is found. But term rewriting will not compute an answer to the equation

$$\text{Plus}(z, \text{Zero}) = \text{Succ(Zero)}$$

because neither rewrite rule for Plus matches Plus(z, Zero). In fact, rewriting can transform a term that includes variables only in special situations. For example, one can rewrite Plus(Zero, z) only because it happens that the exact value of the second argument is not needed to fire rule 1.

Narrowing [5] allows us to solve Plus(z, Zero) = Succ(Zero) when rewriting does not. Trading rigor for clarity, the narrowing process can be described as unifying z with both Zero and Succ(x) in an attempt to find a value for z. The binding z/Zero leads down a path that ends with Zero = Succ(Zero). This equation is false; z/Zero is not a solution. But the other binding leads to

a valid equation: Succ(Zero) = Succ(Zero). Through narrowing the solution to the equation has been found, and that solution is recorded in the bindings used to reach a valid equation. In this case, the binding is z = Succ(Zero).

Extending Gödel to allow user-defined evaluable functions computed with narrowing improves the language by freeing the programmer from the constraints of a predicates-only style. Programs written with functions are often clearer and more intuitively appealing. The implementation of narrowing in our first version of this extended Gödel is leftmost inner-most narrowing[1]. This was a candidate for an early implementation because it can be accomplished with the well understood technique *flattening* [13]. The ultimate goal of this research is the implementation of *needed* narrowing [1], a strategy which performs only steps that are, in a precise technical sense, *needed* to compute a solution. On ground terms needed narrowing performs what is referred to as lazy evaluation in functional programming. This strategy is sound, complete, and optimal for *inductively sequential* rewrite systems, a class that encompasses the first-order programs of functional programming languages such as *ML* and *Haskell*.

[1] In appendix A some comments on future extensions can be found.

Chapter 2

The Extended Gödel

The differences between the original Gödel Language and the extended Gödel are both syntactic and semantic. The following sections discuss the inclusion of rewrite rules in a module, the kind of rules allowed, and overloading. Then we present an example of an extended-Gödel module.

2.1 The Design of the Language

Some of the guiding ideas for the design of the extended Gödel were these:

- A Gödel module should be an acceptable extended-Gödel module.
- The user should receive warnings rather than errors when she provides rewrite rules that violate some requirement of the extended Gödel.

- The flavor of a logic programming language should be retained.
- The capability for defining evaluable functions should not detract from the usefulness of the Gödel language.

The majority of the new code needed for the parsing of the extended Gödel tests rewrite rules. Language design decisions were required as the plan for testing the rules developed. The attractiveness of restricting the accepted rewrite systems to those which are confluent and which allow us to compute normal forms efficiently motivated the choice of the tests of the rewrite rules.

Tests which produce warnings are the test for left-linearity, the test that the condition of a rule is a conjunction of equations, the test for overlapping, and the tests of the variables in a rule. All of these are explained in section 3.5. The decision to print a warning and compile some programs which do not meet all the requirements was motivated by the fact that some attractive properties of rewrite systems are undecidable. Thus there may be programs that are useful and perform in a reasonable manner which do not meet all the requirements we have established to *guarantee* reasonable behavior. The tests which are classified as errors and cause the compilation to abort are the test that the rewrite system is constructor-based and the test that rewrite rules exist for an evaluable function declared with a system-defined

target type. Overloading errors are caught by the original Gödel parser.

2.2 Syntax Details

The appearance of a module written in the extended Gödel differs from that of a module written in Gödel in only one way: it includes rewrite rules. A Gödel module has a section for declarations followed by a (possible empty) section for statements. A *statement* is simply a clause, though Gödel handles many constructs in the body of a clause which are unknown to a Prolog programmer. Corresponding to these two sections, the extended Gödel has three—one for declarations, one for statements and one for rewrite rules. Since either statements or rewrite rules or both may be absent from a module, a Gödel module will present no problem to the extended Gödel system. It simply sees it as a module which has no user-defined evaluable functions.

As an example, consider this small Gödel module which defines the natural numbers, `Plus` and a test for `Zero`:

```
MODULE      Nat.
BASE        Nat.
CONSTANT    Zero: Nat.
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).
```

A module in the extended Gödel which accomplishes the same tasks includes rewrite rules defining `Plus`. Also, the symbol `Plus` is declared in the `FUNCTION` part of the module, rather than the `PREDICATE` part.

```
MODULE          Nat.
BASE            Nat.
CONSTANT        Zero: Nat.
FUNCTION        Succ: Nat -> Nat;
          Plus: Nat * Nat -> Nat. % DIFFERENT HERE
PREDICATE       IsZero: Nat.
IsZero(Zero).
Plus(Zero, x) => x.                        % HERE
Plus(Succ(x), y) =>  Succ(Plus(x, y)).    % HERE
```

The symbol `=>` is a reserved, binary, infix, overloaded operator used for the definition of evaluable functions.

The rewrite rule `Plus(Zero, x) => x` can be transformed into the clause `Plus (Zero, x, x)` by flattening. Flattening creates predicates from equational specifications. One implementation of the extended Gödel which we considered would consist of a preprocessor which would flatten each rewrite rule to produce a regular Gödel module. Though this approach would have simplicity as an asset, it would preclude the implementation of needed narrowing because the evaluation of terms would always be done

by the mechanism provided by Gödel.

Careful consideration of the above extended-Gödel module may excite some protest that the **FUNCTION** declarations lump together **Succ** and **Plus** which have radically different roles to play in the module. This is true, and is certainly less than ideal. The motivation for this approach will be presented in the discussion of disadvantages of this syntax.

Though the presence of rewrite rules is the most striking syntactic difference between Gödel and the extended Gödel, there is much more that needs to be said about syntax. The details concern the form the rewrite rules may have, and the properties the rewrite system must have. Additional information about the property called inductive sequentiality can be found in appendix B.

2.3 Conditional Rewrite Rules

In the extension, the left and right operands of the reserved symbol => are interpreted as the left- and right-hand sides of a rewrite rule. Conditional rewrite rules are also allowed. They have the form l => r <- c. The condition c is a conjunction of equations such as those that can be found in the body of a clause. If c is false, then the rule is not fired, even when the term being narrowed unifies with l.

As an example of the usefulness of conditional rewrite rules, consider this rewrite system presented by Suzuki, Middeldorp, and Ida in their paper on confluence [12]:

$$\begin{aligned}
&\text{Divide(Zero, Succ}(x)) && \Rightarrow && \text{Pair(Zero, Zero).}\\
&\text{Divide(Succ}(x), \text{Succ}(y)) && \Rightarrow && \text{Pair(Zero, Succ}(x))\\
& && && \leftarrow x < y = \text{true.}\\
&\text{Divide(Succ}(x), \text{Succ}(y)) && \Rightarrow && \text{Pair(Succ}(q), r))\\
& && && \leftarrow x \geq y = \text{true}\\
& && && \;\&\; \text{Divide}(x - y, \text{Succ}(y))\\
& && && = \text{Pair}(q, r).
\end{aligned}$$

Here it is assumed that subtraction is suitably defined on the unary representation of natural numbers. Solving the equation

$$\begin{aligned}
&\text{Divide(Succ(Succ(Zero)), Succ(Zero))}\\
&= \text{Pair(quotient, remainder),}
\end{aligned}$$

in the process of rewriting the term

$$\text{Divide(Succ(Succ(Zero)), Succ(Zero))}$$

it becomes apparent that it matches the left-hand sides of both the second and third rules. But the condition, $x < y$, of the second rule is false, therefore the second rule is not fired. The third rule is fired.

This example of the usefulness of conditional rewrite rules overlooks the restrictions placed on rules in the extended Gödel. The above rules are overlapping, for example, meaning that some left-hand side may match more than one rule. The extended

Gödel sends a warning to the user when a rewrite system is overlapping. The restrictions are needed to attain the long term goal of this research - the implementation of needed narrowing.

2.4 Restrictions on Rules and Systems of Rules

The rewrite rules provided by the user are expected to have the properties that they are left linear, constructor-based [4, 8] and non-overlapping.

A rule is left linear only if its left-hand side contains no repeated variables. A group of rules is constructor-based if every rule has at the root of the term on the left-hand side an evaluable function and has as the subterms of the left-hand side terms containing no evaluable functions. For example, the following two rules violate the above conditions.

```
SetUnion(x, x) => x.
SetUnion(x, SetUnion(y, z))
                    =>
                    SetUnion(SetUnion(x, y), z).
```

The second rule above violates the constructor-based restriction even in isolation, but rules cannot be checked one at a time to ensure that a rewrite system is constructor-based. A pair of rules which are legal in isolation can violate the restriction when taken together. For example,

```
Double(Halve(x)) => x.
Halve(Double(x)) => x.
```

In isolation, `Double(Halve(x))` might be seen to have an evaluable function as its root, if `Double` is an evaluable function. Also, it would have no evaluable functions as subterms if `Halve` is a data constructor. But taken together with `Halve(Double(x)) => x`, it forms a rewrite system that is not constructor-based. `Halve`, like `Double` cannot be both an evaluable function and a data constructor.

A rewrite system is overlapping if some (sub)term matches more than one rewrite rule. For example, the rewrite rules

```
F(Zero) => Zero.
F(Succ(x)) => Zero.
F(Succ(Succ(x))) => Succ(F(x)).
```

is overlapping because `F(Succ(Succ(Zero))` matches both the second and the third rewrite rules—the second with `x = Succ (Zero)` and the third with `x = Zero`. Thus the result of evaluating `F(Succ(Succ(Zero))` is both `Zero` and `Succ(F(Zero))`.

The violation of the requirement for left linearity has been given the status of a warning. Some programs which include rewrite rules that are not left linear are well-behaved. For example, the following program (if it is considered to be the whole program and not just a program fragment) is not problematic.

```
MODULE        F.
IMPORT        Integers.
FUNCTION      F: Integer * Integer -> Integer.
F(x, x) => x.
```

Where the extended-Gödel parser prints a warning concerning a rewrite rule, the code includes a stub for changing this to an error. A later implementation of an evaluation strategy may require tighter restrictions.

2.4.1 Requirements of Conditions

The conditional part of a rewrite rule must be a conjunction of equations. For example, the module

```
BASE          Bool.
CONSTANT      MyTrue, MyFalse: Bool.
FUNCTION      InRange: Integer -> Bool.
InRange(x) => MyTrue  <- x > 10 \/ x < 0.
InRange(x) => MyFalse <- x =< 10 & x >= 0.
```

has a problem. Neither of the conditional parts of the rewrite rules are the conjunction of equations.

Next, consider variables. A condition concerning variables is imposed on the rewrite rules of the extension. In general, in a rewrite system, the variables on the right-hand side of a rule must be a subset of those on the left-hand side. A bit more flexibility is obtained by allowing extra variables in the right side

and/or condition of a rule if they satisfy the confluence criteria established by Suzuki, Middeldorp, and Ida [12].

As a first step in guaranteeing confluence (a property of some rewrite systems that involves multiple rewritings of a single term), the class of term rewriting systems under consideration must be described. Suzuki, Middeldorp, and Ida place conditional term rewriting systems, CTRSs, in four categories. The first requires the variables in the right-hand side of a rule and the variables in the conditional part of a rule to be present in the left-hand side of a rule. So, if the form of a conditional rewrite rule is

$$l \Rightarrow r \Leftarrow s_1 = t_1 \ \& \ s_2 = t_2 \ \& \ \ldots \& \ s_n = t_n,$$

then the variables in r together with the variables in the equations that make up the condition must be a subset of the variables in l. As this is the first category of CTRSs, these are called 1-CTRSs. LPG, discussed below, deals with 1-CTRSs.

A 2-CTRS has only the restriction that the variables in r are a subset of the variables in l. The programming language BABEL, also discussed below, deals with 2-CTRSs but includes others that are not 2-CTRSs. The extended Gödel allows the definition of rewrite systems that are 3-CTRSs. These require that variables in r be present in l or in the conditional part of the rule.

The confluence criteria restrict the variables of the equations that form the condition of a 3-CTRS. The restriction on the vari-

ables that appear in the left-hand side of an equation is this: each variable in some s_i must be in l or in some t_j that comes before s_i. Another way of saying this is that *new* variables are not introduced in any s. Rewrite rules with this property are called *properly oriented.*

Variables in the right-hand side of an equation are restricted, too. The variables in any t must not be found in l or in any part of the condition up to t. This property is called *right-stability.* A further restriction on the right-hand sides of the equations requires each to be a certain kind of term. For the extension it is adequate to say that the right-hand sides must have no evaluable functions in them. This requirement is crucial for ensuring confluence. A relaxed version of this requirement allows an evaluable function in the right-hand side of an equation if the subterm whose root is that evaluable function does not match the left-hand side of any rewrite rule. Basically, these restrictions mean that any t must be in normal form with respect to the non-conditional part of the rewrite system.

2.5 Overloading

Gödel disallows the declaration in a module of distinct symbols with the same name and arity in the same category. For the extended Gödel, this prevents declaring a data constructor which

is indistinguishable from an evaluable function. For example, the following extended-Gödel module has an error, since there are two symbols with the same name and arity in the FUNCTIONS category:

```
MODULE          Nat.
IMPORT          Integers.
BASE            Nat.
CONSTANT        Zero: Nat.
FUNCTION        Succ: Nat -> Nat;
                                % SAME NAME, ARITY
                Succ: Integer -> Integer;
                                % SAME NAME, ARITY
                Interpret: Nat -> Integer.
Interpret(Succ(x)) => Succ(Interpret(x)).
```

However, programmers aren't saved from their ability to write bad programs. The following module has no errors.

```
MODULE          Nat.
BASE            Nat.
CONSTANT        Zero: Nat.
FUNCTION        Succ: Nat -> Nat;       % MANY SUCCs
                Succ: Nat * Nat -> Nat.
PREDICATE       Succ: Nat * Nat * Nat;
                Succ: Nat.
```

The extension preserves the behavior of the Gödel module system as it relates to importing a symbol that has the same name and arity as a symbol declared in the importing module. The symbol in the importing module obscures the imported symbol. Two

imported symbols that clash will cause an error in compilation if the symbol appears in a statement or rule.

2.6 Evaluation of the Syntax

From the standpoint of user-friendliness, the syntax of the extended Gödel has the advantages already mentioned: formalization of a problem can include the familiar functional forms and programs are clearer. Also, rewrite rules can be interspersed with statements, which prevents the programmer from being burdened with the detail of sorting them herself.

From the standpoint of implementation, this syntax has the advantage of being only minimally different from standard Gödel. This is the overwhelming advantage of the syntax we chose. By avoiding new categories of declarations and new syntax for declaring data constructors, we have been able to use the parser that came with the Gödel language for the vast majority of the parsing of the extended language. Though this advantage of the syntax requires only a sentence to state, it cannot be over-emphasized: the syntax of the extended Gödel is only slightly different from the syntax of Gödel.

A disadvantage of the syntax we chose is the combining of declarations of data constructors and evaluable functions. An unsophisticated user may well be confused by the presence of re-

write rules defining one symbol declared in the **FUNCTION** section and the absence of rules defining another. A much tidier method of declaration would group a type declaration with all the data constructors that can form objects of that type. For example, it would be preferable to write

```
MODULE          Nat.
BASE            Nat = Zero | Succ: Nat.
                          % DIFFERENT HERE
FUNCTION        Plus: Nat * Nat -> Nat.
PREDICATE       IsZero: Nat.
IsZero(Zero).
Plus(Zero, x) => x.
Plus(Succ(x), y) =>  Succ(Plus(x, y)).
```

where the **BASE** declaration reads, "a term of type **Nat** can be the nullary symbol **Zero** or the unary symbol **Succ** together with its argument of type **Nat**." This not only solves the problem of declaring data constructors side-by-side with evaluable functions, but also allows us to see at a glance all the data constructors that can make a term of type **Nat**. However, this nicer syntax would be much more demanding to implement. The extended parser would require changes in the predicates that parse **BASE** declarations, changes in the predicates that parse **FUNCTION** declarations, and changes to the symbol table generator. The extra effort to make these changes would contribute nothing toward the goal of implementing needed narrowing. Certainly separating the

declarations of data constructors and evaluable functions would be a good choice for a new language.

Another manner in which the declarations of data constructors could be separated from the declarations of evaluable functions does not have the advantage of showing at a glance a type and all its constructors. A section could be added to the declarations for evaluable functions. With this approach, Gödel modules could still be parsed by the extended Gödel parser. This would be less demanding to implement. Since the design and implementation of a new language is a major undertaking, we capitalize on the features Gödel already has. Our approach has been to contain the effort of producing a functional logic language by extending with a functional component an already implemented logic language.

2.7 A Simple Example

The following are the standard and the extended versions of a program fragment dealing with family relations [10]. In Gödel, following a well-established tradition for this kind of relations, the program fragment is

```
PREDICATE Father, Mother, PaternalGrandFather,
          Parent : People * People.

Father(Joe,Tom).
...
PaternalGrandFather(x,y) <- SOME [z] (Father(x,z)
```

```
                                    & Father(z,y)).
Parent(x,y) <- Father(x,y).
Parent(x,y) <- Mother(x,y).
```

where the type **People** and the predicate **Mother** are suitably defined. In the extension, the same program fragment could be coded as

```
FUNCTION  Father, Mother,
          PaternalGrandFather : People -> People.
PREDICATE Parent : People * People.

Father(Joe) => Tom.
...
PaternalGrandFather(x) => Father(Father(x)).
Parent(x,Father(x)).
Parent(x,Mother(x)).
```

The definition of **Father** makes clear that **Tom** is the father of **Joe** rather than the reverse. The definitions of **PaternalGrandFather** and **Parent** avoid several extraneous variables, a quantifier, an operator, and two clause bodies.

Since the relation between a person and their father is, in fact, a function, it seems a burden to require a programmer to cast it as a predicate by writing **Father(Joe, Tom)**. But in identifying all the grandparents of a person, a predicate is superior since backtracking will find the many answers. A system that is flexible enough to model single-valued relations as functions and multi-valued relations as predicates encourages accurate modeling of a

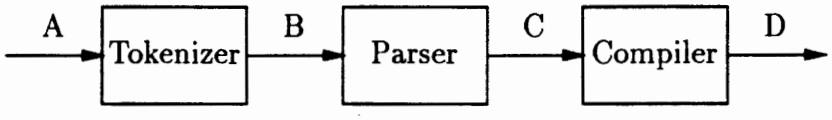
problem.

Chapter 3

Implementation

3.1 The Original Gödel Parser

Modification of the parser to accept evaluable functions was facilitated by the straight-forward structure and careful commenting of the system code. The Gödel parser is written in Prolog. For our purposes, the system code which forms the 1.4 release of Gödel from Bristol can be divided into four parts. First, there are a group of files which define the system-provided Gödel modules (such as Integers and Lists). Second, there are the six files which comprise the parser. Third, a single file contains the compiler. Fourth are some system files not affected by this work. All four parts consist of Prolog code, but some of the code was written in Prolog and some was written in Gödel and compiled to Prolog by machine.



- The input at A is a Gödel module (Gödel code).

- The input at B is a list of tokens.
- The input at C is the representation of the Gödel code.
- The output at D is Prolog code.

The parser begins with a tokenizer which catches errors such as an unexpected end-of-file, incomplete strings, and illegal ASCII characters. Throughout the parser, error messages are processed by means of a call to a generic error-printing predicate. Consequently, error messages can be passed up from lower level predicates to higher level predicates as the list of tokens produced in the first stage makes its way through the parser. For example, if an undeclared predicate name is encountered as the parser is working on a clause, the string "undeclared or illegal symbol" is passed up to the more general module parsing predicate. It calls the error printing predicate and then allows the parsing to continue. In this manner several errors can be found in one compilation instead of aborting the compilation when the first error is found. This error message passing technique is just one example of the tidiness of the parsing code.

The list of tokens produced by the tokenizer is passed to the predicate **parse_module**. There are two separate paths through the parser for groups of tokens. One processes declarations; the other processes statements. The path that parses declarations is itself made up of many paths through the parser. **Module** declarations go through one group of predicates, **Base** declarations

through another, and each of the ten other declarations through their own group of predicates.

Statements are parsed with the help of a symbol table made from the declarations parsed earlier. If no error causes the parsing to abort, the output from the parser is not just confirmation that the module being parsed is well-formed, but also an object representing the module. This object, the internal representation of a module, is written out to a file and is handed to the compiler which generates Prolog code to implement the predicates.

3.2 The Internal Representation of a Module

The details of the internal representation of a module were the main motivation for almost all implementation decisions concerning the parser. Like other parts of the parser, the internal representation has a tidy structure. Objects are wrapped up in logical groups marked with names that hint at their importance and specify the number of objects in the group. Since the internal representation is a Prolog object, it is proper to call these wrappers function symbols. In a sense, these function symbols play the part of data constructors, except that they do not affect an object's type (as Prolog is a typeless language).

Consider, as an example, the function symbol `ProgDefs.Program.F4`. Together with its four arguments, this symbol forms the

internal representation of a program. The choice of the word *program* for this function symbol reflects the fact that Gödel has a module system that allows one module to import another. Thus the internal representation of a program may not be the internal representation of a single module, but rather the internal representation of several modules.

To accomplish the modification of the parser, it was important to distinguish a program from a module. Other names given to objects include language, symbol table, code, constraint, and variable dictionary. It is our hope to spare the reader the details of these names and objects, though some are significant to the explanation of implementation decisions. Most of the objects in the parser have easily understood names, fortunately. There are atoms, predicates, functions, terms, lists, heads, bodies, switches, trees, delays and variables.

The internal representation of a program must provide all the information about the program needed by the compiler. Most important, the representation includes a dictionary of the symbols that make up the language of the program. Since Gödel is strongly typed, has a module system and distinguishes between symbols by the category in which they are declared and their arity, a dictionary entry includes the name of the module in which the symbol was declared, its type, the types of its arguments and its category (for details, see appendix C).

Detailed information concerning the modules that make up a program forms a second part of the internal representation of a program. The name of each module (actually the name of the file in which it was found) is recorded along with an indication that the module is an EXPORT module, a LOCAL module or a CLOSED module. The structure of the importing module is documented so that the dependencies of one module upon another can be found.

The compiler's work is the transforming of Gödel code into Prolog code. Naturally the internal representation of a module needs to include the Gödel code. In the original Gödel, the code is simply the statements that define the behavior of the predicates. In the extended Gödel, rewrite rules are included also. Since the statements and rules have already been through the parser and the type checker, it is expedient to send the compiler a representation of the code. The representation of the code specifies the structure of and the symbols in the statement (or rule). Compilation is facilitated by the grouping together of the pieces of code relating to the behavior of a single predicate of evaluable function.

3.3 Parsing in a Logic Language

Gödel 1.4 was implemented by Anthony Bowers and Jiwei Wang at the University of Bristol. Anthony Bowers designed

the internal representation of a module and Jiwei Wang wrote the parser.

The process of writing a parser in a logic language can be very simple if one is parsing a context-free language. A grammar rule can be changed into a clause in a straightforward manner. In *The Art of Prolog* [11], Sterling and Shapiro give this example: if a context-free grammar includes the rule

sentence → noun_phrase, verb_phrase

then the Prolog program to parse the language of that grammar will include the clause

```
sentence(S) :- append(NP, VP, S), noun_phrase(NP),
                                  verb_phrase(VP).
```

As Gödel is a context-free language, a Gödel parser could be written directly from the grammar describing Gödel. However, this simple method of parser production suffers from a significant inefficiency because of the calls to **append**. A slightly more complicated method of transforming the grammar rules solves this problem using difference-lists to avoid the calls to **append**.

Difference-lists represent a sequence of elements, just as lists do. But difference-lists are purposefully incomplete. A difference-list consists of some elements and a logic variable standing in for the missing part of the sequence. For example, the difference-list that corresponds to the list `[1,2]` is the structure `[1,2|Xs]\Xs`

where `Xs` is the incomplete portion of the difference-list [11]. The benefit of this incompleteness is that the logic variable `Xs` can be instantiated at some suitable time, essentially appending another list to `[1,2]`. To preserve the usefulness of the difference-list, the `Xs` would be instantiated to a difference-list. So, to add the numbers 4 and 5 to the difference-list `[1,2,3|Xs]\Xs`, one would bind `Xs` to `[4,5|Ys]` and form the new difference-list `[1,2,3,4,5|Ys]\Ys`.

Transforming a context-free grammar into clauses using difference-lists is easily done either by hand or by a Prolog program. The group of clauses obtained in this manner is called a definite clause grammar. As Prolog programs, these grammars take advantage of backtracking when an alternative solution is needed.

Backtracking might also come into play if there is ambiguity. In this case, the process of parsing a module is not really functional since the relation between programs and objects representing them is not single valued. As logic programming is suited to calculating relations, one might write a parser that uses backtracking to assemble a list of all the possible parses of the input. If some input results in a list of length zero, it is ill-formed. If it produces a list whose length is greater than one, it is ambiguous. A list of length one would be the desired output.

The Gödel parser is a recursive descent parser that was written using some of the ideas of definite clause grammars. The

grammar describing terms, for example, was manipulated (by hand) into a useful form and used to write the parser. difference-lists were used in some places, and the code for parsing terms uses backtracking to find multiple parse trees for a list of tokens. However, Jiwei Wang was concerned with efficiency. Therefore, he wrote some parts of the parser in a style that is different from the simple definite clause grammars approach. For example, some parts consist of predicates that are intended to succeed exactly once and leave no choice points behind. Also, a stack is used for sorting out the precedence of operations in a term. For details, see appendix D.

On the subject of the efficiency of the parser, Jiwei Wang [14] reports,

> The parser uses the recursive procedure method. By analyzing the Goedel BNF grammar, I found out Goedel grammar can be transformed into a simple recursive pattern. Details can be found in term.pl. This should be true to any mathematical logic based language. The approach turned out to be very efficient. The parser can parse 200 line/second on a Sparc10. Comparing with the 1000 line/sec performance of Sepia's parser written in C, this is quite remarkable.

3.4 The Extended Parser - Implementation Decisions

The extended parser plays the same role in the extended Gödel that the original parser plays in Gödel. It provides an answer to the question, "Is this module well-formed?" and either prints error messages if it isn't or produces an object representing the module if it is. The presence of rewrite rules affects both the Boolean output of the parser and the code-representing output.

3.4.1 Distinguishing Between Evaluable Functions and Data Constructors

We discussed above the negative aspects of declaring both evaluable functions and data constructors in the `FUNCTION` section of a module with regard to program clarity. The implementation of the extension to Gödel requires distinguishing between the two for parsing, compiling and for the execution of goals. Since we chose not to make major changes in the form of declarations, we must resort to examining the rewrite rules included in a module to determine the status of a function symbol. If there is a rewrite rule defining the behavior of a symbol declared in the `FUNCTION` part of a module, it is an evaluable function. If not, it is a data constructor.

The code to identify the evaluable functions of a module would

be simple if not for the fact that rewrite systems must be constructor-based in the extension. One cannot collect the names of the symbols that appear at the root of the term on the left-hand side of the symbol => and treat those and only those as evaluable functions. The problem is that the symbols might be used inconsistently as they are in the rewrite system

```
Double(Halve(x)) => x.
Halve(Double(x)) => x.
```

which was already discussed. The extended-Gödel code gathers the names of the symbols that appear as the left-hand side's root and then checks that these symbols don't occur in the left-hand side's arguments.

The Gödel system writes out several files during the compilation process. The extension adds one to the number of auxiliary files by writing out a `.ef` file. This file holds a list containing the names of the evaluable functions in the module. Referencing the `.ef` file simplifies compilation. The alternative would be to pass the list of evaluable functions from the parser to the compiler when the object representing the module is passed. However, the nature of Prolog programming makes the addition of an argument a serious modification. There is no type system to warn us that some extra argument was inserted in one place but overlooked in another. The code will continue to execute with possibly disastrous results.

3.4.2 Representing Rewrite Rules

The representation of the rewrite rules that were included by the user in an extended-Gödel module must be included in the representation of the module. Naturally, we want to use the structures already in place to represent the parts of an extended-Gödel module which are *identical* to a Gödel module. The object marked by the function symbol `ProgDef.Program.F4` has a section for the representation of statements. Since rewrite rules are very much like statements (statements define the behavior of predicates and rewrite rules define the behavior of evaluable functions), it seems reasonable to include the representation of rewrite rules with the representation of statements.

It was necessary to decide between two approaches. The first extends the representation within `Code` to include rewrite rules marked with their own function symbol. The second incorporates the representation of the rewrite rules into the `Code` section without introducing any new function symbols. In the final analysis, the second proved expedient.

To accomplish the representation of the rewrite rules using the first approach, a new function symbol `ProgDefs.RuleDef.F4` would be needed. Since the object `ProgDefs.Code.F2` has as its second argument a dictionary of symbol which have statements defining their behavior, an entry for each symbol which has a

rule defining its behavior would be inserted into the dictionary. Just as statements are marked as predicate definitions with the function symbol `ProgDefs.PredDef.F4` we could mark rewrite rules with the new symbol.

This plan has the merit of tidiness. Since the writers of the parser were careful to provide a plethora of function symbols that identify objects in the representation, a seamless integration would need to enrich the representation with distinct symbols for the new objects. Also, the production of the representation of rewrite rules with their own identifying function symbol necessitates the creation of a special path through the parser for rules. This parallels the implementation decisions made by the original parser writer, since there is a separate path through the parser for statements (as described above). However, the introduction of a new function symbol has tremendous consequences. Most predicates that look through the representation within `Code` will need to be modified so that they succeed, with some reasonable behavior, when they see `ProgDefs.RuleDef.F4`. Some predicates will not need modification because failure is the desired behavior when they see `ProgDefs.RuleDef.F4`. The number of modifications needed is large. Also, the result of missing even one of the necessary modifications is the complete interruption of compilation.

Another problem emerges when adding a separate path to the parser for dealing with rewrite rules. The type checking code of

the Gödel parser is quite complex. It would be wasteful to expend the effort to write a new type checker, but it isn't easy to see how rewrite rules could be sent to the existing type checker. Type checking is integrated into the parsing process, as one might imagine, not done in a separate pass. Many type checking predicates of the Gödel system were written in Gödel and compiled into Prolog. Thus the code available to us is machine generated. It has no comments and all variable names are the unhelpful choices of a machine. Thus it is very difficult to guess the function of the type checking predicates.

For these reasons, we chose the second approach to producing a representation of rewrite rules.

Incorporating the representation of the rewrite rules into the **Code** section without introducing any new function symbols might be expected to present a different but similarly challenging set of problems. Surprisingly this was not the case. The main modification to the parser that facilitated this approach was small, but more significant than the modifications that would be needed for the first approach.

To accomplish the representation of the rewrite rules using the second approach, a new system-wide symbol, => was defined. The definition is modeled on Gödel's system-wide definition of the equals symbol, =, and makes => an infix predicate. Included in the definition is the information that => is binary, is exported

to every module, and is part of the module named ''. All system-wide symbols are part of the module named ''. The code for adding the symbol => consists of one change to the system file **system.pl** and one change to the system file **parser.pl**. In **system.pl**, a single call to **SharedPrograms.InsertSymbol.P5** was added:

```
'SharedPrograms.InsertSymbol.P5'(
  'MetaDefs.Name.F4'(
    '''',                          % IN THE MODULE ''
    '''=>',                        % DEFINE =>
    'MetaDefs.Predicate.C0',       % AS A PREDICATE
    2),                            % OF ARITY TWO
  F,
  'ProgDefs.Exported.C0',
  'ProgDefs.PredicateDecl.F3'(
    2,
    'Syntax.ZPZ.C0',               % INFIX
    ['MetaDefs.Par.F1'(0),
     'MetaDefs.Par.F1'(0)]),       % MYSTERY PARAMETERS
  G)
```

This call was inserted into a predicate **SharedPrograms.InitializeLanguage.P2**. In **parser.pl**, two lines of new code provide information that the symbol => is reserved.

```
reserved_predicate_name('=>').
reserved_predicate_name('~=>').
```

The symbol ~=> is also reserved and is part of Gödel's implementation of negation.

3.4.3 Declaring Evaluable Functions

An implementation decision already discussed concerns the method of declaring evaluable functions. A change was made to the module system to enable the declaration of a user-defined function which returns a system-defined type. To see the necessity of this change, consider the following version of a module **Nat** which was presented before:

```
MODULE          Nat.
IMPORT          Integers.
BASE            Nat.
CONSTANT        Zero: Nat.
FUNCTION        Succ: Nat -> Nat;
                Succ2: Integer -> Integer;
                Interpret: Nat -> Integer.
Interpret(Succ(x)) => Succ2(Interpret(x)).
Succ2(x) => x + 1.
```

If the Gödel module system is not changed, this module will be rejected because it declares two functions whose target type is non-local (in this case system-defined). Gödel was written under the assumption that any symbol declared a **FUNCTION** will be used as a data constructor. Thus the term `Succ2(3)` would not be reducible and would be some new Integer. Similarly, `Interpret(x)` would be a new Integer.

Clearly the Gödel system can't allow a user to define new Integers. To exclude these problematic modules there is a predic-

ate among the group of predicates which does type checking on the function declarations. It enforces the condition that a function's target type is a type declared in the local module. The first part of the modification involved simply replacing a variable, **ModuleName**, with an underscore everywhere it occured in the predicate `function_type_aux`. This enables success in the test for declarations which would have failed the test when the module name of the target type didn't unify with the name of the module being parsed. Not surprisingly, this change necessitated another change to compensate for the discarded test.

The problem that needed to be solved after the change to `function_type_aux` involves the possibility that the user might declare a symbol in the **FUNCTION** section but neglect to provide rewrite rules. In this case, it would be treated as a data constructor. If its target type is user-defined, this will not present much of a problem, though it may produce behavior the programmer didn't expect. But a system-defined target type should be treated as an error if there are no rewrite rules for the symbol. The tests which finds these errors comes at the end of the parsing process.

3.5 Testing the Rewrite Rules

The most significant implementation decision involving testing of the rewrite rules was this: where in the parser should the tests be done? The original code does many checks of the statements that are part of a program as parsing progresses. Since the new representation of rewrite rules puts them in the structure as statements, it seemed reasonable to test the rewrite rules inside the statement-checking predicates. This would avoid the pitfalls inherent in writing predicates to create a separate path through the parser for rewrite rules. To some extent, this was successful. However, a combination of these two approaches worked best. The tests of the rewrite rules are located in the midst of a predicate called `parse_statements`. The new predicate `check_rules` looks only at the statements whose head has the symbol `=>` at its root—that is, it looks only at the rewrite rules. In this sense, there is a path through part of the parser which is for rewrite rules exclusively. Each statement is either tested by `check_rules` or passed up because it isn't a rule. But afterward, every statement goes through all the tests that the original parser has for statements.

The predicate `parse_statements` deals with one statement at a time. This is a good place to test for left-linearity and for testing the variables in a rule, among others. But the testing of a

rewrite system to see if it's constructor-based cannot be done at this point in the parsing process. All the rewrite rules that were provided in a module must be examined together to determine whether the system is constructor-based. For this reason, there is a second place in the code where testing of rules occurs. This is at the end of a predicate called **parse_module** which is very near the top-level of the parser. This is also the place where the group of rewrite rules defining each distinct evaluable function is tested to see if they are non-overlapping.

Testing the condition of a rewrite rule can be done one rule at a time. At first glance, it might seem that if the condition is not a conjunction of equations, there is no need to see if the variables in the condition meet the confluence requirements. But the richness of the Gödel language becomes an issue for implementing these tests: some Gödel constructs are not actually equations, but can still be tested. Specifically, a rule such as

```
F(x,y) => y <- SOME [x] (x=2)
```

fails the test for the condition being a conjunction of equations, but can still be processed as if it passed. The variable, **x**, in the condition is not the same variable as **x** in the left-hand side of the rule. A reasonable response to this rule would not produce the "not a conjunction of equations" warning. However, the rule is not properly oriented, if we consider **x=2** to be the conjunction

of equations, so a warning is printed.

The tests of the condition of a rewrite rule are located inside **parse_statements** where the writers of the parser put their checks of statements. The representation of the rule is complete at that point in the parsing process, and code which is syntactically wrong has been rejected.

3.6 Parser Output

Because => has been defined as a system-wide predicate symbol, the representation of a rewrite rule appears with the statements in the object representing a module. Luckily, the Gödel parser groups the clauses defining a single predicate together. Thus locating the rewrite rules in the structure (an AVL tree) that holds the statements is not difficult.

As an example, consider the rewrite rule

```
Plus(Zero, x) => x.
```

Its representation appears as one of the clauses defining =>. Since the extended Gödel sees => as an infix predicate, the representation indicates that the rule is a head clause.

```
[ MetaDefs.<-'.F2(
    MetaDefs.Atom.F2(                    % THE HEAD
      MetaDefs.Name.F4(
        ",
```

```
    "=>,                          % HAS ROOT =>
    MetaDefs.Predicate.C0,
    2),
  [ MetaDefs.Term.F2(             % LEFT-HAND SIDE
      MetaDefs.Name.F4(
        "Nov7,
        "Plus,
        MetaDefs.Function.C0,
        2),
      [ MetaDefs.CTerm.F1(        % RIGHT-HAND SIDE
          MetaDefs.Name.F4(
            "Nov7,
            "Zero,
            MetaDefs.Constant.C0,
            0)),
        MetaDefs.Var.F2("x,0)]),
    MetaDefs.Var.F2("x,0)])
MetaDefs.Empty.C0)]               % EMPTY BODY
```

Chapter 4

Conclusion

4.1 Comparison with LPG and BABEL

LPG (*Langage de Programmation Générique*) [2] is a functional-logic language based on Horn clause logic with equality. LPG allows the user to define evaluable functions by providing rewrite rules. Like Gödel, LPG is a strongly typed language and all symbols must be declared.

The rewrite rules of an LPG program need not be constructor-based. The rules must be left-linear. Conditional rewrite rules are allowed, but the restriction on the variables is more strict than in the extended Gödel. Every variable in the condition must be in the left-hand side of the rule. Following the classification system mentioned before, the term rewriting systems of LPG must 1-CTRSs; that is, any variables in the right-hand side of a rule must be in the left-hand side.

LPG allows the use of IF-THEN-ELSE construct on the right-hand side of a rewrite rule, which the extended Gödel does not.

For example, one can write in LPG

```
%insert x into the sorted list cons(y, s)
insert(x, cons(y, s)) ==> if x =< y
                          then cons(x, cons(y, s))
                          else cons(y, insert(x, s))
                          endif.
```

This is equivalent to writing

```
insert(x, cons(y, s)) ==> cons(x, cons(y, s))
                          <== x =< y == false.
insert(x, cons(y, s)) ==> cons(y, insert(x, s))
                          <== x =< y == true.
```

As the above rewrite rules suggest, the correspondence between symbols is this: where the extended Gödel uses =>, LPG uses ==>; where Gödel uses <- and =, LPG uses <== and ==.

LPG has a facility for specifying that an operation is commutative. Commutativity could be expressed by rewrite rules, for example,

$$\text{Multiply(x, y)} \Rightarrow \text{Multiply(y, x)}$$

but a rewrite system that includes such a rule may rewrite some terms forever without finding a normal form. LPG solves this problem by allowing the user to express commutativity in a *property* module rather than expressing commutativity with a rewrite rule.

LPG can accommodate evaluable functions which return a Boolean value. In some cases, defining a function rather than a predicate makes a program more readable. A predicate which could be defined by the clauses

```
in-range(x) <== x > 10
in-range(x) <== x < 0
```

can be written as a function defined by the rewrite rule `in-range(x) ==> x > 10 or x < 0`. This functional version might be, for some programmers, more intuitively appealing. To do the same in the extended Gödel, a user-defined type for Booleans must be created. The Gödel system does not provide a built-in type Boolean. The values `True` and `False` are provided by Gödel, but they are propositions (predicates of arity zero) and have no type. In the context of Gödel's type system this makes sense—the type of a predicate is the cross product of the types of its arguments. So there is no type for a predicate which has no arguments. The following is a fragment of an (somewhat odd) extended-Gödel module which mimics the LPG function.

```
BASE          Bool.
CONSTANT      MyTrue, MyFalse: Bool.
FUNCTION      InRange: Integer -> Bool.
InRange(x) => MyTrue  <- x > 10 \/ x < 0.
InRange(x) => MyFalse <- x =< 10 & x >= 0.
                  % NOTE: CONDITION NOT A
                  % CONJ OF EQUATIONS
```

The unusual form of the conditional part of the rewrite rules will be discussed in section 4.2, but for the moment, notice that a use of **InRange** will always compare the return value of the evaluable function to one of the constants. For example, one might include the equation **InRange(z) = MyTrue** as a part of the body of a clause. Providing a built-in type Boolean with constants True and False is certainly a candidate for the next step in the extension of Gödel.

Like Gödel and LPG, the programming language BABEL[9] requires the declaration of the symbols which are data constructors, the symbols which are evaluable functions and the symbols which are predicates. However, the declarations give the arity of the symbol and no other type information.

BABEL groups rewrite rules and clauses together as *rules*. Every BABEL rule has the form

$$k(t_1, t_2, \ldots, t_n) := C \rightarrow M$$

where C is a condition, called a guard, and $C \rightarrow$ is optional. This makes BABEL a little different from LPG and the extended Gödel since there is no reason that the guard has to be a conjunction of equations. In BABEL, a head clause is equivalent to **p(x) := true** and can be written in the Prolog-like style **p(x)**. Clauses with a non-empty body can be written in the Prolog-like way, too. The result is that the BABEL *rules* section of a program looks like a collection of rewrite rules and clauses.

BABEL rules are required to be left-linear. They must be constructor-based. Since clauses are represented as rules, this prevents programming with clauses that have repeated variables in the head. Working around this requirement is not difficult. Consider a predicate which uses unification to test two lists to see if they begin with the same element.

```
same-head([x|y], [x|z])
```

If clauses, as well as rewrite rules must be left-linear, the clause becomes

```
same-head([x|y], [x2|y2]) := x = x2 -> true.
```

BABEL does allow the omission of the `-> true` part of the rule, which makes it nicer-looking.

The restriction on variables in BABEL rules are not as strict as in LPG. Variables which did not appear in the left-hand side can appear in a guard. Since the conditional rewrite rules of BABEL are not required to have a conjunction of equations as the condition, it is not possible to fit them into the classification scheme of 1, 2, and 3 CTRSs. However, they nearly qualify as 2-CTRSs since the requirement in BABEL is the variables in the right-hand side must be a subset of the variables in the left-hand side. Finally, BABEL rules must be constructor-based and BABEL allows rules of the form

f(x) rewrites to IF y THEN z

and
f(x) rewrites to IF y THEN z ELSE w.

The Three Languages

The extended-Gödel, LPG and BABEL versions of the module Nat follow. Notice that the code is similar.

- The Extended-Gödel Code

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

- The LPG Code

```
type                Nat
sorts               nat
constructors        zero: -> nat
                    succ: nat -> nat
operators           plus: (nat, nat) -> nat
predicates          iszero: nat
variables           x, y: nat
equations
  1 : plus(zero, x) ==> x
  2 : plus(succ(x), y) ==> succ(plus(x, y))
```

```
clauses
  1 : iszero(zero)
end Nat
```

- The BABEL Code

```
constructors
  zero/0, succ/1
functions
  plus/2
predicates
  iszero/1
rules
  plus(zero, x) := x.
  plus(succ(x), y) := succ(plus(x, y))
  iszero(zero).
```

4.2 Correctness and Performance

The parsing code has been tested on well-formed modules and modules that have errors meant to exercise the new code. Also, larger modules written by members of the research team unfamiliar with the parsing code have been used for testing. The modified parser performs the required tasks of rejecting modules that have errors. It produces the object representing a correct module correctly. In these ways, the parser's functionality is everything it needs to be. In addition, the parser never objects to any module that the original Gödel would accept unless it uses the special

symbol =>. Naturally, this is essential if we are to claim that our language is an extension of Gödel.

A number of properties of the rewrite rules which are essential to guaranteeing confluence are checked. But a violation of the restrictions does not always result in an error (as was discussed in section 2.1). The decision to print a warning and compile some programs for which confluence cannot be guaranteed was motivated by the fact that confluence is, in general, undecidable. Many rewrite systems do useful computations but cannot be proved to be confluent. Rather than restrict the programmer to only those systems which are known to be confluent, the extended Gödel prints a warning that confluence may be lost. The programmer has the responsibility for creating rewrite systems that do useful work. If a user chooses to run programs that do not meet all the confluence requirements, she may. An example of a program that does useful work but does not meet the confluence requirements is the module fragment seen above.

```
BASE          Bool.
CONSTANT      MyTrue, MyFalse: Bool.
FUNCTION      InRange: Integer -> Bool.
InRange(x) => MyTrue  <- x > 10 \/ x < 0.
InRange(x) => MyFalse <- x =< 10 & x >= 0.
```

Because the predicates >, <, >= and =< are well-behaved, this system is confluent, even though the conditional part of the rules use

system-defined predicates and are not the conjunction of equations. Allowing such a program with a warning is attractive for the richness of expression of the language.

In addition to being confluent, it is attractive for rewrite systems to be terminating. A non-terminating rewrite system may rewrite some terms forever without finding a normal form. The programmer has the responsibility for termination, also. Termination is, in general, undecidable.

In terms of the object produced, the parser's performance is more than adequate. Except to enable compilation, there is no reason to have an object representing a module. The object produced facilitates compilation in three ways. First, it is no different in structure from the object produced by the original Gödel parser. Second, it has all the rewrite rules represented in one place, and in order. Third, as a side effect, the parser creates a file that lists all the evaluable functions. Thus, during compilation the distinguishing of evaluable functions and data constructors is fast and easy. The usefulness of the extensions to the parser for the task of compiling rewrite rules is its biggest claim to fame.

The amount of time required for parsing a rewrite rule is no different from the amount of time needed to parse a predicate. This is true because the extended Gödel sees a rule as a clause defining the system-declared predicate =>. The extra tests done

on rewrite rules, however, do add to the time needed for parsing. In places it is necessary to look through the representation of a rule several times to check the various properties. The efficiency may be increased slightly by the removal of stubs in the extension that facilitate easy changes between error and warning status for some tests. Still, the new code probably does not detract much, if at all, from the efficiency of the parser.

4.3 A Larger Program

On the following two pages is an example adapted from Bert and Echahed [2]. The module defines dots which are points whose coordinates are integers and the lines that join any two of those points. It also defines quadrilaterals whose vertices are dots. Two lines can be tested to find whether they are parallel. The diagonals of a quadrilateral can be computed.

Because **NotParallel** is truly a predicate in the sense that one wants a yes/no answer from it, it is a predicate in both programs. The predicate **Diagonal** is a relation which is not a function. That is, a quadrilateral has more than one diagonal. Thus **Diagonal** is a predicate in both programs. A line has just one sign, however. So **Sign** is a function in the extended-Gödel code. Similarly, **Dist** is a function in the extended-Gödel code. Notice that the code defining the behavior of **NotParallel** is

quite different in the presence of functions.

4.3.1 Gödel Code

```
MODULE              DotsAndLines.
IMPORT              Integers.
BASE                Dot, Line, Quad, Sign.
CONSTANT            Pos, Neg: Sign.
FUNCTION            CDot: Integer * Integer -> Dot;
                    CLine: Dot * Dot -> Line;
                    CQuad: Dot * Dot * Dot * Dot -> Quad.
PREDICATE           Dist: Integer * Integer * Integer;
                    Sign: Line * Sign;
                    NotParallel: Line * Line;
                    Diagonal: Quad * Line.
Dist(x, y, Abs(x - y)).
Sign(CLine(CDot(x, y), CDot(z, w)), Pos) <- (x =< z) & (y =< w).
                                            % SIGN OF SLOPE
Sign(CLine(CDot(x, y), CDot(z, w)), Pos) <- (x >= z) & (y >= w).
Sign(CLine(CDot(x, y), CDot(z, w)), Neg) <- (x =< z) & (y >= w).
Sign(CLine(CDot(x, y), CDot(z, w)), Neg) <- (x >= z) & (y =< w).
NotParallel(CLine(CDot(x, y), CDot(z, w)),
            CLine(CDot(x2, y2), CDot(z2, w2))) <-
                    Sign(CLine(CDot(x, y), CDot(z, w)), s1)
                    & Sign(CLine(CDot(x2, y2), CDot(z2, w2)), s2)
                    & s1 ~= s2.             % SLOPES HAVE
                                            % DIFFERENT SIGNS
NotParallel(CLine(CDot(x, y), CDot(z, w)),
            CLine(CDot(x2, y2), CDot(z2, w2))) <-
                    Dist(x, z, ans1)
                    & Dist(y2, w2, ans2)
                    & Dist(y, w, ans3)
                    & Dist(x2, z2, ans4)  % SLOPES ARE DIFFERENT
                    & ans1*ans2 ~= ans3*ans4.
Diagonal(Quad(CDot(x1, y1), CDot(_, _), CDot(x3, y3), CDot(_, _)),
         CLine(CDot(x1, y1), CDot(x3, y3)).
Diagonal(Quad(CDot(_, _), CDot(x2, y2), CDot(_, _), CDot(x4, y4)),
         CLine(CDot(x2, y2), CDot(x4, y4)).
```

4.3.2 The Extended Gödel Code

```
MODULE              DotsAndLines.
IMPORT              Integers.
BASE                Dot, Line, Quad, Sign.
CONSTANT            Pos, Neg: Sign.
FUNCTION  % DATA CONSTRUCTORS
                    CDot: Integer * Integer -> Dot;
                    CLine: Dot * Dot -> Line;
                    CQuad: Dot * Dot * Dot * Dot -> Quad.
FUNCTION  % EVALUABLE FUNCTIONS
                    Dist: Integer * Integer -> Integer;
                    Sign: Line -> Sign.
PREDICATE           NotParallel: Line * Line;
                    Diagonal: Quad * Line.
Dist(x, y) => Abs(x - y).
Sign(CLine(CDot(x, y), CDot(z, w))) => Pos
          <- (x =< z) & (y =< w).
                                          % SIGN OF SLOPE
Sign(CLine(CDot(x, y), CDot(z, w))) => Pos
          <- (x >= z) & (y >= w).
Sign(CLine(CDot(x, y), CDot(z, w))) => Neg
          <- (x =< z) & (y >= w).
Sign(CLine(CDot(x, y), CDot(z, w))) => Neg
          <- (x >= z) & (y =< w).
NotParallel(CLine(CDot(x, y), CDot(z, w)),
            CLine(CDot(x2, y2), CDot(z2, w2))) <-
            Sign(CLine(CDot(x, y), CDot(z, w)))
            ~= Sign(CLine(CDot(x2, y2), CDot(z2, w2))).
NotParallel(CLine(CDot(x, y), CDot(z, w)),
            CLine(CDot(x2, y2), CDot(z2, w2))) <-
            Dist(x, z) * Dist(y2, w2)
            ~= Dist(y, w) * Dist(x2, z2).
Diagonal(Quad(CDot(x1, y1), CDot(_, _), CDot(x3, y3), CDot(_, _)),
         CLine(CDot(x1, y1), CDot(x3, y3))).
Diagonal(Quad(CDot(_, _), CDot(x2, y2), CDot(_, _), CDot(x4, y4)),
         CLine(CDot(x2, y2), CDot(x4, y4))).
```

Bibliography

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

[2] D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. Technical report, IMAG-LGI, 1995.

[3] A. Bowers. Personal communication, September, 1995.

[4] 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.

[5] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Mitchie, and J. Richards, editors, *Machine Intelligence 11*, chapter 2, pages 21–56. Claredon Press, Oxford, 1988.

[6] R. Harper. Introduction to Standard ML. 1986-1993. Carnegie Mellon University.

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

[8] 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.

[9] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[10] D. Shapiro S. Antoy and J. Vorvick. Evaluable Functions in the Gödel Programming Language, December 1995. Visions for the Future of Logic Programming Workshop, ILPS, To appear.

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

[12] T. Suzuki, A. Middeldorp, 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.

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

[14] J. Wang. Personal communication, October, 1995. email: jiwei@bnr.ca, Bell-Northern Research Ltd.

Appendix A

Further Extensions

A number of attractive features could be added to the extended Gödel we have produced. One of these would be a warning for the user who provides rewrite rules that only partially define a function. Normally, it is not desirable to define partial functions because it means that some terms will not rewrite even though they include evaluable functions.

Another extension, mentioned with respect to LPG, would define the symbols True and False as constants of some built-in type Boolean. Gödel predefines the symbols `True` and `False` but not their type. With this change, Boolean functions could be defined by the user and would not differ from any other user-defined evaluable function. In particular, predefined Boolean operators, such as `&` and `\/`, could appear in Boolean terms. Likewise, Boolean terms could appear as atoms in the body of a clause.

It would also be useful to allow a conditional expression of the form `IF c THEN e1 ELSE e2` where `c` is a Boolean expression and `e1` and `e2` are expressions of the same type. A conditional expression could be allowed as a subterm of a term. The `ELSE` branch of the conditional expression would be mandatory and the value of the conditional expression would be the value of the expression in either its `THEN` or `ELSE` branch according to the truth of the condition `c`. This change will be difficult because we have defined `=>` as a predicate. Thus the parser would have to accept a conditional expression as an argument to a predicate.

A syntactic change which would clean up programs written in the extended Gödel would allow the user to give a name to a pattern that appears in the left-hand side of a rewrite rule. This is allowed in *ML* [6] programs. For example, the rule

```
Insert(x, Cons(y, s)) => Cons(x, Cons(y, s))
                      <- x =< y = False
```

could be written

```
Insert(x, list as Cons(y, s)) => Cons(x, list)
                              <- x =< y = False
```

thus eliminating the need to repeat complicated terms which were included only for unification.

Appendix B

The Order of the Rules

We adopt a convention concerning the order of presentation of the rewrite rules defining an evaluable function. The order is significant in the extended Gödel as it is in *ML* and *Haskell*. The original Gödel does not attach significance to the order in which statements appear in a file. The meaning of a variable is sometimes affected by the order of the rules. For example, in the rewrite system below (a functional definition of `IsZero` which corresponds to the definition of `IsZero` as a predicate discussed earlier), the variable `x` of the second rule stands for the complement of `Zero`, i.e., `Succ(y)`.

```
IsZero(Zero)  ⇒  True
IsZero(x)     ⇒  False
```

Thus, the second rule is applied to a (sub)term `IsZero(t)` only if `t` unifies with `Succ(y)`. If the order of the rules is reversed, the meaning of `x` is different.

```
IsZero(x)     ⇒  False
IsZero(Zero)  ⇒  True
```

Now the variable `x` of the second rule stands for `Zero` and `Succ(y)`. The second rule is never fired.

In both rewrite systems above, `x` is a variable that is used only once in any rule. It can be replaced with the anonymous variable, _. Gödel takes advantage of the notational convention

that a variable name beginning with an underscore in the body of a statement or goal stands for a unique variable existentially quantified at the front of the atom in which it appears [7]. To achieve a seamless integration of rewrite rules into Gödel, a variable name beginning with an underscore in a the left-hand side of a rewrite rule is allowed.

The convention concerning the order of the rewrite rules is useful for implementing efficient narrowing. For the implementation of needed narrowing, it is necessary to ensure that rewrite systems are inductively sequential. A rewrite system presented with this convention concerning order is automatically inductively sequential, hence non-overlapping. Although the order in which the rules appear in a file is significant, the order in which rules are selected by the narrowing strategy is unspecified. In a similar manner, the order in which the clauses defining a predicate are selected by Gödel's computation rule is unspecified. In this way, writing Gödel code is a bit different from writing Prolog code. Prolog chooses clauses in textual order.

It may be confusing to assess the rewrite system

```
IsZero(Zero)  ⇒  True
IsZero(_)     ⇒  False
```

in light of the fact that the two rules may be selected in any order. In solving the equation IsZero(Zero) = x, the selection of the second rewrite rule might seem problematic. But the left-hand side of the second rewrite rule will not unify with IsZero(Zero). Although there is an underscore in the text, it does not unify with Zero. The underscore has the special meaning, "the complement of Zero." This is the importance of using these conventions: the rewrite system will not have several rules that specify different normal forms for a single term.

A disadvantage results from the need for inductive sequentiality. The clauses defining a predicate cannot necessarily be transformed into functions easily. Consider the predicate `IsZero`. The first definition of `IsZero` presented defined it as a predicate with one clause,

```
IsZero(Zero).
```

Later a function mapping `Nat`s to Boolean values was presented. One of the rewrite rules was made directly from the predicate's clause by adding an arrow and the word `True`.

```
IsZero(Zero) ⇒ True.
```

Some predicates which have attractive clauses cannot be transformed into rewrite rules so easily. Though predicates can be seen as Boolean functions, Gödel adopts no convention on the order of the clauses of a predicate. Thus a predicate regarded as a function may not be inductively sequential. An example of such a predicate is the so-called *parallel-or*.

```
Or(True,_).
Or(_,True).
```

The expression `Or`(t_1, t_2) evaluates to `True` as long as one argument evaluates to `True`, even if the other argument is undefined. Transforming the predicate into a Boolean function in the simple-minded way yields

```
Or(True,_) ⇒ True
Or(_,True) ⇒ True
```

which is not inductively sequential.

Ideally, any predicate could be transformed into a Boolean function directly from its clauses. But the restriction requiring

inductive sequentiality prohibits a simple transformation. Since an efficient narrowing strategy is available for inductively sequential functions, it seems an acceptable loss.

Appendix C

More About the Internal Representation

The four arguments to `ProgDefs.Program.F4` give the module name, the structure of the module, the language defined by the declarations and the statements defining predicates. The first of these is a simple object, but the other three are AVL trees. The tree representing the structure of the module includes information concerning the importing module and all imported modules (if any). The tree representing the language has a node for each module in the previously described tree and one node corresponding to the built-in language of Gödel. Within each node, the symbols of the language are given together with type and arity information. The tree representing the statements also has one node for each module. However, there is no node for holding the representation of statements defining built-in predicates. The built-in predicates are 'hard-wired' rather than defined by Gödel clauses. Within each node, the statements are represented by an object marked with the function symbol `ProgDefs.Code.F2`.

The object `ProgDefs.Code.F2` has as its second argument an AVL tree with a node for each symbol which has a statement defining its behavior. All the statements that define a specific predicate are collected in a single node. They are marked as predicate definitions with the function symbol `ProgDefs.PredDef.F4`. The object representing the module

```
MODULE          Example.
BASE            Type1.
CONSTANT        A.
PREDICATE       IsA: Type1.
IsA(A).
```

consists of `ProgDefs.Program.F4` and its four arguments. The first two are small enough to be included in their entirety. The second has been edited to show the object representing just one symbol, the constant `A`. The complete language object spans one hundred lines. The fourth argument is the most important to the extension.

```
ProgDefs.Program.F4(
  "Example,                                  % FIRST ARG
  AVLTrees.Node.F5(AVLTrees.Null.C0,         % SECOND ARG
                   "Example,
                   ProgDefs.ModDef.F4(
                     ProgDefs.ModuleKind.C0,[],[],[]),
                   AVLTrees.EQ.C0,
                   AVLTrees.Null.C0),
  ProgDefs.Language.F1(                      % THIRD ARG
    AVLTrees.Node.F5(
      AVLTrees.Null.C0,
      ",
      .
      .
      .
            AVLTrees.Node.F5(
              AVLTrees.Null.C0,
              "A,
              [ ProgDefs.Symbol.F2(          % IS A SYMBOL
```

```
                ProgDefs.Hidden.C0,
                ProgDefs.ConstantDecl.F1(
                                        % CONSTANT
                  MetaDefs.BType.F1(
                    MetaDefs.Name.F4(
                     "Example,
                     "Type1,            % ITS TYPE
                     MetaDefs.Base.C0,
                     0))))],
              AVLTrees.RH.C0,
              AVLTrees.Node.F5(
    .
    .
    .

      AVLTrees.Null.C0))),
AVLTrees.Node.F5(                       % FOURTH ARG
  AVLTrees.Null.C0,
  "Example,
  ProgDefs.Code.F2(
    0,
    AVLTrees.Node.F5(
      AVLTrees.Null.C0,
      "IsA,                             % `IsA´ HAS CLAUSES
                                        % DEFINING IT
      [ ProgDefs.PredDef.F4(
          1,
          [ MetaDefs.<-´.F2(            % A CLAUSE
              MetaDefs.Atom.F2(
                MetaDefs.Name.F4(
                  "Example,
                  "IsA,
                  MetaDefs.Predicate.C0,
```

```
                1),
              [ MetaDefs.CTerm.F1(
                  MetaDefs.Name.F4(
                    "Example,
                    "A,
                    MetaDefs.Constant.C0,
                    0))]),
            MetaDefs.Empty.C0)],
                                % EMPTY BODY
        [],
        [])],
    AVLTrees.EQ.C0,
    AVLTrees.Null.C0)),
AVLTrees.EQ.C0,
AVLTrees.Null.C0)))
```

Appendix D

More About the Grammar Used in the Parser

The grammar describing terms is explained in the file **term.pl**. The strategy used for parsing terms focuses first on sorting through the sequence of tokens to form terms and then dealing with the operators. The grammar that provided the starting point for the code was manipulated by hand to remove left recursion. Many of the clauses of the parser in **term.pl** are directly related to clauses of the grammar. As an example of the relationship between the grammar that describes Gödel programs and the parser code, consider this clause which was commented with the grammar rule that inspired it:

```
% This clause is for grammar T -> f1 T | f2 T
term_aux(prefix_function(Functor, Indicator),
         Tokens, Return, SpecLeft,
         PrecLeft, Language) :-
   Indicator =.. [SpecOp, PrecOp],
   term(Tokens, Return2, SpecOp, PrecOp, Language),
   term_aux_prefix(Return2, Return,
                   Tokens, SpecLeft,
                   PrecLeft, Functor,
                   SpecOp, PrecOp).
```

The code is not simply a variation of the grammar rule, but some relationship is evident:

- The rule concerns those terms which are made up of prefix functions and their arguments.
- The code looks for a prefix function and, when one is found, sends the tokens that will give its arguments to the predicate **term** which parses them.