

© 2010 by Christopher Jon Osborn. All rights reserved.

CIRCULAR REASONER:
A PACKAGE IN MATHEMATICA FOR THE EXECUTION OF CERTAIN
OTHERWISE NON-TERMINATING FUNCTIONAL PROGRAMS

BY
CHRISTOPHER JON OSBORN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Research Associate Professor Elsa Gunter

Abstract

We have designed Circular Reasoner, a package in Mathematica for the specification of functional programs using rewrite rules. The package detects certain recursions of a cyclical nature, and through repeated refinement of a set of initial guesses of final values for relevant terms, arrives at a value consistent with the equations used to define the functional program. We discuss this package and its implementation.

To my father.

Acknowledgments

I am eternally grateful to my adviser, Elsa Gunter, without whose patience, support and constant willingness to provide guidance in all topics mathematical and otherwise, this thesis would have been impossible. I am also thankful to all of the members of our research group, who provided endless amounts of moral support, and to Andrei Popescu for very helpful discussions on some of the more theoretical topics in this paper.

Table of Contents

Chapter 1	Introduction	1
Chapter 2	A Brief Mathematica Tutorial	3
2.1	Terms	6
2.2	Identities	7
2.3	Lists	8
2.4	Some Important Functions	9
Chapter 3	Overview of Circular Reasoner	13
3.1	Implementation Overview	16
Chapter 4	An Example: First Sets	19
4.1	Terminology	19
Chapter 5	Theoretical Background	23
5.1	Basics of Term Rewriting	23
5.2	Functional Identities	27
Chapter 6	The Goal of Our Algorithm	29
Chapter 7	Related Works	31
References		32
Appendix - Code Listing		33

Chapter 1

Introduction

Consider the following program for computing the factorial of a number, written in functional style:

```
fac 0 = 1
fac n = n * fac(n-1)
```

The factorial of any number n can easily be computed by starting with the term `fac n` and then repeatedly applying the above rules to arrive at the final answer. But consider the following function to compute the set of numbers in Z_b^+ equivalent (mod d) to x in Z_b^+ :

```
equiv x d b = {x mod b} ∪ equiv ((x + d) mod b) d b
```

This equation represents a true statement: the set of natural numbers equivalent (mod d) to x in Z_b^+ is equal to $\{x \bmod b\}$ unioned with the set of natural numbers equivalent (mod b) to $(x + d) \bmod b$ in Z_b^+ . However, it is not a good recursive program: if one were to attempt to execute it, it would loop forever.

Suppose we wish to find `equiv 3 2 8`, the set of numbers equivalent (mod 2) to 3 in Z_8^+ . We could figure this out in the following way: we first assume that the set of numbers in Z_8^+ that is equivalent (mod 2) to 3 is the empty set. Then, by virtue of the fact that `equiv 3 2 8` = $\{3\} \cup \dots$, we know that `equiv 3 2 8` *at least* contains 3. By similar reasoning we may conclude that `equiv 5 2 8` *at least* contains 5. And, by our equation, we then know that `equiv 3 2 8` must contain at least 3 and 5.

Eventually, this line of reasoning leads us to learn *more and more*, until finally, we have *total knowledge* of the solution ($\{1, 3, 5, 7\}$). Along the way, we will have also computed `equiv 5 2 8`, `equiv 7 2 8`, and `equiv 1 2 8`.

Although we could have written a standard program to compute `equiv` without *too much* extra thought, it would be nice if we had a package that could take a definition of `equiv` such as the one given above, and do the above reasoning automatically.

We have designed such a package, called Circular Reasoner. It allows one to specify a certain set of **special** functions, such that these functions are allowed to be non-terminating under the traditional model

of recursion. For these functions, it is the onus of the user to supply “default” values. In computing an expression, upon running into such a function applied to its arguments, our recursive package is allowed to temporarily assume the default value to prevent non-terminating recursion. As it continues computation, it updates the value assumed to be returned for that particular expression until that value is consistent with the equations.

Where before we had non-termination, we now have a process that “guesses” values and continually updates them. This process terminates as long as the values eventually become consistent with the equations. In order to achieve this, our package continually looks for inconsistencies and removes them. In order for this process to terminate, we need that the guessed values in some way progress – that is, that we do not loop forever.

This tends to work best when the values being guessed have some sort of order on them, so that the iterated guessing may proceed with respect to that order, starting with some very small value and getting larger and larger, until consistency is obtained. One instance of this general idea is the order on sets given by set-inclusion, which is a very natural ordering on sets. With the default value generally being the empty set, signaling in effect, “I have no information,” as the algorithm progresses, a guessed set value will become larger and larger, signifying “I have progressively more and more information about what set should be returned.”

We have implemented our package in Mathematica. Imperative constructs that would have had to have been coded directly and on a case-by-case basis in most languages are now handled “underneath the hood” in our Mathematica implementation.

We start with a tour of Mathematica, then proceed to describe our package and show an example use on the problem of finding first sets for context-free grammars. We then introduce theoretical background and give a brief theoretical description of the algorithm.

Chapter 2

A Brief Mathematica Tutorial

In Mathematica, documents are called *notebooks*. Notebooks are sequences of *cells*, of which there are two types: Input cells (created by the user) and Output cells (created by Mathematica, in response to the evaluation of Input cells.)

Notebooks always begin empty, that is, with no cells. In general, the user creates a new Input cell by clicking the location where he wants the cell and typing in the cell content; in an empty notebook, the user may click anywhere in the notebook to create the first Input cell. After the user enters data into an Input cell, he presses **shift + enter**, causing the Mathematica kernel to process the given input. If the returned result is anything other than `Null`, a new Output cell is created, containing the value returned. The new Output cell is always inserted immediately after the corresponding Input cell.

For example, if the user starts up Mathematica, edits a fresh notebook, clicks inside of it, and enters `3 + 4` into his newly created Input cell, the notebook will look like this (with text entered by the user in blue):

```
In[1]:= 3 + 4
Out[1]= 7
```

The new Input cell has been given the index 1, and the corresponding Output cell shares this index. The value 7 is the result of calling Mathematica's computation engine on the input `3 + 4`.

Now let us focus on the language itself: like most languages, Mathematica is equipped with several “atomic” types of data – for example, integers, strings and booleans. As in most programming languages, one enters an integer by typing a sequence of digits, and one enters a string by typing the character `"` followed by a sequence of characters followed by a terminating `"`. One enters a boolean by typing in either `True` or `False`.¹ Of course, Mathematica can also handle floating-point numbers, complex numbers and even some transcendental numbers (as well as more complicated special types of data such as sparse arrays

¹Technically, `True` and `False` are symbols and therefore fall under the category of standard terms and not special atomic data, but we introduce them here for convenience.

and compiled numerical code), but we do not focus on these types of data here.

Let us see some examples of working with atomic data. For reference, string concatenation is achieved with the infix operator `<>`, and Boolean arithmetic can be performed with the functions `&&`, `||` and `!` (“and”, “or” and “not”, respectively):

```
In[1]:= 2*4 + 23
Out[1]= 31

In[2]:= "Hello " <> "there!"
Out[2]= "Hello there!"

In[3]:= False || ! False
Out[3]= True
```

The user could have created the above notebook, starting from an empty notebook, by first clicking somewhere in the notebook, then typing `2*4 + 23`. Then the user could have pressed **shift + enter**, causing Mathematica to create the Output cell `Out[1]= 31`. Then the user could have clicked below the line `Out[1]= 31` and entered `"Hello " <> "there!"`, pressed **shift + enter**, etc... From this point forward, we will take it for granted that sample notebook sessions can be replicated in actual Mathematica notebooks.

In addition to atomic data such as integers and strings, Mathematica also does computation on *symbols*, which are basically names. For example, `x`, `xyz`, and `Schenectady` are all symbols. Because Mathematica is a computer algebra system, Mathematica will often simplify mathematical expressions involving symbols:

```
In[1]:= x + x
Out[1]= 2 x

In[2]:= x*(x + x)
Out[2]= 2 x^2
```

Some expressions are already fully simplified, and are returned exactly as entered:

```
In[3]:= x + y
Out[3]= x + y
```

and others, not built up from the traditional mathematical operators, cause errors to be generated:

```

In[4]:= x <> y

During evaluation of In[4]:=
StringJoin::string:
String expected at position 1 in x<>y.  >>

During evaluation of In[4]:=
StringJoin::string:
String expected at position 2 in x<>y.  >>

Out[4]= x <> y

```

Here the problem is that the string concatenation operator cannot be applied to generic symbols. One can get the above input to work by first assigning values to `x` and `y`:

```

In[1]:= x = "Hello"
Out[1]= "Hello"
In[2]:= y = " there!"
Out[2]= " there!"
In[3]:= x <> y
Out[3]= "Hello there!"

```

This is the first example we have seen of a *side-effect*: in addition to returning the value gotten by computing the right-hand side, the `=` operator causes some change in the internal state of the current Mathematica session. Specifically, an internal rule is created asserting that from now on, whenever Mathematica sees `x`, it may replace that symbol with the computed value, `"Hello"`, and similarly for `y` and `" there!"`.

But it is quite tedious to enter in these assignments one-by-one in separate cells, and we are forced to view uninteresting intermediate results. We can achieve a one-cell version of the above code using the `;` operator, which allows us to sequence statements:

```
In[1]:= x = "Hello";  
        y = " there!";  
        x <> y  
  
Out[3]= "Hello there!"
```

The `;` operator evaluates the expression or statement to its left, discards the computed value and then evaluates the expression to its right. If there is no expression to its right, then the special symbol `Null` is returned. Consider:

```
In[1]:= 3 + 4;
```

Note that no Output cell was generated, since the final `;` has nothing to its right, causing `Null` to be returned (the computed value 7 is discarded since it was to the left of the `;`.)

2.1 Terms

Mathematica extends the notion of symbols to the very general concept of *terms*, arbitrary expressions which are *constructed* from symbols. Symbols are by themselves terms, and additionally, one may construct a term by *applying* a symbol to a list of arguments, also terms. The application of the symbol f to the terms t_1, \dots, t_n is written as $f[t_1, \dots, t_n]$. f is called the *head* of the expression $f[t_1, \dots, t_n]$. For example:

```
f[a,b,g[3,"hi"]]
```

is a term since `f` is a symbol, and `a`, `b`, and `g[3,"hi"]` are terms. One may enter the above expression into Mathematica, and it will simply return the entered term:

```
In[1]:= f[a, b, g[3, "hi"]]  
Out[1]= f[a, b, g[3, "hi"]]
```

Terms are pervasive in Mathematica; in fact, every application of an operator to its arguments translates into an application of a corresponding *symbol* to those arguments. So, for instance:

```
3 + 4
```

is the same as:

```
Plus[3, 4]
```

and

```
2 * 3 + 4
```

is the same as

```
Plus[Times[2, 3], 4]
```

2.2 Identities

Computation is specified by giving **identities**, equations specifying how to transform terms to get new terms.

One enters an identity with the `:=` operator, like this:

```
In[1]:= f[x_, y_] := x + y^2
```

Recall that earlier we used the `=` operator to assign the value `"hello"` to the symbol `x`. The `:=` operator is similar, but does *not* compute the value of the right-hand side, instead associating the pattern on the left-hand side with the *term* on the right-hand side. Just as entering `x = "hello"` created an internal rule telling Mathematica to replace instances of `x` with the *value* `"hello"`, entering `f[x_, y_] := x + y^2` causes instances of `f[x_, y_]` to be replaced by the *term* `x + y^2`. The blanks after `x` and `y` tell Mathematica that, for the purposes of this identity, `x` and `y` are to be treated as variables and not standard symbols. For example, `f[3, 4]` becomes `3 + 4^2`, and `f[9, 12]` becomes `9 + 12^2`:

```
In[2]:= f[3, 4]
```

```
Out[2]= 19
```

```
In[3]:= f[9, 12]
```

```
Out[3]= 153
```

You can even plug in symbols:

```
In[4]:= f[a, b]
```

```
Out[4]= a + b^2
```

One may restrict patterns further by including a “head symbol” after the blank. For example, the pattern `x_List` only matches terms with head `List`. Additionally, atomic types of data are given special heads that represent their types: for instance, integers have head `Integer` and strings have head `String`.

Here is an example piece of code, which only works on integers:

```
In[1] := f[x_Integer, y_Integer] := x + y^2
```

If you try to execute the code on real numbers, the rewrite rule will not apply:

```
In[2] := f[2.3, 4.2]
Out[2] = f[2.3, 4.2]
```

Although term rewriting is a fairly straightforward concept, it easily facilitates the modeling of almost any mathematical or computational formalism. One example of this versatility, particularly relevant to our thesis, is term rewriting's ability to capture the functional programming paradigm. The following Mathematica program defines the Fibonacci sequence:

```
fib[0] = 0;
fib[1] = 1;
fib[n_] := fib[n - 1] + fib[n - 2];
```

2.3 Lists

Lists are a very common type of data in many programming languages. It is therefore no surprise that Mathematica includes special support for lists. One creates a list by applying the constructor `List` to n arguments, where n is the length of the list. For example, `List[]` represents the empty list, and `List[a, b, c]` is the list of the three elements `a`, `b` and `c`.

Lists are so useful that Mathematica provides special syntax for them: the list `List[t1, ..., tn]` is usually represented as $\{t_1, \dots, t_n\}$.

There are a host of functions provided to manipulate lists:

- `Length[lst]` returns the number of elements in the list `lst`.
- `lst[[i]]` returns the i_{th} element of the list `lst`. Also written as `Part[lst, i]`.
- `First[lst]` returns the first element of the list `lst`, or prints an error if `lst` is empty.
- `Last[lst]` returns the last element of the list `lst`, or prints an error if `lst` is empty.

Note that all of the above functions also work on non-list terms. For example, `Length[f[a, b, c]]` returns 3, even though `f[a, b, c]` has head `f` instead of `List`.

2.4 Some Important Functions

The following functions are used frequently, so we document them here:

- `/@` (mapping)

$f /@ g[t_1, \dots, t_n]$ returns $g[f[t_1], \dots, f[t_n]]$. Also written as `Map[f, g[t1, ..., tn]]`.

Examples:

```
In[1]:= Not /@ {False, False, True}
Out[1]= {True, True, False}
```

```
In[1]:= f[x_] := 2 x;
        f /@ {1, 2, 3, 4}

Out[1]= {2, 4, 6, 8}
```

- `/.` (substitution)

We have discussed the addition of identities to Mathematica's internal database by entering them into a Notebook and pressing **shift** + **enter**. But it is also possible to apply identities locally, by supplying a list of identities to the **substitution** function.

Local identities are defined using the `->` and `:>` operators instead of the `=` and `:=` operators. Like `=`, `->` specifies that the term on the left-hand side should be replaced by the *value* on the right-hand side. Like `:=`, `:>` specifies that the *pattern* on the left-hand side should be replaced by the *term* on the right-hand side.

$a /. lhs \rightarrow rhs$ returns a with all instances of lhs replaced by rhs .

$a /. \{lhs_1 \rightarrow rhs_1, \dots, lhs_n \rightarrow rhs_n\}$ returns a with all identities in the list

$lhs_1 \rightarrow rhs_1, \dots, lhs_n \rightarrow rhs_n$ applied in parallel. Note that for any particular sub-term of a , only one identity from the list may be applied to that sub-term.

Also written as `ReplaceAll`, which can be invoked as follows:

`ReplaceAll[a, lhs -> rhs]` or `ReplaceAll[a, {lhs1 -> rhs1, ..., lhsn -> rhsn}]`.

One may use `:>` instead of `->` for any identity.

Examples:

```
In[1] := f[a, b, g[a, a]] /. a -> b
Out[1] = f[b, b, g[b, b]]
```

```
In[8] := f[a, b, g[a, a]] /. g[x_, x_] :> x
Out[8] = f[a, b, a]
```

```
In[9] := f[a, b, g[a, a]] /. {a -> b, b -> a, g -> h}
Out[9] = f[b, a, h[b, b]]
```

- `@@`

$f @@ g[t_1, \dots, t_n]$ returns $f[t_1, \dots, t_n]$.

Also written as `Apply[f, g[t1, ..., tn]]`. This function replaces the head symbol of a given term with a new head.

Examples:

```
In[1] := f @@ {1, 2, 3}
Out[1] = f[1, 2, 3]
```

```
In[1] := Plus @@ g[1, 2]
Out[1] = 3
```

- `#` and `&` (anonymous functions)

$(e\&)$ is an anonymous function, where e is the body of that function. The scope of `&` is the body e , and the argument appears in e as `#`. Anonymous functions may be nested arbitrarily; each instance of `#` binds to the innermost `&` it can.

Also written as `Function[e]`.

Examples:


```
In[1]:= (# <> " there!" &)[ "Hello"]
Out[1]= "Hello there!"
```

```
In[1]:= (# + 1 &) /@ {1, 2, 3, 4}
Out[1]= {2, 3, 4, 5}
```

- `ClearAll[f]` clears all internally stored identities of the form $f[t_1, \dots, t_n]$.

Example:

```
In[1]:= f[x_] := x;
In[2]:= f[3]
Out[2]= 3
In[3]:= ClearAll[f];
In[4]:= f[3]
Out[4]= f[3]
```

- `??f` displays information about the symbol f , including all active identities of the form $f[t_1, \dots, t_n] := \dots$ (or $f = \dots$). Also displayed are any special attributes that the symbol may have. The only attribute that we are interested in is `Protected`, which states that a symbol may not be modified.

Examples:

```
In[1]:= f[x_] := x;
      ?? f
Global`f
f[x_] := x
```

```
In[1]:= ?? Plus
x+y+z represents a sum of terms. >>
Attributes[Plus]={Flat,Listable,NumericFunction,OneIdentity,
  Orderless,Protected}
Default[Plus]:=0
```

- `Hold[e]` maintains the term e in unevaluated form. Example:

```
In[1] := Hold[3 + 3]
```

```
Out[1]= Hold[3 + 3]
```

Chapter 3

Overview of Circular Reasoner

Our package allows the user to distinguish certain functions by calling `Memo`. The functions marked `Memo` (memoized functions) are the ones for which our package performs reasoning as demonstrated in the introduction. Our package introduces four new Mathematica functions:

- `ResetLFP[v]`

Resets the package, setting v as the new standard default value returned for all memoized function calls. Every function declared via `Memo` will assume this standard value as its default, unless the user overrides the standard value (on a per-function basis) with `Least` (see below.)

- `Memo[f]`

Declares that f is memoized. Internally, the identities for f are modified to invoke our package, and f is given the attribute `Protected` so that its identities cannot be further modified (without calling `ClearLFP` on f , which removes the `Protected` attribute.)

- `Least[f] = v`

Such an assignment sets the default value of f to be v , overriding the standard default value provided to the most recent call of `ResetLFP`. This function is useful if you are writing functions that do not all share the same default value.

- `LFP[e]`

Computes the value of expression e using our package.

- `ClearLFP[f]`

Clears all identities associated with f , a symbol to be memoized. Should be called before the definition of memoized functions to clear all internal Mathematica definitions associated with f , and readies it for memoization.

Before we show how to make use of our package, we invite the reader to consider the following definition, which does *not* invoke our package:

```
In[1] := ClearAll[fff];  
      fff[x_] := fff[x];
```

The definition for `fff` is dubious, given that we have not invoked our package. Let us take a look at the definition, using the `Information` function, abbreviated by the prefix `??` operator:

```
In[3] := ?? fff  
Global`fff  
fff[x_] := fff[x]
```

As expected, we get back the definition exactly as we entered it. Now let us try calling `fff` on an integer input:

```
In[4] := fff[3]  
During evaluation of In[4]:= $IterationLimit::itlim:  
      Iteration limit of 4096 exceeded. >>  
Out[4]= Hold[fff[3]]
```

Not surprisingly, evaluation proceeds to iterate continually, until Mathematica gives up and returns `fff[3]` wrapped in `Hold` to prevent computation from continuing.

Suppose we now load in our package: what happens if we again enter the function `fff`, making use of our package with a default value of `{}` (the empty list, here representing the empty set)?

```
In[68] := ResetLFP[];  
  
      ClearLFP[fff];  
      fff[x_] := fff[x];  
      Memo[fff];
```

First off, let us see what we get when we look at the definition of `fff`:

```
In[69] := ?? fff

Global 'fff

Attributes[fff]=Protected

e$:fff[x_]:=idi$$@@holdID$$[makeid[fff,applyholdID$$[e$],HoldComplete[fff[x]]]]
```

This time we do not get back exactly what we entered; notice that `fff` now has a very complicated definition that has characteristics of the definition we gave, but is clearly doing something else. Furthermore, `fff` now has the attribute `Protected` to prevent the user from accidentally adding or changing definitions of `fff` in a way that is inconsistent with the auto-generated definition of `fff`. We will not cover the details of this definition here, as they are quite complicated and not tremendously revealing.

Now, let us execute our code:

```
In[72] := fff[3]

Out[72]= idi$$[524]
```

Surprise! Well, we certainly avoided non-termination. But we got back something very cryptic. What is `idi$$[524]`? As we will learn shortly, `idi$$[524]` is a special term created by our package, standing for “computation number five hundred and twenty-four,” but certainly we would prefer to get back an actual result.

To get the answer we are expecting, we need to apply the LFP (least-fixed point) function to our expression. Let us try again:

```
In[73] := LFP[fff[3]]

Out[73]= {}
```

We have gotten back an actual answer, and that answer is `{}`. Our package, when confronted with `fff[3]`, realized that `fff` was a memoized function symbol. It further realized that it did not know the value of `fff[3]`, and so “assumed” its value was `{}`, the default value specified for all memoized function symbols. It then recorded the assumption internally and began to evaluate the body of `fff[3]`, which was also `fff[3]`. It used the assumption that `fff[3] = {}` to evaluate the body to `{}`. Finally, it checked that the result computed for `fff[3]` was indeed `{}`, the initial “guess,” indicating no need for further computation.

This example typifies the general procedure for invoking our package: after loading our package, one defines several functions all together in one Input cell. At the top of that Input cell is the line:

```
ResetLFP[...];
```

where ... is replaced by the default value you want functions to have.

Then, for each function, one writes:

```
ClearLFP[f];  
  
.  
  
.  
  
.  
  
Memo[f];
```

where `f` is replaced by the name of the function being defined, and ... is replaced by the rules to define `f`.

3.1 Implementation Overview

When we call `Memo` on a function symbol `f`, the definitions given to `f` are scanned and individually replaced by definitions that invoke our package, then `f` is given the attribute `Protected` to prevent the user from modifying these auto-generated definitions.

Recall that above, the term `fff[3]` was given identifier 524, which we found out by computing `fff[3]` without wrapping `LFP` around it (we got back `idi$$[524]`.) In general, when you do a computation on a term involving calls to memoized functions, instead of directly beginning computation on that term, our package associates the term with an identifier. That identifier is then in turn associated with a *partially* evaluated result, similar to the original term but with simple sub-terms fully evaluated, and subterms containing calls to memoized functions replaced by instances of `idi$$[...]`.

For example, consider the following definition:

```
In[74] := ResetLFP[{}];  
  
ClearLFP[fff];  
fff[x_] := {x + x} ∪ ({x} ∪ fff[x]);  
Memo[fff];
```

Now suppose we ask to compute `fff[3]`. The body of `fff[3]` is equal to $\{3 + 3\} \cup (\{3\} \cup \text{fff}[3])$. Let us assume for sake of argument that the expression `fff[3]` is associated with id 1, that is, were we to compute `fff[3]` without wrapping `LFP` around it, we would get back `idi$$[1]`.

Our package associates id 1 with the following term, represented here as a tree:

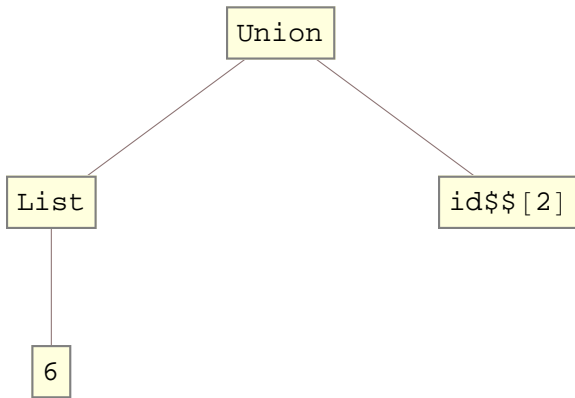


Figure 3.1: tree representation of term associated with id 1

and furthermore, it associates id 2 is with the term:

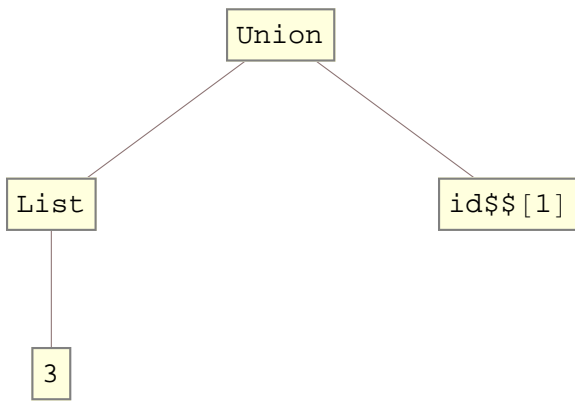


Figure 3.2: tree representation of term associated with id 2

Note that the simple computation taking $\{3 + 3\}$ to $\{6\}$ is done automatically. In general, arguments containing no calls to memoized functions are computed to their final values, whereas arguments containing calls to memoized functions are replaced by identifiers representing the computation tree (either fresh identifiers, or, in the case of an expression tree identical to one previously encountered, the previously assigned id.)

For each id, a guessed final value is maintained at all times, initially the default value. Note that the value of id 1 is clearly affected by that of id 2. For that reason, id 2 is linked to id 1, so that whenever the algorithm discovers a new value for id 2, an internal “update” function is called to re-compute the value to be associated with id 1.

Similarly, in the above example id 2 is affected by the value of id 1, so whenever the value of id 1 changes, “update” is called on id 2.

This process continues until a fixed point is reached, that is, until no id changes value upon update.

Chapter 4

An Example: First Sets

Consider the problem of finding “first sets” for regular grammars, a useful step in verifying that a context-free grammar is suitable for recursive descent parsing.¹ This problem and an algorithmic solution similar to that presented here can be found in [2].

4.1 Terminology

A context-free grammar is parameterized by a set T of *terminal* symbols and a set N of *non-terminal* symbols. T^* is the universe of *strings*.

A context-free grammar G over (T, N) is a set of *productions*. A production is of the form $A \rightarrow \gamma$, where $A \in N$ and $\gamma \in (T \cup N)^*$. One may represent a set of productions $A \rightarrow \gamma_1, \dots, A \rightarrow \gamma_n$ by the notation $A \rightarrow \gamma_1 \mid \dots \mid \gamma_n$.

Definition 1. Consider a context-free grammar G over (T, N) . We say that $\gamma \in (T \cup N)^*$ **reduces to** $\gamma' \in (T \cup N)^*$ if γ can produce γ' by repeatedly choosing a production $(A \rightarrow \delta) \in G$ and replacing an instance of A in γ by δ .

Definition 2. Consider a context-free grammar G . If γ reduces to $\gamma' \in T^*$, then γ **recognizes** γ' .

The former definition states that a list of terminals and non-terminals reduces to another list of terminals and non-terminals if one can get from the first list to the second list by repeatedly applying productions in the given grammar.

The latter definition states that a list of terminals and non-terminals recognizes a string if it reduces to it.

For example, consider the following context-free grammar:

$A \rightarrow aAa \mid aBa$

¹If one verifies that a grammar contains no left-recursion and that, for each non-terminal, the first sets of all of the productions for that non-terminal are disjoint, then one can conclude that the grammar is $LL(1)$.

$B \rightarrow b \mid bB$

The non-terminals are A and B , and the terminals are a and b . This grammar so happens to recognize all strings of the form $a^n b^m a^n$ where $n, m > 0$.

Definition 3. *The first set of $\gamma \in (T \cup N)^*$ is the set of terminals $\{a \in T \mid \gamma \text{ recognizes } a\delta\} \cup \{\bullet \mid \gamma \text{ recognizes } \epsilon\}$*

In our grammar above, the first set of A is $\{a\}$, and the first set of B is $\{b\}$.

We here introduce two mutually recursive functions to determine the first set of a list of symbols: *firstset*, which determines the first set of a list of terminals and non-terminals, and *firstsetTerminal*, which determines the first set of a single non-terminal. For given G , algorithmically determining the first set of a list γ of terminals and non-terminals can be split into three cases:

1. $\gamma = \epsilon$. Then $firstset(\gamma) = \{\bullet\}$.
2. $\gamma = x\delta$ where $x \in T$ and $\delta \in (N \cup T)^*$. Then $firstset(\gamma) = \{x\}$.
3. $\gamma = X\delta$, where $X \in N$ and $\delta \in (N \cup T)^*$. Then $firstset(\gamma) = (firstsetNonterminal(X) \setminus \{\bullet\}) \cup \{x \in firstset(\delta) \mid \bullet \in X\}$

The first set of a single non-terminal (we distinguish between a single non-terminal and the list containing that single non-terminal) is computed as follows:

$$firstsetNonterminal(X) = \bigcup_{(X \rightarrow \gamma) \in G} firstset(\gamma)$$

So far we have dealt with a purely mathematical representation of terminals, non-terminals and the functions *firstset* and *firstsetNonterminal*. Now we shall consider how to represent these notions in Mathematica. To represent symbols in the context-free grammar sense, we will simply use Mathematica's built-in notion of symbols. So, we represent the terminal a with the Mathematica symbol `a`, and the non-terminal A with the Mathematica symbol `A`.

We represent lists of symbols (elements of $(N \cup T)^*$) as Mathematica terms, where the first symbol in the list is applied to the Mathematica term representing the remaining symbols. As a special case, the empty list of symbol is represented by the Mathematica symbol `ε`.

To simplify our code, we require that all productions $A \rightarrow \gamma_1, \dots, A \rightarrow \gamma_n$ for a given symbol be represented by the single statement $A \rightarrow \gamma_1 | \dots | \gamma_n$. This is represented in Mathematica code by `A -> {γ1, ... γn}`, where $\gamma_1, \dots, \gamma_n$ are represented as terms, as explained above.

Here is a sample grammar:

```
{
  A -> {a, A[b], B[A]},
  B -> {c, d, A, ε}
}
```

There are two non-terminals, A and B. A is associated with three productions, $A \rightarrow a$, $A \rightarrow A b$, and $A \rightarrow B A$. B is associated with four productions, $B \rightarrow c$, $B \rightarrow d$, $B \rightarrow A$ and $B \rightarrow \epsilon$.

Our code is split into two functions, `first`, analagous to *firstset* in the mathematical development, which takes in a symbol and a grammar and returns the first set of that symbol, and `firstList`, analagous to *firstsetNonterminal*, which takes in a list of symbols (represented as a Mathematica term) and returns its first set.

The code is as follows:

```
ResetLFP[{}];

ClearLFP[first, firstList];
first[s_Symbol, rls_List] := If[
  MemberQ[First /@ rls, s],
  Union @@ (firstList[#, rls] &) /@ (s /. rls),
  {s}
];
firstList[ε, _] := {•};
firstList[s_Symbol, rls_List] := first[s, rls];
firstList[s_Symbol[rst_], rls_List] :=
  With[{fst = first[s, rls]},
    If[MemberQ[fst, •],
      Complement[fst, {•}] ∪ firstList[rst, rls],
      fst
    ]
  ];
Memo[first, firstList];
```

Let us focus on `first`: the call to `MemberQ` returns `True` if `s` is one of the symbols treated as a non-

terminal in the grammar. If s is *not* a non-terminal, then s must be a terminal. The first-set of a terminal is the set containing only that terminal, so $\{s\}$ is returned. On the other hand, if s is one of the non-terminals, then **first** unions together the first sets of the right-hand sides corresponding to symbol s , as follows:

The term $s /. rls$ gives the list of symbol lists associated with s . (`firstList[#, rls] &)/@ (s /. rls)` maps an anonymous function over this list of symbols, calling **firstList** on each of them. This results in a list of first sets. `Union@@...` takes the union of these first sets, which is the first set for the given symbol.

The function **firstList** is coded using three cases. In the first case, the list of symbols is empty (represented by ϵ). In this case, the grammar does not matter. The result returned is $\{\bullet\}$.

In the case of a singleton list of symbols, represented by a single symbol s , **firstList** simply calls **first** on that symbol, passing along the given grammar, rls .

In the final case, where the symbol list contains multiple elements, represented by an application of a symbol to an argument, **firstList** first calculates the first set of the first symbol (`first[s, rls]`) and locally assigns **fst** to the result. If ϵ is *not* included in **fst**, then **fst** is returned as is. Otherwise, **fst** (minus $\{\bullet\}$) is unioned with `firstList[rst, rls]`, the first set of the remaining symbols.

Note that this code is fairly short, and has no imperative features.

The above implementation terminates with our package, as our package intercepts all recursive calls to **first** and **firstList**. There can only be a finite number of such calls, since each call to **first** has a non-terminal argument (and there are only a finite number of non-terminals) and each call to **firstList** has a postfix of the right-hand side of a production rule as its argument (and there are only a finite number of such postfixes.) Furthermore, since first sets are always finite, only a finite number of updates need to be done to the gussed values for each function call, resulting in termination.

The above implementation fails to terminal without our package, or in a language other than Mathematica. To create an implementation without the package or in another language, one would needs to maintain large structure mapping non-terminals and lists of terminals and non-terminals to symbols known to be in their first sets, and pass this structure through each function call. The elegance of the solution made available by our package is lost.

Chapter 5

Theoretical Background

We now develop the theoretical core of an algorithm that, given a program written in the manner described above, executes it and returns a value (assuming such a value can be discovered.)

Mathematica is, at its core, an implementation of term rewriting. Term rewriting is a very general theoretical framework for specifying programs and systems, based on the theory of universal algebra. We first give a development of the basic theory of term rewriting, and then use this framework to elucidate, at an abstract level, the basic functionality of our algorithm.

We will cover topics that lead to the development of the notion of *functional* identities, a term we introduce to describe the building blocks of programs suitable for use by our algorithm.

5.1 Basics of Term Rewriting

The following development comes mostly from [3].

The most basic notion of universal algebra, and consequently term rewriting, is that of the signature, which is essentially a set of symbols. Whereas in Mathematica the universe of symbols is effectively unlimited, here the set of symbols is specified by the signature.

Definition 4. A signature Σ is a set of pairs (called symbols), where the first part of each pair is a symbol name, and the second part is a natural number indicating the arity of the symbol. We allow individual symbol names to be paired with multiple arities, e.g. both $(f, 2)$ and $(f, 3)$ can appear in the same signature.

From these symbols we derive the notion of terms, similar to terms in Mathematica.

Definition 5. Given a signature Σ and set of variables X , the set of **terms** $T(\Sigma, X)$ is defined by:

1. $X \subseteq T(\Sigma, X)$

2. if $t_1, \dots, t_n \in T(\Sigma, X)$ and $(f, n) \in \Sigma$ then $f(t_1, \dots, t_n) \in T(\Sigma, X)$

We shall find useful the basic notion of a position within a term. Here, positions are represented as strings of natural numbers.

Definition 6. *The set of positions $Pos(t)$ of a term is:*

1. $\epsilon \in Pos(t)$
2. if $1 \leq i \leq n$ and $p_i \in Pos(t_i)$ and $(f, n) \in \Sigma$ then $i p_i \in Pos(f(t_1, \dots, t_i, \dots, t_n))$

For example, $11 \in Pos(f(g(a), b, c))$, but $12 \notin Pos(f(g(a), b, c))$.

We extract the sub-term of term t at position p by $t|_p$, where:

1. $t|_\epsilon = t$
2. $f(t_1, \dots, t_n)|_{i p} = t_i|_p$

For example, $f(g(a), b, c)|_{11} = a$.

We replace a sub-term of term t at position p with term u by $t[u]_p$, where:

1. $t[u]_\epsilon = u$
2. $f(t_1, \dots, t_i, \dots, t_n)[u]_{i p} = f(t_1, \dots, t_i[u]_p, \dots, t_n)$

For example, $f(g(a), b, c)[b]_{11} = f(g(b), b, c)$.

The notion of position allows us to define the sub-terms of a term:

Definition 7. *$Subs(t)$ is the set of all sub-terms of t , that is:*

$$Subs(t) = \{t|_p \mid p \in Pos(t)\}$$

For example, $Subs(f(g(a), b, c)) = \{a, b, c, g(a), f(g(a), b, c)\}$.

Similarly, in the context of a set of variables X , we define $Vars$ by:

Definition 8.

$$Vars(t) = X \cap Subs(t)$$

Informally, a substitution is a mapping from terms to terms.

Definition 9. A $T(\Sigma, X)$ -substitution σ is a function of type $X \rightarrow T(\Sigma, X)$ where there are only finitely many x where $\sigma(x) \neq x$.

One can represent a substitution mapping x_i to t_i for $1 \leq i \leq n$ and every other variable to itself, as follows:

$$[t_1/x_1, \dots, t_n/x_n]$$

Substitutions are extended to terms in the following way:

$$\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$$

We aim to introduce several notions related to the restriction of the types of programs to be defined, culminating in confluence, a powerful restriction on the behavior of a term rewrite system. In the next section, we will show that the exclusive use of yet-to-be defined “functional” identities guarantees confluence. In the mean time, we start by defining a unifier, which is a substitution that “unifies” two terms, i.e. which yields the same thing when applied to either term.

Definition 10. A unifier θ of two terms t_1 and t_2 is a substitution such that $\theta(t_1) = \theta(t_2)$.

A unifier θ of t_1 and t_2 is more general than a unifier σ of t_1 and t_2 if there exists a substitution τ such that $\sigma = \tau \circ \theta$.

A unifier θ of t_1 and t_2 is a most general unifier of t_1 and t_2 if, for all unifiers σ of t_1 and t_2 , θ is more general than σ .

Although we do not need this fact, it is the case that if terms t_1 and t_2 have a unifier, then they have a most-general unifier, as is stated in [3].

The theory of term rewriting has a notion of identities similar to that existing in Mathematica:

Definition 11. A $T(\Sigma, X)$ -identity is a pair $(s, t) \in T(\Sigma, X) \times T(\Sigma, X)$. We will refer to such pairs simply as identities when Σ and X are clear.

Now we are finally ready to define the reduction relation. Recall that in Mathematica terms could reduce to other terms with respect to given identities. Here, the situation is similar: the reduction of terms in our theoretical setting is parameterized by a set of identities.

Definition 12. Given a set E of identities, the reduction relation \rightarrow_E is defined as

$$s \rightarrow_E t \text{ iff } \exists (l, r) \in E, p \in \text{Pos}(s), \sigma. s|_p = \sigma(l) \wedge t = s[\sigma(r)]_p$$

Furthermore, let \rightarrow_E^* be the reflexive and transitive closure of \rightarrow_E .

For example, $a \rightarrow_{\{(a,b),(b,c)\}}^* c$.

We now go on to define critical pair and left linearity, followed by orthogonality, which is defined in terms of the prior two terms. Informally, a critical pair is a pair of two distinct terms that can be written to from a single term, indicating non-determinism.

Definition 13. Let (l_1, r_1) and (l_2, r_2) be two distinct identities in a set of identities E , with variables renamed to be distinct. Let l'_1 be a (not necessarily proper) subterm of l_1 at position p , such that l'_1 is not a variable. If l'_1 and l_2 have most general unifier θ , then $(\theta(r_1), \theta(l_1[r_2]_p))$ is a critical pair of E .

Left linearity specifies that a variable cannot occur more than once on the left-hand side of an identity. More formally:

Definition 14. (l, r) is left linear if for all $x \in \text{Vars}(l)$ and positions p_1, p_2 where $l|_{p_1} = l|_{p_2} = x$, $p_1 = p_2$.

Orthogonality attempts to capture the notion that it is impossible to rewrite a particular term in more than one way:

Definition 15. A system (Σ, E) is orthogonal if:

1. The identities in E are left linear
2. E has no critical pairs

A similar but more powerful concept is that of confluence, which specifies that if you rewrite a term to two other terms, these other terms can be rewritten to a common fourth term:

Definition 16. A system (Σ, E) is confluent if, for all terms x, u, v , if $x \rightarrow_* u$ and $x \rightarrow_* v$, then there exists term y such that $x \rightarrow_* y$.

Proposition 1. If a system (Σ, E) is confluent, then (Σ, E) is orthogonal.

Proof. See [3]. □

5.2 Functional Identities

Expanding on the ideas presented above, we introduce the notion of *functional* identities, such that programs written entirely out of functional identities behave very much like programs written in a standard functional language, and inherit the property of confluence.

It is within this setting, where programs are confluent, that we intend to apply our algorithm.

We first separate out a sub-set of symbols called constructors. Informally, these are symbols used to build up data, not to represent functions. Separating constructors from non-constructors will aid us in our forthcoming definition of functional identities.

Definition 17. Fix a set $C \subset \Sigma$, called the constructors of Σ . Then values are terms in $T(C, \emptyset)$, that is, terms made entirely out of constructors, and value patterns are terms in $T(C, X)$, that is, terms made out of constructors and variables.

And now, we present our main definitions, those of *functional* identities and *functional* sets of equations:

Definition 18. We say that an identity $(l, r) \in E$ is functional if:

1. l is of the form $f(v_1, \dots, v_n)$, where $f \in \Sigma \setminus C$ and $v_1, \dots, v_n \in T(C, X)$.
2. $\text{Vars}(r) \subseteq \text{Vars}(l)$.
3. (l, r) is left linear.

Definition 19. We say that two identities (l, r) and (l', r') overlap if there exist substitutions σ and σ' such that $\sigma(l) = \sigma'(l')$. We say that a set E of identities is functional if every identity in E is functional, and no two distinct identities in E overlap.

Proposition 2. If a set E of identities is functional, then E has no critical pairs.

Proof. Assume E has a critical pair derived from distinct identities (l_1, r_1) and (l_2, r_2) . We thus have the existence of l'_1 , p and θ as specified in the definition of critical pair. By criterion 1 for being a functional identity, l_1 and l_2 must both be function symbols applied to value patterns. Therefore no non-variable subterm of l_1 can unify with l_2 , except possibly l_1 itself. Since θ is the most general unifier of l'_1 and l_2 , we must have that $l'_1 = l_1$. But then we have that $\theta(l_1) = \theta(l_2)$, so l_1 and l_2 overlap, contradicting the fact that E is functional. \square

Corollary 1. *If E is functional, then (Σ, E) is orthogonal.*

Proof. Since E is functional, its identities are left-linear and it has no critical pairs (by the prior proposition), hence orthogonality. \square

Corollary 2. *If E is functional, then (Σ, E) is confluent.*

Proof. This follows from the prior corollary and the fact that orthogonality implies confluence. \square

Thus, the exclusive use of functional identities guarantees confluence, a very strong property that decreases the amount of non-determinism present in our algorithm.

Chapter 6

The Goal of Our Algorithm

Given a signature Σ , a set C of constructors, and a *functional* set of identities E , we distinguish a certain subset S of $\Sigma \setminus C$ to represent our “recursive” function symbols. These symbols, interpreted via the equations in E , are meant to represent recursive functions in our expanded recursion model (by contrast, symbols in $\Sigma \setminus S$ are meant to model standard functions and constructors.) We let F be an environment mapping each such symbol $(f, n) \in S$ to a function of type $value^n \rightarrow value \cup \{\perp\}$, where a *value* is a term restricted to constructors, as defined in the previous section, and \perp is here introduced to represent “undefined.” We denote by $\boxed{f, n}_F$ the function assigned to (f, n) by the environment F . Furthermore, for each $(f, n) \in S$, let $default((f, n))$ be a default value assigned to (f, n) .

We informally define a **context** to be a term with exactly one hole in it. Where x and y are terms, we define the relation $x \rightarrow_E^F y$ by:

1. $C[\perp] \rightarrow_E^F \perp$ for any context C
2. $C[g(v_1, \dots, v_n)] \rightarrow_E^F C[\boxed{g, n}_F(v_1, \dots, v_n)]$ for any context C and values v_1, \dots, v_n
3. If the first two rules do not apply, and $t \rightarrow_E u$, then $t \rightarrow_E^F u$.

Furthermore, \rightarrow_E^{F*} is the transitive reflexive closure of \rightarrow_E^F , and we say that $x \rightarrow_E^{F+} y$ if $x \rightarrow t$ and $t \rightarrow_E^{F*} y$.

Definition 20. For functional equations E , environment F , function symbol $(g, n) \in S$ and values v_1, \dots, v_n , we say that F is **locally consistent** with E at the term $g(v_1, \dots, v_n)$ if either $g(v_1, \dots, v_n) \rightarrow_E^{F+} \boxed{g, n}_F(v_1, \dots, v_n)$ or $\boxed{g, n}_F(v_1, \dots, v_n) = \perp$.

Definition 21. An environment F is **consistent** with a set E of functional equations if, for all function symbols $(g, n) \in S$ and values v_1, \dots, v_n , F is locally consistent with E at $g(v_1, \dots, v_n)$.

Our algorithm is given a set of functional equations E , a “main” function symbol $(f, n) \in S$ and values v_1, \dots, v_n . The goal of the algorithm is to determine a value for $f(v_1, \dots, v_n)$. It will do so by continually updating the functions in F , terminating only at some point when F is consistent with E and $\boxed{f, n}_F(v_1, \dots, v_n) \neq \perp$. The algorithm may fail to terminate.

In cases where the algorithm does terminate, which is the case for well-written programs, the resulting consistency implies that the value returned does in fact agree with the equations defining the function given by the user. The confluence of the given system greatly decreases any non-determinism present in the execution of the algorithm.

Chapter 7

Related Works

As mentioned above, most of the theoretical background on term-rewriting comes from [3].

The heavy restrictions imposed on the set of equations E imply the corresponding system is confluent (a discussion on modeling function programming via term rewriting and the “orthogonality” constraint can be found in [3].) A further property that one would like to prove is that the final F returned from running the algorithm maps recursive symbols to terms in such a way that, if these mappings are added to the set of equations E , the augmented set retains confluence. This is a more natural notion of correctness than what is provided by our definition of **consistency**, but also would require the use of advanced methods for proving confluence of term rewriting systems that are not necessarily terminating, and that do not have strong properties such as orthogonality (which is posed by the original set of equations E). We have been unable to find any method sufficient to show the confluence of the augmented system. One of the first and most popular method for proving confluence, the Knuth-Bendix completion algorithm, fails to apply in cases where the initial system is non-terminating. This algorithm is discussed in [4].

General information about Mathematica (and Mathematica as a programming language) can be found at the official website for Mathematica’s documentation [1]. The documentation for Mathematica 5 and earlier can also be found in book form ([6].) Another good introduction to the language is [5].

Our package shares much in common with systems such as Prolog, where statements asserts that a proposition holds of its arguments if the conjunction of other propositions hold. One major difference is that Prolog is thus restricted to booleans (whether or not some proposition holds) as opposed to general values as with our package. One can create standard functions in Prolog by including the “result” of some computation as the final argument of a proposition. For example, the proposition `Sum` could take three arguments, `x`, `y` and `z`, and `Sum(x, y, z)` would assert that `z` is the sum of `x` and `y`.

References

- [1] Mathematica documentation. <http://reference.wolfram.com/mathematica/guide/Mathematica.html>.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.
- [4] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–267. Pergamon, New York, 1970.
- [5] Leonid Shifrin. *Mathematica programming: an advanced introduction*. 2008.
- [6] Stephen Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, August 2003.

Appendix - Code Listing

Following is a complete listing of the code for our package:

```
BeginPackage["automata`"];

ClearAll[ResetLFP];

ResetLFP::usage = "ResetLFP[default] deletes all memoized
values and sets the new default value for memoized functions to
"default". ResetLFP should be called whenever a symbol definition
is changed in a way that affects the semantics of a memoized
function.";

ResetLFP[default_] :=
(
  ClearAll[LeastFun, Least, symbol$$, args$$, next$$, value$$,
    code$$, children$$, parents$$, parentsNext$$, compute, MemoQ];

  LeastFun[_Symbol] := None;
  Least[_Symbol] := None;
  args$$[_Integer] := None;
  next$$[_Integer] := None;
  value$$[_Integer] := default;
  code$$[_Integer] := NoCode;
  children$$[_Integer] := {};
  parents$$[_Integer] := {};
  parentsNext$$[_Integer] := {};
```

```

compute[_Integer] := Null;
MemoQ[_Symbol] := False;

ClearAll[makeid];
makeid[f_Symbol, r_holdID$$, val_HoldComplete] :=

With[{id = $ModuleNumber++},
  makeid[f, r, _HoldComplete] := id;
  symbol$$[id] = f;
  args$$[id] = r;
  Which[
    LeastFun[f] != None, value$$[id] = LeastFun[f] @@ r,
    Least[f] != None, value$$[id] = Least[f]
  ];

  register[id, List @@ Cases[r, id$$[id2_Integer] :> id2]];
  code$$[id] = val;
  compute[id] :=
    holdID$$ @@ Replace[val, id$$[id2_Integer] :> value$$[id2], {2}];

  update[id, byArgs];

  id
];
);
SetAttributes[ResetLFP, HoldFirst];
SyntaxInformation[ResetLFP] = {"ArgumentsPattern" -> {_}};
ResetLFP[{}];

ClearAll[holdID$$, applyholdID$$];
applyholdID$$[e_] := First@Apply[holdID$$, HoldComplete[e], {1}];

```



```

SetAttributes[applyholdID$$, HoldAllComplete];

(*" ... "*)
ClearAll[id$$, byArgs, byNext, update];
byArgs[id_Integer] :=
  Replace[compute[id], {
    holdID$$@id$$[idr_Integer] :> (setnext[id, idr]; value$$[idr]),
    holdID$$[r2_] :> (setnext[id, None]; r2)
  }];

byNext[id_Integer] := value$$@next$$[id];

update[id_Integer, fn_] :=
  With[{valnew = fn[id]},
    If[value$$[id] != valnew,
      value$$[id] = valnew;
      Scan[({update[#, byArgs] &), parents$$[id]];
      Scan[update[#, byNext] &, parentsNext$$[id]]
    ]];

(*"
  setnext : Integer => Integer => Null

  Sets 'id's 'next' value to be 'idn' and links up 'id',
  and updates parentsNext$$ as necessary.
"*)
ClearAll[setnext];
setnext[id_, idn_] :=
  With[{cn = next$$[id]},
    If[cn === idn, Return[]];

```

```

(* remove current next$$ *)
If[cn != None,
  parentsNext$$[cn] = DeleteCases[parentsNext$$[cn], id, {1}, 1]
];

(* add new next$$ *)
next$$[id] = idn;
If[idn != None, AppendTo[parentsNext$$[idn], id]];
];

(*"
  register : Integer => Integer List => Null

  Informs all of the ids listed in 'args' to inform 'id'
  when they have updated, by calling 'update' on id.
"*)
ClearAll[register];
register[id_Integer, args_List] :=
  Scan[(AppendTo[children$$[id], #]; AppendTo[parents$$[#], id]) &,
    args];

holdID$$[fst___, idi$$[x_], rst___] ^= holdID$$[fst, id$$[x], rst];

e : f_Symbol[___, _idi$$, ___] ^=
(
  With[{e2 = applyholdID$$[e]},
    idi$$ @@ {makeid[f, e2,
      Apply[f, HoldComplete @@ holdID$$[e2], {1}]]}
  ]

```

```

);

ClearAll[ClearLFP, Memo1, Memo];

ClearLFP[f__Symbol] :=
(
  Unprotect[f];
  ClearAll[f]
);
SyntaxInformation[ClearLFP] = {"ArgumentsPattern" -> {_, _ ...}};

Memo::values =
  "symbol '1' has UpValues; only DownValues are allowed for
memoization";
Memo::attributes =
  "symbol '1' has Attributes; this is not allowed for memoization";
Memo::rule = "for symbol '1', could not handle rule '2'";

Memo::usage = "Memo[symb] declares that memoization should be used
when applying symb to its arguments, and protects symb
from further modification.";
Memo1[f_Symbol] := With[{dv = DownValues[f]},
  Which[
    UpValues[f] != {}, Message[Memo::values, f],
    Attributes[f] != {}, Message[Memo::attributes, f],

    True,
    ClearAll[f];

    (* TODO: notice more complex patterns ... *)

```

```

Scan[Replace[#,
{
  (Verbatim[HoldPattern][f[args___]] :> val_) :> (

    e : f[args] :=
      idi$$ @@
      holdID$$[
        makeid[f, applyholdID$$[e], HoldComplete[val]]],
    x_ :> Message[Memo::rule, f, x]
  ]
} &, dv];

MemoQ[f] = True;

SetAttributes[f, Protected];

]];
Memo[e : __Symbol] := Scan[Memo1, {e}];
SyntaxInformation[Memo] ^= {"ArgumentsPattern" -> {_, _ ...}};
SetAttributes[Memo, ReadProtected];

ClearAll[Mapify, MapTo];
Mapify[f_Symbol, default_] := (
  ClearAll[f];
  e : f[_] :=
    idi$$ @@
    holdID$$[makeid[f, applyholdID$$[e], HoldComplete[default]]];
  Protect[f];
);

MapTo[f_Symbol, x_ -> y_] := (

```

```

Unprotect[f];
With[{id = makeid[f, applyholdID$$[f[x]], HoldComplete[y]]},
  compute[id] = y;
  update[id, byArgs];
];
Protect[f];
);

(
  LFP : code => value

  Applies 'eval' to the code.
  If the result is an _id$$, then value$$ is applied to its argument,
  otherwise the result is returned as is.
)

ClearAll[LFP];
LFP::usage = "expr calculates the least
fixed point of expr using monotonic reasoning, or loops infinitely
if no such fixed point can be found.";
LFP[e_] := Replace[holdID$$[e], {
  holdID$$@id$$[id2_] :> value$$[id2],
  holdID$$@r_ :> r
}];

SetAttributes[LFP, {HoldAllComplete, ReadProtected}];
SyntaxInformation[LFP] ^= {"ArgumentsPattern" -> {_}};

ClearAll[aut$$, DoubleBracketingBar, getArg];
DoubleBracketingBar[fst___, idi$$[x_], rst___] ^=
  DoubleBracketingBar[fst, aut$$[x], rst];

```

```

f_Symbol[fst___, aut$$[i_Integer], rst___] /;
  f != DoubleBracketingBar ^:= f[fst, value$$[i], rst];

getArg[f_Symbol*args_DoubleBracketingBar, i_Integer] :=
  Identity[args[[i]]];

```