6-12-1996

Needed Narrowing as the Computational Strategy of Evaluable Functions in an Extension of Goedel

Bobbi J. Barry
Portland State University

THESIS APPROVAL

The abstract and thesis of Bobbi J. Barry for the Master of Science in Computer Science were presented June 12, 1996 and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]

Sergio Antoy, Chair

[redacted]

James Hein

[redacted]

Michael 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 2 August 1996.

ABSTRACT

An abstract of the thesis of Bobbi J. Barry for the Master of Science in Computer Science presented June 12, 1996.

Title: Needed Narrowing as the Computational Strategy of Evaluable Functions in an Extension of Goedel.

A programming language that combines the best aspects of both the functional and logic paradigms with a complete evaluation strategy has been a goal of a Portland State University project team for the last several years. I present the third in a series of modifications to the compiler of the logic programming language Goedel which reaches this goal. This enhancement of Goedel's compiler translates user-defined functions in the form of rewrite rules into code that performs evaluation of these functions by the strategy of needed narrowing. In addition, Goedel's mechanism that evaluates predicates is supplemented so that needed narrowing is still maintained as the evaluation strategy when predicates possess functional arguments.

NEEDED NARROWING AS THE COMPUTATIONAL STRATEGY OF EVALUABLE FUNCTIONS IN AN EXTENSION OF GOEDEL

by

Bobbi J. Barry

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Portland State University

1996

ACKNOWLEDGEMENTS

I wish to express appreciation to a few people who made this thesis possible: Sergio Antoy, Jeff, Valerie, and David Barry. Sergio Antoy, my advisor, coaxed me into being involved in a project of which I knew little. He believed that I had potential and encouraged me to reach my goals. I would not have been as successful without his guidance. Jeff, my husband, allowed me the opportunity to pursue a new career path and was willing to support me in this time of transition. Lastly, my children Valerie and David gave up many hours of mom to allow me time to study.

Also, thanks is extended to the National Science Foundation who partially funded this work under grant CCR-9406751 and to my two predecessors on the Goedel project, David Shapiro and Janet Vorvick.

Thank you to all.

Bobbi

Abstract

I. Introduction

For the past several years a research team at Portland State University has investigated the theoretical and practical aspects of developing a functional-logic language. In the fall of 1995, the project team reached a major goal by extending the logic language Goedel with a functional component.

The computational strategy for this functional extension is eager narrowing. While eager narrowing does allow for the realization of a functional-logic language, it has one major drawback. The operational semantics of eager narrowing are incomplete for non-terminating rewrite systems. Consider the following functional code, written in a Goedel-like syntax [5]:

```
Primes => Sieve(Ints_from(2)).

Sieve([a|b]) => [a| Sieve(Filter(b,a)].

Ints_from(a) => [a| Ints_from(a+1)].

Filter([a|b],c) => IF (a Mod c)=0 THEN Filter(b,c)
                                  ELSE [a| Filter(b,c)].

Show(x,[a|b]) => IF x=0 THEN [] ELSE [a|Show(x-1,b)].
```

Based on the above term rewriting system, a request to `Show` the first n primes, i.e. `Show(n,Primes)`, should succeed, but eager narrowing will loop forever on a call of `Primes`. Needed Narrowing is a computational strategy that is sound, complete, and optimal for inductively sequential rewrite systems. Thus a request to produce the first n primes will succeed with needed narrowing as the computational strategy. This is the next major goal of the P.S.U. project: a functional-logic language with complete semantics for non-terminating rewrite systems.

This thesis represents an attempt to integrate a functional and logic language with needed narrowing as the functional computational strategy. Specifically my predecessors on the P.S.U. project have modified the logic language Goedel to accept/parse functions [14], and to eagerly evaluate functions [13]. My work is to replace the eager evaluation of functions in this extension with the lazy evaluation strategy of needed narrowing. (Hereafter I will refer to these two extensions as the eager-functional extension and the lazy-functional extension of Goedel.)

To compute functions by needed narrowing, I have implemented two major changes within the compiler. The Goedel compiler now builds definitional trees for appropriately defined functions. And once a definitional tree is built a function called *Codegen* [4] recurs on the tree to generate appropriate code.

The remainder of the thesis discusses theoretical and practical concepts relevant to this alteration of the code generator within the Goedel compiler. Section II briefly recalls some fundamental ideas of rewriting and narrowing. Section III describes the concept of definitional tree and provides two strategies to build them. Section IV gives the algorithm for production of needed narrowing code into the Prolog language. Section V discusses optional optimizations of the code produced. Section VI describes the implementation within the Goedel compiler environment. Section VII compares lazy-functional Goedel with two other functional-logic languages. Section VIII provides some examples and Section IX concludes the thesis.

II. Narrowing

We start this section by recalling the concepts of rewriting and narrowing, and we finish the section with a short discussion of lazy narrowing and eager narrowing.

A symbol is a mark or collection of marks that we usually assign some type of meaning. In arithmetic, symbols are of the form 3,6,+, or =. The following are symbols that are useful to us:

```
{Succ, Zero, Add, False, True, Leq }.
```

Each symbol has an associated type referred to as a sort. From the set above, the first three symbols `{Succ, Zero, Add}` have sort natural numbers and the last three symbols `{False, True, Leq}` have sort boolean.

Likewise each symbol has a signature which associates the symbol with a list of sorts. Using the symbols above, the following are their signatures:

```
Succ:  natural,natural;
Zero: natural;
Add: natural, natural, natural;
False: boolean;
True: boolean;
Leq: natural, natural, boolean.
```

The signature of a symbol describes the type of the symbol's arguments and the type of the symbol itself. If the signature is of length n, where n>0, the first n-1 elements are the sorts of the n-1 arguments of the symbol and the nth element is the sort of the symbol itself. From the signature of `Leq`, we can deduce the symbol `Leq` has two arguments of type natural and the sort of `Leq` is boolean.

A term consists of symbols that are well typed. Thus `Succ(Zero)` and `Add(Zero,Zero)` are terms. While `Add(False,Zero)` is not a term. It should be noted that terms may contain unknowns or variables and the unknown assumes the type of the argument to produce a term. For example in the term `Succ(x)`, the unknown `x` assumes type natural so that the term is valid.

A substitution is an assignment of the variables in a term by a term of the appropriate sort. Thus a substitution of the term `Succ(z)` by `{z|->Zero}` yields the term `Succ(Zero)`.

A rewrite rule is an ordered pair of terms of the same sort represented by L=>R where the term L is not a variable and the variables of term R are a subset of the variables contained in the term L . L is referred to as the left-hand side of the rewrite rule and R is referred to the right-hand side of the rewrite rule. A rewrite system is just a set of rewrite rules. A rewrite system using the symbols from above could consist of the following rewrite rules. In this thesis, rewrite rule are represented in a Goedel-like syntax.

```
R1: Leq(Zero,y) => True

R2: Leq(Succ(x),Zero) => False

R3: Leq(Succ(x),Succ(y)) => Leq(x,y)

R4: Add(y,Zero) => y

R5: Add(y, Succ(x)) => Succ(Add(y,x))
```

Rewriting is an operation performed upon a term (or subterm) using a substitution applied to a rewrite rule. To perform rewriting of a term, we first find a substitution of the variables of the left-hand side of a rewrite rule such that when the substitution is applied, the left-hand side of the rule and the term are identical. If a substitution exists, the term is replaced by the corresponding right-hand side of the rewrite rule with the same substitution applied. For example, the term `Leq(Succ(Zero),Succ(Zero))` can be rewritten using rule `R3` and under the substitution `{x|->Zero, y|->Zero}`. Thus the term is rewritten as `Leq(Succ(Zero),Succ(Zero))-->Leq(Zero,Zero)`. If we apply rewriting a second time with rule `R1` and the substitution `{y|->Zero}` we obtain `Leq(Succ(Zero),Succ(Zero))-->Leq(Zero,Zero)-->True`. In essence we have used rewriting to compute that `Leq(1,1)=True`, or 1≤1 is true.

Narrowing also is an operation performed upon a term using a substitution

and rewrite rule, but narrowing allows a bilateral substitution. Both the variables of the left-hand side of a rewrite rule and the variables of the term may be substituted. Consider the term `Leq(z,Zero)`, using rule `R1` and the substitution `{z|->Zero, y|->Zero}` we can narrow the above term to yield the term

```
Leq(z,Zero)-->Leq(Zero,Zero)-->True.
```

Using the same original term, an alternate substitution of `{z|->Succ(x)}` with rule `R2` will yield the term of

```
Leq(z,Zero)-->Leq(Succ(x),Zero)-->False.
```

Thus narrowing allows the computation to proceed by "guessing" at values for variables. The choice of substitution is made nondeterministically.

There are many different strategies which are used to narrow terms or subterms. The discussion here centers on an eager strategy versus a lazy strategy. The eager strategy under scrutiny is the strategy which is implemented in the eager-functional extension of Goedel - leftmost innermost narrowing. The lazy strategy under discussion is the strategy chosen to replace leftmost innermost narrowing in the eager-functional extension of

Goedel - needed narrowing.

If there is a choice of subterms to narrow, leftmost innermost narrowing selects the leftmost innermost term first. Consider the term,

```
Add(Add(Succ(Zero), Succ(Zero)), Zero).
```

Leftmost innermost narrowing begins by narrowing the subterm in position 1, `Add(Succ(Zero), Succ(Zero))`. Thus it may have a derivation with corresponding substitutions similar to the following:

```
Add(Add(Succ(Zero), Succ(Zero)),Zero)-->
                                {y|->Succ(Zero), x|->Zero} with R5
Add(Succ(Add(Succ(Zero),Zero)), Zero)-->
                                        {y|->Succ(Zero)} with R4
Add(Succ(Succ(Zero)),Zero)-->
                                    {y|->Succ(Succ(Zero))} with R4
Succ(Succ(Zero)).
```

Needed narrowing is a lazy strategy which delays the evaluation of arguments until the value is needed to compute the result. In the example given above the needed position for `Add` is position 2, so we begin narrowing with the

subterm in position 2- `Zero`. The needed term is in head normal form so we apply the substitution to the complete term. A derivation may look like the following:

```
Add(Add(Succ(Zero), Succ(Zero)),Zero)-->
                          {y|->Add(Succ(Zero),Succ(Zero))} with R4
Add(Succ(Zero), Succ(Zero))-->
                                {y|->Succ(Zero), x|->Zero} with R5
Succ(Add(Succ(Zero),Zero))-->          {y|->Succ(Zero)} with R4
Succ(Succ(Zero)).
```

In the example above both needed narrowing and leftmost innermost narrowing narrow the term to `Succ(Succ(Zero))` in four steps. Both strategies do not always yield such similar steps and results.

Needed narrowing has two advantages over leftmost innermost narrowing. The operational semantics of leftmost innermost narrowing are incomplete whereas needed narrowing's operational semantics are complete. Completeness guarantees the finding of all solutions to an equation, if there are solutions. Also needed narrowing delays evaluation of arguments until the value is needed, while leftmost innermost may perform some unnecessary computations. Consider the following function `Times` with the given signature

and corresponding rewrite rules:

```
Times: natural, natural, natural
Times(Zero,y)=>Zero
Times(Succ(x),y)=>Add(y,Times(x,y)).
```

Now let us compare the two derivations of the following term by each strategy.

```
Times(Zero,Add(Succ(Zero),Succ(Zero))).
```

Leftmost innermost narrowing yields the following derivation:

```
Times(Zero,Add(Succ(Zero), Succ(Zero)))-->
Times(Zero, Succ(Add(Succ(Zero),Zero)))-->
Times(Zero, Succ(Succ(Zero)))-->Zero
```

A derivation with the strategy of needed narrowing will not compute the second argument because it is not needed for the result. Thus a derivation and corresponding substitution is:

```
Times(Zero,Add(Succ(Zero),Succ(Zero)))--> Zero,
                                {y|->Add(Succ(Zero),Succ(Zero))}.
```

The above example is a clear demonstration where leftmost innermost computes unnecessarily. But both strategies do compute the correct answer. Let us now consider the case where leftmost innermost narrowing fails to compute an answer.

Consider the following code borrowed from Hanus [9] written in a Goedel-like style:

```
First(Zero,l)=>[].
First(Succ(n),[e|l])=>[e|First(n,l)].
From(n)=>[n|From(Succ(n))].
```

The function `First(n,l)` will compute the first `n` elements from a list `l` and the function `From(n)` computes the infinite list of natural numbers beginning with `n`.

The call `First(Succ(Succ(Zero)),From(Zero))` yields the result `[Zero,Succ(Zero)]` via a needed narrowing computational strategy. The

same call with the leftmost innermost narrowing strategy will not terminate.

III. Definitional Trees

Definitional trees are the only known "tool" by which we are able to generate needed narrowing code. This section will explain what a definitional tree is, when it is possible to build a definitional tree for a function, and two techniques to build definitional trees.

Definitional trees are defined for a set of rewrite rules that are inductively sequential. For rewrite rules to be inductively sequential they must satisfy two requirements: left-linearity and non-overlapping. Left-linearity ensures that a rewrite rule contains any given variable only once in the left-hand side of a rule. Thus `Equal(x,x)` is not allowed. Non-overlapping of arguments ensures that a function cannot have two rewrite rules that have unifying left-hand side arguments. Thus rewrite rules for a function `Equal` with left-hand sides of the form `Equal(Zero,x)` and `Equal(_,x)` are not allowed.

A definitional tree is a hierarchical structure that contains and organizes a function's rewrite rules. To aid our understanding of a definitional tree, let's consider the following user defined function `Equal` where `Zero` and `S` are

constructors of the sort natural numbers:

```
R1: Equal(Zero,Zero)=>true.

R2: Equal(Zero,S(x))=>false.

R3: Equal(S(x),Zero))=>false.

R4: Equal(S(x),S(y))=>Equal(x,y).
```

Figure 1 shows definitional tree for the function `Equal`. To contain a rewrite rule, a definitional tree must posses a pattern within a node that is a variant of a left-hand side of a rewrite rule. Two terms are variants if one is obtained from another by renaming variables. The pattern `Equal(Zero,S(w))` and the left-hand side of rule `R2` are variants. Thus `R2` is contained in the definitional tree below. In fact all four rewrite rules from above are contained in the definitional tree below.

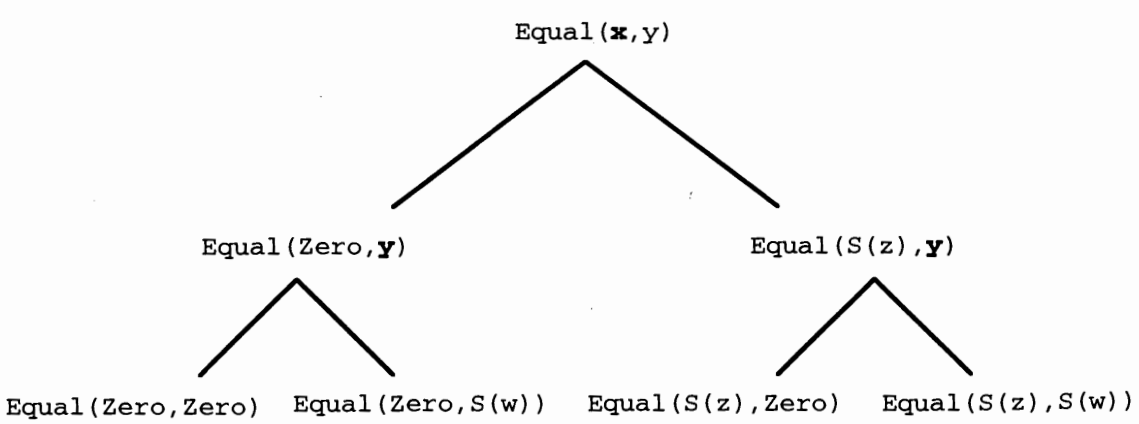


Figure 1

In general a definitional tree is made up of branch nodes, rule nodes, and exempt nodes [2]. A branch node is a node that does not contain a rewrite rule. A branch node has a pattern with a variable which is referred to as the inductive variable. The inductive variable is bold-faced in the example above. A rule node is a node that contains a rewrite rule for a function. An exempt node is neither a rule or branch node. Thus an exempt node does not contain a rewrite rule or an inductive variable. At the root node of a definitional tree is a function with arguments consisting entirely of variables. As you progress down a definitional tree, the inductive variables are replaced by constructor-rooted arguments of the appropriate sort until a rule node is obtained.

From the figure above, the branch nodes are `Equal(x,y)`, `Equal(Zero,y)`, `Equal(S(z),y)` and the rule nodes are `Equal(Zero,Zero)`, `Equal(Zero,S(w))`, `Equal(S(z),Zero)`, and `Equal(S(z),S(w))`. There are no exempt nodes in the tree for `Equal` given above.

A more formal definition is given in [4] which we rephrase here. A pattern is a term of the form $f(t_1,...,t_n)$ where f is a defined operation and, for all i, t_i is either a variable or a constructor term. A pattern π subsumes π' if π' is obtained from π by replacing a variable of π with a shallow constructor term of an appropriate

sort. A shallow constructor term is a term that is constructor-rooted and whose arguments, if any, consist of only variables. As an example, `S(z)` is a shallow constructor term while `S(Zero)` is not. Also, if π=Equal(x,y) and π'=Equal(S(z),y) then π subsumes π'. A definitional tree of an operation f is a nonempty set `T` of patterns partially ordered by subsumption and having the following properties up to renaming of variables.

•Root Property. The minimum element, called the root of `T`, is $f(X_1,...,X_n)$ where $X_1,...,X_n$ are distinct fresh variables.

•Leaf Property. The maximal elements, called the leaves of `T`, are variants of the left-hand sides of the rewrite rules defining f. Non-maximal elements are referred to as branches.

•Parent Property. If π is a pattern of `T` and not the root, there exists in `T` a unique pattern π' strictly preceding π such that there exists no other pattern between π and π'. The parent of π is π' and π is the child of π'.

•Induction Property. All children of a same parent differ from each other only at the position of a variable, the inductive variable of the parent.

Although the formal definition and picture of a definitional tree help us

understand the concept, a non-pictorial representation of a tree is useful in some further discussions. In the following sections of the thesis, a definitional tree may be represented by the form branch(π,P,T) or rule(π) where π represents the pattern of the node, P is a list of numbers indicating the position of the inductive variable of the node, and T is a list of definitional trees which have pattern π' such that π' is a child of π [2].

Using the pictorial example of `Equal` given above, we can now represent `Equal`'s definitional tree in the following manner.

```
branch(Equal(x,y),[1],                    %root node
    [branch(Equal(Zero,y),[2],            %left child
        [rule(Equal(Zero,Zero)),
          rule(Equal(Zero,S(w))]),
      branch(Equal(S(z),y),[2],           %right child
        [rule(Equal(S(z),Zero),
         rule(Equal(S(z),S(w)))])]).
```

Given a function's rewrite rules there are two strategies for building definitional trees: top-down and bottom-up. Let's consider each technique.

Bottom-Up Strategy

In the bottom-up strategy, there are two main processes, called "chaining" and "merging"[4]. The first step is to "chain" the arguments of the left-hand side of a rewrite rule until we reach arguments that are entirely variables. For example, the function `Leq` can be defined by the following rules where `Zero` and `S` are the constructors of the sort natural numbers:

```
R5: Leq(Zero,y) => True,
R6: Leq(S(x),Zero) =>False,
R7: Leq(S(y),S(x)) => Leq(y,x).
```

Each left-hand side of a rule goes through a chaining process where one shallow constructor term at a time is replaced by a variable. A partial chaining sequence for the rewrite rule `R7` may look like the following:

```
Leq(S(y),S(x)) --> Leq(S(y),w) --> Leq(z,w).
```

There are two noteworthy items. First, this is not the only chaining sequence possible for this particular rule. Second, for convenience we view this incomplete chain in reverse order of generation and as a sequence of terms:

```
Leq(z,w), Leq(S(y),w), Leq(S(y),S(x)).
```

For each term in the sequence there is a position that relays information about which argument was replaced (i.e. which is the inductive argument). Thus from above we have a sequence of positions of ([1],[2]); again viewed in reverse order of generation. The position of [1] represents the replacement of `S(y)` with `z`, and [2] represents the replacement of `S(x)` with `w`. The chain is now completed by alternating functional terms with a corresponding term from the position list. From the rewrite rules for `Leq` given above, we obtain the following chains:

```
C5: Leq(z,y),[1],Leq(Zero,y)

C6: Leq(z,w),[1],Leq(S(x),w),[2],Leq(S(x),Zero)

C7: Leq(z,w),[1],Leq(S(y),w),[2],Leq(S(y),S(x)).
```

After all rules' left-hand sides are chained, the second step of the bottom-up strategy is to merge the chains into a tree, if possible. The merging process involves two distinct substeps. The initial step of the merging process is confirming that all the chains for the given function have equal positions as the second member of their chain (i.e. they have the same inductive variable). From the example above it means that all chains of rewrite rules for the function `Leq` have `[1]` as their leading position. If the positions are not equal there are two possibilities: either the "wrong" argument was replaced by a

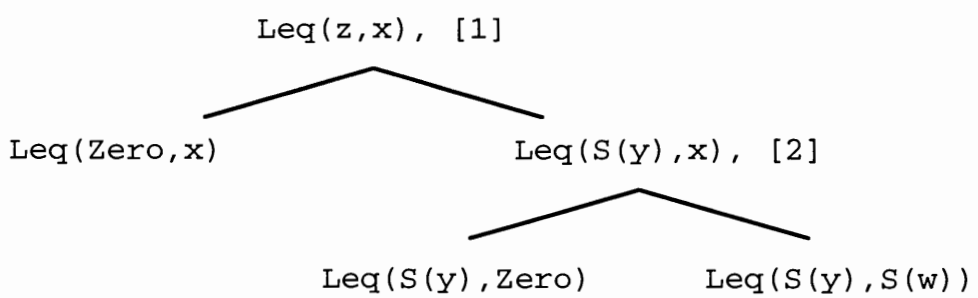
variable in the chaining process or a definitional tree does not exist. If the positions are equal the first term of the chains represent a node in the definitional tree with inductive variable specified by the position list. From here we shorten the chains by disregarding the lead term and following position of a chain. With the remaining portion of the chains, we proceed to the next substep of the merging stage.

The shortened chains above are

```
C5': Leq(Zero,y)

C6': Leq(S(x),w),[2],Leq(S(x),Zero)

C7': Leq(S(y),w),[2],Leq(S(y),S(x)).
```

To continue with the merging process, we split the chains into smaller lists. All chains with variant arguments for the new leading term of their chain sequence will be placed into the same list. Above `C6'` and `C7'` will be in the same list while `C5'` will be in its own list. These two steps of verifying equality of the lead position and creating smaller lists of variant arguments are repeated on each list until eventually a list containing a variant of a left-hand side of a rewrite rule is obtained. Chains in the same list are in the same branch of the tree. The pattern of variant arguments represents a branch node in the definitional tree, while a list containing a variant of a rule represents a

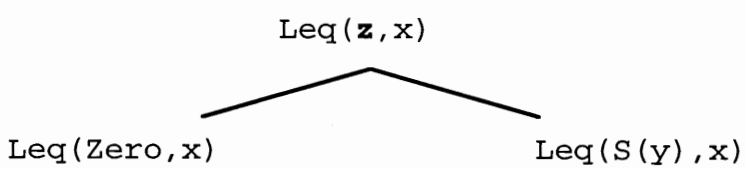
rule node of the definitional tree. If this complete process of chaining and merging succeeds a definitional tree for the given function exists and is built. If this process fails a definitional tree does not exist and needed narrowing code cannot be generated for the given function. A definitional tree for `Leq` is given below.



Top-Down Strategy

Another technique for building definitional trees is the top-down method. The top-down method begins by searching for an inductive position. The left-hand sides of rewrite rules for a given function are examined. The search is for a position where all left-hand side arguments are nonvariable. If there does not exist such a position a definitional tree is not possible and needed narrowing code will not be generated. If there exists such a position, this is an inductive position and we can begin to grow the definitional tree. The growing of the

definitional tree is accomplished by substituting in the inductive position of a shallow functional-rooted term a shallow constructor term of an appropriate sort. These newly created terms are nodes within the definitional tree. For example using `R5`, `R6`, and `R7` from above, we discover position 1 for all the rewrite rules of `Leq` is nonvariable. So we start with `Leq(z,x)` and determine `z`'s sort is the natural numbers. Shallow constructor terms of the natural numbers are `Zero` and `S(y)`. So substituting both constructor terms in place of z, position 1, we obtain `Leq(Zero,y)` and `Leq(S(y),x)` The initial growth of the definitional tree for `Leq` is shown below.



We continue the top-down method by splitting the left-hand sides of the rewrite rules into smaller lists. All left-hand sides which unify with a node in the definitional tree will be in the same list. Thus there is a correspondence between a list and a node of the definitional tree. If the list which corresponds to a node in the definitional tree is empty, the processing of that portion of the tree is complete and the node is an exempt node. If the list contains one element which is a variant of the pattern within the corresponding node, the

processing of that particular node is complete and the node is labelled a rule node. If neither of the two above cases occur, we recur on the list. The search back in step 1 is now for a new inductive position for a particular list of rewrite rules. The growth of the tree continues from the list's corresponding node of the definitional tree.

To continue with the example `Leq`, we split the left-hand sides of the rewrite rules into the following two lists: `[Leq(Zero,y)]`, `[Leq(S(x),Zero), Leq(S(y),S(x))]`. The first list contains a variant of the node `Leq(Zero,x)` and thus processing is complete and it is labelled a rule node. The second list relies on recursion where we will detect position 2 is a new position that contains nonvariable terms. Thus in position 2 shallow constructor terms of the natural numbers are substituted in the node pattern `Leq(S(y),x)`. Here two new terms are obtained `Leq(S(y),Zero)` and `Leq(S(y),S(z))`. Again the tree grows two new nodes and we obtain the complete tree given above for the bottom-up method. At step two these two new nodes are labelled rule nodes and the definitional tree for `Leq` is completed.

Given rewrite rules for a function which are inductively sequential, each technique does produce a definitional tree for the evaluable function. Though each technique requires different information within the process. The top-down method requires that more information is known in advance to build the

tree. In the `Leq` case discussed above, information about argument sorts and argument constructors are immediately needed for the substitution that is performed within step 2. Also if a user omits a rewrite rule it is incorporated into the definitional tree in the form of an exempt node. This produces a bushier form of the tree.

The bottom-up method requires little knowledge in advance, but may lose in efficiency. We need only to differentiate between what is a constructor and what is not a constructor before the chaining process can begin. Also if a user omits a rewrite rule the tree is built with one less rule node. The bottom-up method has a potential drawback. It may be less efficient in the situation where a wrong argument is mistakenly chosen to be replaced within the chaining process. Backtracking will eventually find the correct argument to replace.

The bottom-up strategy is the technique I chose for implementation in the Goedel compiler. The lack of advance knowledge, the ease of implementation, and the backtracking capability of Prolog, which the Goedel compiler is written in, were all determining factors in this choice.

IV. Codegen

After building a definitional tree, the next step is to generate the appropriate needed narrowing code. One strategy for generating needed narrowing code involves a recursive procedure called *CodeGen*[4]. The *CodeGen* procedure takes two arguments, the functor of a predicate and a definitional tree or subtree of the function. *Codegen's* output is our desired needed narrowing code mapped into the Prolog language.

The following terminology is useful within the explanation of Codegen. A shallow constructor term is a term $c(X_1, \ldots, X_k)$ where c is a constructor, X_i are distinct variables and k is the arity of c. A constructor term enumeration for a sort s with m constructors is an enumeration $c_1, \ldots, c_m$ such that each c_i is a shallow constructor term with fresh variables[4]. Codegen generates many new functor symbols, so it will be convenient to enumerate them as f_0, f_1, f_2, ... where f_0 represents the original function name. Anytime a new functor is required the index of the last element in the list will be increased by one and that symbol is placed on to the end of the list. In this way we can generate as many new functors as is required for a particular tree.

The behavior of Codegen depends on whether the input tree is a branch, rule, or exempt node. To simplify the explanation we will discuss each case

independently. The initial call to Codegen contains arguments f_0, the name of the function for which code is being generated, and a definitional tree for f_0.

Case 1: The case where **T is a branch** with call Codegen(f_h,T).

Let $f(t_1, \ldots, t_n)$ be the pattern of the branch. Since it is a branch node there exists an inductive position call it t_i. Let s be the sort of t_i with a constructor term enumeration $c_1, \ldots, c_j$. The call to Codegen will generate j+2 clauses where the heads and portions of the bodies are shown below. The variables Y and H are fresh variables.

0 $f_h(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n,Y)$:- var(X), !, ...

1 $f_h(t_1,\ldots,t_{i-1},c_1,t_{i+1},\ldots,t_n,Y)$:- !, ...

...

j $f_h(t_1,\ldots,t_{i-1},c_j,t_{i+1},\ldots,t_n,Y)$:- !, ...

j+1 $f_h(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n,Y)$:- ftp_s(X,H),...

Each of these j+2 clauses are mutually exclusive. The clause numbered 0 will be executed when the inductive position contains a variable, the clauses numbered 1 to j will be executed when the inductive position contains a term rooted by the appropriate constructor and the j+1 clause will be executed

when the inductive position contains an operation-rooted term.

The bodies of clauses labelled 1 to j will be of the following form where h_k is the next unused index from the functor list .

$$f_h(t_1,\ldots,t_{i-1},c_k,t_{i+1},\ldots,t_n,Y):-$$
$$!,\ f_{h_k}(t_1,\ldots,t_{i-1},c_k,t_{i+1},\ldots,t_n,Y).$$

The definition of predicate f_{h_k} will be generated on a recursive call to Codegen, $\texttt{Codegen}(f_{h_k}, T_k)$, where T_k is the subtree of T whose pattern is a variant of $f_0(t_1,\ldots,t_{i-1},c_k,t_{i+1},\ldots,t_n)$.

The body of clause 0 is completed as follows where h_0 is the next unused index from the functor list.

$$f_h(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n,Y):-$$
$$\texttt{var(X)},\ !, f_{h_0}(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n,Y).$$

The definition for the predicate f_{h_0} consists of j clauses identical to the 1 to j clauses of f_h but without the cut and with the functor of f_{h_0} instead of functor f_h.

The body of clause j+1 is completed as follows where again $f_{h_{(j+1)}}$ is the next unused index from the functor list.

$$f_h(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n,Y):-$$
$$ftp_s(X,H),\ f_{h_{(j+1)}}(t_1,\ldots,t_{i-1},H,t_{i+1},\ldots,t_n,Y).$$

The definition of predicate $f_{h_{(j+1)}}$ consists of j clauses identical to the 1 to j clauses of f_h with the functor $f_{h_{(j+1)}}$ instead of functor f_h. The ftp_s clause evaluates the operation-rooted term X to head normal form H. Since H is in a needed position and is now in head normal form, further reduction is possible.

Below is typical output from Codegen for a branch node with the function symbol `add` and with add's definitional tree. Both the call and initial output is listed where the non-pictorial representation of the definitional tree is used.

```
call:
Codegen(add, branch(add, [1],
    [rule(add(zero,Y)), rule(add(succ(X),Y))])

initial output:
    0 add(A,B,C):- var(A), !, add_1(A,B,C).
    1 add(zero,B,C):- !, add_2(zero,B,C).
```

```
2 add(succ(A),B,C):- !, add_3(succ(A),B,C).
```

```
3 add(A,B,C):- ftp_nat(A,H), add_4(H,B,C).
```

In this paper I will typically use more expressive symbols for the new functors being generated such as `add_10` and `add_1s` instead of `add_2` and `add_3` respectively. In the functors `add_10` and `add_1s` the 1 represents the inductive position and `0/s` represents the content of that position.

Case 2: The case where **T is a rule** and the call to Codegen is of the form `Codegen(`f_h`,`T_h`)`. Let $f(t_1, \ldots, t_n)$ be the pattern of the rule node and L=>R be the corresponding rewrite rule where L and $f(t_1, \ldots, t_n)$ are variants. There are three subcases: R is a variable, R is a constructor-rooted term, or R is an operation-rooted term.

2.1 Suppose R is a variable, say X. Since the rewrite rules are left-linear there exists a unique occurrence of X in $t_1, \ldots, t_n$. Let s be the sort of X with j constructors and $c_1, \ldots, c_j$ represent a shallow constructor term enumeration for sort s with fresh variables. The following j+2 clauses are generated.

```
0   f_h(t_1, . . .,t_n,X):- var(X), !, f_h0(t_1, . . .,t_n,X).
```

```
1   f_h(t_1', . . .,t_n',c_1):- !.
```

```
...
j    f_h(t_1', . . .,t_n',c_j):- !.
j+1  f_h(t_1, . . .,t_n,Y):- ftp_s(X,Y).
```

The predicate $f_{h_0}(t_1, \ldots, t_n, X)$ is defined by the clauses identical to clauses numbered 1 to j for f_h except for the cut and the application of the new functor f_{h_0}. The arguments $t_1', \ldots, t_n'$ are identical to $t_1, \ldots, t_n$ except for the instantiation of X with an appropriate member of the constructor term enumeration. The predicate ftp_s, as described previously, belongs to a family of clauses that evaluates an operation-rooted term X of sort s to head normal form.

To continue with the example of `add` from Case 1, a call to Codegen with functor `add_10` and definitional tree consisting of rule(add(zero,Y)) with corresponding rewrite rule `add(zero,Y)=>Y` will generate the following clauses.

```
add_10(zero,B,C):- var(B), !, add_10_2v(zero,B,C).
add_10(zero, zero, zero):- !.
add_10(zero, succ(B), succ(B)):- !.
add_10(zero,B,Y):- ftp_nat(B,Y).
```

2.2 Suppose R is a constructor-rooted term. The following single clause is generated

$f_h(t_1, \ldots, t_n, R)$.

An example is in generation of the definition of `add_1s` from above with corresponding rewrite rule `add(succ(A),B)=>succ(add(A,B))`. The following clause is our result.

```
add_1s(succ(A),B,succ(add(A,B)).
```

2.3 Suppose R is an operation-rooted term. Let $R=g(W_1, \ldots, W_n)$. Then the following single clause is generated where W is a fresh variable.

$f_h(t_1, \ldots, t_n, W)$:- $g(W_1, \ldots, W_n, W)$.

An example of this case is in the generation of the clauses for `leq_1s` and with the rewrite rule `leq(succ(A),succ(B)) => leq(A,B)`. The following clause is generated:

```
leq_1s(succ(A),succ(B),C):- leq(A,B,C).
```

Case3: The case where **T is exempt** with call `Codegen(`f_h`,T)`.

Since T is exempt there is no corresponding rewrite rule for the pattern $f(t_1, \ldots, t_n)$. Thus no needed narrowing steps are possible and we generate the following clause.

```
fh(t1,...,tn,_):- !, fail.
```

V. Optimizations

The procedure Codegen described in the last section generates code that allows functions to be evaluated lazily. The code that is generated has several known optimizations [4] which can be implemented. This section describes several of the optimizations and discusses the implementation aspects of the optimizations.

Below I give the rewrite rules used in following subsections. The rules and code are presented in a Prolog-like syntax.

```
R1: add(zero,Y)=>Y.
R2: add(succ(X),Y) => succ(add(X,Y)).
```

```
R3: leq(zero,_)=>true.

R4: leq(succ(_),zero) => false.

R5: leq(succ(X),succ(Y)) => leq(X,Y).
```

For convenience I also give parts of the unoptimized code output from *Codegen* for each of these functions.

```
C1: add(X,Y,Z):- var(X),!, add_1v(X,Y,Z).

C2: add(zero,Y,Z):- !, add_10(zero,Y,Z).

C3: add(succ(x),Y,Z):- !, add_1s(succ(X),Y,Z).

C4: add(X,Y,Z):- ftp_nat(X,H), add_1h(H,Y.Z).

C5: leq(X,Y,Z):- var(X),!, leq_1v(X,Y,Z).

C6: leq(zero,Y,Z):- !, leq_10(zero,Y,Z).

C7: leq(succ(X),Y,Z):- !, leq_1s(succ(X),Y,Z).

C8: leq(X,Y,Z):- ftp_nat(X,H), leq_1h(H,Y,Z).
```

1. Rule Call Elimination

When only one clause is generated by Codegen for a rule node, the definition of the clause can be merged with its call.

An example of this optimization is present in the generation of the `add` clauses. Using the rewrite rules given above for `add`, and on the call

```
Codegen(add_1s,rule(add(succ(A),B))
```

the following code is generated:

```
add_1s(succ(A),B,succ(add(A,B)).
```

This predicate, `add_1s`, is called from only one place in the code, in `C3` above, so it is possible to merge the definition with the call in the following way.

```
C3: add(succ(A),B,succ(add(A,B)):- !.
```

This particular optimization is not implemented at the current time. To implement this optimization requires the knowledge at the time of generating the branch code that the corresponding rewrite rule has a right hand side that is constructor rooted.

2. Leading Constant/Constructor Elimination

In the generation of the clauses, often times there are a family of clauses that contain constructors or constants as arguments. This constructor or constant

can be eliminated from the definition if all calls are appropriately modified.

An example of this optimization for constants is present in the `add` clauses given above. A constructor of natural numbers is the constant zero. In the generation of the constructor clauses in codegen there is a predicate `add_10(zero,B,C)`. All the clauses for `add_10` contain the information about constant zero. This may be eliminated from all clauses. So in `C2` above and in the definition of `C2`'s auxiliary predicate we can code

```
C2:add(zero,B,C):- !, add_10(B,C).
    add_10(B,C):- var(B)...
    add_10(zero,C):-...
    add_10(succ(B),C):-...
    add_10(B,C):- ftp_nat(B,H),...
```

This is implemented in the equality but not in the definitional tree code. The implementation is fairly simple. I build the clause body separate from the head. So upon building the clause body where the constant first appears I build it with one less argument. Since this body is reused and not rebuilt the alteration affects all places which use the constant.

An example of this optimization when applied to a leading constructor is

visible in clause C7 and in the definition of its auxiliary predicate leq_1s. The clauses before the optimization is applied appear below.

```
C7: leq(succ(X),Y,Z) :- !, leq_1s(succ(X),Y,Z).
    leq_1s(succ(X),Y,Z) :- var(Y), !, leq_1s_2v(succ(X),Y,Z).
    leq_1s(succ(X),zero,Z) :- !, leq_1s_20(succ(X),zero,Z).
    leq_1s(succ(X),succ(Y),Z) :- !,
                                 leq_1s_2s(succ(X),succ(Y),Z).
    leq_1s(succ(X),Y,Z) :- ftp_nat(Y,H),
                                      leq_1s_2h(succ(X),H,Z).
```

Notice that each one is headed by the constructor succ of the natural numbers. The use of this optimization transforms these into the following clauses assuming all other auxiliary predicates are transformed suitably.

```
    leq(succ(X),Y,Z) :- !, leq_1s(X,Y,Z).
    leq_1s(X,Y,Z) :- var(Y), !, leq_1s_2v(X,Y,Z).
    leq_1s(X,zero,Z) :- !, leq_1s_20(X,zero,Z).
    leq_1s(X,succ(Y),Z) :- !, leq_1s_2s(X,succ(Y),Z).
    leq_1s(X,Y,Z) :- ftp_nat(Y,H), leq_1s_2h(X,H,Z).
```

This is not implemented in the compiler. This implementation may be trickier

than the others. The information of which constructor was eliminated may be needed in further reductions. So this particular information must be carried somehow. An example where the constructor information needs to be retained is in the following clause of an equality code.

```
sen_ls(X,Y):- var(Y),!, Y=succ(Z), strict_sseq_nat(X,Z).
```

Here the original clause head was `sen_ls(succ(X),Y)` and in the body we use the fact that X has head functor `succ`. So the knowledge of which constructor was eliminated must not be forgotten.

3. First Argument Indexing

If the Prolog compiler indexes the functions based on the principal functor of the first argument, we can take advantage of this by rearranging the arguments so the inductive argument appears first. This will gain an advantage in the lookup time of a particular function.

An example is in the `leq_ls` clauses given above. The inductive argument is the second and we can transform them into the following.

```
leq(succ(X),Y,Z) :- !, leq_ls_r(Y,succ(X),Z).
```

```
leq_1s_r(Y,succ(X),Z) :- var(Y), !,leq_1s_2v_r(Y,succ(X),Z).
leq_1s_r(zero,succ(X),Z) :- !, leq_1s_20_r(zero,succ(X),Z).
leq_1s_r(succ(Y),succ(X),Z) :- !,
                                leq_1s_2s_r(succ(Y),succ(X),Z).
leq_1s_r(Y,succ(X),Z) :- ftp_nat(Y,H),
                                        leq_1s_2h_r(H,succ(X),Z).
```

This is not yet implemented into the compiler. This shouldn't be too difficult to implement with some care being given to reorder the arguments if ever a call is made back to the original function. For example below we need to reorder to continue the computation with a call to `leq`.

```
leq_1s_2s_r(succ(Y),X,Z) :- !, leq(X,Y,Z).
```

4. FTP Elimination

For each sort there is a whole family of ftp (function to predicate) clauses. If all operations are known for a particular sort this call can be eliminated by replacing the call by all appropriate operation-rooted patterns.

An example of this is in the leq clauses.

```
leq(X,Y,Z) :- ftp_nat(X,H), leq_1h(H,Y,Z).
```

If it is known that add and mult are the only operations on the natural numbers we can replace this particular clause with the following two clauses.

```
leq(add(X,W),Y,Z) :- !, add(X,W,A), leq_1h(A,Y,Z).
leq(mult(X,W),Y,Z) :- !, mult(X,W,A), leq_1h(A,Y,Z).
```

This particular optimization appears difficult to implement. It requires knowledge of all defined operations of a particular sort at the time of code generation.

5. Eager Discrimination

When a function is defined by a call to another function, the called function's generated code may be integrated into the callee's code.

An example of this is in the definition of `Double`, where I use Goedel syntax.

```
Double(x,y) => Add(x,x,y).
```

Using information about `add` we can integrate the code for `add` into the call by `double` in the following manner. Rule call elimination was also applied in this example and the definitions of the auxiliary predicates have been omitted.

```
double(X,Y):- var(X),!, double_1v(X,Y).

double(zero,zero):- !.

double(succ(X),succ(add(X,succ(X)))):- !.

double(X,Y):- ftp_nat(X,H), double_1h(H,Y).
```

This is not implemented. The implementation is not straightforward. Knowledge of the called function and its optimized code will need to be accessible to have a successful integration of code.

6. Variable Flattening

Rules that call predicates with deep patterns may not invoke all clauses. When the deep argument is a variable we can eliminate clauses that are never called by flattening the subtree and generating the remaining clauses.

A typical example of this is in the definition of Half. Consider the following `Half` rewrite rules:

```
Half(Zero)=>Zero.
```

```
Half(Succ(Zero)) => Zero.
Half(Succ(Succ(y))) => Succ(Half(y)).
```

A call to Codegen will generate among others the following clauses:

```
half(X,Y):- var(X),!, half_1v(X,Y).
half_1v(zero,Y):- half_10(zero,Y).
half_1v(succ(X),Y):- half_1s(succ(X),Y).
```

Now the definition of `half_1s(succ(X),Y)` has four cases: `X` is a variable, `X` is zero, `X` is `succ(Z)`, or `X` is operation-rooted. It is clear that `X` is a variable in this case. Thus we need to execute the subclauses of half_1s(succ(X),Y) where X is a variable. So only one of the four options apply and we can eliminate the others from consideration by integrating the appropriate case of `half_1s(succ(X),Y)` with `half_1v` in the following fashion.

```
half(X,Y):- var(X),!, half_1v(X,Y).
half_1v(zero,Y):- half_10(zero,Y).
half_1v(succ(zero),Y):- half_1s(succ(zero),Y).
half_1v(succ(succ(X)),Y):- half_1ss(succ(succ(X)),Y).
```

This optimization is not implemented. To implement this, we must observe the

pattern given in the rewrite rules and realize we can optimize. Then replacement of the variable with appropriate constructors will generate the desired clauses.

7. Last cut Elimination

The last clause generated by a call to Codegen with a branch yields subclauses whose heads are exclusive clauses. Each clause contains a cut within the body to avoid choice points. The last of these clauses need not contain a cut.

An example of this is

```
add(A,B,C) :- ftp_nat(A,H), add_1h(H,B,C),
add_1h(zero,B,C) :- !, add_10(zero,B,C),
add_1h(succ(A),B,C) :- !, add_1s(succ(A),B,C).
```

The last clause can be rewritten as

```
add_1h(succ(A),B,C) :-  add_1s(succ(A),B,C).
```

This optimization is not implemented. It will be fairly easy to implement.

Testing has shown this particular optimization is not one of key optimizations that produces noticeable savings in time and garbage creation.

VI. Implementation Issues

This section describes the transformation of eager-functional Goedel to lazy-functional Goedel. Topics discussed include obtaining all rewrite rules, detecting overlapping, building definitional trees, generating needed narrowing code, discovering sorts, creating FTP modules, evaluation of functions and issues regarding the corresponding implementations.

In the construction of a definitional tree, all rewrite rules for a given user-defined function must be available simultaneously. This is a new requirement for the compiler. Previously the compiler used a strategy of eager narrowing to evaluate functions where rewrite rules were processed independently [13]. To satisfy this new requirement for the compiler, a file listing the names of all evaluable functions is created during parsing, <Module_name>.ef. From the file, I retrieve names and corresponding arities of evaluable functions. I use this information within the compiler environment to obtain a list of all the rewrite rules for a given evaluable function. One minor omission in the creation of this file is that the signatures of the functions should be included in

this file and not just the names and arities of the functions.

A definitional tree is defined for inductively sequential rewrite systems. So the rewrite rules must be left-linear and non-overlapping. The left-linear check is performed during parsing by the modified parser [14]. I test for overlapping rewrite rules within the merging process during the construction of definitional trees. If overlapping is detected a message is printed to the user and the process of constructing a definitional tree is abandoned for that particular function.

A future project is to force all rewrite rules to be non-overlapping through appropriate conditionals. For example a factorial function defined in Goedel could take on the following definition:

```
MODULE Fact.
IMPORT Integers.
FUNCTION
Fact :Integer -> Integer.
Fact(0)=> 1.
Fact(n)=> n * Fact(n-1) <- n>0.
```

These two rules overlap since `Fact(0)` unifies with the head of both clauses

Thus a message to the user `"***Fact is not inductively sequential**"` is printed. The intent of these rewrite rules is that clause 2 is executed when n>0. It would be convenient if these two clauses were viewed as non-overlapping and a definitional tree built.

From this point forward we will assume the rewrite rules that are processed are inductively sequential. After obtaining all associated rewrite rules for a user-defined function and deciding on a bottom-up strategy to build definitional trees, we can begin the chaining process on the rewrite rules' left-hand sides. One left-hand side of a rule is chosen and two recursive predicates will chain it. The predicate `select_term` returns the first unselected argument from a left-hand side that is not a variable or returns all the arguments if it is a shallow term. The predicate `candidate_to_chain` analyzes this choice and decides whether `select_term` selected a term that was appropriate or if further refinement is required. Remember the idea is to choose a shallow constructor term to replace, but with the limited knowledge of variable, term, or constant. If `select_term` returned all the arguments this is the stopping point of the chain process as all arguments are variables. Otherwise `select_term` will return an argument that might possibly have inner arguments. `Candidate_to_chain` recognizes constants and terms. If it receives a constant it generates a variable and replaces the constant. If it receives a term it calls `select_term` to choose an inner term to replace.

Within `select_term` is built the nondeterminism required for backtracking in case a wrong term is chosen for replacement and therefore yielding chains that cannot be merged.

Consider the call to `select_term` of `add(X,Y)`. The complete term is returned with position=0. This completes the processing of this term.

Consider the call to `select_term` of `leq(X,succ(Y))`. The term `succ(Y)` is returned, position=2, and a call to `candidate_to_chain` is made. `Candidate_to_chain` recognizes `succ(Y)` as a term, calls `select_term` with argument `Y`. `Select_term` returns the term `Y` with position=0. Upon receipt of this information, `candidate_to_chain` replaces the term `succ(Y)` with a generated or fresh variable call it `Genvar_1`. All this information is integrated to obtain a partial chain which takes the following form: `leq(X,succ(Y)), [2], leq(X, Genvar_1)`. Here the new transformed term, `leq(X, Genvar_1)`, is recurred on. The process of chaining will end with this term since it is a shallow term.

After all a given function's rewrite rules' left-hand sides are chained, we are ready to see if they can be merged. The process of merging follows the steps of verifying positions are equal, partitioning into smaller lists and recurring on

these lists. When the merging process is complete a definitional tree is built.

Consider the following rewrite rules and corresponding chains.

```
rules:
    R1: half(zero)=>zero.
    R2: half(succ(zero))=>zero.
    R3: half(succ(succ(zero)))=>succ(half(X)).
chains:
    C1: half(Y),[1],half(zero)
    C2: half(Y),[1],half(succ(Y)),[1,1], half(succ(zero))
    C3: half(Y),[1],half(succ(Y)),[1,1], half(succ(succ(Y)))
```

We first verify the chains all begin with the same position. Here they all have `[1]`. The top part of the tree takes on the form `branch(half(Y),[1],...)`. Then we cut the heads from the chains and partition the new shortened chains into the following smaller lists:

`[half(zero)]`, and

```
[half(succ(Y)),[1,1],half(succ(zero)),
  half(succ(Y)),[1,1],half(succ(succ(Y)))].
```

The first list is a variant of a rule, thus is a rule node. So we grow part of the tree `branch(half(Y),[1],[rule(half(zero)),...])`. We recur on the second list and verify the positions are the equal. This time the positions are

[1,1]. Again, remove the lead two elements of the chains and partition the shortened chains into lists of [half(succ(zero))], [half(succ(succ(Y)))]. These remaining lists are labelled as rule nodes and the finished tree takes on the following form.

```
branch(half(Y),[1],
    [rule(half(zero)),
    branch(half(succ(Y)),[1,1],
        [rule(half(succ(zero))),
        rule(half(succ(succ(Y))))])])
```

Up to this point in the compiler, the eager-functional compiler was intact and coexisted with the definitional tree code. To make progress toward code generation in the compiler we require additional information regarding signatures of various functions. This information was not carried through the eager-version of the compiler and so the two compilers became separated. Much akin to the eager-functional Goedel necessity to carry Rulelist over much of the code , lazy-functional Goedel needed a structure that is referred to as a Sort_tree. Basically I extracted from a Module Descriptor an AVL-tree that contained signature information for functions defined in the current module. Since Sort_tree is an AVL-tree, which the original Goedel compiler uses, there exists methods which can extract the appropriate signature given the function name. With the new knowledge of sorts for functions, coding of

Codegen is now possible.

Codegen has two main predicates, one for branch clauses and one for rule clauses. Since trees are built in a bottom-up method there are no exempt nodes in the definitional trees and thus no need to code clauses for exempt nodes. The predicates that generate the code generally follow the description as provided in the Codegen section with a few subtleties. Let's briefly consider the highlights of the coding process for each the branch case and rule case.

The generation of branch clauses involves construction of clauses for a variable term, for constructor terms, and for an operation-rooted term. The process begins by consulting the lead position term to determine the inductive variable. The Sort_tree is referenced to find the corresponding sort for the given function and inductive position. Then the clause for a variable term is constructed. The new functor generated is saved for definition later.

Continuing with the example of half from above, the inductive position is 1 and the sort is natural. The variable term clause is then constructed.

```
half(X,Y):- var(X),!, half_1v(X,Y).
```

Construction of constructor term clauses involves some foresight. Rather than finding all the appropriate constructors to substitute in place of the inductive variable, we need only observe that the information is already encoded within the definitional tree as either branch or rule nodes. Thus by looking ahead in the definitional tree the appropriate constructor clauses are generated. Also in this constructor clause phase, new functors are created that are saved for reuse later.

Referring to the example of half, all our required information for generation of constructor clauses is present in the definitional tree.

```
branch(half(Y),[1],
    [rule(half(zero)),
    branch(half(succ(Y)),[1,1],
        [rule(half(succ(zero))),rule(half(succ(succ(Y))))])])
```

We generate the following clauses utilizing the information contained in the rule node and branch node.

```
half(zero,Y):- !, half_10(zero,Y).
half(succ(Y),X):- !, half_1s(succ(Y),X).
```

To create the operation-rooted clause, the sort information is required. Sort_tree supplies the information that in position 1 for `half` is a natural number. Thus we generate the operation-rooted clause.

```
half(X,Y):- ftp_nat(X,H), half_1h(H,Y).
```

Both in the phase of the variable and operation-rooted clause generation are new functors that now need definitions. Recall they should be defined as the constructor clauses are defined but with or without cuts and with new functors. Rather than generate the constructor clauses a second and third time, I save the clauses and use a procedure that cuts off the lead functor and pastes in a new functor to obtain the desired definitions. To side step the issue of whether to cut or not, two predicates are encoded which take as arguments a head, body and optional conditional. One predicate prints clauses with a cut and the other prints clauses without a cut. They both rely on a Prolog clause called `portray_clause` which uses a dictionary to name variables appropriately.

The clauses for `half` are now supplemented with the following clauses.

```
half_1v(zero,Y):- half_10(zero,Y).
half_1v(succ(Y),X):- half_1s(succ(Y),X).
```

```
half_1h(zero,Y):- !, half_10(zero,Y).
half_1h(succ(Y),X):- !, half_1s(succ(Y),X).
```

We now recur on the list of trees from this branch, carrying with the subtrees functors generated in the constructor clause phase. In the example of `half` we would call Codegen with functors `(half_10, half_1s)` and with the remaining portion of the tree unprocessed of which they functors correspond to. For example the two calls that generate definitions for the new functors appear below.

```
Codegen(half_10,rule(half(zero)).
Codegen(half_1s,branch(half(succ(Y)),[1,1],
    [rule(half(succ(zero))),rule(half(succ(succ(Y))))])
```

In the rule node clause generation, there are three main predicates depending on whether the right hand side of the corresponding rule is a constructor, variable, or operation-rooted term. To detect if the right hand side is constructor rooted, compare the term with the constructors obtained from <Module_name>.ef file which contains constructor information. The constructor-rooted clause is then built.

The call `Codegen(half_10,rule(half(zero))` is an example where the

corresponding rewrite rule `half(zero)=>zero` is constructor-rooted. The output from Codegen is similar to the following.

```
half_10(zero,zero).
```

The second case for a rule is a variable on the right-hand side. The detection of a variable on the right-hand side is straightforward, but the generation of the corresponding clauses requires additional information. To generate the desired clauses, we need knowledge of the range sort of the defined function and the corresponding constructors. Once the constructors are known the variable occurring on the right-hand side must be replaced with the constructors. In addition the variable must also be found in the left-hand side as well as in the conditional and replaced by the appropriate constructor. From here the generation of clauses is very similar to that described in the branch section.

This case is not demonstrated in the `half` rewrite rules, so let's reconsider the rule `add(zero,Y)=>Y` with constructors of Y `zero` and `succ`. The clauses generated under these conditions are shown below.

```
add_10(zero,B,C):- var(B), !, add_10_2v(zero,B,C).
```

```
        add_10_2v(zero,zero,zero).
        add_10_2v(zero, succ(B), succ(B)).
    add_10(zero, zero, zero):- !.
    add_10(zero, succ(B), succ(B)):- !.
    add_10(zero,B,Y):- ftp_nat(B,Y).
```

If neither of the two previous cases is satisfied the right-hand side is assumed to be an operation-rooted term. Recall from the Codegen description, if the right-hand side is an operation-rooted term we just expand the arity by one with a variable through which to return an answer. This works as expected with user-defined functions, but for system-defined functions additional coding is required.

Let's consider two examples. Suppose we have an operation `sumlist` that can be defined via the built in operation + or a user-defined operation `add`. The two rewrite rules may be written as follows.

```
R1: sumlist_int([])=>0                    %sytem defined +
R2: sumlist_int([A|B])=> A + sumlist_int(B)
R3: sumlist_nat([])=>zero.                %user defined add
R4: sumlist_nat([A|B])=>add(A,sumlist_nat(B)).
```

Part of the generated code for the rules that are operation-rooted follows:

```
R2': sumlist_int([A|B],C):- Integers.+.(A,sumlist_int(B),C).
R4': sumlist_nat([A|B],C):-  add(A, sumlist_nat(B),C).
```

Now with a goal `sumlist_nat([add(succ(zero),succ(zero))])=x` we use `R4'` and call

```
add(add(succ(zero),succ(zero)),sumlist_nat([]),C).
```

This will execute the operation-rooted clause of ftp_nat within the add clauses. The call to predicate ftp_nat will evaluate `add(succ(zero),succ(zero))`. Thus we eventually end up with answer `x=succ(succ(zero))`.

With goal `sumlist_int([1+1])=x`, we call `R2'` and hope to prove `Integers.+.(1+1,sumlist_int([]),C)`. Since + is a system-defined function it is assumed that + is called with all functional arguments already evaluated. Clearly this is not the case in this situation. To compensate for this fact additional coding was required which defined an equality for integers and an ftp for integers. This code is generated for system-defined functions so that arguments that are needed for evaluation of the goal are evaluated before calling the built-in function. Thus we are able to utilize built-in functions within modules.

Within the code generated by Codegen is a series of calls to predicates headed by the functor ftp_sort. The purpose of the ftp_sort clause is to evaluate a functional-rooted argument as it occurs in a needed position. A clause ftp_sort is defined for each function which has a definitional tree. After construction of a definitional tree for a given function, the Sort_tree is consulted to determine the range of the function and a clause for that given function is constructed. For example an `add` function that requires two arguments with range of natural numbers would have the following clause generated.

```
FTP:  ftp_nat(add(A,B),C):- add(A,B,C).
```

The lead symbol `'FTP'` is the module name where the clause resides. The `FTP` clauses are generated during compilation of the module where the function is defined, but we need all the clauses of ftp_sort to reside in a FTP module where they are accessible to functions with operation-rooted arguments having range of the given sort. For example there might be a ftp_nat for `add`, `times`, and `half` where each function is defined in its own module, but all calls to ftp_nat would reference the `FTP` module. Suppose we had the goal `half(add(zero,succ(zero)))=x`, to solve, since half's argument is functional-rooted, we consult the clause of ftp_nat for add. We

reduce the argument to head normal form and then return to computation in the half clauses.

There was considerable effort to define a predicate, ftp_sort, in many files and have them coexist in the same module. The solution is to declare a predicate as multifile. Thus during the parsing phase of files which declare bases `nat` and `bool`, clauses of the following form are generated.

```
:-module('FTP',[]).
:-multifile ftp_nat/2.
:-multifile ftp_bool/2.
```

These clauses are written into a new file called <Module_name>_ftp.pl This allows for the definition of `ftp_nat` or `ftp_bool` to be spread across many modules, but for all their clauses to coexist in the FTP module.

There are associated problems with the multifile declaration. One problem was obtaining the knowledge of base information to make the appropriate declaration(s). This information was gleaned from the infamous <Modname>.ef file which is created during parsing. Other problems include where to generate the multifile declarations and where to load the associated information. The generation of clauses is accomplished in the code where the

<Module_name>.ef file is created and we load the file prior to loading the system-defined files. The major problem encountered was resolved only through communication with M. Carlsson [7]. Prolog gives messages to the user when a module is redefined. Thus when a file is loaded and then a new file is loaded Prolog prints a message about redefinition of the FTP module to the user. The user should not be aware of an FTP module because it is a creation of the compiler. Also the FTP module should be redefined when a new file is loaded. I tried many ideas, but it took personal communication from M. Carlsson to accomplish the feat of turning off the appropriate warnings of module redefinition.

The last major change required to perform needed narrowing is a supplement to Goedel's evaluation mechanism. Predicates which perform a semi-strict equality evaluation on functional expressions was added to the existing compiler environment. The idea is when a user-defined function occurs as an argument to a predicate only evaluate the function as much as necessary to obtain a normal form for the predicate to compute with. To generate the semi-strict equality code, the <Module_name>.ef file is consulted to obtain a list of all constructors. All the constructors of a given sort are bundled together and a method similar to the definitional tree code generation constructs clauses for a variable term, constructor terms and operation-rooted terms.

For example consider the natural number with constructors `zero` and `succ`. The following code for equality is generated.

```
sseq_nat(A,B):- var(A),!, sseq_nat_1v(A,B).
    sseq_nat_1v(A,B):- var(B), !, A=B.
    sseq_nat_1v(zero,B):- sseq_nat_10(zero,B).
    sseq_nat_1v(succ(A),B):- sseq_nat_1s(succ(A),B).
sseq_nat(zero,B):- !, sseq_nat_10(zero,B).
    sseq_nat_10(zero,A):- var(A), !, A=zero.
    sseq_nat_10(zero,zero):- !.
    sseq_nat_10(zero,succ(_)):- !,fail.
    sseq_nat_10(zero,A):- ftp_nat(A,zero).
sseq_nat(succ(A),B):- !, sseq_nat_1s(succ(A),B).
    sseq_nat_1s(succ(A),B):-
                        var(B), !,B=succ(C), sseq_nat(A,C).
    sseq_nat_1s(succ(_),zero):- !,fail.
    sseq_nat_1s(succ(A),succ(B)):- !,sseq_nat(A,B).
    sseq_nat_1s(succ(A),B):-
                         ftp_nat(B,succ(C)), sseq_nat(A,C).
sseq_nat(A,B):- ftp_nat(A,H), sseq_nat_1h(H,B).
    sseq_nat_1h(zero,B):- !, sseq_nat_10(zero,B).
```

```
sseq_nat_1h(succ(A),B):- !, sseq_nat_1s(succ(A),B).
```

The equality created above works well when sort information is known for arguments of constructors, but there are problems with polymorphic types. For example a function on lists can have the following signature.

```
FUNCTION                append:List(a)->List(a).
```

It is not known at compile time what the argument sort of List is. The solution to this difficulty is to create an untyped equality. The untyped equality becomes a typed equality based upon the arguments it is called with. For instance, part of the equality for lists has the following code:

```
sseq_List([A|B], [C|D]):- untyped_equality(A,B),
                                        sseq_List(B,D).
```

Now whether I have a goal of `append([1,2],[3])=append([1],[1+1,3])` or `append([succ(zero)], [succ(succ(zero))])= append([succ(zero)], [add(succ(zero),succ(zero))])` based on the arguments of `1,2,3` or `zero, succ(zero)` the untyped equality will call the right semi-strict equality of `sseq_Integer` or `sseq_nat`.

VII. Functional Logic Languages

In this section, we briefly compare two other functional-logic languages Escher and Curry with lazy functional-logic Goedel.

1. Escher

"Escher is a declarative, general-purpose programming language which integrates the best features of both functional and logic programming languages" [13]. For comparison with lazy-functional Goedel we will consider aspects of the operational semantics and some language features of Escher.

Escher uses residuation to evaluate functions. The idea behind residuation is to delay the evaluation and unification of functions until the arguments are instantiated to ground terms. Although residuation preserves the deterministic nature of functions it is incomplete. Consider the following definition for a function `Rev` which reverses a list where Append is system defined within the Lists module:

```
MODULE Rev.

IMPORT Lists.
```

```
FUNCTION  Rev:List(a) * List(a)  -> Boolean.

MODE(NONVAR,_).

Rev([],w) => w=[].

Rev([x|xs],w) =>  Append(Rev(xs),[x],w).
```

A call of Rev(x, [A,B,C]) which requests the list that is the reverse of the list [A,B,C] would flounder, although it is clear that x=[C,B,A].

A similar program in lazy-functional Goedel does not suffer this same affliction. Below is the same program written in Goedel syntax, where `Lists` is a system defined module and `App` is a user defined module which contains a definition of a functional append - `Appendf`.

```
MODULE Rev.

IMPORT Lists, App.

FUNCTION Rev: List(a) -> List(a).


Rev([]) => [].

Rev([x|xs]) => Appendf(Rev(xs),[x]).
```

Escher considers predicates as boolean functions and it distinguishes between constructors and functions. Thus using the predicate example of

parent where `Mother` and `Father` are appropriately defined boolean functions, the following is the syntax for Escher:

```
MODULE Relative.
CONSTRUCT Person/0.
FUNCTION Mother: Person * Person -> Boolean.
FUNCTION Father: Person * Person -> Boolean.
FUNCTION Parent: Person * Person  -> Boolean.

Parent(x,y) => Mother(x,y) \/ Father(x,y).
```

Here the intended interpretation by Escher is if Parent succeeds it maps to `True` and if Parent fails it maps to `False`. Thus Escher does not fail as in traditional logic languages but returns `False`. As a result of Escher's dependency on boolean functions, it has a built-in module `Booleans` which defines = and many other useful boolean operators.

Goedel is a logic language with built in facilities to handle predicate definitions. The same example of parent has the following syntax, where `Mother` and `Father` are appropriately defined functions.

```
Parent(x,Mother(x)).
```

```
Parent(x,Father(x)).
```

Escher has three features which Goedel does not: it contains higher-order functions, declarative input/output, and local variable definitions. The well-known map function from functional languages can be implemented. Also Escher's input/output is performed via a suitable abstract data type of the "world" and then restricting the operations on this type. This is called monadic I/O. As is common in functional languages with the `let` construct Escher provides a local definition in the form of `E where F`. The idea is that some extra computation is prevented when used appropriately. Following is an example given in the Escher report.

```
MODULE  LocalDefinition
IMPORT  Integers, Lists.
FUNCTION  Halve:  List(a) -> (List(a) * List(a)).
Halve(x) => <Take(half,x),Drop(half,x)>
                              WHERE half=Length(x) Div 2.
```

A goal of Halve([6, 28, 96, 32, 64]) reduces to the answer <[6,28], [96, 32, 64]>.

Escher contains many similarities to Goedel. First is the module system, it is almost identical to the Goedel module system with portions labelled `EXPORT`,

IMPORT, and LOCAL. Escher has meta-programming facilities via an abstract data type PROGRAM which again mimicks the Goedel facilities. Escher has a polymorphic type system and boasts lazy evaluation. Both of these concepts are demonstrated in the example below borrowed from the Escher report.

```
MODULE  Lazy.
IMPORT  Lists.
FUNCTION  First:  Integer * List(a) -> List(a).
  First(n,x)=>  IF n=0 THEN []
    ELSE [Head(x)|First(n-1, Tail(x))].
FUNCTION  From:  Integer -> List(Integer).
  From(n)=>  [n|From(n+1)].
```

A goal of `First(4,From(2))` reduces to the answer `[2, 3, 4, 5]`.

I am not aware of a complete implementation for Escher.

2. Curry

The functional logic language Curry is an attempt to "combine the best ideas of existing declarative languages" [11]. This section briefly discusses and compares the functional logic language Curry and lazy-functional Goedel.

The emphasis of this discussion is on aspects of the operational semantics and language features.

In an attempt to combine features of many languages, Curry's operational semantics are somewhat diverse. Curry's functional evaluation is based on a combination of narrowing (lazy or eager) and residuation, as well as a choice of breadth or depth first search strategies. Residuation is a computation strategy which delays function calls until they can be evaluated deterministically, while narrowing is a computation strategy which uses unification and computes nondeterministically. Curry allows the user to choose the evaluation strategy by specifying evaluation restrictions. If the user does not specify evaluation restrictions Curry chooses a strategy which performs according to the following criteria:

1. If there exists a solution to a goal, the solution is computed
2. If an expression is reducible to a value, Curry computes the value.

The lazy-functional extension of Goedel has an evaluation strategy for functions and a search strategy for predicates. Functional evaluation is performed by the strategy of needed narrowing and predicates use resolution (with modifications if an argument is a functional call). Goedel is mapped to Prolog, thus a depth first search strategy is employed. As a consequence of

the depth first search strategy, an existing solution may not be computed, but if an expression is reducible to a value Goedel computes the value.

The language features of Curry and functional Goedel have similarities and differences. Within the syntax of the language, Curry distinguishes between data constructors and user-defined functions. Curry introduces the symbols 'datatype' and 'function' to denote constructors and functions, respectively. Thus the function `Leq` used earlier in this paper would have the following format in Curry:

```
datatype nat = 0 | succ nat
datatype boolean = true | false
function leq: nat -> nat -> boolean

    leq 0 N = true
    leq (succ M) 0 = false
    leq (succ M) (succ N) = leq M N
```

Both functional extensions of Goedel have no distinction between constructors and user-defined functions for compatibility with original Goedel. Thus the same code from above would appear in a file in the following format:

```
MODULE  LEQ.
BASE Nat, Boolean.
CONSTANT Zero : Nat;
    True, False: Boolean.
FUNCTION Succ : Nat -> Nat;          %constructor
    Leq: Nat * Nat -> Boolean.

Leq(Zero,n) => True.
Leq(Succ(m),Zero) => False.
Leq(Succ(m),Succ(n)) => Leq(m,n) .
```

Now let's compare the restrictions on the rewrite rules of the two languages. Both languages have a requirement of left-linearity on rewrite rules and both languages allow extra variables in the condition which are not present in the left-hand side of a rewrite rule. The two languages differ in the aspect of the strength of non-overlapping requirements. Whereas functional Goedel requires rewrite rules to non-overlap, Curry requires non-ambiguity of rewrite rules. Here the difference is that Curry allows the rules to overlap as long as upon rewriting the same result is obtained.

Curry does not have predicates, but allows boolean functions. So a predicate

of `parent` may have the following definition where `father` and `mother` are appropriately defined functions:

```
parent X Y  = true  <= Y = father X | Y = mother X.
```

Another difference worth noting is Curry's higher order functions. Thus, for example, Curry can implement the well-known map function. Functional Goedel is equipped to evaluate first-order functions.

As of this writing I am not aware of an existing implementation for Curry.

VIII. Examples

Throughout this paper different functional evaluation strategies are mentioned. Let's briefly revisit each strategy through examples and determine for which goals each evaluation technique is able to compute solutions.

Before proceeding to the examples, let's briefly recall the evaluation strategies which we will compare. The residuation principle maintains the deterministic

nature of functions by delaying the unification process until arguments are sufficiently instantiated. Eager narrowing with the strategy of leftmost innermost evaluates the leftmost innermost reducible expression first. Lazy narrowing via the strategy of needed narrowing selects a needed position to reduce first. Leftmost innermost narrowing and residuation are incomplete evaluation techniques whereas needed narrowing is a complete and optimal evaluation strategy with respect to inductively sequential systems

The following examples are presented in a Goedel-like syntax.

```
FUNCTION
    append: List(Integer)* List(Integer)-> List(Integer).
append([],l)=>l.
append([x|xs],l)=> [x|append(xs,l)].
```

A goal of `append([3,4],[5,6])=x` produces solution `x=[3,4,5,6]` regardless of the evaluation strategy utilized.

A goal of `append(l,[5,6])=[3,4,5,6]` will be delayed when residuation is the evaluation strategy. The correct solution of `l=[3,4]` is returned whether innermost narrowing or needed narrowing is the evaluation strategy.

Now let's consider an example where eager narrowing does not perform as a programmer would hope.

```
FUNCTION:  First: Nat* List(Nat)->List(Nat).
FUNCTION:  From: Nat->List(Nat).

First(Zero,_)=>[].
First(Succ(y),[x|xs])=>[x|(First(y),xs)].
From(n)=>[n|From(Succ(n)].
```

A goal of `First(Succ(Succ(Succ(Zero))),From(Zero))=x` under the evaluation strategy of eager narrowing will not terminate, it will loop forever on the evaluation of `From(Zero)`. With needed narrowing and lazy-residuation the answer of `x=[Zero,Succ(Zero),Succ(Succ(Zero)]` is returned. Note: if residuation is not coded to evaluate lazily it suffers the same affliction as eager narrowing does in this case.

The next example is a depth-first state-transition search framework for solving a problem similar to the one on page 285 of Sterling and Shapiro, The Art of Prolog. It is used to solve a version of the wolf-goat-cabbage problem. It is decomposed into two modules a Search module and a WolfGoatCabbage module.

```
%WolfGoatCabbage export file

EXPORT      WolfGoatCabbage.

IMPORT      Lists.

BASE        Object, State, Move.

CONSTANT    Farmer, Wolf, Goat, Cabbage : Object;

            Initial:State.

FUNCTION    St: List(Object) * List(Object) -> State;

            LeftToRight: List(Object) -> Move;

            RightToLeft: List(Object) -> Move;

            Diff: List(Object)*List(Object)->List(Object);

            Union: List(Object)*List(Object)->List(Object);

            ApplyMove: Move * State -> State.

PREDICATE   Final: State;

            Applicable: Move * State;

            Legal: State.


%WolfGoatCabbage local file

MODULE WolfGoatCabbage.

%CONSTANT  Initial:  State.- all 4 start on the left bank

Initial=> St([Farmer,Wolf,Goat,Cabbage],[]).
```

```
%PREDICATE Final:  State.-finish when all on the right bank
Final(St([],[_,_,_,_])).

%PREDICATE Applicable: Move * State.-finding a move to make
Applicable(LeftToRight([Farmer]),St(left,_))<-
        Member(Farmer,left).       %move farmer to right bank
Applicable(RightToLeft([Farmer]),St(_,right))<-
        Member(Farmer,right).      %move farmer to left bank
%find a passenger to travel with Farmer
Applicable(LeftToRight([Farmer,x]), St(left,_))<-
       Member(Farmer,left) & Member(x,left).
Applicable(RightToLeft([Farmer,x]),St(_,right))<-
       Member(Farmer,right) & Member(x,right).

%FUNCTION  ApplyMove: Move * State -> State.
%move passengers to opposite bank
ApplyMove(LeftToRight(cargo), St(left,right))=>
       St(Diff(left,cargo),Union(right,cargo)).
ApplyMove(RightToLeft(cargo),St(left,right))=>
       St(Union(left,cargo),Diff(right,cargo)).
```

```
%PREDICATE Legal: State.
Legal(St(left,right))<- ~Illegal(left) & ~Illegal(right).

PREDICATE  Illegal: List(Object).
%Can't leave goat with cabbage or wolf with goat.
Illegal(bank)<- ~Member(Farmer,bank) &
        ((Member(Goat,bank) & Member(Cabbage,bank)) \/
        (Member(Wolf,bank) & Member(Goat,bank))).

%FUNCTION Union: List(Object) * List(Object) -> List(Object)
%union of two lists of objects
Union([],y)=>y.
Union([a|b],y)=>z<-IF Member(a,y) THEN z=Union(b,y)
                                  ELSE z=Cons(a,Union(b,y)).

%FUNCTION  Diff: List(Object) * List(Object) -> List(Object)
%difference of two lists of objects
Diff([],y)=>[].
Diff([a|b],y)=>z<-IF Member(a,y) THEN z=Diff(b,y)
                                  ELSE z=Cons(a,Diff(b,y)).
```

```
%Search file requires only a local part
MODULE Search.
IMPORT     Lists, WolfGoatCabbage.

PREDICATE  Solve:  State * List(State) * List(Move).
Solve(current_state,_,[]) <- Final(current_state).
Solve(current_state, history, [move|moves]) <-
       Applicable(move,current_state) & %find a move to make
       new_state=ApplyMove(move,current_state) &  %make move
       Legal(new_state) &  %will move finish in legal state?
       NoLoops(new_state, history) &     %ensure not looping
       Solve(new_state, [new_state|history], moves).

PREDICATE  NoLoops: State * List(State).
NoLoops(_,[]).
NoLoops(state, [first_state|rest]) <-
       state~=first_state &   %ensure move isn't a repeat
       NoLoops(state, rest).

PREDICATE  Run: List(Move).
Run(moves)<- Solve(Initial,[Initial],moves).
```

All three evaluation strategies when given the goal of `Run(x).` will produce an initial answer of

```
x=[LeftToRight(Farmer,Goat), RightToLeft(Farmer),
LeftToRight(Farmer,Wolf), RightToLeft(Farmer,Goat),
LeftToRight(Farmer,Cabbage), RightToLeft(Farmer),
LeftToRight(Farmer), RightToLeft(Farmer,Wolf),
LeftToRight(Farmer,Goat), RightToLeft(Farmer,Cabbage),
LeftToRight(Farmer,Cabbage), RightToLeft(Farmer,Goat),
LeftToRight(Farmer,Wolf), RightToLeft(Farmer),
LeftToRight(Farmer,Goat)].
```

If this game is coded such that the computation of the search space is separated from the evaluation of the goals, a lazy evaluation strategy is the clear winner in terms of performance.

IX. Conclusion

When a project is completed it is always beneficial to point out the key

elements which helped bring the project to a successful conclusion, and to pinpoint those issues which require further efforts. In the following I give a brief summary of both of these aspects.

Success of a project depends on many factors, but often there are a few that are pivotal. Some of the key factors that ensured success of this project include a prior implementation of functional Goedel, the ease of implementing the bottom-up method within Goedel's compiler, and familiarity with the theory of Codegen before attempting to implement it within the compiler environment.

The project also raised issues which require further study or work. Some of these include implementing optimizations, researching polymorphic functions, considering implementation of sets, and examining the interaction of needed narrowing evaluation and conditionals.

In conclusion, this thesis and associated work represents one of the first implementations of the evaluation strategy of needed narrowing for functions within a functional-logic language. This thesis demonstrates that a functional-logic language that evaluates functions by "need" is feasible and provides a stepping stone for many more functional-logic languages to come.

Appendix A

The following are some typical rewrite rules, chains generated for the rules, corresponding definitional trees, and needed narrowing code. The rules are represented in Goedel syntax, whereas the code is represented in Prolog syntax.

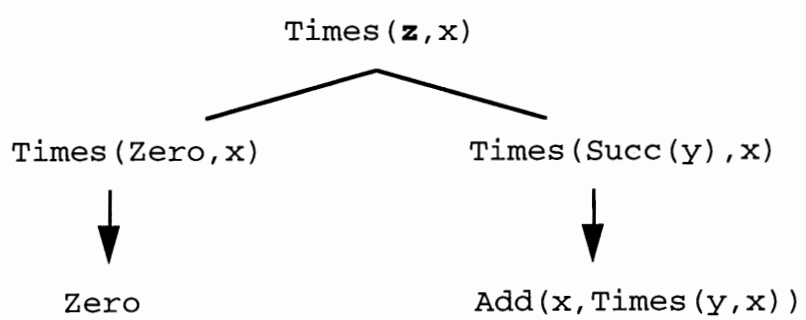
Rules:

```
Times(Zero,y)=>Zero.

Times(Succ(x),y)=>Add(y,Times(x,y))
```

Chains:

```
Times(x,y),[1],Times(Zero,y)

Times(x,y),[1],Times(Succ(x),y)
```

Tree:



Code generated:

```
times(X,Y,Z):- var(X), !, times_1v(X,Y,Z).    %var clause

   times_1v(zero,Y,Z):- times_10(zero,Y,Z)..

   times_1v(succ(X),Y,Z):- times_1s(succ(X),Y,Z).

times(zero,Y,Z):- !, times_10(zero,Y,Z).      %const. clause

   times_10(zero,_,zero).

times(succ(X),Y,Z):- !, times_1s(succ(X),Y,Z).        %const.

   times_1s(succ(X),Y,Z):- add(Y,times(X,Y),Z).

times(X,Y,Z):- ftp_nat(X,H), times_1h(H,Y,Z).         %ftp

   times_1h(zero,Y,Z):- !,times_10(zero,Y,Z)..

   times_1h(succ(X),Y,Z):- !,times_1s(succ(X),Y,Z).
```

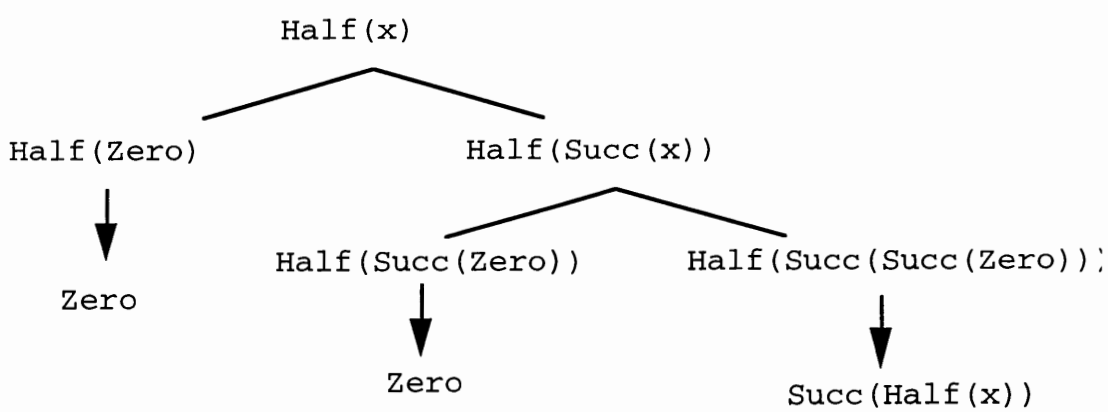
Rules:

```
Half(Zero)=>Zero.

Half(Succ(Zero))=>Zero.

Half(Succ(Succ(x)))=>Succ(Half(x)).
```

Chains:

```
Half(x),[1],Half(Zero)

Half(x),[1],Half(Succ(x)),[1,1],Half(Succ(Zero))

Half(x),[1],Half(Succ(x)),[1,1],Half(Succ(Succ(x)))
```

Tree:



Code generated:

```
half(X,Y):- var(X), !, half_1v(X,Y).    %var clause

   half_1v(zero,Y):-  half_10(zero,Y).

   half_1v(succ(X),Y):- half_1s(succ(X),Y).

half(zero,Y):- !, half_10(zero,Y).      %const. clause
```

```
    half_10(zero,zero).
half(succ(X),Y):- !, half_1s(succ(X),Y).      %const clause
    half_1s(succ(X),Y):- var(X), !, half_1sv(succ(X),Y).
    half_1sv(succ(zero),Y) :- half_1s0(succ(zero),Y).
    half_1sv(succ(succ(X)),Y) :- half_1ss(succ(succ(X)),Y).
half_1s(succ(zero),Y):- !, half_1s0(succ(zero),Y).
    half_1s0(succ(zero),zero).
    half_1s(succ(succ(X)),Y):- !, half_1ss(succ(succ(X)),Y).
    half_1ss(succ(succ(X)), succ(half(X))).
    half_1s(succ(X),Y):- ftp_nat(X,H), half_1sh(succ(H),Y).
    half_1sh(succ(zero),Y) :- !, half_1s0(s(zero),Y).
    half_1sh(succ(succ(X)),Y) :- ! half_1ss(succ(succ(X)),Y).
half(X,Y):- ftp_nat(X,H), half_1h(H,Y).       %ftp clause
    half_1h(zero,Y):-  !, half_10(zero,Y).
    half_1h(succ(X),Y):- !, half_1s(succ(X),Y).
```

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] S. Antoy. Definitional Trees, In Proc. of the 3rd International Conference on Algebraic and Logic Programming, pages 143-157. Springer LNCS 632, 1992.

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

[4] S. Antoy, Needed Narrowing in Prolog. TR96-2, Portland State University, May 1996.

[5] 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.

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

[7] M. Carlsson. Personal Communication. April 1996.

[8] M. Carlsson, Johan Widen, Sicstus Prolog User's Manual. 1991.

[9] M. Hanus, The Integration of Functions into Logic Programming: From Theory to Practice, J. Logic Programming 1994:19,20:583-628

[10] M. Hanus, H. Kuchen, J. Moreno-Navarro, Curry: A Truly Functional Logic Language. Visions for the Future of Logic Programming, pages 95-107, Portland, OR, December 1995.

[11] P. Hill, J.W. Lloyd. The Goedel Programming Language. MIT Press, 1993.

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

[13] D. Shapiro, Compiling Evaluable Functions in the Goedel Programming Language, Thesis for Master of Science in Computer Science, Jan. 1996

[14] J. Vorvick, Evaluable Functions in the Goedel Programming Language: Parsing and Representing Rewrite Rules, Thesis for Master of Science in Computer Science, October 1995.