Towards an Intermediate Language based on Graph Rewriting

H.P. Barendregt$_2$, M.C.J.D. van Eekelen$_2$, J.R.W. Glauert$_1$, J.R. Kennaway$_1$, M.J. Plasmeijer$_2$ and M.R. Sleep$_1$.

[1]School of Information Systems, University of East Anglia, Norwich, Norfolk NR4 7TJ, U.K., partially supported by the U.K. ALVEY project,
[2]Computing Science Department, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, partially supported by the Dutch Parallel Reduction Machine Project.

Abstract.

Lean is an experimental language for specifying computations in terms of graph rewriting. It is based on an alternative to Term Rewriting Systems (TRS) in which the terms are replaced by graphs. Such a Graph Rewriting System (GRS) consists of a set of graph rewrite rules which specify how a graph may be rewritten. Besides supporting functional programming, Lean also describes imperative constructs and allows the manipulation of cyclic graphs. Programs may exhibit non-determinism as well as parallelism. In particular, Lean can serve as an intermediate language between declarative languages and machine architectures, both sequential and parallel.

1. Introduction.

Emerging technologies (VLSI, wafer-scale integration), new machine architectures, new language proposals and new implementation methods [VEG84] have inspired the computer science community to consider new models of computation. Several of these developments have little in common with the familiar Turing machine model. It is our belief that in order to be able to compare these developments, it is necessary to have a novel computational model that integrates graph manipulation, rewriting, and imperative overwriting. In this paper we present Lean, an experimental language based on such a model. In our approach we have extended Term Rewriting Systems [O'DO85, KLO85] to a model of general graph rewriting. With this model it should be possible to reason about programs, to prove correctness, and to port programs to different machines. The language as presented here does not yet meet all these goals, but we believe that it is a good step in the right direction.

A Lean computation is specified by an initial graph and a set of rules used to rewrite the graph to its final result. The rules contain graph patterns that may match some part of the graph. If the graph matches a rule it can be rewritten according to the specification in that rule. This specification makes it possible to first construct an additional graph structure and then link it into the existing graph by redirecting arcs.

Lean programs may be non-deterministic. The semantics also allows parallel evaluation where candidate rewrites do not interfere. There are few restrictions on Lean graphs (cycles are allowed and even disconnected graphs). Lean can easily describe functional graph rewriting in which only the root of the subgraph matching a pattern may be overwritten. Through non-root overwrites and use of global nodeids in disconnected patterns imperative features are also available.

In this paper we first introduce Lean informally. Then we show how a Lean program can be transformed to a program in canonical form with the same meaning. The semantics of Lean is explained using this canonical form. The semantics adopted generalises Staples' model of graph rewriting [STA80], allowing, for example, multiple redirections. A formal description of the graph rewriting model used in this paper can be found in [BAR87], as it applies to the special case of purely declarative term rewriting. After explaining the semantics we give some program examples to illustrate the power of Lean. The syntax of Lean and the canonical form is given in appendix A. Appendix B contains a summary of the predefined Lean rules.

2. General description of Lean.

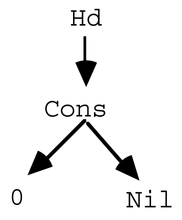
2.1 Lean graphs.

__The object that is manipulated in Lean is a directed graph called the *data graph.* When there is no confusion, the data graph is simply called *the graph*. Each node in the graph has an unique identifier associated with it (the *node identifier* or *nodeid*). Furthermore a node consists of a *symbol* and a possibly empty sequence of nodeids which define arcs to nodes in the graph. We do not assume that symbols have fixed arities. The data graph is a *closed* graph, that is, it contains no variables. It may be cyclic and may have disjoint components. This class of data graphs is, abstractly, identical to that discussed in [BAR87]. We refer to that paper for a formal discussion of the precise connection between graphs and terms.

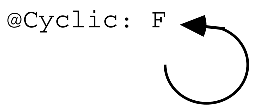
Programming with pictures is rather inconvenient so we have chosen a linear notation for graphs. In this notation we use brackets to indicate tree structure and repeated nodeids to express sharing, as shown in the examples below. Nodeids are prefixed with the character ‘@’. Symbols begin with an upper-case character.

Lean notation: Graphical equivalent:

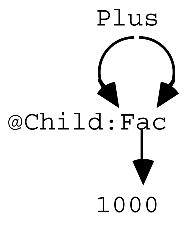
```
Hd (Cons 0 Nil);
```



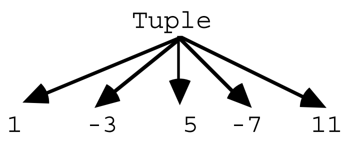
```
@Cyclic: F @Cyclic;
```



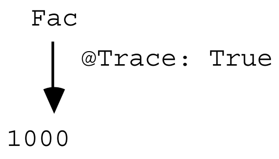
```
Plus @Child @Child,
     @Child: Fac 1000;
```



```
Tuple 1 -3 5 -7 11;
```



```
Fac 1000,
@Trace: TRUE;
```



2.2 Lean programs.

A *Lean program* consists of a set of *rewrite rules* including a *start rule.* A rewrite rule specifies a possible transformation of a given graph. The initial graph is not specified in a Lean program (see also section 4.2).

The left-hand-side of a rewrite rule consists of a Lean graph which is called a *redex pattern.* The right-hand-side consists of a (possibly empty) Lean graph called the *contractum pattern* and, optionally, a set of *redirections.* The patterns may be disconnected graphs and they are *open*, that is, they may contain *nodeid variables*. These are denoted by identifiers starting with a lower-case letter. Nodeids of the data graph may also occur in the rules. These are called *global nodeids*. When there can be no confusion with the nodeids in the data graph, we sometimes refer to the nodeid variables and the global nodeids in the rules just as nodeids. Here is an example program:

```
Hd (Cons a b)                  →  a                               ;

Fac 0                          →              1
Fac n:INT                      →              *I n (Fac (-I n 1))

F (F x)                        →  x                               ;

Start                          →  Fac (Hd (Cons 1000 Nil))        ;
```

The first symbol in a redex pattern is called the *function symbol.* Rules starting with the same function symbol are collected together forming a *rule-group*. The members of a rule-group are separated by a ‘|’. Note that function symbols may also occur at other positions than the head of the pattern. A symbol which does not occur at the head of any pattern in the program is called a *constructor symbol.*

2.3 Rewriting the data graph.

The initial graph of a Lean program is rewritten to a final form by a sequence of applications of individual rewrite rules. A rule can only be applied if its redex pattern matches a subgraph of the data graph. A redex pattern in general consists of variables and symbols. An *instance* of a redex pattern is a subgraph of the data graph, such that there is a mapping from the pattern to that subgraph which preserves the node structure and is the identity on constants. This mapping is also called a *match*. The subgraph which matches a redex pattern is called a *redex* (*red*ucible *ex*pression) for the rule concerned.

We will use the following rules which have a well-known meaning, as a running example to illustrate several concepts of Lean.

```
Add Zero z                     →  z                            |  (1)
Add (Succ a) z                 →              Succ (Add a z)   ;
```

Now assume that we have the following data graph:

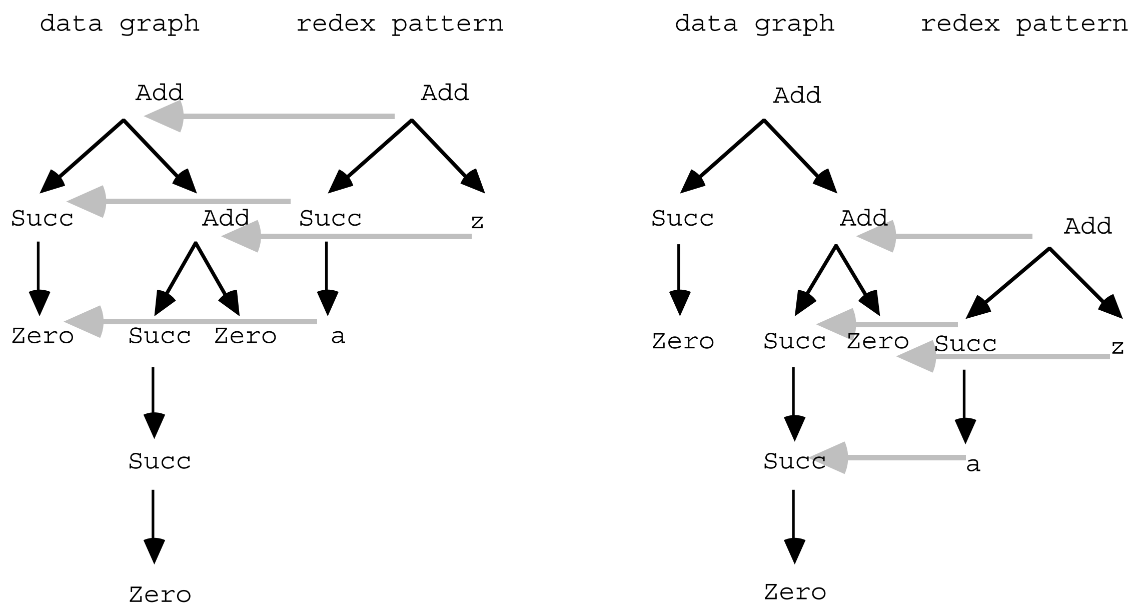
```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero);
```

There are two redexes:

```
                                   a                  z
          ____  ______  _____  ________________________________
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)              redex matching rule 2

                                       a          z
                 ____  _____  __________  _____
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)              redex matching rule 2
```

In graphical form this is:



Note that there may be several rules for which there are redexes in the graph. A rule may match several redexes and a redex can match several rules. For instance, in the example above there is only one rule which matches any part of the data graph, but it matches two redexes. In general, therefore, there are many rewriting sequences for a given graph.

Evaluation of a Lean program is controlled by a *rewriting strategy*. In its most general form:

(1) It decides which rewritings to perform.
(2) It decides when to perform no further rewritings. The graph at this point is said to be in *strategy normal form*, or briefly, in *normal form*.
(3) It specifies what part of the resulting graph is the *outcome* of the computation.

For the purposes of graphical implementations of functional languages, strategies need only consider the subgraph of nodes accessible from the data root, for the purposes of identifying both redexes and terminal states. However, more general applications of Lean may not wish to be constrained in this way: for example, graphical rewrite rules may be used to represent non-terminating behaviours of practical interest such as operating systems.

The choices made by a rewriting strategy may affect the efficiency of rewriting, as well as its termination properties. In the future we aim to incorporate facilities into Lean to permit programmer control of strategy where necessary.

Once the strategy has chosen a particular redex and rule, rewriting is performed. The first step is to create an instantiation of the graph pattern specified on the right-hand-side of the chosen rule. This instantiation is called the *contractum*. In general this contractum has links to the original graph since references to nodeid variables from the left-hand-side are linked to the corresponding nodes identified during matching. A new data graph is finally constructed by redirecting some arcs from the original graph to the contractum. In most cases all arcs to the root node of the redex are redirected to the root node of the contractum as in Staples' model [STA80]. This has an effect similar to "overwriting" the root of the redex with the root of the contractum. This is what happens when no redirections are given explicitly in the rule. Explicit redirection of arbitrary nodes is also possible.

The process of performing one rewrite step is often called a *reduction*. The graph after one reduction is called the *result* of the reduction. Initially, the data graph contains a node with the symbol `Start`. Hence, the rewriting process can begin with matching the start rule and hereafter rewriting is performed repeatedly until the strategy has transformed the graph to one which it deems to be in normal form.

[BAR87] gives a formal discussion of how graph rewrite rules with root-only redirection model term rewriting, and proves certain soundness and completeness results. The definition of rewriting given in that paper only covers rules of this form, but the extension of the formal description to the general cases of multiple and/or non-root redirection is straightforward.

The data graph of the previous example can be rewritten in the following way:

```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)                     → (2)
Succ (Add Zero (Add (Succ (Succ Zero)) Zero))                     → (1)
Succ (Add (Succ (Succ Zero)) Zero)                                → (2)
Succ (Succ (Add (Succ Zero) Zero))                                → (2)
Succ (Succ (Succ (Add Zero Zero)))                                → (1)
Succ (Succ (Succ Zero))
```

Note that in this example the graph was actually a tree, and remained a tree throughout. There was no difference with a Term Rewriting System. In the following example there is a data graph in which parts are shared. Rewriting the shared part will reduce the number of rewriting steps compared to an equivalent Term Rewriting System.

```
Add @X @X,              @X: Add (Succ Zero) Zero                  → (2)
Add @X @X,              @X: Succ (Add Zero Zero)                  → (1)
Add @X @X,              @X: Succ Zero                             → (2)
Succ (Add @Z @X),       @X: Succ @Z,               @Z: Zero       → (1)
Succ (Succ Zero)
```

2.4 Predefined delta rules.

For practical reasons it is convenient that rules for performing arithmetic on primitive types (numbers, characters etc.) are predefined and efficiently implemented. In Lean a number of basic constructors for primitive types such as `INT`, `REAL` and `CHAR` are predefined. Representatives of these types can be denoted: for instance `5` (an integer), `5.0` (a real), `'5'` (a character). Basic functions, called *delta rules*, are predefined on these basic types.

The actual implementation of a representative of a basic type, is hidden for the Lean programmer. It is possible to denote a representative, pass a representative to a function or delta-rule and check whether or not an argument is of a certain type in the redex pattern.

```
Nfib 0              →  1                                              |
Nfib 1              →  1                                              |
Nfib n:INT          →  ++I  (+I  (Nfib  (-I  n  1))  (Nfib  (-I  n  2)))        ;
```

In this example ‘`0`’, ‘`1`’ and ‘`2`’ are denotations for some representation of the numbers 0, 1 and 2, ‘`+I`’, ‘`-I`’ and ‘`++I`’ are function symbols for predefined delta rules defined on these representations. Hence, an integer consists of the unary constructor `INT` and an unknown representation. Note that in general one is allowed to specify just the constructor in the redex pattern of a rule. The value can be passed to a function by passing the corresponding nodeid (`n` in the example).

These predefined rules are however not strictly necessary. For instance, one could define numbers as: `INT Zero` to denote `0`, `INT (Succ Zero)` to denote `1`, `INT (Succ (Succ Zero))` to denote `2` etc., and define a function for doing addition

```
PlusI (INT x) (INT y)            →  INT (Add x y)                            ;
```

where `Add` is our running example. This kind of definition makes it possible to do arithmetic in a convenient way. However, for an efficient implementation one would probably not choose such a Peano-like representation of numbers, but prefer to use the integer and real representation and the arithmetic available on the computer.

3. Translating to canonical form.

Lean contains syntactic sugar intended to make programs easier to read and write. Explaining the semantics of Lean will be done with a form with all syntactic sugar removed known as *Canonical Lean*. In this section we show how a Lean program can be transformed to its canonical form. Canonical Lean programs are valid Lean programs and are unaffected by this translation procedure.

Every Lean program can be seen as a shorthand for its canonical form. Note that this section is all about syntax. The semantics of the canonical form are explained in section 4.

In the canonical form every node has a definition and definitions are not nested. Every redirection, including any redirection of the root, is done explicitly and in patterns all arguments of constructors are specified. In this canonical form a rewrite rule has the following syntax:

```
Graph              →           [ Graph , ] Redirections
```

The first `Graph` is the redex pattern. The second is the optional contractum pattern. Each pattern is represented as a list of node definitions of the form:

```
Nodeid:  Symbol { Nodeid }
```

Braces mean zero or more occurrences. The initial *Nodeid* identifies the node, *Symbol* is some function or constructor symbol and the sequence of nodeids identifies zero or more child nodes. Occurrences of nodeids before a colon are *defining* occurrences. Every nodeid must have at most one defining occurrence within a rule. Defining occurrences of global nodeids are allowed on the left-hand-side only. Within a rule a nodeid which appears on the right-hand-side must either have a definition on the right-hand-side or it must also appear on the left-hand-side.

3.1 Add explicit nodeids and flatten.

In the canonical form all nodes have explicit nodeids and there are no nested node definitions. Hence in each rule we have to introduce a new unique nodeid variable for every node that does not yet have one. Every nested node definition in the rule is then replaced by an application of the corresponding nodeid variable, and the definitions are moved to the outer level. Applying this transformation to our running example gives:

```
x: Add    y   z,
y: Zero                   →  z                           |
x: Add    y   z,
y: Succ   a               →  m: Succ n,
                             n: Add  a z                 ;
```

All arguments of symbols (such as `Add` and `Succ`) have now become nodeids and brackets are no longer needed.

3.2 Specify the arguments of constructors.

In Lean one may write the following function which checks to see if a list is empty:

```
x: IsNil n,
n: Nil             →           t: TRUE                     |
x: IsNil n,
n: Cons              →            t: FALSE                              ;
```

`Cons` is a binary constructor symbol, but in Lean one may omit the specification of the arguments if they are not used elsewhere in the rule. This is not allowed in the canonical form hence the arguments are made explicit by introducing two new nodeid variables. Transformation of the example above will give:

```
x: IsNil n,
n: Nil               →            t: TRUE                               |
x: IsNil n,
n: Cons y z          →            t: FALSE                              ;
```

3.3 Make root redirections explicit.

The meaning of both rules in the running example is that the root of the pattern is redirected to the root of the contractum. Redirections are always made explicit in the canonical form. If no redirections are specified explicitly, a redirection is introduced to redirect the redex root to the contractum root. Note that if the right-hand-side of a rule consists only of a nodeid, the root of the redex is redirected to this nodeid. The running example with explicit redirections now becomes:

```
x: Add    y   z,
y: Zero                        →  x := z                            |
x: Add    y   z,
y: Succ   a                    →  m: Succ n,
                                  n: Add  a z,
                                  x := m                            ;
```

3.4 Define delta rules.

The predefined rules which are used in the program have to be explicitly added. We can assume that these rules are already in canonical form.

4. Semantics of Lean.

4.1 Graph terminology.

- Let **F** be a set of symbols and **N** be a set of nodes.
- Further, let **C** be a function (the *contents function*) from N to F $\times$ N*.
- Then C specifies a *Lean Graph* over F and N.
- If node n has contents F n_1 n_2 ... n_k we say the node contains *symbol* F and *arguments* $n_1,n_2,...,n_k$.

- There is a distinguished node in the graph which is the *root* of the graph.

In standard graph theory, a Lean graph is a form of directed graph in which each node is labelled with a symbol, and its set of out-arcs is given an ordering. In Lean nodes are denoted by their names, i.e. their nodeids. The canonical form defined in section 3 can be regarded as a tabulation of the contents function. We will explain the semantics of Lean using this canonical form.

4.2 The initial graph.

The initial graph is not specified in a program. It always takes the following form:

```
@DataRoot:   Graph @StartNode @GlobId1 @GlobId2 ... @GlobIdm,
@StartNode:         Start,
@GlobId1:    Initial,
@GlobId2:    Initial,
...
@GlobIdm:    Initial;
```

The root of the initial graph contains the nodeid of the start node which initially contains the symbol `Start`. The root node will always contain the root of the graph to be rewritten. Furthermore the root node contains all global nodeids addressed in the Lean rules. The corresponding nodes are initialised with the symbol `Initial`.

4.3 Operational semantics for rewriting.

Let G be a Lean graph, and R the set of rewrite rules. A reduction option, or *redop*, of G is a triple T which consists of a redex g, a rule r and a match μ. The match μ is a mapping from the nodes of the redex pattern p to the nodes of the graph G such that for every node x of p, if $C_p(x) = s\ x_1\ x_2\ ...\ x_n$ then $C_g(\mu(x)) = s\ \mu(x_1)\ \mu(x_2)\ ...\ \mu(x_n)$. That is, μ preserves node structure. A redop introduces an available choice for rewriting the graph. A redop that is chosen is called a *rewrite* of the graph. The process of performing a rewrite is also called *rewriting*.

The contractum pattern may contain nodeid variables which are not present in the redex pattern. These correspond to the identifiers of new nodes to be introduced during rewriting. The mapping μ' is introduced taking as its domain the set of nodeid variables which only appear in the contractum pattern. Each of these is mapped to a distinct, new, nodeid which does not appear in G or R.

The domains of μ and μ' are distinct, but every nodeid variable in the contractum pattern is in the domain of one or the other. In order to compute the result of a rewrite one applies the mapping μ'' formed by combining μ and μ', to the contractum pattern resulting in the *contractum*.

Finally the new graph is constructed by taking the union of the old graph and the contractum, replacing nodeids in this union (and in the case that global nodeids are mentioned also in the rules) as specified by the redirections in the rewrite rule of the chosen redop.

Hence rewriting involves a number of steps:

1. A redop is chosen by the rewriting strategy. This gives us a redex in the graph G, a rule which specifies how to rewrite the redex and a mapping μ.

2. The contractum is constructed in the following way.
   - invent new nodeids (not present in G or R) for each variable found only in the contractum pattern. This mapping is called μ'.
   - apply μ'', the combination of μ and μ', to the contractum pattern of the rule yielding the contractum graph C. Note that the contractum pattern, and hence C, may be empty.

3. The new graph G' is constructed by taking the union of G and C.

4. Each redirection in a rule takes the form O := N. In terms of the syntactic representation, this is performed by substituting N for every applied occurrence of O in the graph G' and in the rules R. The definition of O still remains. The nodeids O and N are determined by applying μ'' to the left-hand-side and the right-hand-side of the redirection. All redirections specified in the rule are done in parallel. This results in the new graph G''.

The strategy will start with a rewrite rule which matches the symbol `Start` in the initial graph. When a computation terminates, its *outcome* is that part of the final graph which is accessible from the root. Thus a “garbage collection” is assumed to be performed at the end of the computation only. A real implementation may optimise this by collecting nodes earlier, if it can predict that so doing will not affect the final outcome. Which nodes can be collected earlier will in general depend on the the rule-set of the program and the computation strategy being used. Note that before the computation has terminated, nodes which are inaccessible from the root may yet have an effect on the final outcome, so they cannot necessarily be considered garbage. For certain strategies and rule-sets they will be, but inaccessibility is not in itself the definition of garbage.

Redirection of global nodeids has as a consequence that all references to the original global nodeid have to be changed. This includes the references to global nodeids made in the rewrite rules. Hence global nodeids can be viewed as *global* variables (they have a global scope), where nodeid variables are *local* variables (they have a meaning only within a single rule).

4.4 A small example.

We return to our running example with a small initial graph and see how rewriting proceeds. The rewriting strategy we choose will rewrite until the data graph contains no redexes only examining nodes accessible from the @Dataroot.

```
   x: Add  y   z,
   y: Zero                         →  x := z                            |  (1)
   x: Add  y   z,
   y: Succ a                       →  m: Succ n,
                                      n: Add  a z,
                                      x := m                            ;  (2)
___x: Start                        →  m: Add  n o,
                                      n: Succ o,
                                      o: Zero,
                                      x := m                            ;  (3)
```

Initially we have the following graph:

```
@DataRoot : Graph @StartNode,
@StartNode: Start;
```

We now follow the rewrite steps.

1. The start node is the only redex matching rule (3). The mapping is trivial: μ(x) = @StartNode and the redex in the graph is:

```
@StartNode: Start;
```

2. Applying μ to the contractum scheme will leave this as it is while x does not appear in it. For all variables in the scheme we invent new nodeids and map the variables as follows: μ'(m) = @A, μ'(n) = @B, μ'(o) = @C, μ'(p) = @C. The contractum can now be constructed:

```
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

3. The union of C and G is G':

```
@DataRoot : Graph @StartNode,
@StartNode: Start,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

4. We have to redirect μ"(x) = @StartNode to μ"(m) = @A. All applied occurrences of @StartNode will be replaced by occurrences to @A. The graph G" after rewriting is now:

```
@DataRoot : Graph @A,
@StartNode: Start,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

This completes one rewrite. The start node will not be examined by the strategy anymore. Therefore it can be considered as garbage and it will be thrown away. The strategy will not stop yet because the graph still contains a redex accessible from the `@DataRoot`.

1. The strategy will choose the only redop. It matches rule 2: μ(x) = `@A`, μ(y) = `@B`, μ(z) = `@C`, μ(a) = `@C`;

2. Invent new nodeids and map the variables as follows: μ'(m) = `@D`, μ'(n) = `@E`. The contractum can now be constructed:

```
@D: Succ @E,
@E: Add  @C @C;
```

3. The union of C and G is G':

```
@DataRoot: Graph @A,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero,
@D: Succ @E,
@E: Add  @C @C;
```

4. We have to redirect μ''(x) = `@A` to μ''(m) = `@D`. The graph G'' after rewriting and removing garbage is:

```
@DataRoot: Graph @D,
@C: Zero,
@D: Succ @E,
@E: Add  @C @C;
```

It is now clear how this process may continue: `@E` is a redex and it matches rule 1: μ(x) = `@E`, μ(y) = `@C`, μ(z) = `@C`. The strategy chooses this redop, there is no new contractum graph but just a single redirection which takes μ''(x) = `@E` to μ''(z) = `@C` yielding the expected normal form:

```
@DataRoot: Graph @D,
@C: Zero,
@D: Succ @C;
```

5. Some Lean programs.

5.1 Merging lists.

The following Lean rules can merge two ordered lists of integers (without duplicated elements) into a single ordered list (without duplicated elements).

```
Merge     Nil       Nil         →  Nil                                |
Merge     f:Cons    Nil         →  f                                  |
Merge     Nil       s:Cons      →  s                                  |
Merge     f:(Cons a b)
          s:(Cons c d)          →                    IF (<I a c)
                                       (Cons a (Merge b s))
                                       (IF (=I a c)
                                           (Merge f d)
                                           (Cons c (Merge f d)))      ;
```

<I, =I and IF are predefined delta rules (see appendix B) with the obvious semantics. Note that the right-hand-side of the last rule uses an application of the argument as a whole as well as its decomposition.

5.2 Higher order functions, currying.

In this example we show how higher-order functions are treated in Lean, by giving the familiar definition of the function `Map`.

```
Map       f         Nil         →  Nil                                | (1)
Map       f         (Cons a b)  →                    Cons (Ap f a) (Map f
b)        ;         (2)
Ap        (*I a)    b           →                    * I       a       b
;         (3)
Start                           →  Map (*I 2) (Cons 3 (Cons 4 Nil))   ; (4)
```

This can be rewritten, for example, in the following way:

```
Start                                                         → (4)
Map (*I 2) (Cons 3 (Cons 4 Nil))                              → (2)
Cons (Ap @L 3) (Map @L (Cons 4 Nil)),              @L:*I 2    → (3)
Cons (*I 2 3) (Map @L (Cons 4 Nil)),               @L:*I 2    → (*I)
Cons 6 (Map @L (Cons 4 Nil)),                      @L:*I 2    → (2)
Cons 6 (Cons (Ap @L 4) (Map @L Nil)),              @L:*I 2    → (3)
Cons 6 (Cons (*I 2 4) (Map @L Nil)),               @L:*I 2    → (*I)
Cons 6 (Cons 8 (Map @L Nil)),                      @L:*I 2    → (1)
Cons 6 (Cons 8 Nil)
```

Rule `(3)` of this example will rewrite `(Ap (*I 2) 3)` to its uncurried form `(*I 2 3)` which makes multiplication possible. One will need such an “uncurry” rule for every function which is used in a curried manner. Note that during rewriting the node `@L:(*I 2)` is shared. In this case sharing only saves space, but not computation.

5.3 Graphs with cycles.

```
Ham       → Cons 1 (Merge (Map (*I 2) Ham) (Map (*I 3) Ham))          ;
```

A more efficient solution to this problem can be obtained by means of creating cyclic sharing in the contractum making heavy use of computation already done. The new definition is:

```
x: Ham    → Cons 1 (Merge (Map (*I 2) x) (Map (*I 3) x))             ;
```

5.4 Copying a tree structure.

This example is very straightforward if the structure of tree nodes is known. Here is a program which copies a binary tree structure.

```
Copy      (Bin left right)      → Bin (Copy left) (Copy right)        |
Copy      (Leaf x)                → x                                 ;
```

In the present version of Lean it is not possible to copy an arbitrary unknown data structure. We hope to support more general solutions in a future version of Lean.

5.5 Counting specific rewrites via global assignment.

```
r: Hd (Cons a b),
@HdCount: Total n:INT          →  newvalue: Total (++I n),
                                  r := a,
                                  @HdCount := newvalue               ;

r: Start                       →                    nr: Hd (Cons 1 (Cons 2
Nil)),
                                  initvalue: Total 0,
                                  @HdCount := initvalue              ;
```

We are dealing with disconnected graphs and patterns in this example. The global nodeid `@HdCount` in the graph is addressed in a rewrite rule. The integer value in `@HdCount` will be increased each time a head of a list is taken. Global nodeids and arbitrary redirections in rewrite rules make other styles of programming possible involving globals and side effects. Here, the retention of the canonical notation forces the user to make his text inelegant. Perhaps a useful danger signal, both to reader and writer?

5.6 Unification using redirection.

This program implements a simple unification algorithm. It operates on representations of two types, returning “cannot unify” in case of failure. The types are contructed from three basic types `I`, `B` and `Var` and a composing constructor `Com`. Different type variables are represented by distinct nodes. Repeated type variables are represented by shared nodes. References to such a shared node are taken to be references to the same variable.

```
root: Start                          →  Unify t1 t2 root,
                                              t1: Com i t1,
                                              t2: Com i (Com i t2),
                                              i: I                        ;

   Unify x              x            r  →  x
 o:   Unify t1:(Com x y)             t2:(Com p q)       r               → n:
Com (Unify x p r) (Unify y q r),
 o:   Unify t1:Var      t2           r  →  o := t2, t1 := t2          |
 o:   Unify t1                       t2:Var             r               → o
:= t1, t2 := t1         |
   Unify t1:Com         t2:I         r  →  n: "cannot unify", r := n    |
   Unify t1:Com         t2:B         r  →  n: "cannot unify", r := n    |
   Unify t1:I           t2:Com       r  →  n: "cannot unify", r := n    |
   Unify t1:B           t2:Com       r  →  n: "cannot unify", r := n    |
   Unify t1:I           t2:B         r  →  n: "cannot unify", r := n    |
   Unify t1:B           t2:I         r  →  n: "cannot unify", r := n    ;
```

5.7 Combinatory logic.

Here we show the Lean equivalent of a well-known TRS using explicit application: combinatory logic.

```
  Ap (Ap (Ap S a) b) c)       → Ap (Ap a c) (Ap b c)                    |
  Ap (Ap K a) b)              → a                                       ;

  Start           →          Ap (Ap (Ap S (Ap K K)) (Ap S K)) (Ap (Ap K K)
K))               ;
```

6. Future work.

Lean is the result of collaboration between two research groups: the Dutch Parallel Reduction Machine (DPRM) group at Nijmegen and the Declarative Alvey Compiler Target Language (DACTL) group at UEA. Recognising the current instability of emerging languages and architectures, both groups wish to identify a computational model appropriate to a new generation rewriting model of computing. The DPRM group has developed a subset of Lean, called Clean [BRU87], for the support of purely functional languages. Dactl0 [GLA87] predates Lean, and includes some concepts not present in Lean. In the future, our groups plan to continue to collaborate on further developing and refining the computational model and the Lean language based on it. It is intended that later versions of Lean and Dactl will converge.

Because rewriting strategies have a critical influence on efficiency and outcome, future versions of Lean aim to offer the programmer explicit control. Strategies should be based mainly on local information so that concurrent evaluation is not constrained. One approach is to employ fine grain control annotations so that a rule may nominate which of the nodes it creates should be considered as roots for future redexes. Dactl0 adopts this approach. Its main advantage is that a simple execution model is obtained. Another approach is to have a high level specification of strategies and a formalism for combining strategies during evaluation. This approach holds out promise for global reasoning [EEK86]. We believe that the way forward should involve a careful combination of these approaches. At the high level formally specified strategy information should be used, allowing analysis and transformation of programs using abstract interpretation techniques. Correctness preserving translation tools would then convert such a program into a form using a small set of well-designed control primitives suitable for efficient parallel implementation.

Besides strategies, there are several other concepts that may be incorporated in Lean in the near future. These include: more general typing; annotations to allow compiler optimisations; interfacing with the outside world; modules and separate compilation facilities; support for unification.

7. Conclusions.

Lean is an experimental language for specifying computations in terms of graph rewriting. It is very powerful since there are few restrictions on the graph that is transformed and the transformations that can be performed.

The graph rewriting model underlying Lean is of independent interest as a general model of computation for parallel architectures. It includes as special cases, more restricted systems, such as Graph Rewriting Systems which model Term Rewriting Systems. For these GRSs certain soundness and completeness results are shown in [BAR87].

Lean is designed to be a useful intermediate language for those language implementations which rely on graph rewriting. Compilers targetted to Lean are being implemented for functional languages. Interpreters for Lean are under development [JAN87]. A first version of a compiler for a restricted subset of Lean (Clean) is now running on a Vax750 (Unix) [BRU87]. The performance is encouraging.

8. Acknowledgements.

We would like to thank Jan-Willem Klop of the Centre for Mathematics and Computer Science in Amsterdam for his explanations, Tom Brus and Maarten van Leer of the University of Nijmegen and Nic Holt of ICL for their valuable comments.

9. References.

[BAR87] Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., “Term Graph Reduction”, these proceedings, 1987.

[BRU87] Brus, T., Eekelen, M.C.J.D. van, Leer, M. van, Plasmeijer, M.J., “Clean - A Language for Functional Graph Rewriting”, University of Nijmegen, Internal Report nr. 95, February 1987.

[EEK86] Eekelen, M.C.J.D. van, Plasmeijer, M.J., “Specification of rewriting strategies in Term Rewriting Systems”, University of Nijmegen, Workshop on Graph Reduction, Santa Fe, 1986.

[GLA87] Glauert, J.R.W., Kennaway, J.R., Sleep, M.R., “DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction”, University of East Anglia 1987, Internal Report SYS-C87-03.

[JAN87] Jansen, T, “Interpreting Lean”, Masters thesis, University of Nijmegen, may 1987.

[KLO85] Klop, J.W., “Term rewriting systems”, Notes for the Seminar on Reduction Machines, Ustica 1985, to appear.

[O’DO85] O’Donnell, M.J., “Equational Logic as a Programming Language”, Foundations of Computing Series, MIT Press, 1985.

[STA80] Staples, J., “Computation on Graph-like Expressions”, Theor. Comp. Sc. 10, North-Holland, 1980, pp. 171-185.

[VEG84] Vegdahl, Steven R., “A Survey of Proposed Architectures for the Execution of Functional Languages”, IEEE Transactions on Computers, vol. c-33, no. 12, december 1984.

Appendix A : Syntax

```
LeanProgram      =   { RuleGroup }.
RuleGroup        =   Rule { '|' Rule } ';'.
Rule             =   Graph '->' Graph [ ',' Redirections ]  |  Graph '->' Redirections.
Graph            =   [ Nodeid ':' ] Node { ',' NodeDefinition }.
NodeDefinition   =   Nodeid ':' Node .
Node             =   Symbol { Term }.
Term             =   Nodeid  |  [ Nodeid ':' ] Symbol  |  [ Nodeid ':' ] '(' Node ')'.
Redirections     =   Redirection { ',' Redirection }  |  Nodeid { ',' Redirection }.
Redirection      =   Nodeid ':=' Nodeid.
```

For the canonical form of Lean replace the following rules in the syntax above;

```
Rule             =   Graph '->' [Graph ','] Redirections.
Graph            =   NodeDefinition { ',' NodeDefinition }.
Term             =   Nodeid.
Redirections     =   Redirection { ',' Redirection }.
```

Appendix B : Predefined Delta Rules.

Name	Type	Description
EXP	REAL -> REAL	exponent
LOG	REAL -> REAL	logarithm base e
LOG10	REAL -> REAL	logarithm base 10
SQRT	REAL -> REAL	square root
POW	(REAL,REAL) -> REAL	x^y
NOT	BOOL -> BOOL	invert
AND	(BOOL,BOOL -> BOOL	and
OR	(BOOL,BOOL) -> BOOL	or
XOR	(BOOL,BOOL) -> BOOL	exclusive or
=B	(BOOL,BOOL) -> BOOL	equal
<>B	(BOOL,BOOL) -> BOOL	not equal
+C	(CHAR,CHAR) -> CHAR	plus
-C	(CHAR,CHAR) -> CHAR	minus
++C	CHAR -> CHAR	increment
--C	CHAR -> CHAR	decrement
=C	(CHAR,CHAR) -> BOOL	equal
<>C	(CHAR,CHAR) -> BOOL	not equal
<C	(CHAR,CHAR) -> BOOL	lower than
>C	(CHAR,CHAR) -> BOOL	higher than
<=C	(CHAR,CHAR) -> BOOL	lower or equal
>=C	(CHAR,CHAR) -> BOOL	higher or equal
+S	(STRING,STRING) -> STRING	concatenate strings
SLICE	(STRING,INT,INT) -> STRING	slice string
INDEX	(STRING,INT) -> CHAR	index string
UPDATE	(STRING,INT,CHAR) -> STRING	replace char
LENGTH	STRING -> INT	length of string
=S	(STRING,STRING) -> BOOL	equal
<>S	(STRING,STRING) -> BOOL	not equal
<S	(STRING,STRING) -> BOOL	lower than
>S	(STRING,STRING) -> BOOL	higher than
<=S	(STRING,STRING) -> BOOL	lower or equal
>=S	(STRING,STRING) -> BOOL	higher or equal
RTOI	REAL -> INT	convert REAL to INT
ITOR	INT -> REAL	convert INT to REAL
CTOI	CHAR -> INT	convert CHAR to INT
ITOC	INT -> CHAR	convert INT to CHAR
ITOS	INT -> STRING	transform to string
RTOS	REAL -> STRING	transform to string
BTOS	BOOL -> STRING	transform to string
CTOS	CHAR -> STRING	transform to string
ABORT	STRING -> halt	write string, stop.

Name	Type	Description
IF	(BOOL,*,*) -> *	normal IF
READLINES	STRING -> [STRING]	read file
READCHARS	STRING -> [CHAR]	read file
+I	(INT,INT) -> INT	plus
-I	(INT,INT) -> INT	minus
*I	(INT,INT) -> INT	times
/I	(INT,INT) -> INT	divide
DIV	(INT,INT) -> INT	divide
MOD	(INT,INT) -> INT	modulo
NOT%	INT -> INT	bit NOT
AND%	(INT,INT) -> INT	bit AND
OR%	(INT,INT) -> INT	bit OR
XOR%	(INT,INT) -> INT	bit XOR
ROTL%	(INT,INT) -> INT	bit rotate left
++I	INT -> INT	increment
--I	INT -> INT	decrement
=I	(INT,INT) -> BOOL	equal
<>I	(INT,INT) -> BOOL	not equal
<I	(INT,INT) -> BOO	lower than
>I	(INT,INT) -> BOOL	higher than
<=I	(INT,INT) -> BOOL	lower or equal
>=I	(INT,INT) -> BOOL	higher or equal
RANDOM	-> INT	random number
+R	(REAL,REAL) -> REAL	plus
-R	(REAL,REAL) -> REAL	minus
*R	(REAL,REAL) -> REAL	times
/R	(REAL,REAL) -> REAL	divide
++R	REAL->REAL	increment
--R	REAL->REAL	decrement
=R	(REAL,REAL) -> BOOL	equal
<>R	(REAL,REAL) -> BOOL	not equal
<R	(REAL,REAL) -> BOOL	lower than
>R	(REAL,REAL) -> BOOL	higher than
<=R	(REAL,REAL) -> BOOL	lower or equal
>=R	(REAL,REAL) -> BOOL	higher or equal
SIN	REAL -> REAL	sinus
COS	REAL -> REAL	cosinus
ASIN	REAL -> REAL	arcsinus
ACOS	REAL -> REAL	arccosinus
ATAN	REAL -> REAL	arctangens
ATAN2	(REAL,REAL) -> REAL	arctangens x/y