The Ontic Inference Language

Carl Roger Witty *
B.S., Computer Science
Stanford University, 1991

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology

June 1995

©1995 Massachusetts Institute of Technology. All rights reserved.

Signature of Author

Certified by

David A. McAllester, Associate Professor
MIT Artificial Intelligence Laboratory
Thesis Advisor

Accepted by

Frederic R. Morgenthaler,
Chairman, Committee on Graduate Students

*This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.
The Ontic Inference Language

by

Carl Roger Witty

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 1995 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

OIL, the Ontic Inference Language, is a simple bottom-up logic programming language with equality reasoning. Although intended for use in writing proof verification systems, OIL is an interesting general-purpose programming language. In some cases, very simple OIL programs can achieve an efficiency which requires a much more complicated algorithm in a traditional programming language.

This thesis gives a formal semantics for OIL and some new results related to its efficient implementation. The main new result is a method of transforming bottom-up logic programs with equality to bottom-up logic programs without equality. An introduction to OIL and several examples are also included.

Thesis Supervisor: Dr. David A. McAllester

Title: Associate Professor
## Contents

Abstract .................................................. 2

Table of Contents ......................................... 3

1 Introduction ............................................. 5

2 An Informal Presentation of OIL .......................... 7
   2.1 Some Trivial Examples ............................... 7
   2.2 Parsing ........................................... 8
   2.3 Arithmetic ....................................... 10
   2.4 Primality ........................................ 11
   2.5 Goldbach's Conjecture .............................. 13
   2.6 Another Look at Primality .......................... 14
   2.7 All-Pairs Shortest Paths ............................ 14

3 The OIL Programming Language ......................... 16
   3.1 The Syntax of OIL ................................ 16
   3.2 An Operational Semantics for OIL ............... 18
      3.2.1 The Semantics of Simple OIL ............... 19
      3.2.2 Equality .................................. 21
      3.2.3 Removing Equality ........................... 23
      3.2.4 Bounds ..................................... 40
      3.2.5 The Full OIL Language ..................... 40

4 The Power of OIL ....................................... 41

5 Complexity Results and Optimizations .................. 41
   5.1 Complexity for OIL Without Equality or Negated Antecedents 42
      5.1.1 Binary Rule Sets ............................ 42
      5.1.2 The Binary Rule Transform ................. 43
      5.1.3 Acyclic Rules ................................ 44
   5.2 Complexity and Optimizations with Equality .......... 46
   5.3 Complexity and Optimizations with Negated Antecedents .... 47

6 Examples ................................................ 47
   6.1 Binary Arithmetic ................................ 48
   6.2 de Bruijn Numbers ................................ 50
   6.3 Efficient Object Language .......................... 52

7 External Syntax ........................................ 52

8 Rulesets and Orcfuns .................................. 53

9 The Ontic Rule Compiler ................................ 60
1 Introduction

Bottom-up logic programming has received considerable attention in recent years in the database literature because of its ability to concisely represent a wide variety of algorithms [NR91]. Algorithms well suited to such logic program representation include transitive closure, context free parsing, and shortest paths in graphs. Bottom-up interpreters for logic programs cache intermediate results. This makes them well suited for the representation of dynamic programming algorithms. It also makes bottom-up logic programming well suited for implementing inference systems.

This thesis describes a method for handling equality in bottom up logic programming together with various source to source transformations and implementation techniques. Equality is a difficult issue in bottom-up logic programming both because the basic equality axioms generate an infinite number of statements and because equality must be taken into account during the match phase of rule execution. Here we show that equality can be fully handled with a source to source transformation on bottom-up logic programs.

The Ontic Inference Language, OIL, is a bottom-up logic programming language with equality reasoning. OIL is also, essentially, a production system like OPS5. OIL can be used as a general-purpose programming language, although the original application, writing inference rules for some kind of inference system, guides several design choices. In some cases, such as context-free parsing, very simple OIL programs can achieve an efficiency which requires a much more complicated algorithm in a traditional programming language.

As the name suggests, OIL is designed for use in the Ontic proof verification system [McA]; it is a variant of the language in which Ontic’s inference rules are currently written. The existing Ontic low-level inference language has a number of irregularities and special cases which make it difficult to understand and use. Also, the implementation is old, and has grown complex by accretion over the years. OIL is more general and expressive.

In [NR91], Naughton and Ramakrishnan describe the Magic Templates transfor-
mation, which transforms a pure Prolog program (intended to be evaluated top-down) into an equivalent program which can be evaluated efficiently with bottom-up fixpoint evaluation. They also describe several program transformations which, in conjunction with the Magic Templates algorithm, yield even more efficient programs. Essentially, the transformation gains efficiency by providing full memoing. OIL programs are typically written in a style similar to the results of the Magic Templates transformation; an implementation of OIL could be used as the back end of a system which used Magic Templates.

In [GST90], the authors present a method for splitting up parallelizing bottom-up logic programs for distributed memory computers. The method depends on splitting the program so that parts of it execute on different processors, with minimal communication between the processors. Unfortunately, their work is not directly applicable to OIL, because the only known efficient algorithm for implementing OIL depends on a shared data structure, the congruence grammar data structure.

OIL is also related to existing work in production systems and logic programming. Production systems are often used to create expert systems. Essentially, a production system [New73] consists of a set of rules (sometimes known as productions; hence the name) and a database encoding the current knowledge of the system (this database is known as the working memory, and the facts in the database are called working memory elements, or WME's). The rules are of the form $A_1, \ldots, A_n/C$, where the $A_i$ are the antecedents and $C$ is the conclusion. The antecedents of the rules are patterns which are matched against the WME's.

A production system repeatedly finds a rule which can fire (one such that all the antecedents match WME's) and fires it (executes its conclusion). Each possible rule firing (a rule and a sequence of WME's that match the antecedents of the rule) is called an instantiation. Some production systems compute the set of all instantiations (the conflict set) and select one of them. (For instance, the instantiation chosen might be the one added to the conflict set most recently.) The conclusion of a rule may add or delete working memory elements, or it may specify other actions, such as I/O. OPS5 [For81] is one of the best-known languages for writing production systems. It
is often used for writing expert systems.

Except for its handling of equality, OIL is a particularly simple production system. The only action allowed in the conclusion of an OIL rule is to add a fact to the database. The language of equality-free OIL can be implemented as a simple variant on an existing production system. In fact, it is possible to compile an equality-free OIL program into OPS5 such that the OPS5 program will have only a linear factor more rules firings than the OIL program; the only difficult point in the translation is ensuring fairness of the resulting OPS5 program.

As with other production systems, the match process (computing the conflict set) is the performance bottleneck in OIL. Much work has been done on discovering efficient match algorithms for production systems. One of the most popular match algorithms is RETE [For82], which is used in many implementations of OPS5. Another match algorithm called TREAT [Mir90] was developed by Miranker. TREAT was designed for parallel machines, but experiments show that it is more efficient than RETE on serial architectures as well.

2 An Informal Presentation of OIL

The basic operation of OIL is to take a set of facts \( \Sigma \), a set of rules (a program) \( R \), and a fact \( \Phi \), and report whether \( \Phi \) can be deduced from \( \Sigma \) by \( R \). Equality reasoning is built into OIL.

2.1 Some Trivial Examples

Here are some examples of simple OIL programs.

The simplest program is the empty program, \( R = \emptyset \). Suppose that \( R = \emptyset \) and \( \Sigma = \emptyset \). The only facts which can be deduced in this case are the reflexive equality facts, such as \( = (f(g(a)), f(g(a))) \).

Suppose \( \Sigma \) contains the facts \( = (a, b) \) and \( P(a) \). From this, it is possible to deduce facts such as \( P(b) \) and \( = (f(a, b), f(b, a)) \).
Now consider a simple OIL program, which consists of a single rule: \((f(x), g(x))/special(x)\). This says that for any \(x\), if \(f(x)\) is known to be equal to \(g(x)\), then \(x\) is special. For instance, if \(\Sigma\) contains \((f(a), g(a))\), then we can deduce that \(a\) is special \((special(a))\). If \(\Sigma\) contains \((f(a), g(b))\) and \((a, b)\), then we can deduce both \(special(a)\) and \(special(b)\). (And, as always, we can deduce random equality facts like \((h(f(b)), h(g(a)))\).)

2.2 Parsing

Let’s move on to a slightly more interesting example, parsing. Suppose we want to test whether sentences can be produced by the following context free grammar.

\[
\begin{align*}
\langle\text{Sent}\rangle & ::= \langle\text{NP}\rangle\langle\text{VP}\rangle \\
\langle\text{NP}\rangle & ::= \langle\text{Art}\rangle\langle\text{Noun}\rangle \\
\langle\text{VP}\rangle & ::= \langle\text{Trans}\rangle\langle\text{NP}\rangle \mid \langle\text{Intrans}\rangle \\
\langle\text{Art}\rangle & ::= a \mid \text{the} \\
\langle\text{Noun}\rangle & ::= \text{cat} \mid \text{rat} \mid \langle\text{Adj}\rangle\langle\text{Noun}\rangle \\
\langle\text{Adj}\rangle & ::= \text{green} \mid \text{happy} \mid \text{small} \\
\langle\text{Trans}\rangle & ::= \text{ate} \\
\langle\text{Intrans}\rangle & ::= \text{died}
\end{align*}
\]

This grammar produces sentences such as “A cat ate the rat,” “The rat died,” and “A happy green happy small green cat ate a green small mouse.”

We need to encode such sentences as terms, which will look like

\[
\text{cons}(a, \text{cons}(\text{cat}, \text{cons}(\text{ate}, \text{cons}(\text{the}, \text{cons}(\text{rat}, \text{nil})))))).
\]

Given this encoding, we will say that \(x\) is produced by the above grammar iff

\[
\{\text{parse}(x)\} \overset{\text{OIL}}{\vdash} R_P \Sent(x, \text{nil}),
\]

where \(R_P\) is the following set of rules. It should be clear how to transform any context free grammar into a similar set of rules.
propagate-parse
\[
\begin{align*}
\text{parse}(\text{cons}(x, y)) & \quad \text{parse}(y) \\
\text{parse}(y) & \\
\end{align*}
\]

parse-np
\[
\begin{align*}
\text{parse}(x) & \\
\text{Art}(x, y) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\end{align*}
\]

parse-vp
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-art
\[
\begin{align*}
\text{parse}(x) & \\
\text{Art}(x, y) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\end{align*}
\]

parse-vp2
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-vp3
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-noun
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-noun2
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-intrans
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

parse-sent
\[
\begin{align*}
\text{parse}(x) & \\
\text{NP}(x, y) & \\
\text{VP}(y, z) & \\
\text{Sent}(x, z) & \\
\end{align*}
\]

parse-np1
\[
\begin{align*}
\text{parse}(x) & \\
\text{Trans}(x, y) & \\
\text{NP}(y, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-np2
\[
\begin{align*}
\text{parse}(x) & \\
\text{Art}(x, y) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\end{align*}
\]

parse-vp1
\[
\begin{align*}
\text{parse}(x) & \\
\text{Trans}(x, y) & \\
\text{NP}(y, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-art1
\[
\begin{align*}
\text{parse}(x) & \\
\text{Trans}(x, y) & \\
\text{NP}(y, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-noun1
\[
\begin{align*}
\text{parse}(x) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-noun2
\[
\begin{align*}
\text{parse}(x) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-noun3
\[
\begin{align*}
\text{parse}(x) & \\
\text{Noun}(y, z) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-adj
\[
\begin{align*}
\text{parse}(x) & \\
\text{Adj}(x, y) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-adj1
\[
\begin{align*}
\text{parse}(x) & \\
\text{Adj}(x, y) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-adj2
\[
\begin{align*}
\text{parse}(x) & \\
\text{Adj}(x, y) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-adj3
\[
\begin{align*}
\text{parse}(x) & \\
\text{Adj}(x, y) & \\
\text{NP}(x, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-trans
\[
\begin{align*}
\text{parse}(x) & \\
\text{Trans}(x, y) & \\
\text{NP}(y, z) & \\
\text{VP}(x, z) & \\
\end{align*}
\]

parse-intrans
\[
\begin{align*}
\text{parse}(x) & \\
\text{Intrans}(x, y) & \\
\text{VP}(x, y) & \\
\end{align*}
\]

Here, there is a binary relation for every nonterminal, and it is easy to prove by
induction that for a nonterminal $N$, $\{\text{parse}(x)\} \overset{\text{OIL}}{\Rightarrow} N(x, y)$ iff $N$ produces $z$ and $x = \text{append}(z, y)$.

If $\text{propagate-parse}$ was removed and the $\text{parse}(x)$ antecedent was removed from each rule (call the resulting rule set $P'$), we would have $\emptyset \overset{\text{OIL}}{\Rightarrow} N(x, y)$ iff $N$ produces $z$ and $x = \text{append}(z, y)$. However, the resulting OIL program would not terminate. (As we will see below, the semantics of OIL programs are well-defined even if they don’t terminate.)

You may wish to compare this treatment of parsing with the parsing example in Section 8.

### 2.3 Arithmetic

In this section we’ll see how to write OIL programs to do unary arithmetic.

We will represent zero as $\text{zero}$, and the successor function as $s$, so 3 will be represented as $s(s(s(\text{zero})))$. First, we show how to compute addition.

\[
\begin{align*}
\text{add-base-try-1} & \quad \text{add-rec-try-1} \\
\overset{=}{(+ (\text{zero}, x), x)} & \quad 
\overset{=}{(+ (s(x), y), +(x, s(y)))}
\end{align*}
\]

This has the disadvantage that it asserts facts like

\[
= (+ (s(\text{zero}), \text{cons}(\text{the}, \text{nil})), s(\text{cons}(\text{the}, \text{nil}))).
\]

We can avoid this by adding $\text{compute}$, which restricts our attention in much the same way that $\text{parse}$ did in the above section.

\[
\begin{align*}
\text{add-base-try-2} & \quad \text{add-rec-try-2} \\
\text{compute}(x) & \quad \text{compute}(x) \\
\overset{=}{(x, +(\text{zero}, z))} & \quad \overset{=}{(x, +(s(y), z))} \\
\overset{=}{(x, z)} & \quad \overset{=}{(x, +(y, s(z)))}
\end{align*}
\]

This version terminates, unlike the try-1 version, but it requires you to assert $\text{compute}$ facts. In this section we’ll take a third approach which doesn’t terminate but which is slightly simpler to work with. First, we’ll mark all the numbers.
Now we can compute all the addition facts for all the numbers.

\[
\text{add-base} \quad \frac{\text{number}(x)}{=+(\text{zero}, x), x}
\]

\[
\text{add-rec} \quad \frac{\text{number}(x) \quad \text{number}(y)}{=+(s(x), y), +(x, s(y))}
\]

Multiplication is almost as easy.

\[
\text{mult-base} \quad \frac{\text{number}(x)}{=*(\text{zero}, x), \text{zero}}
\]

\[
\text{mult-rec} \quad \frac{\text{number}(x) \quad \text{number}(y)}{=*(s(x), y), +(s(x), y))}
\]

Let \( R_A \) be the set of rules \text{number-base}, \text{number-rec}, \text{add-base}, \text{add-rec}, \text{mult-base}, \) and \text{mult-rec}. When we get to the formal semantics, we will see that it takes time \( \omega \) to compute the closure of the empty set under the rule set \( R_A \).

### 2.4 Primality

It’s time to take a look at two more features of OIL; priorities and negated antecedents.

Until now, all the rules have had the default priority number of 0. OIL never fires a rule when there is a rule firing available with a lower priority number.

Let’s look at a variant of \( R_A \).
Given the above rule set, OIL will first mark all the numbers as numbers, then it will compute all addition facts, then it will compute all multiplication facts. (The time taken is thus $3\omega$.) However, OIL will come up with exactly the same final set of facts, so the priorities don’t really make a difference here. In fact, without negated antecedents, priorities never change the meaning of a program, although they may change its running time.

Here’s a case where priorities are important. Consider the following rules, which will be called $R_{Pr}$.

The rule composite says that the product of any two numbers, both greater than one, is composite. (If $x$ is a number, then $s(s(x))$ is a number which is greater than one.) The rule prime says that any number which is neither zero, one, nor composite, is prime. The latter rule uses negated antecedents; the rule can fire if there is a set of variable bindings such that all the positive antecedents match known facts and none of the negated antecedents match known facts.

The rules $R_A \cup R_{Pr}$ do indeed compute primality. We get $\emptyset \models^{OIL}_{R_A \cup R_{Pr}} \text{prime}(x)$ iff $x$ is the unary representation of a prime number. (The rules take time $2\omega$ to compute.)
Here, the priorities are vital. If \textit{prime} were at the default priority of zero, then OIL could start off by deducing the following facts:

\begin{align*}
number(zero) \\
number(s(zero)) \\
number(s(s(zero))) \\
number(s(s(s(zero))))
\end{align*}

and decide that 4 was prime, although it would continue on and eventually decide that 4 was composite, as well. Alternatively, OIL could fire the rules in the “right” order and correctly compute the set of primes. Thus, without priorities, these rules are nondeterministic, and may give the wrong answer.

### 2.5 Goldbach’s Conjecture

Let’s write an OIL program to decide Goldbach’s Conjecture. Goldbach’s Conjecture states that every even number greater than 2 is the sum of two primes. Consider the following set of rules, $R_G$.

\[
\begin{align*}
prime(x) \\
prime(y) \\
\textit{goldbach-ok}(+(x,y)) \\
\text{number}(x) \\
=(y,s(s(x))) \\
=(z,+(y,y)) \\
\neg \textit{goldbach-ok}(z)
\end{align*}
\]

If Goldbach’s Conjecture is true, then $\emptyset \vdash_{R_A \cup R_P \cup R_G} \text{yes}$. If Goldbach’s Conjecture is false, then $\emptyset \vdash_{R_A \cup R_P \cup R_G} \text{no}$. Of course, a real implementation of OIL will not terminate when presented with this program; it takes time $2\omega + 2$. 

\[\text{number}(zero)\]
\[\text{number}(s(zero))\]
\[\text{number}(s(s(zero)))\]
\[\text{number}(s(s(s(zero))))\]
\[\text{prime}(s(s(s(s(zero)))))\]
2.6 Another Look at Primality

It’s hard to find an example of an OIL program that takes time $\omega^2$ or more. Here’s one; it’s an alternative method for computing primes. This set of rules will be called $R_{P_{r2}}$.

\[
\begin{align*}
\text{ready}(s(s(zero))) & \quad \text{ready}(x) \\
\text{done}(x) & \quad \text{done}(x) \\
\text{prime}(x) & \quad \text{prime}(x) \\
\text{number}(y) & \quad \text{number}(y) \\
\text{composite}(*(s(s(y)), x)) & \quad \text{composite}(x) \\
\text{composite}(x) & \quad \text{composite}(x) \\
\text{done}(x) & \quad \text{done}(x)
\end{align*}
\]

This program implements the Sieve of Eratosthenes. It searches through the numbers in order, starting with 2. Whenever it finds an unmarked number, it marks that number as prime, and all its multiples as composite, and then continues with its search. Since it has to mark all the multiples of every prime number, the program $R_A \cup R_{P_{r2}}$ takes time $\omega^2$.

2.7 All-Pairs Shortest Paths

There’s one feature of OIL we haven’t dealt with yet: bounds. Every term can have a nonnegative upper bound associated with it. Here’s an example of an OIL program which uses bounds facts. The following set of rules will be called $R_D$. 

14
This is a particularly simple implementation of an all-points shortest-paths algorithm for directed graphs. The input contains a fact of the form \(\leq (\text{arc}(a, b), w)\), for every arc between nodes \(a\) and \(b\) of weight \(w\). It will deduce a fact of the form \(\leq (\text{distance}(c, d), w')\) iff there is a path from \(c\) to \(d\) of weight less than or equal to \(w'\).

Rules \textit{refl-dist1} and \textit{refl-dist2} are particularly interesting. It’s tempting to replace them with a single rule \([\text{refl-dist}] / \leq (\text{distance}(x, x), 0)\) saying that the distance between any node and itself is 0, but this rule could fire an infinite number of times, leading to inevitable non-termination.

It’s slightly more complicated to keep track of the actual paths. The following set of rules will be called \(R_{D2}\).

\[
\begin{align*}
\leq (\text{arc}(x, y), d) & \leq (\text{distance}(x, y, \text{cons}(\text{arc}(x, y), \text{nil})), d) \\
\leq (\text{distance}(x, y, p), d_1) \quad & \leq (\text{distance}(y, z, p_2), d_2) \\
\leq (\text{distance}(x, z, \text{append}(p_1, p_2)), +(d_1, d_2)) \\
\leq (\text{arc}(x, y), d) & \leq (\text{distance}(x, x, \text{nil}), 0) \\
\leq (\text{arc}(x, y), d) & \leq (\text{distance}(y, y, \text{nil}), 0)
\end{align*}
\]

The above rule set doesn’t terminate, because it can deduce arbitrarily large facts of the form

\(\leq (\text{distance}(n1, n1, \text{append}(\text{append}(\text{nil}, \text{nil}), \text{nil})), 0)\),

and it can examine paths which are obviously non-optimal, in that the same arc appears multiple times.

It’s an open question whether OIL can use a program similar to \(R_D\) to efficiently
compute all-pairs shortest paths in a form which actually allows the paths to be extracted.

3 The OIL Programming Language

The OIL programming language deals with a (possibly infinite) database of facts about terms; the database consists of the facts which are currently "known." An OIL program consists of a set of rules. A rule has a set of antecedents and a conclusion; if there is a variable substitution such that every antecedent in the rule matches a known fact, then the rule can fire, and the conclusion (under the variable substitution) will be added to the conclusion.

3.1 The Syntax of OIL

Definition 1 (term) A term $\sigma$ is defined by the following grammar.

$$
F := x^f \mid f \\
\sigma := x \mid F(\sigma_1, \sigma_2, \ldots, \sigma_n)
$$

Definition 2 (formula) A formula $\Phi$ is defined by the following grammar.

$$
b := n \mid x^b \\
\Phi := \sigma \mid = (\sigma_1, \sigma_2) \mid \leq (\sigma, b)
$$

Definition 3 (fact) A fact is a ground formula.

There is an infinite set of function symbols, which is disjoint from the set of predicates. Predicates include $=$, $\leq$, and $=_{f}$ (we will see $=_{f}$ in the grammar for rules below); we will discuss other predicates later.

There are three infinite distinct sets of variables; term variables, function variables, and bounds variables. In the grammars above, $x$ is a term variable, $x^f$ is a function variable, and $x^b$ is a bounds variable. Function variables can be bound to arbitrary function symbols. Bounds variables can be bound to nonnegative integers.

Here are some examples of ground terms.
\(+ (\text{succ(succ(zero)}), \text{succ(zero)})\
\text{arc(node1, node7)}\
\text{forall}(x, \text{exists}(y, \text{greater}(y, x)))\

And here are some examples of terms with variables.

\text{is(a\_brother\_of(a\_sister\_of(x)}, a\_brother\_of(x))\
\text{and(subset(x, y), subset(y, x))}\

Note that every term is a formula; the above sets of examples are also examples of facts and formulas. Here are some more facts.

\text{=(the\_mother\_of(fred), the\_mother\_of(harry))}\
\text{greater(big\_omega(exp(n)), big\_oh(*(n, n)))}\

And some more formulas.

\text{\leq(arc(x, y), xylen)}\
\text{=(x, *(one, x))}\

**Definition 4 (rule)** A rule \( \Lambda \) is defined by the following grammar and constraints.

\[ B := b \mid +(B_1, B_2) \mid *(B_1, B_2) \]
\[ \hat{\Phi} := \Phi \mid \leq(\sigma, B) \]
\[ A := \Phi \mid \neg \hat{\Phi} \mid =_f(F_1, F_2) \mid \neg =_f(F_1, F_2) \]
\[ \Lambda := A_1, \ldots, A_k/\hat{\Phi} (n) \]

The \((n)\) is the priority of the rule; priorities are integers. Lower numbers mean higher priorities. The default priority is 0. All rules must have nonnegative priorities.

**Definition 5 (program)** An OIL program is a set of rules.

There are several source-to-source transformations in this thesis. The target language of these transformations is slightly richer than the language given by the grammars above; it allows for an infinite set of predicates.

**Definition 6 (extended formula)** An extended formula \( \Phi_e \) is defined by the following grammar.

\[ \delta := \sigma \mid F \mid b \]
\[ \Phi_e := \sigma \mid P(\delta_1, \delta_2, \ldots, \delta_n) \]
Definition 7 (extended fact) An extended fact is a ground extended formula.

Definition 8 (extended rule) An extended rule \( \Lambda_e \) is defined by the following grammar.

\[
\begin{align*}
\hat{\Phi}_e & := \Phi_e \mid \leq (\sigma, B) \\
A_e & := \Phi_e \mid \neg \hat{\Phi}_e \\
\Lambda_e & := A_{e1}, \ldots, A_{ek} / \hat{\Phi} (n)
\end{align*}
\]

Extended rules are allowed to have negative priorities.

Definition 9 (extended program) An extended program is a set of extended rules.

3.2 An Operational Semantics for OIL

This section defines a formal, mathematical semantics for OIL; it also defines a standard for implementations of OIL, which is slightly different from the mathematical specification in ways which will be explained later.

As stated above, the basic operation of OIL is to say whether a fact \( \Phi \) can be deduced from an OIL program \( R \) and a set of facts \( \Sigma \). OIL is nondeterministic, so it is possible that \( \Phi \) could be deduced along some computation paths but not others. If \( \Phi \) can be deduced along every computation path, we write \( \Sigma \models_{\text{OIL}} R \Phi \); if it cannot be deduced along any computation path, we write \( \Sigma \not\models_{\text{OIL}} R \Phi \). Note that it is possible for neither \( \Sigma \models_{\text{OIL}} R \Phi \) nor \( \Sigma \not\models_{\text{OIL}} R \Phi \) to be true.

An implementation of OIL also takes \( R, \Sigma, \) and \( \Phi \). Under some conditions (which will be exactly specified later) the implementation is allowed not to terminate. If it does terminate, and \( \Sigma \models_{\text{OIL}} R \Phi \), then the implementation is required to return yes; if \( \Sigma \not\models_{\text{OIL}} R \Phi \), then the implementation is required to answer no. If neither \( \Sigma \models_{\text{OIL}} R \Phi \) nor \( \Sigma \not\models_{\text{OIL}} R \Phi \), the implementation may answer yes, no, or ambiguous.

The semantics of the full OIL programming language are rather complex; I will build up to the full semantics by describing the semantics of ever more complete subsets of OIL. We will actually define the semantics of extended OIL, which is OIL with arbitrary predicates and without certain syntactic restrictions on rules. Remember that extended OIL exists only to be the target of source-to-source transformations on rules; user programs must be written entirely in unextended OIL.
3.2.1 The Semantics of Simple OIL

First, I will describe the semantics of “Simple OIL”.

**Definition 10 (Simple OIL)** Simple OIL is OIL without equality or bounds facts.

The semantics given in this section are somewhat complicated to allow a particularly simple statement of the semantics of equality.

This section defines the relation of “simple implication,” where a set of extended facts $\Sigma$ simply implies $\Phi_e$ under $R$ ($\Sigma \vdash_R \Phi_e$) if $\Sigma$, $R$, and $\Phi_e$ do not include any equality or bounds formulas and $\Sigma \vdash_R \Phi_e$.

**Definition 11 (variable substitution)** Throughout this document, a substitution $\rho$ is a function which maps term variables to terms, function variables to function symbols, and bounds variables to nonnegative integers.

**Definition 12 (rule firing)** A rule firing is a pair $(r, \rho)$, where $r$ is an extended rule and $\rho$ is a variable substitution whose domain is the set of variables in $r$.

**Definition 13 (acceptable rule firing)** Let $r = A_1, \ldots, A_k/C(n) \in R$. A rule firing $(r, \rho)$ is acceptable for $\Sigma$ under $R$, if for every positive antecedent $A_i$ of $r$, $A_i\rho \in \Sigma$; for every negative antecedent $A_j = \neg \Phi_e$ of $r$, $\Phi_e\rho \notin \Sigma$; for the conclusion $C$ of $r$, $C\rho \notin \Sigma$; and there is no acceptable rule firing of a rule in $R$ with a priority smaller than $n$.

Note that a rule firing which would add a fact which is already known is not acceptable.

**Definition 14 (FR)** Let $FR(\Sigma)$ be the set of acceptable rule firings for $\Sigma$ under $R$.

**Definition 15 (IR)** Let $IR(\Sigma) = \{C\rho \mid (A_1, \ldots, A_k/C(n), \rho) \in FR(\Sigma)\}$. That is, $IR(\Sigma)$ is the set of extended facts which can be derived from $\Sigma$ in one step, using $R$.

**Definition 16 (state history)** A state history for $\Sigma$ under $R$ is a function $f$ from a prefix of the ordinals to states with the following properties. Let the domain of $f$ be $\kappa$. First, $f(0) = \Sigma$. Second, if $\eta < \kappa$ is a successor ordinal, then $f(\eta) = f(\eta-1) \cup \{\Phi_e\}$, for some $\Phi_e \in IR(f(\eta-1))$. Third, if $\eta < \kappa$ is a limit ordinal, then $f(\eta) = \bigcup_{\eta' < \eta} f(\eta')$. 

19
An OIL computation is completed when there are no more acceptable rule firings; that is, when $I_R(\Sigma) = \emptyset$. This motivates the following definition.

**Definition 17 (complete state history)** A complete state history for $\Sigma$ under $R$ is a state history $f$ for $\Sigma$ under $R$ such that the domain of $f$ has a largest element $\eta$, and $I_R(f(\eta)) = \emptyset$.

**Definition 18 (closure)** Let $f$ be a complete state history for $\Sigma$ under $R$, and let $\eta$ be the largest element of the domain of $f$. Then $f(\eta)$ is a closure of $\Sigma$ under $R$.

**Definition 19 ($C_R$)** Let $C_R(\Sigma)$ be the set of all possible closures of $\Sigma$ under $R$.

**Definition 20 ($CIR$)** If $|C_R(\Sigma)| = 1$, then $CIR(\Sigma)$ is the unique member of $C_R(\Sigma)$. Otherwise, $CIR(\Sigma)$ is undefined.

We can also define “fair” versions of the above concepts.

**Definition 21 (fair state history)** A fair state history is a state history $f$ (with domain $\kappa$) with the additional property that for every rule firing $(r, \rho)$, the set $\{\eta \mid \eta < \kappa \text{ and } (r, \rho) \in F_R(f(\eta))\}$ is finite. In other words, no rule firing will be acceptable infinitely many times without actually being fired.

**Definition 22 (fair complete state history)** A fair complete state history is a function which is both a fair state history and a complete state history.

**Lemma 1** For every $R$ and $\Sigma$, there exists a fair complete state history for $\Sigma$ under $R$.

**Definition 23 (fair closure)** Let $f$ be a fair complete state history for $\Sigma$ under $R$, and let $\eta$ be the largest element of the domain of $f$. Then $f(\eta)$ is a fair closure of $\Sigma$ under $R$.

**Definition 24 ($CIR'$)** Let $CIR'(\Sigma)$ be the set of all possible fair closures of $\Sigma$ under $R$.

**Definition 25 ($CIR'$)** If $|CIR'(\Sigma)| = 1$, then $CIR'(\Sigma)$ is the unique member of $CIR'(\Sigma)$. Otherwise, $CIR'(\Sigma)$ is undefined.
Lemma 2  For every $R$ and $\Sigma$, if there are at most $n$ distinct priority levels used in $R$, then no fair state history for $\Sigma$ under $R$ can have a domain larger than $\omega^n + 1$. In other words, fair inference always terminates, in some transfinite number of steps.

Definition 26 (Simple OIL implication) If $\Phi_e$ is a member of every member of $C'_R(\Sigma)$, we say $\Sigma \vdash_R \Phi_e$. If $\Phi_e$ is not a member of any member of $C'_R(\Sigma)$, we say $\Sigma \not\vdash_R \Phi_e$. Otherwise, if $\Phi_e$ is a member of some members of $C'_R(\Sigma)$ but not of others, the truth value of $\Sigma \vdash_R \Phi_e$ is undefined.

Definition 27 (termination) A program $R$ terminates on $\Sigma$ if every member of $C'_R(\Sigma)$ is finite.

Definition 28 (terminating program) A program $R$ is terminating if it terminates on every finite set of extended facts.

In the absence of bounds and equality facts, if $R$ is a set of rules, $\Sigma \vdash^\text{OIL}_R \Phi$ iff $\Sigma \vdash_R \Phi$.

In the absence of bounds and equality facts, if $R$ terminates on $\Sigma$, then an implementation of OIL is required to terminate when asked whether $\Sigma \vdash^\text{OIL}_R \Phi$.

Lemma 3 If $R$ has no negated antecedents, then $C_R(\Sigma) = C'_R(\Sigma)$ and $|C_R(\Sigma)| = 1$. (Thus, if $R$ has no negated antecedents, $C_I_R(\Sigma)$ is always defined.)

3.2.2 Equality

We can define the semantics of OIL with equality simply by specifying a set of extended rules which are to be incorporated into every OIL program.

Definition 29 (equality rules) The following set of rules will be called $R_\text{ex}$.
symmetric
\[ (x, y) \quad \Rightarrow \quad (y, x) \quad (-1) \]

transitive
\[ (x, y) \quad \Rightarrow \quad (y, z) \quad \Rightarrow \quad (x, z) \quad (-1) \]

equal-truth
\[ (x, y) \quad \Rightarrow \quad y \quad (-1) \]

function-equality
\[ f(x', x') \quad \Rightarrow \quad (-1) \]

\textit{subst-k}
\[ (x_1, y_1) \quad \Rightarrow \quad (x_2, y_2) \quad \Rightarrow \quad \cdots \quad \Rightarrow \quad (x_k, y_k) \quad \Rightarrow \quad (x'(x_1, \ldots, x_k), x'(y_1, \ldots, y_k)) \quad (-1) \]

There is one instance of rule \textit{subst-k} for every nonnegative integer \( k \).

These are not legal rules in unextended OIL, for several reasons. First, they have negative priorities. Second, the conclusion of \textit{function-equality} is a \( =_f \) fact, which is not allowed as a conclusion in unextended OIL.

Note that the \textit{subst-k} rules imply that equality is reflexive.

\textbf{Lemma 4} \( |C'_{R_{=}^c}(\Sigma)| = 1 \). That is, equality reasoning is deterministic.

\textbf{Definition 30} \( (C_{=}) \) The closure under equality of \( \Sigma \) \( (C_{=}^c(\Sigma)) \) is the sole member of \( C'_{R_{=}^c}(\Sigma) \).

Clearly, the rule set \( R_{=} \) does not terminate on any \( \Sigma \); nor does any rule set which contains \( R_{=} \). We will define a new concept, user termination.

\textbf{Definition 31 (user rule)} A \textit{user rule} is a rule with nonnegative priority.

\textbf{Definition 32 (system rule)} A \textit{system rule} is a rule with negative priority. (All system rules are thus extended rules.)

\textbf{Definition 33 (user termination)} An \textit{extended rule set} \( R \) user terminates on \( \Sigma \) if every fair state history for \( \Sigma \) under \( R \) has only a finite number of rule firings of user rules.

22
Definition 34 (user terminating program) An extended rule set is user terminating if it user terminates on every finite $\Sigma$.

In the absence of bounds facts, if $R$ is a set of user rules, $\Sigma \vdash^\text{OIL}_R \Phi$ iff $\Sigma \vdash^\text{OIL}_{R \cup R_\text{=} \cup R_\text{U}} \Phi$.

In the absence of bounds facts, if $R$ is a set of user rules and $R \cup R_\text{=} \cup R_\text{U}$ user terminates on $\Sigma$, then an implementation of OIL is required to terminate when asked whether $\Sigma \vdash^\text{OIL}_R \Phi$.

As I stated above, the simplest rule set $R$ such that $R \cup R_\text{=} \cup R_\text{U}$ user terminates but does not terminate is $R = \emptyset$. Here’s another example; this set of rules will be called $R_U$.

\[
\frac{P(x)}{P(+x, \text{zero})} \quad \frac{P(x)}{=(x, +(x, \text{zero})))}
\]

Suppose $\Sigma$ is $P(a)$. To see that $R_U \cup R_\text{=} \cup R_\text{U}$ user terminates on $\Sigma$, consider a fair state history $f$ for $\Sigma$ under $R_U \cup R_\text{=} \cup R_\text{U}$. Consider the rule firing $\langle \text{equal-plus-zero}, \{x \mapsto a\} \rangle$. If this is fired, it deduces $=(a, +(a, \text{zero}))$, and the equality rules in $R_\text{=} \cup R_\text{U}$ take over to deduce all the facts that $P$-plus-zero could have deduced.

Since $f$ is fair, this rule firing will be acceptable only a finite number of times before being fired. Since it is acceptable every time another user rule fires, there can be only a finite number of user rule firings in $f$; therefore, $R_U \cup R_\text{=} \cup R_\text{U}$ user terminates on $\Sigma$.

3.2.3 Removing Equality

It’s not obvious how to implement OIL such that the above termination condition is met, in the presence of equality. In [McA92], McAllester presents an algorithm which will work, but here we’ll take a different approach. I will present a transformation $T$ on unextended rule sets such that $T(R) \cup R_\text{=} \cup R_\text{U}$ is essentially equivalent to $R \cup R_\text{=} \cup R_\text{U}$, but whenever $R \cup R_\text{=} \cup R_\text{U}$ user terminates on $\Sigma$, $T(R) \cup R_\text{=} \cup R_\text{U}$ terminates on $\Sigma$. (There are restrictions on the rule sets and sets of facts for which this result holds; but it holds for every terminating rule set. The exact restrictions will be stated below.)

23
Here's what I mean by “essentially equivalent”. For every $R$ and $\Sigma$ there is a
function from members of $C'_T(R)\cup R_\Sigma$ to members of $C'_R\cup R_\Sigma$ such that the following
holds. Let $\Gamma$ be a member of $C'_T(R)\cup R_\Sigma$, and let $\Delta$ be the corresponding member of
$C'_R\cup R_\Sigma$; then the unextended facts of $C_\Sigma(\Gamma)$ are exactly $\Delta$.

**Congruence Grammars** The transformation is essentially a method for encoding
congruence grammars [McA92] in rules. A congruence grammar is a very simple normal
form for sets of facts which are closed under $R_\Sigma$; it can also be quite compact. The
compactness is because congruence grammars work in terms of equivalence classes,
and don’t store equivalent facts about different members of the same equivalence
class.

For instance, suppose that the following facts are known at some point in the
inference process.

\begin{align*}
  &= (f(a), a) \\
  &= (f(b), f(c)) \\
  P(f(a), f(b))
\end{align*}

From these facts, the rule set $R_\Sigma$ can derive an infinite set of further facts, such as
$=(f(f(f(f(a)))), f(f(a)))$. The congruence grammar representation of this set follows.

\begin{align*}
  TRUE & \to Z \\
  Z & \to P(V, Y) \\
  V & \to a \\
  V & \to f(V) \\
  Y & \to f(W) \\
  Y & \to f(X) \\
  W & \to b \\
  X & \to c
\end{align*}

As another example, here’s the congruence grammar representation of the closure
of $R_U \cup R_\Sigma$ on $P(a)$.
\[ TRUE \rightarrow Z \]
\[ Z \rightarrow P(X) \]
\[ X \rightarrow a \]
\[ X \rightarrow +(X, Y) \]
\[ Y \rightarrow \text{zero} \]

**Definition 35 (congruence grammar)** A congruence grammar is a context free grammar of the following form. There is a distinguished nonterminal TRUE. Every production with TRUE as its left hand side has another nonterminal as the right hand side. All other productions have the application of a function symbol to a (possibly empty) list of nonterminals as the right hand side. No two productions have the same right hand side.

**Lemma 5** For every congruence grammar \( G \), and every pair of distinct nonterminals \( X \) and \( Y \) of \( G \), the sets of terms produced by \( X \) and \( Y \) are disjoint.

If we view the sets of terms produced by different nonterminals of a congruence grammar \( G \) as equivalence classes, then such a grammar encodes a congruence relation (hence the name "congruence grammar").

**Definition 36 (fact set of a congruence grammar)** The fact set of a congruence grammar \( G \) (written as \( \Sigma_G \)) is the set of facts produced by the nonterminal TRUE, plus some equality facts. Specifically, let \( \Sigma'_G \) be the set of facts produced by the nonterminal TRUE, plus, for every nonterminal \( X \) of \( G \) and every pair of terms \( \sigma_1 \) and \( \sigma_2 \) produced by \( X \), the fact \( = (\sigma_1, \sigma_2) \). Then \( \Sigma_G = C_= (\Sigma'_G) \).

A congruence grammar can easily be encoded as a set of facts. For every nonterminal \( N \), let \( \sigma_N \) be some term produced by that nonterminal. Then the following is a direct translation of the above congruence grammar.
\[
\begin{align*}
\sigma_Z &= (\sigma_Z, P(\sigma_Y, \sigma_Y)) \\
&= (\sigma_Y, a) \\
&= (\sigma_Y, f(\sigma_Y)) \\
&= (\sigma_Y, f(\sigma_X)) \\
&= (\sigma_Y, f(X)) \\
&= (\sigma_Y, b) \\
&= (\sigma_X, c)
\end{align*}
\]

From now on, we will consider the set of facts to be the true representation of a congruence grammar, and the set of productions to be syntactic sugar. When we speak of a nonterminal of a congruence grammar, we will mean some term \( \sigma_N \).

**Lemma 6** For every set of facts \( \Sigma \), there is an equivalent congruence grammar \( Gr(\Sigma) \) such that \( C = (S) = C = (Gr(\Sigma)) \).

**Definition 37** Given a congruence grammar \( G \) and a nonterminal \( N \) of \( G \), \( G(N) \) is the set of terms which can be produced by \( N \), when \( G \) is viewed as a context free grammar.

**Definition 38 (interned)** If \( \sigma \in G(N) \) for some nonterminal \( N \) of \( G \), then \( \sigma \) is said to be interned in \( G \).

**Lemma 7** For any congruence grammar \( G \) and nonterminal \( N \) of \( G \), if \( \sigma_1 \in G(N) \), then \( G \vdash_{R_m} = (\sigma_1, \sigma_2) \) iff \( \sigma_2 \in G(N) \).

Thus, it’s easy to test whether \( G \vdash_{R_m} = (\sigma_1, \sigma_2) \) if either \( \sigma_1 \) or \( \sigma_2 \) is interned in \( G \). Otherwise, you can test for equality recursively; \( \sigma_1 \) and \( \sigma_2 \) are equal in \( G \) (when neither is interned in \( G \)) iff they have the same top-level function symbol and number of arguments, and their arguments are pairwise equal in \( G \).

To check whether a term \( \sigma \) is true in a given congruence grammar \( G \), find the equivalence class that it belongs in and check if it’s marked as true. If \( \sigma \) is not interned in \( G \), then it is not true in \( G \).

**Definition 39 (essentially finite)** Call a set of facts \( \Gamma \) “essentially finite” if \( C = (\Gamma) = \Sigma_G \) for some finite \( G \).
Lemma 8  If $\Sigma$ is finite, then $C_=(\Sigma)$ is essentially finite.

Lemma 9  If $\Sigma$ is essentially finite, then there is some finite $\Sigma'$ such that $\Sigma = C_=(\Sigma')$.

Lemma 10  A congruence grammar $G$ represents an essentially finite set of facts iff there are only a finite number of productions with TRUE on the left hand side and there are only a finite number of nonterminals which appear on the left hand side of more than one production.

The Transformation $T$  We want to transform $R$ into $T(R)$ such that $R \cup R_\neq$ is equivalent to $T(R) \cup R_\neq$. To see how this works, it’s best just to consider $R_\neq$ as a “black box” that makes sure that all possible equality reasoning gets done between each rule firing of $R$. In this view of things, OIL has a “current state” which is a set of facts closed under equality reasoning, and it searches through $R$ to find a rule to fire. To allow this process to happen in a terminating rule set, we need to represent the current state finitely. Fortunately, we know how to finitely represent a set of facts closed under equality reasoning; we use a congruence grammar. The idea of the transformation, then, is to transform the rules so that they operate on a congruence grammar representation of the current state. Throughout this section, we will speak of the current state (as a congruence grammar) as $G$.

For instance, suppose that the current state is $C_=(\{P(f(a)),=(f(a),g(b))\})$. This state might be represented by the following congruence grammar, $G_{ex}$.

$$
\begin{align*}
  P(f(a)) &= (f(a),g(b)) \\
  =(p(f(a)), p(f(a))) &= (f(a), f(a)) \\
  =(a, a) &= (b, b)
\end{align*}
$$

(Remember that the transformed ruleset will not include $R_\neq$, so we can’t just take reflexive equality facts for granted.)

Equivalence classes have a distinguished representative, which represents them in facts. In the above example, $f(a)$ is the representative for the equivalence class
containing \( f(a) \) and \( g(b) \), and \( P(f(a)) \) is the representative for the equivalence class containing \( P(f(a)) \) and \( P(g(b)) \).

The rule set \( R_\sim \) is presented here. It maintains the congruence grammar data structure. Nonterminals of the grammar are represented as terms \( \sigma \) of which \( =_\sigma(\sigma, \sigma) \) are true. When two nonterminals (equivalence classes) are merged (that is, when a new equality fact is discovered), then one of the terms is marked as \emph{dead}, and all facts about this term are transferred to the other term. A \( \sim \) arc is introduced from the \emph{dead} term to the new term. Remember that the transformed rule set \( T(R) \) ignores all \emph{dead} terms.

The rule set \( R_\sim \) also manages the predicates \( \equiv \) and \emph{interned}. We deduce \( \equiv \) only of syntactically identical terms; we deduce \emph{interned} only of the nonterminals of the grammar. The predicates \emph{dead}, \( \equiv \), \( \sim \), and \emph{interned} are internal predicates, which means that they cannot occur in user-written rules. Remember also that predicates are not in the range of function variables.

\begin{align*}
\textit{syntactic-identity} & \quad \sim(x, y) \\
& \quad \equiv(x, x) \quad (-5) \\
\textit{death} & \quad \sim(x, y) \\
& \quad \equiv(x, y) \\
& \quad \text{dead}(x) \quad (-4) \\
\textit{subst-production-n} & \quad \sim(x, y) \\
& \quad \text{dead}(y) \\
& \quad =_\sigma(x, z^f(x_1, \ldots, x_n)) \\
& \quad \sim(x_1, y_1) \\
& \quad \text{dead}(y_1) \\
& \quad \vdots \\
& \quad \sim(x_n, y_n) \\
& \quad \text{dead}(y_n) \\
& \quad =_\sigma(y, z^f(y_1, \ldots, y_n)) \quad (-3)
\end{align*}
**propagate-equality-n**

\[ \equiv (x, z^l(z_1, \ldots, z_n)) \]

\[ \equiv (y, z^l(z_1, \ldots, z_n)) \]

\[ \neg \text{dead}(x) \]

\[ \neg \text{dead}(y) \]

\[ \neg \text{dead}(z_1) \]

\[ \vdots \]

\[ \neg \text{dead}(z_n) \]

\[ \leadsto (x, y) \] (-2)

**recursively-intern-n**

\[ \equiv (x, x^l(x_1, \ldots, x_n)) \]

\[ \neg \text{dead}(x) \]

\[ \neg \text{dead}(x_1) \]

\[ \vdots \]

\[ \neg \text{dead}(x_n) \]

\[ \equiv (x_i, x_i) \] (-2)

**check-interned-n**

\[ \equiv (x, x^l(x_1, \ldots, x_n)) \]

\[ \neg \text{dead}(x) \]

\[ \neg \text{dead}(x_1) \]

\[ \vdots \]

\[ \neg \text{dead}(x_n) \]

\[ \text{interned}(x^l(x_1, \ldots, x_n)) \] (-1)

**intern-fact**

\[ (x, x) \]

\[ \neg \text{dead}(x) \]

\[ \equiv (x, x) \] (-1)

**intern-equal-1**

\[ (x, y) \]

\[ \neg \text{dead}(x) \]

\[ \equiv (x, x) \] (-1)

**intern-equal-2**

\[ (x, y) \]

\[ \neg \text{dead}(y) \]

\[ \equiv (y, y) \] (-1)

The rule set \( R_{\omega} \) is infinite, because it contains an instance of \( \text{subst-production-n}, \)

\( \text{propagate-equality-n}, \) \( \text{recursively-intern-n}, \) and \( \text{check-interned-n} \) for each \( n \). However,

these rules are only needed for arities which actually occur in \( R \), so there is a finite

restriction of \( T(R) \cup R_{\omega} \) which is equivalent to the infinite set.

Several of the rules serve only to convert an initial set of facts into a congruence grammar.

Now I will present the transformation \( T \).

As stated above, there are restrictions on the rule sets \( R \) and fact sets \( \Sigma \) for which

the transformation \( T \) is valid. First, \( R \) and \( \Sigma \) must both be finite. The reason for this

restriction is to allow \( P(x), \neg \equiv (x, y)/Q(x) \) to be replaced by \( P(x)/Q(x) \). The rules

are equivalent assuming that for every \( x \), there is some term which is not equal to \( x \);

which is guaranteed if \( R \) and \( \Sigma \) are finite. Second, for any rule \( r \in R \), and any finite

set of facts \( \Sigma \), \( I(r)(C_=(\Sigma)) \) is finite. For instance, the rule \( \equiv (x, f(y))/P(x, y) \) would

not be allowed. Subsequently it will be assumed that these restrictions are met.

I will present \( T \) as a series of rewrite rules on rules. These rewrite rules are given
in order of priority. Whenever a later rewrite changes a rule, all of the earlier rewrites should be rechecked for applicability.

1. The first step is to remove the nested structure from rule antecedents and conclusions, so that the antecedents are in terms of productions. To see why this is necessary, consider the congruence grammar $G_{ex}$ presented earlier. Suppose that there is an antecedent $P(g(x))$. There is no fact in $G_{ex}$ which matches this antecedent. However, if the antecedent is transformed to the set of antecedents \( \{y, = (y, P(x)), = (z, g(x))\} \) (which is equivalent to the original antecedent under \( R = \)) then there is a match to the example, with \( y = P(f(a)), z = f(a) \), and \( x = b \).

In these rewrite rules, a variable which appears on the right but not on the left must be a new variable; one which does not appear elsewhere in the given rule. We use \( \tau \) to range over terms which are not variables.

\[
\begin{align*}
\Sigma, \tau / C & \Rightarrow \Sigma, x = (x, \tau) / C \\
\Sigma, = (\tau_1, \tau_2) / C & \Rightarrow \Sigma, = (x, \tau_2), = (x, \tau_1) / C \\
\Sigma, \neg \tau / C & \Rightarrow \Sigma, \neg x, = (x, \tau) / C \\
\Sigma, \neg = (\tau_1, \tau_2) / C & \Rightarrow \Sigma, \neg = (x, \tau_2), = (x, \tau_1) / C \\
\Sigma, = (x, F(\sigma_1, \ldots, \sigma_{k-1}, \tau, \sigma_{k+1}, \ldots, \sigma_n)) / C & \Rightarrow \Sigma, = (x, F(\sigma_1, \ldots, \sigma_{k-1}, y, \sigma_{k+1}, \ldots, \sigma_n)), = \Sigma, = (x, F(\sigma_1, \ldots, \sigma_{k-1}, x, \sigma_{k+1}, \ldots, \sigma_n)) / C \\
\Sigma / F(\sigma_1, \ldots, \sigma_{k-1}, \tau, \sigma_{k+1}, \ldots, \sigma_n) & \Rightarrow \Sigma, = (x, \sigma_k) / F(\sigma_1, \ldots, \sigma_{k-1}, x, \sigma_{k+1}, \ldots, \sigma) \\
\Sigma / = (\sigma_1, \sigma_2) & \Rightarrow \Sigma, = (x, \sigma_1) / = (x, \sigma_2) \\
\Sigma / = (\sigma_1, \sigma_2) & \Rightarrow \Sigma, = (x, \sigma_2) / = (\sigma_1, x)
\end{align*}
\]

2. Since we’re operating on a congruence grammar, if two antecedents \( = (x, F(z_1, \ldots, z_n)) \) and \( = (y, F(z_1, \ldots, z_n)) \) match, \( x \) and \( y \) must be the same term.
\[ \Sigma, = (x, F(z_1, \ldots, z_n)), = (y, F(z_1, \ldots, z_n))/C \implies \Sigma, = (x, F(z_1, \ldots, z_n)), = (x, y)/C \]

3. If there is an antecedent of the form \( = (x, y) \), then we can replace \( x \) by \( y \) throughout the rule and discard the antecedent.

\[ \Sigma, = (x, y)/C \implies \Sigma[y/x]/C[y/x] \]

4. Similarly, \( =_f \) antecedents can be removed.

\[ \Sigma, =_f (x^f, F)/C \implies \Sigma[F/x^f]/C[F/x^f] \]
\[ \Sigma, =_f (F, y^f)/C \implies \Sigma[F/y^f]/C[F/y^f] \]
\[ \Sigma, =_f (F, F)/C \implies \Sigma/C \]

5. If there is an antecedent of the form \( \neg = (x, x) \) or \( \neg =_f (F, F) \), or an antecedent \( =_f (F_1, F_2) \), where \( F_1 \) and \( F_2 \) are distinct function symbols, or a pair of antecedents \( x \) and \( \neg x \), then the rule can never fire. Delete the rule.

After the above rewrites, every antecedent of a rule will be of one of the following forms:

- \( x \)
- \( = (x, F(y_1, \ldots, y_n)) \)
- \( \text{interned}(x) \)
- \( \neg x \)
- \( \neg = (x, y) \)
- \( \neg =_f (F_1, F_2) \)
- \( \neg \text{interned}(\sigma) \)

Antecedents of the form \( = (x, F(y_1, \ldots, y_n)) \) are called production antecedents.

Call the transformation defined by the above rewrite rules \( T_1 \). It should be clear that \( R \cup R_\approx \) is equivalent to \( T_1(R) \cup R_\approx \).

Now we must consider how to transform \( T_1(R) \) so it does not depend on \( R_\approx \). The only remaining problem is that \( T_1(R) \) may depend on reflexive equality facts which are not present in \( G \). That is, for a rule \( r \), there may be equality antecedents in \( T_1(r) \).
which fail because reflexive equality facts are missing from $G$. Unfortunately, while the problem is quite simple, the solution is very complicated. Here’s an example of a rule which might fail.

\[
\begin{align*}
& P(x) \\
& Q(y) \\
& = (z, f(x)) \\
& = (z, f(y)) \\
& R(x, y)
\end{align*}
\]

If $\Sigma$ includes \{\{P(a), Q(a)\}\}, this rule should fire using the reflexive equality fact $= (f(a), f(a))$. However, this fact might not be interned in $G$, so the rule must be modified so it will work regardless of whether the reflexive equality fact is interned.

The rule \text{rem-eq-1} will be used to illustrate the remaining parts of the transformation. The rule \text{rem-eq-1} is the result of applying $T_1$ to \text{rem-eq-1-orig}.

\[
\begin{align*}
\text{rem-eq-1-orig} & \quad \text{rem-eq-1} \\
& \quad \text{rem-eq-1-orig} & \quad \text{rem-eq-1} \\
& f(x) & x \\
& = (z, g(g(z))) & = (x, f(y)) \\
& = (f(g(z)), f(z)) & = (z, g(w)) \\
& = (i(z), i(h(g(z)))) & = (w, g(z)) \\
& i(z) & = (v, h(w)) \\
& & = (u, i(v)) \\
& & = (t, i(z)) \\
& & = (s, f(w)) \\
& & = (s, f(z)) \\
& & = (t, u) \\
& & t
\end{align*}
\]

First, a bit of terminology. Let $V$ be the set of term variables used in the antecedents of $T_1(r)$. For variables $x, y \in V$, we say that $x$ dominates $y$ if there is an antecedent $= (x, F(\ldots, y, \ldots))$. In \text{rem-eq-1}, $V = \{x, y, z, w, t, u, v, s\}$ and the domination relation is as follows.
x dominates y
z dominates w
w dominates z
v dominates w
u dominates v
t dominates z
s dominates w, z

We wish to mark each variable as either safe or unsafe. Initially, the variables are unmarked. The idea behind these markings is that if x is a safe variable, and $T_1(r)$ would fire on $C_x^=(G)$, then x must be interned in G.

First, mark a variable x as safe if it is involved in a domination cycle, if there is an antecedent consisting solely of x, if there is an antecedent interned(x), or if x is dominated by a safe variable.

At this point, x, y, z, and w are marked as safe.

Now, while there are any unmarked variables in r, let x be some unmarked variable which is not dominated by another unmarked variable. (There must be such a variable, because there are no domination cycles among unmarked variables.) If x is the left hand side of at most one production antecedent, then mark x as unsafe and start over. Otherwise, let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be the right hand sides of the production antecedents with x as a left hand side. If these terms cannot be unified, then mark x as safe (and also mark the variables which are transitively dominated by x as safe) and start over. (The rule set $R_\omega$ will add interned facts as appropriate.) Otherwise, this rule must be split into two. In one of these rules, add an antecedent interned($\sigma_1$) and mark x (and its transitively dominated variables) as safe. For the other rule, let $\rho$ be a most general unifier of $\sigma_1, \sigma_2, \ldots, term_k$. Apply this substitution to r and add the result to the rule set. (Remember to apply any applicable rewrites from earlier in the transformation.)

We can immediately mark u and t, and then v, as unsafe, but s is more complicated, since it is the left hand side of two different production antecedents. The right hand sides are $f(w)$ and $f(z)$. Since these terms can be unified (with a most general unifier mapping w to z, for instance), we must split the rule into two. Here are the two new rules.
In rem-eq-2a, s is marked as safe. In rem-eq-2b, s will be marked as unsafe. Subsequently, we’ll follow rem-eq-2b. Note that rewrites from $T_1$ were used to remove duplicate antecedents in rem-eq-2b.

Now, every variable in every rule is marked either safe or unsafe, and every unsafe variable occurs on the left hand side of at most one production antecedent. We’re going to divide the unsafe variables into based and unbased, according to this recursive definition: an unsafe variable is based if it is the left hand side of a production antecedent such that every variable on the right hand side is either safe or based; it is unbased if it is not the left hand side of a production antecedent at all, or if it is the left hand side of a production antecedent with at least one unbased variable on the right hand side. The idea is that if we assume that all the safe variables are bound, the based variables have unique bindings, but the unbased variables have infinitely many bindings which satisfy all the positive antecedents.

In rem-eq-2b, all of the unsafe variables ($v$, $u$, $t$, and $s$) are based.

Now, there cannot be any unbased variables in the conclusion of any rule. If there were, then on some finite set of facts $\Sigma$, that rule could fire an infinite number of times, which we have assumed cannot happen. Our restrictions also ensure that if there is an unbased variable in a negated antecedent, then that antecedent has no effect; so we can remove any negated antecedent which mentions an unbased variable. Now unbased variables can occur only in production antecedents, and every production antecedent which has an unbased variable on the right hand side also has an unbased variable on
the left hand side. Since no unbased variable is used anywhere besides a production antecedent, we can remove all antecedents which contain unbased variables.

Now we are left with a rule where every variable is either safe or based, and every based variable is the left hand side of exactly one production antecedent. For convenience, if \( x \) is some based variable, we will say “the right hand side of \( x \)” to mean the right hand side of the production antecedent that has \( x \) as its left hand side. For instance, in \( \text{rem-eq-2b} \), the right hand side of \( s \) is \( f(z) \).

If a based variable appears in only one antecedent (which must be as the left hand side of a production antecedent) then the variable, and that antecedent, is useless. Remove the antecedent.

In \( \text{rem-eq-2b} \), we can remove the antecedent \( = (s, f(z)) \), to get \( \text{rem-eq-3} \).

\[
\begin{align*}
\text{rem-eq-3} \\
x = (x, f(y)) \\
= (z, g(z)) \\
= (v, h(z)) \\
= (u, i(v)) \\
= (t, i(z)) \\
\vdash = (t, u) \\
t
\end{align*}
\]

Remember that we’re in the middle of solving the problem of rules that fail to fire on \( G \), even though they would fire on \( C_\cong(G) \), because of terms which are not interned in \( G \). We’re making progress on this goal; only the based variables remain to be taken care of. Furthermore, every based variable appears in a negated antecedent or the conclusion, or is transitively dominated by a variable which does so; otherwise it would have been removed.

Now, let \( S \) be the set of based variables. We’re going to create a new rule for every subset \( Q \) of \( S \) which is closed under downward domination. For every such set \( Q \), we will create a new rule as follows.

1. For every variable \( x \in Q \), where the right hand side of \( x \) is \( \sigma \), add the antecedent \( \text{interned}(x) \), and mark \( x \) as safe.
2. For every variable $x \in S \setminus Q$, where the right hand side of $x$ is $F(x_1, \ldots, x_k)$, mark $x$ as uninterned. If all of $x_1, \ldots, x_k$ are in $Q$ or are safe, then add an antecedent $\neg \text{interned}(F(x_1, \ldots, x_k))$. Remove the antecedent which has $x$ as its left hand side, but remember what it was; we will continue to refer to $F(x_1, \ldots, x_k)$ as the right hand side of $x$.

Now every variable is safe or is known to be uninterned. (Of course, the variables which are known to be uninterned no longer appear as left hand sides of production antecedents in the rule; such antecedents could never fire.)

For rem-eq-3, $S$ is $\{t, u, v\}$, and the downwardly closed subsets of $S$ are $\{t, u, v\}$, $\{t, v\}$, $\{u, v\}$, $\{v\}$, $\{t\}$, and $\emptyset$.

Here are two of the six new rules, rem-eq-4a (which corresponds to $\{u, v\}$), and rem-eq-4b (which corresponds to $\emptyset$).

\[
\begin{align*}
\text{rem-eq-4a} \\
{x} &= (x, f(y)) \\
&= (z, g(z)) \\
&= (v, h(z)) \\
&= (u, i(v)) \\
&\quad \text{interned}(v) \\
&\quad \text{interned}(u) \\
&\quad \neg \text{interned}(i(z)) \\
&\quad \neg = (t, u) \\
&\quad t
\end{align*}
\]

In rem-eq-4a, $u$ and $v$ are safe and $t$ is uninterned. In rem-eq-4b, $t$, $u$, and $v$ are uninterned.

Now, every negated antecedent involves only uninterned variables and safe variables. While there are unprocessed negated antecedents, pick one and transform it as follows.

1. If the antecedent mentions no uninterned variables, then mark the antecedent as processed and start over.

2. If the antecedent is of the form $\neg x$, where $x$ is uninterned, remove the antecedent.
3. If the antecedent is of the form \( \neg=(x,y) \), where one of \( x \) and \( y \) is uninterned and the other is safe, remove the antecedent.

4. Otherwise, the antecedent is of the form \( \neg=(x,y) \), where both \( x \) and \( y \) are uninterned. Let the right hand side of \( x \) be \( F_1(x_1,\ldots,x_j) \), and let the right hand side of \( y \) be \( F_2(y_1,\ldots,y_k) \). If \( F_1 \) and \( F_2 \) are different function symbols, or \( j \neq k \), then simply remove the antecedent. Otherwise, create \( j+1 \) new rules. In each of them, remove the antecedent under consideration, and add a new antecedent; in \( j \) of the rules, the antecedent should be \( \neg=(x_i,y_i) \), for \( 1 \leq i \leq j \), and in the other rule, the antecedent should be \( \neg=(F_1,F_2) \). (If \( F_1 \) and \( F_2 \) are the same function symbol, then this antecedent can never be satisfied; the rule may then be omitted.)

Now, all negated antecedents mention only safe variables.

Here are \textit{rem-eq-5a}, \textit{rem-eq-5b1}, and \textit{rem-eq-5b2}. The rule \textit{rem-eq-5a} is the result of applying the above transformation step to \textit{rem-eq-4a}. Similarly, \textit{rem-eq-5b2} is the result of applying the above transformation step to \textit{rem-eq-5b}; \textit{rem-eq-5b1} is an intermediate step in this process.
The only modification in \textit{rem-eq-5a} is to remove the antecedent \( \neg = (t, u) \), because \( u \) is safe and \( t \) is uninterned in \textit{rem-eq-4a}. In \textit{rem-eq-5b1}, the antecedent \( \neg = (t, u) \) was replaced by the antecedent \( \neg = (v, z) \). We could have added another rule where the antecedent \( \neg = (t, u) \) was replaced by the antecedent \( \neg = f(i, i) \), but this rule could never fire, so it was omitted. In \textit{rem-eq-5b2}, the antecedent \( \neg = (v, z) \) was removed because \( v \) is uninterned and \( z \) is safe in \textit{rem-eq-5b1}.

Now we’ve massaged the antecedents so that the rule will fire exactly when it ought to; the only remaining problem is that the conclusion may contain uninterned variables.

Let \( S \) be the set of uninterned variables which appear in the conclusion or which are transitively dominated by such a variable. If \( S \) is nonempty, then let \( x \) be some such variable which dominates no other uninterned variable. Let \( \sigma \) be the right hand side of \( x \). Then replace the conclusion of the current rule by \( = (\sigma, \sigma) \). This will have the effect of interning \( \sigma \), at which point one of this rule’s sibling rules will be able to fire.

The rule \textit{rem-eq-5a} gets transformed into \textit{rem-eq-6}.
We’ve finally finished off the problem of uninterned terms. Now it’s time to do a few final cleanups on the rules.

For every variable which occurs in a rule, add an antecedent \( \lnot \text{dead}(x) \).

Replace conclusions of the form \( = (x, y) \) with \( \lnot = (x, y) \).

After these two transformations, \textit{rem-eq-6} becomes \textit{rem-eq-7}.

\textit{rem-eq-7}

\[
\begin{align*}
  & x \\
  & = (x, f(y)) \\
  & = (z, g(z)) \\
  & = (v, h(z)) \\
  & = (u, i(v)) \\
  & \text{interned}(v) \\
  & \text{interned}(u) \\
  & \lnot \text{interned}(i(z)) \\
  \hline
  & = (i(z), i(z))
\end{align*}
\]

Remember that \textit{rem-eq-7} is just one of several rules which are the collective result of transforming \textit{rem-eq-1}.

That’s it! The transformed rule set is now ready to be merged with \( R_\ldots \).
3.2.4 Bounds

The only part of the full OIL language we haven’t covered yet is bounds. A set of facts \( \Sigma \) can contain facts of the form \( \leq (\sigma, n) \), where \( n \) is a nonnegative integer. (Note that such a fact is not a term, so an antecedent consisting of a single term variable \( x \) cannot be mapped to such a fact.)

Negated bounds antecedents and bounds conclusions have a special feature; they may be of the form \( \leq (\sigma, B) \), where \( B \) is an expression made of nonnegative integers and bounds variables, with addition and multiplication. In this form, the value of the expression is computed before the conclusion is asserted or the negated antecedent is checked.

OIL may be implemented using finite-precision integers for bounds. In this case, if the conclusion of a rule asserts a bound expression with a value which is too large to represent, the implementation simply does not fire that rule. (The implementation is not allowed to silently truncate a bound to its low-order bits.)

We define the semantics of OIL with bounds by specifying a set of extended rules which are to be incorporated into every OIL program.

**Definition 40 (bounds rules)** The following set of rules will be called \( R_\leq \).

\[
\begin{align*}
equal_{\leq} & \quad (x, y) \leq (x, b) \\
\leq (y, b) & \quad (\neg 1)
\end{align*}
\]

\[
\begin{align*}
\leq_{\text{closed-up}} & \quad \leq (x, n) \\
\leq (x, + (n, 1)) & \quad (\neg 1)
\end{align*}
\]

3.2.5 The Full OIL Language

We’ve now covered the semantics of the full OIL language; here’s a short summary.

**Definition 41 (OIL implication)** We have \( \Sigma \vdash^\text{OIL}_R \Phi \iff \Sigma \vdash_{R \cup R_\leq} \Phi \).

**Definition 42 (conforming OIL implementation)** Given \( R, \Sigma, \) and \( \Phi \), an implementation is required to terminate if \( R \cup R_\neg \cup R_\leq \) user terminates on \( \Sigma \) (equivalently, if \( C'_{R \cup R_{\neg} \cup R_{\leq}}(\Sigma) \) is essentially finite). If the implementation terminates, it
must say yes if $\Sigma \vdash_R \Phi$, no if $\Sigma \not\vdash_R \Phi$, and either yes, no, or ambiguous otherwise.

As mentioned above, there is one time a conforming implementation is allowed to give an incorrect answer. The implementation may implement bounds with finite-precision integers. If so, when a rule firing should create a bounds fact with a bound which is out of the range of the integers used, that rule firing may be ignored.

4 The Power of OIL

I'll just state a few results without proof.

Lemma 11 Any language in the arithmetical hierarchy can be decided by an OIL program. That is, suppose $L$ is in the arithmetical hierarchy; then there is an OIL program $R_L$ such that $\emptyset \vdash_{R_L}^{\text{OIL}} \text{yes}(x)$ iff $x \in L$ and $\emptyset \vdash_{R_L}^{\text{OIL}} \text{no}(x)$ iff $x \notin L$.

Lemma 12 Any recursive language can be decided by a terminating OIL program.

Definition 43 (syntactically local) A rule is syntactically local if every proper subexpression of the conclusion occurs in an antecedent.

Lemma 13 The closure of $\Sigma$ under any syntactically local rule set can be computed in time polynomial in the size of $\Sigma$, where $\Sigma$ is represented as a congruence grammar.

Lemma 14 Any language of terms which can be computed in time polynomial in the size of the term (when represented as a DAG) can be represented as a syntactically local rule set (see [GM92]).

Thus, the syntactically local rule sets form a syntactic characterization of the set $P$.

5 Complexity Results and Optimizations

The results in this section assume that hash table operations take constant time.

There are many ways one could implement an OIL program; some of them may be much more efficient than others. Consider the following rule.
Suppose that $\Sigma$ contains about a thousand facts each about parenthood and siblinghood. One way to find all the cousins (all the possible firings of $\text{cousin}$) is to go through the antecedents in order, and consider each triple of two parenthood facts and one siblinghood fact. If the facts match up according to the rule, assert a cousinhood fact. This approach takes about a billion operations.

A much better method of dealing with this rule is to first go through all the parenthood facts to find possibilities for $x$ and $y$. For each parenthood fact, go through the siblinghood facts, checking for ones which match $x$, to find possibilities for $z$. Then go through the parenthood facts again, checking for ones which match $z$, to find possibilities for $w$. This approach takes only a few million operations. Notice that we had to reorder the antecedents to achieve this savings.

Since each person almost certainly has a fairly small number of parents, children, and siblings, an even better approach is to keep a list associated with each person of the parents, children, and siblings of that person. With such a data structure, the set of all cousins can be found with only a few thousand operations.

### 5.1 Complexity for OIL Without Equality or Negated Antecedents

The results about binary rule sets and the binary rule transform for OIL without equality and without negated antecedents are from [McA93]. Throughout the rest of this section $R$ and $\Sigma$ do not use equality or negated antecedents.

#### 5.1.1 Binary Rule Sets

**Definition 44 (binary rule set)** A binary rule set is one where every rule has at most two antecedents.
Lemma 15 If $R$ is a binary rule set, then $C_l_R(\Sigma)$ can be computed in time proportional to the number of rule firings of $R$ on $C_l_R(\Sigma)$ plus the size of $\Sigma$.

To see the significance of this, consider splitting the work done in computing $C_l_R(\Sigma)$ into two parts: the match work and the update work. The match work is the work done to find possible rule firings. The update work is the work done to fire the rule and update the database. The update work, in our model, takes constant time for each rule firing. According to the above result, the match work can also be done in constant time per rule firing plus time proportional to the size of $\Sigma$. Without some kind of source optimization of the rule set, to reduce the number of rule firings to compute the same result (i.e., by eliminating duplicate derivations of some facts), this is as good as you’re going to get, since you’ll always have to at least look at each fact in $\Sigma$.

The above result can be used to derive other complexity results, such as the following.

5.1.2 The Binary Rule Transform

Definition 45 (prefix firing) A prefix firing of a rule $r = A_1, \ldots, A_n/C$ on a set of facts $\Sigma$ is a pair $(k, \rho)$, where $1 \leq k \leq n$, and the antecedents $A_1$ through $A_k$ are in $\Sigma$ under the variable substitution $\rho$, and where the domain of $\rho$ is the set of variables mentioned in antecedents $1$ through $k$.

Definition 46 (binary rule transform) Define $B(R)$ (the “binary rule transform [McA93]”) as the following transformation. Let $r$ be a rule with more than two antecedents, $r = A_1, A_2, \ldots, A_n/C$. Replace $r$ with two extended rules, $A_1, A_2/\mathcal{P}(x_1, x_2, \ldots, x_k)$ and $\mathcal{P}(x_1, x_2, \ldots, x_k), A_3, \ldots, A_n/C$, where $\mathcal{P}$ is a new predicate and $x_1, x_2, \ldots, x_k$ are the variables which appear in $A_1$ or $A_2$ and which are also mentioned in the remainder of $r$. Repeat this process until no rule has more than two antecedents. (This results in an extended OIL program.)

Lemma 16 The above transformation doesn’t change the semantics of the rule set; $\Sigma \vdash^OIL_R \Phi$ iff $\Sigma \vdash^OIL_{B(R)} \Phi$, as long as neither $\Sigma$ nor $\Phi$ include extended facts.
Since $B(R)$ is a binary rule set, we can compute $Cl_{B(R)}(\Sigma)$ in time proportional to the number of rule firings of $B(R)$ in $Cl_{B(R)}(\Sigma)$ plus the size of $\Sigma$. But what is this number? It's merely the number of prefix firings of $R$ in $Cl_R(\Sigma)$ plus the size of $\Sigma$.

**Lemma 17** $Cl_R(\Sigma)$ can be computed in time proportional to the number of prefix firings of $R$ on $Cl_R(\Sigma)$ plus the size of $\Sigma$.

If the binary rule transform is used in an implementation, the ordering of the antecedents is critical. Consider two variants of the example used above.

\[
\begin{array}{cccc}
\text{cousin-1} & \text{cousin-2} \\
\text{parent}(x, y) & \text{parent}(x, y) \\
\text{parent}(z, w) & \text{sibling}(x, z) \\
\text{sibling}(x, z) & \text{parent}(z, w) \\
\text{cousin}(y, w) & \text{cousin}(y, w)
\end{array}
\]

With the above-mentioned database, the number of prefix firings of cousin-1 is a few million, but the number of prefix firings of cousin-2 is a few thousand. An implementation might simply use the antecedent order the user used (in which case the user had better be aware of these issues), it might use various heuristics to reorder the antecedents (such as trying to make the second antecedent share a variable with the first antecedent, like cousin-2 but not cousin-1), or it might try various antecedent orderings on real data, and pick the one which works best.

### 5.1.3 Acyclic Rules

As explained above, it is in some sense "optimal" to compute $Cl_R(\Sigma)$ in time proportional to the number of rule firings of $R$ on $Cl_R(\Sigma)$ plus the size of $\Sigma$, and this is achieved in the case of binary rule sets. Also, arbitrary rule sets can be computed in time proportional to the number of prefix rule firings of $R$ on $Cl_R(\Sigma)$ plus the size of $\Sigma$, so one way to optimize rules is to transform them so that the number of prefix firings is proportional to the number of firings.

Consider the following rule set.
In this rule set, the rule opt-cousin has only as many prefix firings as it has firings, because the extra information in the new facts ensures that every prefix firing can be completed to a full firing. The extra rules in the rule set are unary or binary, and are limited in their number of firings to the size of $E$, so the entire rule set can run in time proportional to the size of $E$ plus the number of pairs of cousins.

We will now consider under what conditions a transformation like the one above can be used.

Definition 47 (antecedent graph) Consider a rule $r$. Create a graph which has one node for each antecedent of $r$ and one node for each variable used in an antecedent of $r$. There is an edge between an antecedent node and a variable node iff the variable is mentioned in the antecedent. This is the antecedent graph of $r$.

Definition 48 (acyclic rule) If the antecedent graph of $r$ is acyclic, then $r$ is called an acyclic rule.

An acyclic rule can be transformed like opt-cousin, as follows.
Let $x$ be some variable mentioned in $r$. Consider the antecedent graph of $r$ to be a tree rooted at $x$. Eliminate leaf nodes of this tree which are variable nodes. Create a new predicate for each remaining node of the tree that has children; the predicate for variable nodes should have arity 1, and the predicate for antecedent nodes should have arity of the number of variables mentioned in the antecedent. Each node in this tree will have an associated new antecedent. For variable nodes, the new antecedent will be the new predicate for the variable, applied to the variable. For antecedent nodes with children, the new antecedent will be the new predicate, applied to the variables mentioned by the antecedent. For antecedent nodes without children, the new antecedent will be the same as the original antecedent. Consider some preorder traversal of this tree (that is, consider some ordering for the nodes of the tree such that every parent node precedes all its children). The transformed rule will consist of the new antecedent for each node in the tree, in order of this traversal.

We also need to create helper rules, one for each node that has an associated new predicate. The conclusion of the rule will be the associated antecedent; the antecedents of the rule will be the associated antecedents of the children of the node.

With this transformation, the closure of any rule set $R$ consisting of acyclic rules can be computed on $\Sigma$ in time proportional to the number of rule firings of $R$ on $C_lR(\Sigma)$.

There are related techniques for handling rules which are not acyclic. One possibility is to break the rule into cyclic and acyclic rules, use the above transformation on the acyclic rules, and leave the cyclic rules alone. Another possibility is to transform the rules in ways which break the cycles.

5.2 Complexity and Optimizations with Equality

The binary rule transform and the acyclic rule transform are still correct (meaning-preserving) in the presence of equality. However, the complexity results are not; in the presence of equality, there are rule sets $R$ such that any straightforward computation of $C_l'R(\Sigma)$ will take infinitely long, even though $C_R'(\Sigma)$ is essentially finite. The problem is repeated inference...duplicate inference which is performed on terms $\sigma_1$ and $\sigma_2$, ...
when $\sigma_1$ and $\sigma_2$ are later deduced to be equal.

To reduce the problem of repeated inference as much as possible, within a priority level, an implementation could fire rules with equality conclusions before rules with other types of conclusions. While this no longer strictly adheres to the definition of “fairness” given above, it preserves termination.

5.3 Complexity and Optimizations with Negated Antecedents

The binary rule transform and the acyclic rule transform are not correct in the presence of negated antecedents; they must be modified to treat negated antecedents specially (and less efficiently). The new internal predicates collect information about sets of antecedents; with negated antecedents, this information can be made incorrect when new facts are deduced.

If $r$ is a rule, possibly with negated antecedents, let $\text{pos}(r)$ be the rule with all negated antecedents removed (the “positive part” of $r$), and let $\text{neg}(r)$ be the negated antecedents of $r$. The binary rule transform or the acyclic rule transform can be performed on $\text{pos}(r)$, and then $\text{neg}(r)$ can be added to the antecedent list of the transformed rule which asserts the conclusion of $r$. The result is a correctly transformed rule.

Consider the acyclic rule transform. Without negated antecedents, you can say that the transformed $r$ can be run on $\Sigma$ in time proportional to the number of firings of $r$ on $\text{C}l_r(\Sigma)$. With negated antecedents, you get the slightly weaker result that the transformed $r$ can be run on $\Sigma$ in time proportional to the number of firings of $\text{pos}(r)$ on $\text{C}l_r(\Sigma)$. The extent to which this matters depends on the fraction of firings of $\text{pos}(r)$ which are not legal firings of $r$.

6 Examples

Here are some examples of fairly large OIL programs, as well as some tips and tricks for using OIL.
6.1 Binary Arithmetic

Here is an example showing one way to do computation in OIL. This method is rather inefficient; in a real application, it would probably be better to either extend your implementation of OIL (some implementations of OIL can be extended in C or Lisp) or to try to use the arithmetic built into bounds variables. On the other hand, it's far more efficient than the unary arithmetic used in Section 2.3.

This method represents numbers by terms encoding the binary representation. To encode the number 6, turn it into binary as 110, reverse the digits as 011, and add various punctuation to get \( d0(d1(d1(z))) \). Here, \( z \) can be read as 0, \( d0 \) can be read as “two times,” and \( d1 \) can be read as “one plus two times,” to get “two times (one plus two times (one plus two times zero)),” which is indeed 6. We will also use the unary function \( p1 \) (“one plus”), and the binary functions + and *.

The first try might include a rule like this.

\[
p1\text{-broken}\]

\[
\frac{}{(p1(d0(x)), d1(x))}
\]

However, this can fire on any term \( x \), and does not terminate. We need to restrict the rules somehow, to tell them what we want to compute. We will do this by asserting \( \text{compute}(x) \) to say that we want to compute the value of the expression \( x \) and assert that \( x \) is equal to that value, where \( x \) is either \( p1(a) \), \( +(a, b) \), or \( *(a, b) \), and \( a \) and \( b \) are terms built from \( z \), \( d0 \), and \( d1 \).
\[ p1-z \]
compute\( p1(z) \)
\[ = (p1(z), d1(z)) \]

\[ p1-dl-a \]
compute\( p1(dl(x)) \)
\[ = (p1(dl(x)), d0(p1(x))) \]

\[ plus-z-x \]
compute\( +(z, x) \)
\[ = (+ (z, x), x) \]

\[ plus-dl-dO-a \]
compute\( +(dl(x), dO(y)) \)
\[ = (+ (dl(x), dO(y)), dO(+ (x, y))) \]

\[ plus-dl-dl-a \]
compute\( +(dl(x), dl(y)) \)
\[ = (+ (dl(x), dl(y)), dO((+ (x, y))) \]

\[ plus-dl-dO-b \]
compute\( +(dl(x), dO(y)) \)
\[ = (+ (dl(x), dO(y)), dO(+(x, y))) \]

\[ times-z-x \]
compute\( *(z, x) \)
\[ = (* (z, x), z) \]

\[ times-dl-y-a \]
compute\( *(d1(x), y) \)
\[ = (* (d1(x), y), d0(* (x, y))) \]

\[ times-dl-y-b \]
compute\( *(d1(x), y) \)
\[ = (* (d1(x), y), +(y, d0(* (x, y))) \]

\[ times-dl-y-c \]
compute\( *(d1(x), y) \)
\[ = (* (d1(x), y), d0(* (x, y))) \]

\[ p1-dO \]
compute\( p1(d0(x)) \)
\[ = (p1(d0(x)), d1(d0(x))) \]

\[ p1-dl-b \]
compute\( p1(dl(x)) \)
\[ = (p1(dl(x)), d0(p1(x))) \]

\[ plus-x-z \]
compute\( +(x, z) \)
\[ = (+ (x, z), x) \]

\[ plus-dO-dO-a \]
compute\( +(d0(x), d0(y)) \)
\[ = (+ (d0(x), d0(y)), d0(+ (x, y))) \]

\[ plus-dO-dO-b \]
compute\( +(d0(x), d0(y)) \)
\[ = (+ (d0(x), d0(y)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]

\[ plus-dO-dl-b \]
compute\( +(d0(x), d (x)) \)
\[ = (+ (d0(x), d (x)), dO(+(x, y))) \]
One problem with the above solution is that working a few arithmetic problems will clutter the database with equality facts about expressions built of natural numbers. While real implementations of OIL will handle equality fairly efficiently, it may be more efficient to use a different “calling convention” for the arithmetic module. Here’s another possibility: to compute the value of an expression \( x \), we assert \( \text{compute}(x) \). When the arithmetic module has computed the value, it will assert \( \text{value}(x, y) \), where \( y \) is the value. I’ll just show how to do one of the cases of + with this convention.

\[
\text{compute}(+(x, y)) \quad \text{value}(+(x, y), z)
\]

In this example, the use of \( \text{compute} \) as a sort of function call is apparent.

### 6.2 de Bruijn Numbers

The original (and, so far, only) use of OIL is to implement the inference engine for a proof verification system, Ontic. In this system, object-language quantified terms are implemented using de Bruijn notation. In this notation, \( \forall x. P(x) \lor \exists y. Q(x, y) \) might be represented internally as

\[
\forall \text{dbn}(\text{dbz}), \exists \text{dbn}(\text{dbs}(\text{dbz})), \text{dbn}(\text{dbs}(\text{dbz})))
\]

(This is not the real internal or external syntax of Ontic.) Here, \( \text{dbc} \) is a marker which introduces a binding level, \( \text{dbn} \) means “de Bruijn number,” and \( \text{dbs} \) (“de Bruijn successor”) and \( \text{dbz} \) (“de-Bruijn zero”) encode a de Bruijn number in unary notation.

Here’s an example of a program that works on formulas in the above notation. If you assert \( \text{compute}(\text{instantiate}(\sigma_1, \sigma_2)) \), where \( \sigma_1 \) is a closed \( \forall \) or \( \exists \) formula and \( \sigma_2 \) is a ground term, it will compute the instantiation.
Here I'm assuming that the representation of the object language doesn't use functions of arity greater than two. It's easy to extend it to allow any bounded arity, but it's unfortunately impossible to allow arbitrary arities.

It takes a little more work, but not much, to do the right thing when the terms contain free de Bruijn references.
6.3 Efficient Object Language

When using OIL to write an inference system, the object language must be distinguished from the metalanguage. For instance, the fact \( compute(+z,z) \) is a statement in the metalanguage about the object language term \(+z,z\). Facts in the object language, such as \( prime(d1(d0(d1(z)))) \), could be represented in the metalanguage something like \( true(prime(d1(d0(d1(z)))) \). However, it’s more efficient to omit the \( true \) and simply use \( prime(d1(d0(d1(z)))) \) as the metalanguage representation.

Consider a rule about implication in such a system.

\[
\text{implies-1} \\quad \text{implies}(x, y) \\
\quad x \quad y
\]

This is the reason that lone variables are allowed as antecedents and conclusions in a rule.

7 External Syntax

The Ontic Rule Compiler (ORC) is an implementation of OIL in Common Lisp. It uses a syntax for rules and terms which is based on Lisp s-expressions. Here’s what the grammar looks like for this external syntax.

\[
F ::= \text{?f} \mid f \\
\sigma ::= \text{?x} \mid (F \sigma_1 \sigma_2 \ldots \sigma_n) \\
b ::= n \mid \text{?b} \\
\Phi ::= \sigma \mid (= \sigma_1 \sigma_2) \mid (\leq \sigma \ b)
\]

Any symbol which begins with a \( ? \) is a variable; term, function, and bounds variables are distinguished by context.

Here are the examples of terms and formulas from Section 3.1, redone in the ORC syntax.
(+ (succ (succ (zero))) (succ (zero)))
(arc (node1) (node7))
(forall (x) (exists (y) (greater (y) (x))))
(is (a-brother-of (a-sister-of ?x)) (a-brother-of ?x))
(and (subset ?x ?y) (subset ?y ?x))
(= (the-mother-of (fred)) (the-mother-of (harry)))
(greater (big-omega (exp (n))) (big-oh (* (n) (n))))
(<= (arc ?x ?y) ?xylen)
(= ?x (* (one) ?x))

Here is the external grammar for rules.

\[
\begin{align*}
B & := b \mid (+ B_1 B_2) \mid (*) B_1 B_2 \\
\check{\phi} & := \phi \mid (\leq \sigma B) \\
A & := \phi \mid (\text{not-proved} \ \check{\phi}) \\
\Lambda & := (\text{rule} n \ \text{rule-name} \ (A_1 A_2 \ldots A_k) \ \check{\phi})
\end{align*}
\]

The rule-name can be an arbitrary Lisp s-expression.

8 Rulesets and Orcfuns

ORC provides a couple of macros for easily specifying certain types of rules or sets of rules. The first of these is ruleset, which is used to specify sets of rules which share some antecedents. For instance, here is a ruleset from the Ontic source code.

(ruleset 0 variables
  (when ((variable-of-type ?c ?type))
    (is ?c ?type)
    (at-most-one ?c)
    (when ((there-exists ?type))
      (there-exists ?c))))

This expands into three rules:
There are several forms which can occur in the body of a ruleset; these include
when, if, nonmon-if, progn, compute-value, and selectmatch forms, and conclu-
sion forms. Rulesets are transformed into rules by applying the RS transformation
to the body of the ruleset.

\[
RS[\Gamma, (\text{when } (A_1 \ldots A_j) B_1 \ldots B_k)] \Rightarrow \bigcup_{i=1}^{k} RS[\Gamma, [A_1, \ldots, A_j], B_i]
\]

\[
RS[\Gamma, (\text{progn } B_1 \ldots B_k)] \Rightarrow \bigcup_{i=1}^{k} RS[\Gamma, B_i]
\]

\[
RS[\Gamma, (\text{if } A B_T B_F)] \Rightarrow RS[\Gamma, [A], B_T] \cup RS[\Gamma, [(\text{not } A)], B_F]
\]

There's no special meaning in OIL to the term (not A), so if is only useful if
your own rules use not.

\[
RS[\Gamma, (\text{nonmon-if } A B_T B_F)] \Rightarrow \\
RS[\Gamma, [A], B_T] \cup RS[\Gamma, [(\text{not-proved } A)], B_F]
\]

\[
RS[\Gamma, (\text{compute-value } (\ldots ) B)] \Rightarrow RS[\Gamma, B]
\]

\[
RS[\Gamma, (\text{compute-value } ((v c) . R) B)] \Rightarrow \\
RS[\Gamma, (\text{progn } (\text{compute! } c) (\text{when } (\text{has-value } c v) (\text{compute-value } R B)))]
\]

The selectmatch ruleset form does pattern matching on its first argument, which
must be a term variable. The other arguments are patterns and sets of ruleset forms
to be executed if that pattern matches. Here is an example from Ontic.
(ruleset 0 db-numberness
  (when ((interned! ?x))
    (selectmatch ?x
      ((db-one) (is-db-number ?x))
      ((?const) (not (is-db-number ?x)))
      ((db-succ ?y) (is-db-number ?x))
      ((?const ?arg) (not (is-db-number ?x)))
      ((?const ?arg1 ?arg2) (not (is-db-number ?x))))))

Rather than try to give a formal description of the selectmatch transformation, I'll just give an example. The selectmatch form in the above example translates into

(progn
  (when ((= ?x (?c)))
    (when ((= ?x (db-one)))
      (is-db-number ?x))
    (when ((not-proved (fun-equal ?c db-one)))
      (not (is-db-number ?x))))
  (when ((= ?x (?c ?a)))
    (when ((= ?x (db-succ ?y))
      (is-db-number ?x))
    (when ((not-proved (fun-equal ?c db-succ)))
      (not (is-db-number ?x))))
  (when ((= ?x (?c ?al ?a2))
    (not (is-db-number ?x))))

In Ontic, bound variables are represented with de Bruijn numbers. The first de Bruijn number is (db-one), the second is (db-succ (db-one)), and so on. The above rule takes any interned! term and checks its structure to see if it's a de Bruijn number. (Note that in Ontic, no term has more than two arguments.)

Here is the final part of the RS transformation, which is used if none of the above cases applies.

\[ RS[\Gamma, B] \Rightarrow \{(\text{rule } n \text{ rule-name } (\Gamma) \ B)\} \]

Here's an example of the use of rulesets; this is a rewrite of the subst rules from Section 6.2.
(ruleset 0 subst
  (when ((compute! ?subst)
    (= ?subst (subst ?x ?y ?z)))
  (selectmatch ?y
    ((?const)
      (has-value ?subst ?y))
    ((dbc ?body)
      (compute-value ((?res (subst ?x ?body (dbs ?z))))
        (has-value ?subst (dbc ?res))))
    ((dbn ?db-num)
      (nonmon-if (= ?db-num ?z)
        (has-value ?subst ?x)
        (has-value ?subst ?y)))
    ((?const ?arg)
      (compute-value ((?res (subst ?x ?arg ?z)))
        (has-value ?subst (?const ?res))))
    ((?const ?arg1 ?arg2)
      (compute-value ((?res1 (subst ?x ?arg1 ?z))
                      (?res2 (subst ?x ?arg2 ?z)))
        (has-value ?subst (?const ?res1 ?res2)))))
)

Notice how I created a variable ?subst to be a shorthand for (subst ?x ?y ?z), to save typing.

The compute-value ruleset form works together with the deforcfun macro to provide a form of function call. The body of a deforcfun is the same as the body of a ruleset, with the addition of an additional form, the return form. Here's an example of deforcfun, which is equivalent to the above subst ruleset.
(deforcfun (subst ?x ?y ?z)
  (selectmatch ?y
    (?const)
    (return ?y))
  ((dbc ?body)
    (compute-value ((?res (subst ?x ?body (dbs ?z))))
      (return (dbc ?res))))
  ((dbn db-num)
    (nonmon-if (= ?db-num ?z)
      (return ?x)
      (return ?y)))
  ((?const ?arg)
    (compute-value ((?res (subst ?x ?arg ?z)))
      (return (?const ?res))))
  ((?const ?arg1 ?arg2)
    (compute-value ((?res1 (subst ?x ?arg1 ?z))
                   (?res2 (subst ?x ?arg2 ?z)))
      (return (?const ?res1 ?res2))))
)

As you can see, a deforcfun is almost equivalent to a ruleset, except that there is an implicit compute! antecedent and within a deforcfun you can use return as a shorthand for a has-value.

It’s interesting to note that deforcfun and compute-value act almost like function definition and function call in a Lisp-style language even though deforcfun and compute-value expand into relatively straightforward rules in a language that doesn’t have any notion of flow control. One difference between deforcfun/compute-value and Lisp, though, is that deforcfun functions can be nondeterministic; that is, they can return more than one value for a given input. If they do, the compute-value call will go ahead and “execute” for each return value.

Let’s do one more example with deforcfun. The efficient handling of equality of a good OIL implementation can result in very simple algorithms for tasks such as parsing. Suppose you want to parse a string according to a given context-free grammar. For example, let’s take the following grammar.
\[ A := BC | CD \]
\[ B := AB | EC \]
\[ C := x | CC \]
\[ D := x | y | BA \]
\[ E := y | z | DE \]

(This grammar has a special form: the right hand side of each production is either
two nonterminals or a single terminal. It’s easy to transform an arbitrary context-free
grammar into this special form.)

Suppose that the representation for strings is such that a single character \( x \) is
represented as \((x)\), and longer strings are represented as \((app \ ?x \ ?y)\), where \(app\)
stands for “append.” Thus, one possible representation for the string \(xyzx\) would be
\((app \ (app \ (x) \ (y)) \ (app \ (z) \ (x)))\). In this case, here’s a function which, given
an arbitrary string which is parenthesized to match an expected parse tree, will return
the nonterminal associated with that parse tree.
The above function is a straightforward encoding of the CFG. Note that the function `parse` is nondeterministic; it can give several possible answers to a given query.

But what if you want to parse strings with arbitrary parenthesization? In that case, you can merely add the following rule to make sure that all parenthesizations are used.
(ruleset 0 associate-string
  (when ((compute! (parse ?s)))
    (associate! ?s))

  (when ((associate! (app ?s1 ?s2)))
    (associate! ?s1)
    (associate! ?s2))

  (when ((associate! (app (app ?s1 ?s2) ?s3)))
    (= (app (app ?s1 ?s2) ?s3) (app ?s1 (app ?s2 ?s3))))

  (when ((associate! (app ?s1 (app ?s2 ?s3))))
    (= (app (app ?s1 ?s2) ?s3) (app ?s1 (app ?s2 ?s3)))))

This is beginning to look like a very inefficient method of parsing, if you ignore the equality handling and caching. First, all possible parse trees are generated, then the parse function nondeterministically searches through each one to see if it is possible to assign a nonterminal to it (and to all its substructure). Actually, however, a good implementation of OIL will execute this program in time $O(n^3)$, where $n$ is the length of the input string.

9 The Ontic Rule Compiler

The Ontic Rule Compiler (ORC) is an environment for compiling and executing OIL programs. It is currently partially implemented; the implementation is a macro package on top of Common Lisp. The current version compiles a set of OIL rules into a C program, using GCC 2.x extensions, which is then executed using non-portable features of either Lucid Common Lisp or Clisp; it would be easy to port this to any Common Lisp which has similar features. It is mostly completed, but the optimizations described above are not implemented; also, it implements an older version of the OIL language.

There are several possibilities for further work. One is to implement and analyze the above optimizations. Another is to improve the efficiency of interactive work with the system. Currently, if a rule is removed from the system interactively, all the work of matching the antecedents against the database is still done for that rule; it is only
when a match for the rule is found that the system notices that the rule has been removed. A smarter system could avoid at least some of this wasted match work.

Another important improvement would be profiling. Some rules are far more expensive to match against the database than others, and it’s not always easy to tell whether a rule will be cheap or expensive just by looking at it. A good profiler could identify the most expensive rules, possibly allowing the user to rewrite the rule so it can be matched more cheaply. (Such a profiler could also be an important part of an antecedent-sorting optimizer.)

Another important but currently unimplemented tool for debugging is some way to deal with infinite loops. Currently, if the user inadvertently creates a nonterminating computation, the only way to find the problem is examining the program by hand; an improved system could include tools to stop the computation in the middle and examining the current state of the system, or to determine which rules are involved in the infinite loop.

10 Match Algorithms

An important part of an implementation of OIL is the match algorithm; the algorithm that checks rules against the database to find rule firings. I describe two well-known match algorithms, RETE and TREAT.

For our purposes, the essential features of RETE can be summarized using the binary rule transform (see Section 5.1.2). (This summary only holds in the positive case; that is, when there are no negated antecedents and working memory elements are never removed.)

RETE is an incremental match algorithm, which means that it runs when new elements are added to the working memory, and computes only the changes to the conflict set. For every antecedent in every rule in $B(R)$, RETE maintains the set of working memory elements which match that antecedent; for antecedents in $R$, these memories are called alpha memories, and for antecedents which were added by the binary rule transform, the memories are called beta memories.
The difference between TREAT and RETE is that TREAT does not keep beta memories, and hence is not equivalent to the binary rule transform. It does keep the alpha memories. After every WME addition that adds to an alpha memory, it checks the rules that contain that antecedent. For every such rule, it does a match using the new contents of that alpha memory and the current contents of the other alpha memories for the rule.

The TREAT algorithm doesn't specify how the match is to be done among these alpha memories, although Miranker [Mir90] mentions several possibilities. These include static ordering, where the match is done with a depth-first search in the order the antecedents appear in the rule, and seed ordering, where the antecedent with new WME's is ordered first. Another optimization is semijoin reduction. This is a two-pass method. In the first pass, the new WME is matched against the other alpha memories one at a time, and any WME's in the other alpha memories which are inconsistent with the current WME are temporarily disabled. If one of the alpha memories has no remaining enabled WME's, then the rule cannot fire, and the second pass may be skipped. Then, the second pass is just like the non-semijoin match, using static ordering or seed ordering.

RETE and TREAT are both very effective in normal uses of OPS5. Although either could be used to implement OIL, several aspects of OIL as the Ontic inference language may make RETE less suitable for this application. OPS5 supports a programming style where WME's are used as temporary storage; when the information represented by a WME is no longer needed, it is removed from the working memory. In many OPS5 systems, the average number of WME's is quite small, and RETE and TREAT work well in this situation. In contrast, OIL facts are irrevocable, so the execution of a complex OIL program may result in the creation of many WME's. RETE may have trouble in this situation, because of the large amounts of storage involved in the beta memories.

The current implementation of OIL uses a variant of TREAT.
11 Ideas for Further Research

If the language were changed so that `deforcfun` and `compute-value` were language primitives, instead of just macros on top of rules, then they could be compiled into efficient function calls. If this optimization is done in a way which preserves the semantics given above for `deforcfun` and `compute-value`, it may not gain much efficiency, because `compute!` and `has-value` facts must still be entered into the database. On the other hand, if `compute-value` is compiled into a function call in the most straightforward way, the semantics become much more difficult to describe.
References


