LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Id Nouveau

Computation Structures Group Memo 265

July 23, 1986

Rishiyur S. Nikhil
Keshav Pingali
Arvind

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-83-K-0125.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

1. Introduction 1

1.1. Document Outline 2

2. Functional Id Nouveau(*or* Id Nouveau without I-structures) 3

2.1. Numbers, Strings, Booleans and nil 4

2.2. Identifiers, or Names 4

2.3. Grouping with Parentheses 5

2.4. Applications 5

2.5. Conditionals 5

2.6. Infix Operators 6

2.7. Let-Blocks 6

2.7.1. Renaming Identifers in Blocks 8

2.8. Tuples 9

2.8.1. Simplifying Tuple-structured Bindings 10

2.8.2. General Recursive Definitions 11

2.9. Curried vs. Multi-Argument Procedures 12

2.10. Loops 14

3. I-structures 18

3.1. I-Structure Operations 18

3.1.1. I-structure Allocation 18

3.1.2. I-structure Component Assignment 18

3.1.3. I-structure Component Selection 19

3.1.4. I-structure Index Bounds 19

3.1.5. Nested I-structures 20

3.2. Using I-structures in Programs 21

3.2.1. Expressions Revisited 21

3.2.2. Let-Blocks Revisited 21

3.2.3. Procedures Revisited 22

3.2.4. Loops Revisited 23

3.3. Discussion 23

3.3.1. Efficiency 24

3.3.2. Determinacy 26

3.3.3. Implementation 26

4. A Rewrite-Rule Operational Semantics for Id Nouveau 28

4.1. Operational Semantics for a Functional Subset of Id Nouveau 28

4.1.1. The Language 28

4.1.2. Reduction Rules and Redexes 29

4.1.3. Computation, Answers and Normal Forms 30

4.1.4. Computation Rules, Determinacy, Termination and Efficiency 31

4.2. Identifier-Bindings and Expressions When There Are Side Effects 33

4.3. Operational Semantics for Id Kernel 35

4.3.1. The Language 35

4.3.2. Reduction Rules, Redexes, Substitution Rules and Subexes 37

4.3.3. Computation and Answers 41

4.3.4. The Dataflow Computation Rule 43

4.3.5. An Example 43
4.4. Translating an Id Nouveau Program into an Id Kernel Program 46
4.4.1. Implementing Tuples Using I-structures 47
4.4.2. Naming Copyable Subexpressions 48
4.4.3. Lambda-Lifting 48
4.5. Conclusion 51
Appendix A. Example Programs 53
Appendix B. Id Kernel: Collected Syntax and Semantics 58
B.1. Id Kernel Syntax 58
B.2. Reduction Rules 58
B.3. Redexes and Reduction 60
B.4. Substitutable Forms 60
B.5. Subexes and Substitution 61
B.6. Computation 61
B.7. Final Form of the Query-Expression 61
B.8. The Answer Computed by a Program 61
B.9. Dataflow Computation Rule 62

List of Figures

Figure 4-1: Id Kernel Syntax 36

Figure 4-2: Substitutable Forms 41

Figure 4-3: Final Form of the Query-Expression 42

Figure 4-4: The Answer Computed by a Program 43

Figure 4-5: Dataflow Computation Rule 43

1. Introduction

In this document we define the syntax and abstract operational semantics of the programming language "Id Nouveau". Id Nouveau is an an evolutionary successor to Id [3, 9] and Id/83s [12], languages used to program the MIT Tagged-Token Dataflow Machine [2, 1].

Our underlying motivation is the design of general-purpose computer architectures that achieve high performance by exploiting parallelism. But a parallel architecture is not enough-- we need to explore programming languages, programming methodology and compiling technology that will make effective use of such architectures. Indeed it is our belief that these research directions must be pursued together-- the architecture will depend heavily on the choice of programming language and compiling technology.

A programming language for parallel machines must admit the division of a problem into smaller tasks that can be performed in parallel. Axiomatic in our approach is the belief that the programmer should be insulated from the details of this division. We are thus led to certain design choices:

- The programmer should not have to annotate the program to specify what can be done in parallel. This information is implicit in the data-dependencies in the program and should be recognized and exploited automatically (by the compiler/architecture).

- The programmer should not have to change the program when the machine configuration changes (e.g. doubling the number of processors, faster technology, different interconnection, failure of one processor, etc.). Preferably, the program should not even have to be recompiled for a new machine configuration.

Surprisingly, *very* few other researchers (and *no* commercial products) aim for this degree of machine independence.

An important consequence of these requirements is that the language be *determinate, i.e.* that the result computed by a program be independent of the particular schedule chosen by the machine for executing parallel tasks. Without this property of the programming language, we believe that the programmer will be hopelessly bogged down, battling the vagaries of machine configuration and technology-dependent timing considerations.

For these reasons, functional programming languages have attracted many researchers in this field. The semantics of functional languages admits much parallelism, and these languages are determinate (they are Church-Rosser). Indeed Id [3], a predecessor of Id Nouveau, was a functional language.

But experience with functional languages reveals a serious problem-- the treatment of data-structures. In functional languages, there is no concept of *in situ* update, and this can lead to excessive copying and lack of parallelism in data-structures. However, unrestricted *in situ* updates would bring us back into the class of imperative languages, and therefore reintroduce indeterminacy. It was to solve this problem that a new data-structuring mechanism called "I-structures" was introduced in [5] and subsequently refined. I-structures are a non-functional data-structuring mechanism, but the operations on I-structures are regulated so that determinacy is preserved. The behaviour of I-structures is analogous to the behaviour of first-order terms in a logic programming language such as Prolog.

The language Id has been running for many years, first on simulated dataflow machines and more recently on a real multi-processor emulation of a dataflow machine. Id/83s was a first cut at a major redesign of the language based on that experience (July 1985). Id Nouveau, the language described in this document, is roughly the same as Id/83s-- but it is now accompanied by an abstract operational semantics, and we have a new understanding of compilation techniques for all its constructs [14].

There are two documents that may be considered companions to this one. "I-Structures" [4] concentrates on the basic data-structuring mechanism in Id, comparing it with structuring mechanisms in other imperative and functional languages. "Dataflow Graphs from Id Nouveau" [14] describes how to compile Id Nouveau programs into Dataflow Graphs. These documents together supercede [12].

The MIT Tagged-Token Dataflow Machine, the programming language Id Nouveau and Dataflow Graphs reflect our current thinking on architectures, languages and compiling techniques for parallel machines. This is a very active and exciting area of research and so it would not be surprising to see many changes as our understanding of these issues deepens.

1.1. Document Outline

In Section 2 we present the subset of Id Nouveau that is purely functional (*i.e.* without I-structures). It is a fairly typical functional language.

In Section 3 we introduce I-structures. This takes Id out of the class of purely functional languages-- I-structures have similarities to so-called "logic variables" of languages such as Prolog. We revisit some of the affected language constructs as necessary, and discuss the motivations for and implications of introducing such a non-functional feature.

In Section 4 we describe an operational semantics-- based on rewrite rules-- for a simple subset of Id Nouveau called Id Kernel. The kernel language is chosen to facilitate the exposition of the semantics without losing any of the power of the full language. In Section 4.4, we describe the translation of an Id Nouveau program into Id Kernel.

Appendix A contains numerous example Id Nouveau programs. Appendix B contains the collected syntax and semantics of Id Kernel.

Syntax Conventions: In the following, we use the usual syntax notations of "{..}" to denote 0 or more occurrences of a construct, "[..]" to denote 0 or 1 occurrences of a construct (*i.e.* optional items).

Acknowledgements: Vinod Kathail has inspired much of the evolution of Id to Id Nouveau. In particular, the sections on I-structures and rewrite rule semantics owe much to the insights he gathered while writing the Id compiler.

2. Functional Id Nouveau(*or* Id Nouveau without I-structures)

An Id Nouveau program is *a.* a collection of *definitions* and *b.* an *expression* (also called the "query"). Syntactically, a definition gives a name to an expression. The named expression may optionally be parameterized. Here is an example of an Id program with four definitions and a query:

```
def pi = 3.14159 ;

def power x y = if (x = 0) then 1
                else y * (power (x - 1) y) ;

def square = power 2 ;

def circarea r = pi * (square r) ;

circarea 14.5
```

We see some similarities and some striking differences from programs in conventional languages. Like most languages, some definitions are clearly recognizable as defining procedures (e.g. **power**, **circarea**) because of the syntactic presence of formal parameters. Unlike some languages, however, definitions need not define procedures at all (e.g. **pi**)! And again unlike most languages, some definitions define procedures even though there are no formal parameters to signal this (e.g. **square**).

An interesting point is that procedures like **power** take their arguments one at a time, which is why we were able to say **power 2**, supplying **2** as the value for **power**'s formal parameter **x**, but not yet supplying any value for the formal parameter **y**. Such procedures are called "Curried" procedures, after H.B.Curry, a prominent logician who studied such procedures (more about this in Section 2.9).

These nuances are explained uniformly in functional languages by the fact that procedures are themselves *values* and can be computed as results of expressions. Thus, just as the expressions **3.14159** and **2 * 1.570795** compute a particular floating-point value (a point in the space of floating-point numbers), the expression **power 2** computes a procedure value: that point in procedure space that is *the* procedure that squares numbers.

Thus semantically speaking, all definitions uniformly bind names to *values*. Every expression (in definition bodies or the query) may refer to any of the defined identifiers. In other words, the definitions may be mutually recursive.

A program thus represents a value to be computed -- the answer to the query. The definitions are a convenient way to name values that are useful in computing the value of the program (later we shall see that they also are useful in implementations).

Dijkstra-like-comment

In mathematics, the concept of a "function" is very clear. It is a value that can be applied to an input value to produce an output value; it *always* produces the same output value for a given input value (*i.e.* the output value depends only on the input value); and it has *no* meaning other than this mapping.

In most programming languages, this terminology is abused-- things called functions are not really functions in the mathematical sense. First, the output value depends not only on the input value, but also on the "state" of the computation at the time of application. In addition to producing an output value, they may also have side-effects, *i.e.* change the state of the computation.

In Functional Id Nouveau, what we are calling "procedures" are indeed "functions" in the mathematical sense. The reason we call them procedures instead of functions is that later, when we introduce I-structures, they will have additional effects and cannot be identified with mathematical functions anymore.

tncmmoc-ckil-artskjiD

Expressions are central to Functional Languages, and we shall now examine the various forms they may take.

2.1. Numbers, Strings, Booleans and nil

We follow the usual syntax for numbers -- a sequence of digits with an optional decimal point.

For strings, we allow any sequence of characters enclosed in double-quotes. For example:

```
"God was dead"
```

There are two Boolean constants, written `true` and `false`.

There is a distinguished value called `nil` different from every other value. It is most useful in building finite recursive structures such as lists and trees.

2.2. Identifiers, or Names

We follow the usual programming language rules for identifiers, *i.e.* a letter followed by zero or more letters, digits, underscores, question-marks, etc. We do not distinguish lower- and upper-case letters. For example:

```
x
x1
j
cosine
Max_Salary
average_hiccup
Good_Guess?
```

2.3. Grouping with Parentheses

When one wishes to make clear the textual extent of a construct (either for legibility or to override certain defaults), one can always surround it with parentheses. For example:

```
3 * 4 + 5
(3 * 4 + 5)
((3 * 4) + 5)
(((3) * 4) + 5)
```

are all equivalent expressions (assuming that * has a higher operator precedence than +), and none of them are equivalent to:

```
3 * (4 + 5)
```

This is the only rule for use of parentheses in the language.

2.4. Applications

Applications[1] are expressions written by merely juxtaposing the two parts: a function-part on the left and an argument-part on the right. Some examples:

```
sine 2.4
cosine pi
square 2.45
power 2
```

The application notation associates to the left. For example,

```
power 2 4
(power 2) 4
```

are equivalent expressions. This left-association may be overridden with parentheses, as in

```
square (square 2.45)
cosine (2 * pi)
```

2.5. Conditionals

The conditional construct is also an expression, *i.e.* it represents a value. The syntax is:

if expr$_1$ *then* expr$_2$ *else* expr$_3$

expr$_1$ should evaluate to a Boolean value. If true, the value of the entire conditional expression is the value of expr$_2$. If false, the value is that of expr$_3$.

The conditional can be viewed as special syntax for the application of a function `op_if` to 3 arguments:

[1] Also called *Combinations*.

op_if $\texttt{expr}_1$ $\texttt{expr}_2$ $\texttt{expr}_3$

2.6. Infix Operators

Infix operators are special identifiers. In addition to denoting a value (as all identifiers), they also play a *syntactic* role. For every infix operator **x**, let us assume another identifier **op_x** that denotes the same value, but is devoid of any special syntactic role. Then the notation

$\texttt{expr}_1$ x $\texttt{expr}_2$

can be considered syntactic sugar for

op_x $\texttt{expr}_1$ $\texttt{expr}_2$

i.e. infix operators always denote functions of two arguments, and are just a syntactic convenience[2]. Of course, one must be wary of the usual pitfalls with infix operators: *precedence* and *associativity*. If we wish to override defaults or if we lose our nerve, we use parentheses for explicit grouping.

Another example:

```
3 + 4 * 5
```

is just another way of writing

```
op_+  3  (op_*  4  5)
```

2.7. Let-Blocks

A Let-Block is an expression with two parts: a set of Bindings, and a Return-Expression. First, a simple example:

```
let
   aSq = 3 * 3 ;
   bSq = 4 * 4
in
   sqrt (aSq + bSq)
```

Here, there are two Bindings[3], introducing identifiers **aSq** and **bSq** representing the values 9 and 16 respectively. The value of the Let-Block is the value of the Return-Expression: **sqrt (9 + 16)**, *i.e.* **5**.

Let-Blocks are like **begin...end** blocks in Algol60: the Bindings correspond to local variables declared in such a block. With this basic intuition, there are several complications that we will now explore.

[2] They are also used as hints for the compiler to generate better code.

[3] In previous versions of Id, the symbol "=" was written "←".

First, Let-Blocks can be nested, and follow the usual static scoping rules for identifiers that are redefined in inner Let-Blocks. For example:

```
let
   x = 2
in
   let
      y = 3 ;
      x = 4
   in
      x + y
```

is an expression representing the value 7. The x in the inner Return-Expression x + y represents the value bound in the *nearest* statically enclosing scope,, *i.e.* 4.

An identifier may be defined only once in the Bindings of a Let-Block. Thus the following block:

```
let
   x = 2 ;
   x = 3
in
   . . .
```

is meaningless.

Let-Blocks may be nested in many ways. For example:

```
let
   xQuad = let
                xSq = x * x
            in
                xSq * xSq
in
   xQuad * xQuad
```

In general, a Let-Block may appear anywhere that an expression may appear.

The values bound in a Let-Block may be arbitrary values, including procedure values. For example:

```
let
   square x = x * x
in
   sqrt (square 3 + square 4)
```

binds a local squaring procedure. This is analogous to the Pascal facility for defining local procedures.

The Bindings in a Block may be recursive and mutually recursive. For example:

```
let
  pi = 3.14159 ;

  power x y = if (x = 0) then 1
              else x * (power (x - 1) y) ;

  square = power 2 ;

  circarea r = pi * (square r)
in
  circarea 14.5
```

from which it should be evident that the top-level of an Id program is just different syntax for a Let-Block![4]

The textual ordering of the Bindings in a Let-Block is unimportant. The only ordering, if any, arises out of data-dependencies, i.e. the binding of an identifier *vs.* the use of that identifier. So, the above block could also have been written:

```
let
  square = power 2 ;

  power x y = if (x = 0) then 1
              else x * (power (x - 1) y) ;

  circarea r = pi * (square r) ;

  pi = 3.14159
in
  circarea 14.5
```

which is just a textual re-ordering of the bindings.

2.7.1. Renaming Identifers in Blocks

Identifiers introduced via bindings in a block are sometimes called "dummy identifiers" because they can be renamed uniformly without changing the meaning of the program. Consider the following program.

```
let
  x = 2 ;
  y = 3 ;
  z = let
        x = 4
      in
        x + y
in
  x + y + z
```

[4] The only reason we use different syntax at the top level is to facilitate interactive program development where we can edit and compile individual top-level definitions in isolation.

Using standard "static scoping" rules, we see that **x** in the inner Return-Expression represents the value 4 whereas **x** in the outer Return-Expression represents the value 2. The **y**'s in both Return-Expressions represent the value 3. To avoid worrying about scope rules, we can uniformly rename identifiers so that a given identifer has exactly one binding in the entire program. For instance, we can rename the identifiers above as follows:

```
let
  x1 = 2 ;
  y1 = 3 ;
  z1 = let
         x2 = 4
       in
         x2 + y1
in
  x1 + y1 + z1
```

Each identifier now has exactly one binding in the entire program. While the meaning of this program is identical to that of the previous one, it is a lot more transparent to the reader. From now on, we take for granted the transformation:

```
rename : Source-Program → Source-Program
```

which renames the identifiers in the program to unique names so that each identifier has exactly one binding.

2.8. Tuples

Tuples are a basic data-structuring mechanism in the language : they allow us to build compound objects. The primary means of constructing an *n*-tuple is to write *n* expressions separated by commas. For example:

```
"One", 2, "Many",
```

is a 3-tuple with three components: a string, a number and a string.

Arbitrary expressions may be placed in a tuple-construction. Thus

```
2 + 3, square 2.5, ("a gauche", "a droit"), power 2
```

is an expression representing a 4-tuple containing the number 5, the number 6.25, a 2-tuple containing two strings, and a procedure that is the squaring function.

Components of tuples are selected by binding names to them, using tuple-structured bindings. For example,

```
let
  x, y = 23, 45 ;

  a, (b, c), d = e
in
  ...
```

In the first Binding, the right-hand side is an expression that represents a 2-tuple; the binding then associates the names x and y with the first and second components of that tuple, *i.e.* it associates x with the value 23 and y with the value 45.

In the second Binding, the right-hand side e should be an expression which represents a 3-tuple whose second component is itself a 2-tuple. The binding then accociates a with the first component, b with the first component of the second component, c with the second component of the second component, and d with the third component.

The left-hand side of a tuple-structured binding can also be viewed as a "pattern" describing the structure of the value of the right-hand side, and the names that should be given to the components of that structure.

Note that our syntax does not allow us to build 1-tuples! For completeness, we could imagine a built-in function mk-1-tuple that takes any object and returns a 1-tuple containing that object, and a built-in function select-1-1 that selects the single component of a 1-tuple. But 1-tuples are never used, and so these functions are unnecessary.

2.8.1. Simplifying Tuple-structured Bindings

Tuple-structured bindings may be considered as just a syntactic convenience. Consider this example again:

```
let
  x, y = 23, 45 ;

  a, (b, c), d = e
in
  ...
```

For every pair of integers i and j such that $1 \leq i \leq j$, let select-i-j be a built-in function that extracts the i^{th} component of a j-tuple. Then, the program above can be re-written as:

```
let
  p     = 23, 45 ;
  x     = select-1-2 p ;
  y     = select-2-2 p ;

  q     = f x ;
  a     = select-1-3 q ;
  (b,c) = select-2-3 q ;
  d     = select-3-3 q
in
  ...
```

which, in turn, can be written as:

```
let
  p     = 23, 45 ;
  x     = select-1-2 p ;
  y     = select-2-2 p ;

  q     = f x ;
  a     = select-1-3 q ;
  r     = select-2-3 q ;
  d     = select-3-3 q ;

  b     = select-1-2 r ;
  c     = select-2-2 r
in
  ...
```

which has no tuple-structured bindings. Here **p**, **q** and **r** are new identifiers.

Thus, in general, an arbitrary tuple-structured binding of the form

```
t_1,...,t_n = expr
```

may be replaced by a collection of bindings of the form

```
t_new = expr ;
t_1   = select-1-1 t_new ;
 ... ;
t_n   = select-n-n t_new
```

where $\mathbf{t}_{new}$ is a new, previously unused identifier. If any of the $\mathbf{t}_j$'s are themselves tuple-structures, we simply repeat the process on those bindings, until there are no tuple-structured bindings left. In fact, this is how a compiler handles tuple-structured bindings.

Analysis of the left- and right-hand sides of tuple-structured binding quite often enables a compiler to generate code that does not involve tuples at all. For example, in the expression discussed above, it could generate code for the binding

```
x, y = 23, 45 ;
...
```

as if it had been defined as

```
x = 23 ;
y = 45 ;
...
```

2.8.2. General Recursive Definitions

One final nuance. We are all used to seeing recursive *procedures* in modern programming languages; but recursive bindings, whether at the top level or deep inside a Block, need not define procedure values at all! For example

```
let
  x = "Lara", x
in
  select-1-2 x
```

binds **x** to:

```
a 2-tuple whose
first component is the string Lara, and
second component is
        a 2-tuple whose
        first component is the string Lara, and
        second component is
                a 2-tuple whose
                first component is the string Lara, and
                second component is
                          .
                          .
                          .
```

i.e. a 2-tuple nested infinitely to the right:

```
("Lara", ("Lara", ("Lara", ... )))
```

Whether we can actually *implement* such a beast is another question (and in fact we will show later that we can indeed!), but semantically there is no problem! The Return-Expression of the Block selects the first component of this tuple, which is the string **"Lara"**. And so the Block as a whole is a perfectly meaningful expression and represents the string **"Lara"**.

(Incidentally, **"Lara"** stands for "*L*ara's *a*nother *r*ecursive *a*cronym"; perhaps you already know about Eine, Fine, and Zwei? Gnu? Ah well ...)

2.9. Curried vs. Multi-Argument Procedures

We must be careful not to confuse Curried procedures with so-called "Multi-Argument" procedures. Thus the two definitions:

```
def max1 x y   = if (x > y) then x else y ;

def max2 (x,y) = if (x > y) then x else y ;
```

have identical bodies, but their parameterization makes them quite different procedures.

max1 is a Curried procedure. When applied to a numeric argument (**x**), it returns a *new* procedure; when that procedure in turn is applied to another numeric argument (**y**), it returns a number (the larger of **x** and **y**).

On the other hand, **max2** is a procedure which, when applied to an argument that is a 2-tuple of numbers (x,y), returns a number (the larger of x and y).

max2 may be viewed as syntactic sugar for

```
def max2 z = let
                 x,y = z
               in
                 if (x > y) then x else y ;
```

Here we have explicitly named the 2-tuple argument by z, and then separately bound x and y to the two components of z.

There is another uniform principle behind this: *All procedures take exactly one argument.* But that argument could be an arbitrarily complex object! Thus what we usually think of as an "*n*-argument" procedure is really a procedure that operates on a single *n*-tuple argument.

The difference between the two procedures is sometimes explained in this more formal way: they have different *types*:

```
max1 : number → (number → number)

max2 : (number × number) → number
```

Thus the *domains* of the two procedures are quite different. It makes sense to apply max1 to one numeric argument, but not to apply max2 to one numeric argument -- the latter simply is rejected as an attempt to apply the procedure to something outside its valid domain.

The following example using max1 demonstrates the use of Currying:

```
def clipBelow5 vec = forall (max1 5) vec ;
```

Here, clipBelow5 is a function that takes a vector of numbers vec as argument, and returns a vector of numbers as result which is an image of vec except that components of vec less than 5 have been "clipped" up to 5. The application of max1 to a single argument 5 returns a function which, when later applied to any number y, returns 5 or y, whichever is larger. The forall expression applies this function to each component of vec, returning the "clipped" image.

Procedures like max1 are also called **Higher-Order** procedures: they return procedures as results or take procedures as arguments. We also say that max1 has "arity" 2.

It may appear that there is some inefficiency involved in the use of functions like max2, *i.e.* the overhead of explicitly building an *n*-tuple just to pass arguments. However, a good compiler can generate code that elides this entirely. In sequential implementations, the caller may leave the tuple components at fixed offsets on a stack where they are accessed directly by the callee. In dataflow implementations, the caller may send the component tokens directly to separate receiving points in the callee. But it is important to realize that these are only optimizations.

2.10. Loops

Loops are expressions that encapsulate a certain common control structure. For example the following expression uses a loop to compute the 20th Fibonacci number:

```
let
  x,y = 1,1 ;
  j = 1
in
  while j <= 20 do
    new x, new y = y, x+y ;
    new j = j+1
  return x
```

This is just a convenient way of writing the following equivalent expression:

```
let
  fibloop (x,y,j) = if j <= 20 then
                       fibloop (y, x+y, j+1)
                     else
                       x
in
  fibloop (1, 1, 1)
```

`fibloop` is a classic example of a *tail-recursive* procedure, a special class of recursive procedures that are characterized by the property that the recursive call is the *last* activity performed by the body, *i.e.* the value returned by the recursive call is passed back untouched and unexamined to the caller. Tail-recursion corresponds exactly to *iteration*, and Loop expressions are a convenient way to express iterations (they also provide good optimization hints for implementation).

As in most conventional languages, in addition to unbounded iteration (While-Loops), we also have bounded iteration (For-Loops). For example:

```
let
  x,y = 1,1
in
  for j from 1 to 20 do
    new x, new y = y, x+y
  return x
```

is a more succinct way of writing the same program.

Each Loop has three parts: an Iteration Control, a Loop-Body and a Loop-Return-Expression.

The Iteration Control has one of the following three forms:

while e *do*

for j *from* e_1 *to* e_2 *by* e_3 *do*

for j *from* e_1 *downto* e_2 *by* e_3 *do*

In the **for-to** form, the loop index takes all values e_1, e_1+e_3, e_1+2*e_3, etc. as long as it is not greater than e_2. In the **for-downto** form, the loop index takes all values e_1, e_1-e_3, e_1-2*e_3, etc. as long as it is not less than e_2. The forms

for j *from* e_1 *to* e_2 *do*

for j *from* e_1 *downto* e_2 *do*

are abbreviations for the full forms where the step e_3 is 1.

The Loop-Body contains Bindings, similar to the Bindings in a Let-Block, except that the identifiers introduced in the left-hand sides of bindings may be prefixed by the keyword **new** (we call such bindings New-Bindings).

The Loop-Return-Expression of a Loop is evaluated after the last iteration (if any), and that value is the value of the entire loop.

Informally, the semantics of a Loop can be explained as follows. The identifiers in a Loop fall into three disjoint categories:

1. those bound in the Loop-Body and qualified by **new**,

2. those bound in the Loop-Body without the keyword **new**, and

3. all others.

Imagine the loop being unfolded (perhaps infinite times) into separate copies of the body, one for each iteration. Identifiers in category 3 represent values bound in the scope surrounding the entire loop and hold through all iterations; they are thus also known as "Loop Constants". Identifiers in category 2 represent values bound within the Loop-Body for a particular iteration, and their scope is just the Loop-Body for that iteration. Identifiers in category 1 are values bound in the Loop-Body during a particular iteration, and their scope is the *next* iteration of the loop.

More formally, the semantics of a Loop are explained by their correspondence to an equivalent expression without Loops; this equivalent expression will involve a tail-recursive procedure. We had an example of this in the Fibonacci progam above, but here is a systematic procedure to perform this transformation:

Suppose the loop has the general form:

```
for-or-while ... do
   ... Loop-Body ...
return ret-expr
```

Step 1: If the Iteration Control is a **for-to** iterator:

for j *from* e_1 *to* e_2 *by* e_3 *do*

convert it into a **while** iterator like this:

```
let
  j      = e₁ ;
  bound  = e₂ ;
  step   = e₃
in
  while j <= bound do
    ... Loop-Body ...
    new j = j + step
  return ret-expr
```

If the Iteration Control is a `for-downto` iterator:

```
for j from e₁ downto e₂ by e₃ do
```

convert it into a `while` iterator like this:

```
let
  j      = e₁ ;
  bound  = e₂ ;
  step   = e₃
in
  while j >= bound do
    ... Loop-Body ...
    new j = j - step
  return ret-expr
```

From now on, let's assume we're dealing only with while-loops of the form:

```
while do_next? do
    ... Loop-Body ...
return ret-expr
```

Step 2: Separate the Loop-Body into ordinary bindings and New-Bindings.

```
while do_next? do
    ... Ordinary-Bindings ... ;
    ... New-Bindings ...
return ret-expr
```

This may involve breaking up tuple-structured bindings into a collection of separate bindings as shown in Section 2.8.1.

Step 3: The while-loop can now be replaced by the following tail-recursive definition:

```
let
  loopfunction(...,x,...) = if do_next? then
                               let
                                 ... Ordinary-Bindings ...
                               in
                                 loopfunction(...,e,...)
                             else
                               ret-expr
in
  loopfunction(...,x,...)
```

where, for every New-Binding of the form

```
new x = e
```

in the Loop-Body,

- `loopfunction` should have a formal parameter `x`,

- the recursive call to `loopfunction` should have `e` as the corresponding actual parameter, and

- the initial call in the Return-Expression of the outer Let-Block should have `x` as the corresponding actual parameter.

The reader is invited to confirm that the scope rules for this transformed version of the loop are exactly those outlined informally earlier.

3. I-structures

We now introduce a new data-structuring mechanism called *I-structures.* Unlike tuples, I-structures are *not* functional; in fact, they are closer in spirit to first-order terms in a logic programming language such as Prolog. The introduction of I-structures complicates the language to some extent, and some nice properties of functional languages such as referential transparency are lost. The motivation for this departure is to permit more efficiency, avoiding the excessive copying that is often implied by purely functional operations on data structures. Nevertheless, the introduction of I-structures does not complicate the detection of parallelism in programs. Programs remain "determinate" in the sense that the output of a program does not depend on the order in which computations in the program are performed. A primitive form of I-structures was first proposed in [5].

3.1. I-Structure Operations

I-structures are similar to arrays in conventional programming languages. There are three basic I-structure operations: allocation, component assignment and component selection.

3.1.1. I-structure Allocation

An I-Structure is allocated by an expression of the form

```
array(l..u)
```

The operational meaning of this expression is: allocate an "empty" I-structure with the specified lower and upper index bounds, and return a pointer to it. The index-bounds expressions l and u may be arbitrary numeric expressions (with l $\leq$ u).

Note that this is an expression with a side-effect, *i.e.* every execution of this expression produces a *new* I-structure and returns a pointer to it. Thus the expression

```
let
  x = array(1..5)
in
  x,x
```

is not equivalent to the expression

```
array(1..5), array(1..5)
```

The former returns two copies of a pointer to one I-structure, whereas the latter returns two pointers to two distinct I-structures.

3.1.2. I-structure Component Assignment

A component of an I-Structure may be set to a value, *provided it is empty*, using the following familiar construct:

```
a[j] = expression
```

a must be an identifier bound to an I-structure[5] and j must be a numeric expression.

A component of an I-Structure can be assigned *at most once*; an attempt to assign twice to the same component of an I-structure results in a run-time error, and the program returns the value "error". This is one difference from the usual concept of arrays in imperative languages in which a component may be assigned any number of times. We will describe the motivation behind the "single-assignment" restriction in Section 3.3.

The assignment statement is our first example of a "Command", *i.e.* a construct executed only for its side-effect. It is different from, and may not be used as an "Expression".

3.1.3. I-structure Component Selection

A component of an I-Structure may be selected using the following familiar syntax:

```
    a[j]
```

This is an Expression, and represents the value stored in the designated component of the I-Structure. a is an I-structure-valued expression, and j is a numeric expression.

Because of concurrency, an execution of a program may attempt to evaluate an array-selection expression while the designated component is still empty, *i.e.* before it has been assigned a value by some other concurrently executing part of the program. In such a situation, the evaluation of this expression is deferred-- it merely waits until the component is assigned, after which it returns the value. If the component is *never* assigned by other parts of the progam, the evaluation of this selection expression hangs forever, and it can be considered to have an undefined value. This is a second difference from the usual concept of arrays.

3.1.4. I-structure Index Bounds

The index bounds of an I-structure may be queried using the built-in function **bounds** which returns a 2-tuple containing the lower and upper bounds. For example:

```
    let
      l,u = bounds x
    in
      x[(l+u) div 2]
```

is an expression that returns the middle element of an I-structure x, and

[5] In principle, a could be an *expression* that evaluates to an I-structure, but we restrict it to be an identifier to simplify parsing. There is no loss in generality, since any assignment of the form

```
    expression[j] = ...
```

can always be written as

```
    a = expression ;
    a[j] = ...
```

to meet our syntactic restriction.

```
def upper x = let
                  l,u = bounds x
              in
                  u ;
```

defines a function that returns the upper index bound of an I-structure.

3.1.5. Nested I-structures

The components of an I-structure can be any values; thus multi-dimensional arrays can be modelled by I-structures whose components are themselves I-structures. For example,

```
let
  x = array(1..5)
in
  for j from 1 to 5 do
    x[j] = array(1..5)
  return x
```

allocates an empty 5-by-5 matrix. This common construction may be abbreviated

```
array(1..5, 1..5)
```

(with generalizations to any number of dimensions and any index bounds).

The i,j^{th} component of this matrix can be assigned using

```
y = x[i] ;
y[j] = expression
```

The first line binds y to the i^{th} component of x which is a 5-array, and the second line assigns to the j^{th} location of this array.[6] This common construction may be abbreviated

```
x[i,j] = expression
```

The i,j^{th} component of the matrix can be selected using the expression

```
x[i][j]
```

i.e.

```
(x[i])[j]
```

which may be abbreviated

```
x[i,j]
```

[6] If it were not for our syntactic restriction that the I-structure reference on the left-side of an assignment statement must be an identifier and not an expression, we could have expressed this in one line as

```
x[i][j] = expression
```

In both the above abbreviations, the "i,j" is only a syntactic shorthand, and may *not* be replaced by an arbitrary 2-tuple expression.

3.2. Using I-structures in Programs

Given these primitive I-Structure operations, let us examine how they are used in Id Nouveau.

3.2.1. Expressions Revisited

It is important to keep in mind that Expressions, which previously represented only *values*, now may also have side-effects.

We call Expressions that have no side-effects "Pure Expressions".

3.2.2. Let-Blocks Revisited

So far, Let-Blocks contained only Bindings and a Return-Expression. We now also allow Commands to appear in Let-Blocks. For example,

```
let
  prime = array(1..5) ;
  prime[1] = 2 ;
  prime[2] = 3 ;
  prime[3] = 5 ;
  prime[4] = 7 ;
  prime[5] = 11
in
  prime
```

contains one Binding, five Commands, and a Return-Expression. As before, the textual ordering is unimportant; all components may be executed in parallel.

There is an important difference to keep in mind between the following two similar-looking expressions:

```
let                                   let
  x = 5                                 y[1] = 5
in                                    in
  x                                     y
```

The former contains a Binding, *i.e.* it introduces a new identifier x that hides any x extant in the current scope. The x in the Return-Expression refers to this locally-introduced x.

The latter contains a Command, and does *not* introduce a new identifier; the y must already be extant in the current scope, bound to an I-structure, and all it does is to assign a value into this existing I-structure. The reader should note that the "effect" of the component assignment is not restricted to the extent of the *let* - thus, the expression y[1] will now return the value 5 even in an outer scope.

Sometimes we wish to use a Let-Block *only* for the side-effects in its Bindings, and we are not interested in any value from the Return-Expression. We express this using the special symbol "()". For example:

```
let
  prime[1] = 2 ;
  prime[2] = 3 ;
  prime[3] = 5 ;
  prime[4] = 7 ;
  prime[5] = 11
in
  ()
```

Such Let-Blocks are called "Command-Blocks".

3.2.3. Procedures Revisited

We have already seen how to define a Procedure by abstracting over an expression, parameterizing it with respect to some of the identifiers occurring in it. So far, procedures were functions in the mathematical sense, because expressions only represented values and had no side-effects,

Since expressions can now also have side-effects, procedures built out of expresssions can no longer be regarded as pure functions, *i.e.* the *application* of a procedure to its arguments not only represents a value but may also have side-effects.

At the extreme, if we abstract over a Command, parameterizing it with respect to some of the identifiers occurring in it, we get a Procedure that represents *only* side-effects. For example,

```
def zero_diag_5 a = let
                       a[1,1] = 0 ;
                       a[2,2] = 0 ;
                       a[3,3] = 0 ;
                       a[4,4] = 0 ;
                       a[5,5] = 0
                    in
                       () ;
```

is a procedure executed *only* for its side-effects: it zeroes the diagonal of its 5-by-5 matrix argument.

Such procedures are called Command Procedures. An application of such a procedure is itself a Command. For example:

```
let
  x = array(1..5, 1..5) ;
  = zero_diag_5 x
in
  x
```

uses a procedure-call to perform the side-effect of zeroing the diagonal of I-structure **x**.

In general, any expression **e** may be treated as a Command and executed purely for its side-effect, as follows:

```
let
   ... ;
   = e ;
   ...
in
   ...
```

which can be viewed as syntactic shorthand for

```
let
   ... ;
   dummy = e ;
   ...
in
   ...
```

where dummy is an identifier not used anywhere in the program.

3.2.4. Loops Revisited

A Loop-Body, in addition to containing Bindings and New-Bindings, may also contain Commands (and, as indicated earlier, Expressions treated as Commands). Here is an expression that defines and returns an array of the first twenty Fibonacci numbers:

```
let
  fib_20 = array(1..20) ;
  x,y = 1,1
in
  for j from 1 to 20 do
     fib_20[j] = x ;
     new x, new y = y, x+y
  return fib_20
```

We can also write Loops that are Commands. For example:

```
for j from 1 to upper(a) do                      -- a is a square matrix [1..n,1..n]
   a[j,j] = 0
return ()
```

is a Command-Loop that zeroes the diagonal of a square matrix a.

3.3. Discussion[7]

To understand I-structures better, it is useful to contrast them with functional data-structures such as tuples or functional arrays. A tuple is defined by writing down the expressions whose values will be its components. Thus, the entire value of a tuple is defined "in one place" in the program. In contrast, the allocation of an I-structure is separate from the operations of filling in its components. For example, an I-structure may be allocated in one procedure and then passed to a

[7] For a detailed exposition on I-structures, the reader is invited to read [4].

number of other procedures each of which is responsible for filling in one of its components. Thus functional data-structures are completely defined "all at once", whereas I-structures begin as "empty" structures and are then incrementally refined.

3.3.1. Efficiency

I-structures allow us to avoid the excessive copying often implied by purely functional operations on data-structures. To illustrate this, consider the following problem. Suppose we have an array **x** with index bounds 1 to 10, and we wish to produce an array **y** that is a "shifted" version of **x**, *i.e.* **y[1]=x[2]**, **y[2]=x[3]**, ..., **y[9]=x[10]**, **y[10]=x[10]**.

A functional array assignment is an expression of the form

```
fassign a i v
```

which returns a *new* array containing the same components as array **a**, except at index **i** where it contains value **v**. Note that **fassign** conceptually copies all but the **i**th component of **a** into the new array.[8]

We can now write a purely functional program to solve our problem:

```
for j from 1 to 9 do
   new x = fassign x j x[j+1]
return x
```

Note that this program allocates 9 arrays, only the last of which is the result array. The remaining 8 arrays are intermediate, "temporary" arrays existing only during loop execution. Each component of the final array arrives there only after it has been copied 9 times through the intermediate arrays. In each iteration, there is only one "useful" component assignment, and nine "unnecessary" copies of the other components. Each iteration is responsible for defining only one component of the final array. What we would like is for each iteration to write its component value directly into the final array, independent of the other iterations. The final array would thus begin as a collection of 10 empty slots, which are then filled incrementally by 10 independent activities.

It was in recognition of this need for data-structures that can be directly and incrementally filled that a primitive version of I-structures was first proposed in [5]. The current syntax and semantics of I-structures reflects considerable refinement of that early proposal.[9]

With I-structures, we would write the program as follows:

[8] This may be alleviated somewhat by tree-structured representations of arrays.

[9] A refinement that owes much to Vinod Kathail.

```
let
  y = array(1..10) ;
  for j from 1 to 9 do
    y[j] = x[j+1]
  return ()     ;
  y[10] = x[10]
in
  y
```

which allocates only one array (the result array), copies each result component exactly once, and potentially performs all the assignments in parallel.

One may be tempted to refute such examples as follows: we first extend the language to include functional array operators that are more powerful and complex than the simple `fassign` shown above (for example, the languages APL and TALE [6] have such operators). We then try to make our compiler clever enough to generate code for such operators that matches the storage efficiency and parallelism of the I-structure solution.

For example, suppose (following the language TALE) we had a "tabulation" function to create new arrays:

```
tabulate (l,u) f
```

which creates a new array with bounds `l..u` and components `f(l)`, `f(l+1)`, ..., `f(u)` respectively. A program to solve our problem is then:

```
let
  f j = if j < 10 then x[j+1]
        else x[10]
in
  tabulate (1,10) f
```

A clever compiler may be able generate code that allocates only one array (the result) and, in parallel, executes `f(1)`, `f(2)`, ..., `f(10)`, and writes all the components. (There is still some inefficiency in that each activity `f(j)` executes a conditional.)

But it is as yet unclear whether this general approach is feasible-- *i.e.* is there a repertoire of array operations and associated compilation techniques that will always produce code as efficient as the I-structure solution? Furthermore, while functional programs with such operators retain referential transparency, they are sometimes not very perspicuous (APL one-liners are functional, but can be *very* hard to read!).

It is evident that we can implement operators such as `tabulate` very efficiently in Id Nouveau using I-structures, so that the programmer can choose to stay within the functional paradigm as long as it is convenient. On the other hand, it is clear that even if I-structures are omitted from the language, the architecture must support something like I-structures if the compiler is to generate efficient code for operators like `tabulate`.

3.3.2. Determinacy

The separation of data-structure allocation from component assignment cannot be done indiscriminately, as in imperative languages. Since a single I-structure **x** may be referred to by several concurrently executing parts of the program, this raises two threats to the determinacy of the program:

1. One part of the program may try to select x[1] before some other part, responsible for assigning x[1], has completed. This is a race condition-- without some form of synchronization, the select operation may read garbage from that location. The "deferred select" rule is exactly such a synchronization mechanism-- the selection always returns the same, valid value.

2. Two (or more) parts of the program may attempt to assign x[1] with different values. Consider the following (concocted) program:

```
let
   x = array(1..10) ;
   x[1] = 2 ;
   x[1] = 3 ;
   y = x[1]
in
   y
```

Depending on what order the assignments and the selection are done, **y** may receive the value 2 or 3. The "single-assignment" rule ensures that such programs are in error-- if multiple writes ever occur in any component of an I-structure, the entire program aborts, returning the result "error".

3.3.3. Implementation

While a full understanding of how I-structures are implemented requires some knowledge of dataflow, we provide the following overview for the curious reader. I-structures are implemented in *I-structure storage* which is a tagged memory with a smart memory controller [8]. An I-structure with bounds 1..u is implemented as (u-1+1) contiguous locations in I-structure memory. Each location has a tag consisting of status bits to indicate that it is in one of three possible states :

- ALLOCATED : no data is present in the location and no attempt has been made to read from it. The location may be written to as in a conventional memory.

- PRESENT : the location contains data which can be read as in a conventional memory. An attempt to write new data into this location is signalled as an error.

- READ PENDING : no data is present in the location but at least one attempt has been made to read its contents. When (and if) the location is written to, all reads waiting for this value are notified.

There are various ways to implement this. In the dataflow computer, each select request comes with a "destination address" where the result value is to be sent. For each such request that arrives before the location is written, the destination address is simply appended to a list associated with the location in a special portion of I-structure memory. When a value gets written into that location, a copy of the value is forwarded to all destinations on that list.

The I-structure memory controller is responsible for the allocation of I-structures, and it changes the states of locations in the obvious way when presented with read and write requests.

4. A Rewrite-Rule Operational Semantics for Id Nouveau

In this section we describe a simple operational semantics for Id Nouveau. We use an abstract "Rewriting Model"-- Dataflow and Graph-Reduction Machines may be regarded as lower-level implementations of this abstract model. Id Nouveau has been implemented on the Tagged-Token Dataflow machine [14] and this implementation is "congruent" with the semantics given here.

The operational semantics of Id Nouveau are described in terms of a subset of the language called *Id Kernel*. While Id Kernel does not have loops, nested procedure definitions and some other "features" of Id Nouveau, any Id Nouveau program can be easily translated into an Id Kernel program-- this simple transformation is given in Section 4.4. The meaning of Id Kernel programs is given by rewrite rules. Thus, the meaning of an Id Nouveau program can be determined by first transforming it to a corresponding Id Kernel program, and then applying the rewrite rules for Id Kernel. While this two-step process of giving the semantics may seem a little indirect, it enables us to concentrate on the essentials of the rewrite semantics, and to ignore features of Id Nouveau which are there mainly for programming convenience.

To introduce the reader gently to rewrite rules, we first give the semantics of a simple functional subset of Id Nouveau, *i.e.* a subset without I-structures.

4.1. Operational Semantics for a Functional Subset of Id Nouveau

4.1.1. The Language

The functional subset that we begin with is very simple. As before, programs have the form:

$$\begin{array}{l}
\textit{def}\;\; f_1\; x_{1_1}\; \ldots\; x_{1_n} \;=\; e_1 \;; \\
\ldots \\
\textit{def}\;\; f_m\; x_{m_1}\; \ldots\; x_{m_n} \;=\; e_m \;; \\
e_0
\end{array}$$

but each expression e_0, e_1 ... , e_m has a certain "simplified" form, recursively defined as follows:

atoms : constants and identifiers
1.2, 3, *, "fony", phud, ...

conditionals
if e_c *then* e_t *else* e_e

applications
$e_f\; e_a$

As before, free identifiers in e_0 must be from the set $f_1, \ldots, f_m$, and free identifiers in each of the other expressions e_j must be from the set $f_1, \ldots, f_m, x_{j_1}, \ldots, x_{j_n}$.

The circarea program discussed in Section 2 is a program in this language. It is reproduced below.

```
def pi = 3.14159 ;

def power x y = if (x = 0) then 1
                else y * (power (x - 1) y) ;

def square = power 2 ;

def circarea r = pi * (square r) ;

circarea 14.5
```

4.1.2. Reduction Rules and Redexes

The query expression in a program in some sense "represents" the answer to be computed. Reduction rules can be regarded as a method to compute that answer-- they are simplification rules that specify how to repeatedly rewrite the query expression until it *becomes* the result of the program.

For each built-in operator, we imagine a (possibly infinite) set of reduction rules that defines its behavior. For example,

```
0 + 0 ⇒ 0
0 + 1 ⇒ 1
.
.
523 + 1062 ⇒ 1585
.
.
16 * 3 ⇒ 48
.
.
if true   then e_t  else e_e ⇒ e_t
if false  then e_t  else e_e ⇒ e_e
.
.
```

Each of the top-level definitions in the program can also be regarded as a Reduction Rule. For example,

```
def power x y = if (x = 0) then 1
                else y * (power (x - 1) y) ;
```

can be regarded as the reduction rule

```
power x y ⇒ if (x = 0) then 1
            else y * (power (x - 1) y)
```

The right-hand side of such a rule is also called the "body" of the rule.

A *Redex* (for *red*ucible *ex*pression) is an expression that matches the left-hand side of a reduction rule. Parameter identifiers on the left-hand side of a reduction rule are "wild-cards" which match any expression. For example, the expression `(circarea 14.5)` matches the left-hand side of the reduction rule

```
circarea r ⇒ pi * (square r)
```

with r matching the expression 14.5.

A redex may be *Reduced* by replacing it with the corresponding right-hand side of the rule, with each parameter identifier replaced by the expression it matched. For example, the redex (circarea 14.5) may be replaced by the expression

```
pi * (square 14.5)
```

(Sometimes the resulting expression may be larger than the original, so the use of word "Reduction" is admittedly strange!)

4.1.3. Computation, Answers and Normal Forms

To compute with a program, we begin with the query expression and repeatedly rewrite it by reducing redexes contained in it. Hopefully, this process terminates with the expression in some kind of "simplified" form. At this point we say that we have computed the answer.

There are various choices for what this "simplified" form for answers should be. A typical choice is for the expression to be in its so-called *Normal Form.* An expression is in *Normal Form* if it contains no redexes, *i.e.* neither it nor any of its subexpressions is a redex.

We urge the reader to work through the following example to get a feel for computation using reduction rules. In each line, we have underlined the redex to be reduced next.

```
    circarea 14.5
    -------------
⇒
    pi * (square 14.5)
    --   -------------
⇒
    3.14159 * (power 2 14.5)
              --------------
⇒
    3.14159 * (if (2 = 0) then 1
                  -------
               else 14.5 * (power (2 - 1) 14.5))
⇒
    3.14159 * (if false then 1
               ----------------
               else 14.5 * (power (2 - 1) 14.5))
               ----------------------------------
⇒
    3.14159 * (14.5 * (power (2 - 1) 14.5))
                             -------
⇒
    3.14159 * (14.5 * (power 1 14.5))
                      --------------
```

```
⇒
    3.14159 * (14.5 * (if (1 = 0) then 1
                          -------
                       else 14.5 * (power (1 - 1) 14.5)))
⇒
    3.14159 * (14.5 * (if false then 1
                      ----------------
                       else 14.5 * (power (1 - 1) 14.5)))
                       ---------------------------------
⇒
    3.14159 * (14.5 * (14.5 * (power (1 - 1) 14.5)))
                                      -------
⇒
    3.14159 * (14.5 * (14.5 * (power 0 14.5)))
                              --------------
⇒
    3.14159 * (14.5 * (14.5 * (if (0 = 0) then 1
                                  -------
                               else 14.5 * (power (0 - 1) 14.5))))
⇒
    3.14159 * (14.5 * (14.5 * (if true then 1
                              ---------------
                               else 14.5 * (power (0 - 1) 14.5))))
                               ---------------------------------
⇒
    3.14159 * (14.5 * (14.5 * 1))
                      ----------
⇒
    3.14159 * (14.5 * 14.5)
              -------------
⇒
    3.14159 * 210.25
    ----------------
⇒
    660.51935
```

4.1.4. Computation Rules, Determinacy, Termination and Efficiency

One subtle point is that an expression may have more than one redex in it. For example, in the expression `(power (2 - 1) 14.5)`, there are two redexes: the sub-expression `(2 - 1)` which can be reduced to `1`, and the entire expression itself, which can be reduced to the expression

```
if ((2 - 1) = 0) then 1
else 14.5* (power ((2 - 1) -1) 14.5)
```

by matching `x` with `(2 - 1)` and `y` with `14.5` in the reduction rule for `power`.

Thus, we have left unspecified the question of *which* redexes of the expression are to be reduced at each step. Any rule that specifies this choice is called a "Computation Rule". For example, we might specify that at each step, the "leftmost innermost" redex must be reduced.

But this raises the following important question: does the choice of computation rule affect the answer computed for a given program? One nice feature of functional languages is that they are

Determinate-- normal forms are independent of the computation rule, *i.e.* the answer for a given program is unique, and any computation rule that produces an answer will produce that answer. This property of functional languages is also called the Church-Rosser property, and is what makes them attractive programming languages for parallel machines-- the exact schedule chosen for parallel activities does not affect the answer.

On the other hand, computation rules may differ with respect to

1. *termination*: for some expressions, one computation rule may lead to a normal form while another computation rule may not

2. *efficiency*: the number of reductions performed to reach the normal form.

3. *parallelism*: the number of redexes which are reduced at each step

These differences are illustrated by the so-called "applicative-order" and "normal-order" computation rules. An applicative-order interpreter always chooses the leftmost-innermost redex for reduction: thus, it evaluates the arguments of a function before invoking the function itself. On the other hand, a normal-order interpreter always selects the leftmost redex for reduction: thus, it invokes a function before evaluating its arguments. Applicative order interpreters also give special treatment to the conditional operator-- they do not reduce any redex that is in either arm of a conditional (thus the conditional expression must be reduced to one of the arms first, before any redex in that arm is reduced).

Termination: It can be shown that a normal-order interpreter is *safe* in the sense that it always terminates whenever *any* rule of computation does so. An applicative-order interpreter is not safe; consider this example:

```
def runaway x = runaway x

def power x y = if (x = 0) then 1
                else y * (power (x - 1) y) ;

power 0 (runaway 3)
```

A normal-order interpreter would terminate with answer `1` because it would not even attempt to reduce `(runaway 3)`. On the other hand, an applicative-order interpreter would keep reducing `(runaway 3)` (to itself) because it is the innermost redex, and thus would never terminate.

Efficiency: The two interpreters differ with respect to efficiency as well. A normal-order interpreter may evaluate the arguments of a function-application many times over. Consider the expression

```
power (2 - 1) 3
```

A normal-order interpreter would substitute the expression `(2 - 1)` twice (for the two occurrences of the parameter identifier `x` in the body of `power`) and thus reduce it twice. On the other hand, an applicative-order interpreter would reduce it to `1` first before substituting for `x`, and thus evaluate it just once.

In Section 4.2 we will see that when we include I-structures, it becomes important to control precisely how many times a redex is reduced.

Parallelism: Both interpreters are completely sequential in the sense that they reduce exactly one redex at each step.

4.2. Identifier-Bindings and Expressions When There Are Side Effects

In Id Nouveau, a variable identifier may be bound to an expression and used multiple times. This occurs in two situations. First, in a situation like this:

```
def makepair x = (x, x) ;

makepair expression
```

x is bound to expression and is used twice in the body of makepair. Second, in a situation like this:

```
let
  x = expression
in
  (x, x)
```

Again, x is bound to expression and is used twice in the body of the Let-Block. These two situations are actually semantically equivalent.[10] In both cases, the semantic intent is that x denotes the *value* represented by expression and that the x's in the bodies may be replaced by denotations of this value.

In a purely functional language, an expression represents a value and nothing else. Therefore, it makes sense to replace each x by the expression it is bound to. Thus, given the expression

```
let
  x = (2 + 3)
in
  (x, x)
```

it makes sense to replace it by

```
((2 + 3), (2 + 3))
```

The fact that the expression (2 + 3) may be evaluated twice is certainly a concern about efficiency, but it does not change the meaning of the expression.

In summary: *In a functional language, an identifier may freely be substituted by its binding expression.*

[10] This is in fact Landin's famous "Principle of Correspondence".

When we extend our language to include constructs that have side-effects (such as I-structures operations), our view of expressions and bindings must change. There are now *two* important aspects about an expression-- the side-effect it performs *and* the value it represents. For example, the expression `array(1..10)` has a side-effect (allocating a new array in storage) and a value (the pointer to that array). When an identifier is bound to an expression, it represents only the *value* denoted by the expression, and not the side-effect. Thus it no longer makes sense to replace an identifier by its binding expression-- it can only be replaced by a denotation of the *value* of its binding expression.

For example, the expression

```
let
  x = array(2..5)
in
  (x, x)
```

is *not* equivalent to the expression

```
(array(2..5), (array(2..5))
```

In the former: we allocate *one* array in storage; x is bound to the value of the expression, *i.e.* the pointer to this array; and the entire expression returns two pointers to this single array. In the latter: we allocate *two* arrays in storage; and the entire expression returns two pointers to these two separate arrays. The meanings of the two expressions are thus quite different.

In summary: *When expressions can have side-effects, an identifier may not be substituted by its binding expression, but only by a denotation of the value represented by its binding expression.*

Our rewrite rules must take this into account, *i.e.* when an identifier is bound to an expression, the rewrite rules must separate the value and the side-effect of the expression, after which the value part can be substituted for the identifier wherever it is used.

But this seems to imply a certain restriction on parallelism. Consider a rewrite rule and application

```
f x ⇒ ... x ... x ...

f e
```

where e has a side-effect. We cannot reduce the application before evaluating e, because we must first discharge the side-effect of e and rewrite it to a denotation of its value, and only then may we substitute this value for the multiple occurrences of x on the right-hand side. In other words, the possibility that argument expressions have side-effects seems to imply that we cannot invoke a procedure until we have evaluated its arguments.

But consider the following procedure and its application:

```
def f x = let
              a    = array(1..2) ;
              a[1] = x ;
              a[2] = x
          in
              a ;

f array(1..10)
```

We should not have to wait for the evaluation of **array(1..10)** before we begin executing the body of **f** -- we can go ahead and allocate the array **a**, and return the pointer to **a** immediately.

To allow this kind of parallelism in our rewrite rules, we must separate the reduction of an application from the substitution of formal-parameter identifiers by actual values. In a pre-processing step, we first replace an application of the form

$$e_f\ e_1\ e_2\ \ldots\ e_n$$

with the form

```
let
   dummy_1 = e_1 ;
   dummy_2 = e_2 ;
   ...
   dummy_n = e_n
in
   e_f dummy_1 dummy_2 ... dummy_n
```

Function applications can then be reduced as before: the application is rewritten to its body with the dummy_j's substituted for formal parameter identifiers. Evaluation of the body can then proceed concurrently with the evaluation of actuals e_j.

4.3. Operational Semantics for Id Kernel

4.3.1. The Language

The syntax of Id Kernel is given in Figure 4-1.

Figure 4-1: Id Kernel Syntax

Programs:

def $f_1 \; x_{1_1} \; \ldots \; x_{1_n} \; = e_1$;
$\ldots$
def $f_m \; x_{m_1} \; \ldots \; x_{m_n} \; = e_m$;
let
$\ldots$
in
e_0

None of the definitions have arity 0, *i.e.* they all have at least one parameter.

In each e_j ($j > 0$), all free identifiers must be from the set $f_1, \ldots, f_m, x_{j_1}, \ldots, x_{j_n}$.

In the query-expression, all free identifiers must be from the set $f_1, \ldots, f_m$.

Expressions:

atoms : constant or identifier
`1.2, *, phony, ..`

conditionals
if e_1 *then* e_2 *else* e_3

I-structure allocations
`array` $(e_1 \, .. \, e_2)$

I-structure selections
$e_1[e_2]$

applications
e_f `atom`

Let-Blocks
let
$x_1 = e_1$;
$\ldots$
$x_i[e_j] = e_k$;
$\ldots$
in
e_0

Motivated by the discussion of Section 4.2, there are a few differences from the functional subset considered earlier:

- The reintroduction of Let-Blocks, with Commands such as I-structure assignments. The query-expression is always a Let-Block.

- The arguments in procedure applications must be atoms, *i.e.*, constants and identifiers.

- All top-level definitions must have arity > 1. (0-arity definitions are moved into the query-expression Let-Block).

- The introduction of I-structure allocations and selections

Section 4.4 describes how to translate an Id Nouveau program into this kernel language.

4.3.2. Reduction Rules, Redexes, Substitution Rules and Subexes

In the functional subset, the substitution of an identifier by its binding expression was just another reduction rule. In light of the discussion in Section 4.2, this is no longer possible. We thus consider "Reduction Rules" and "Substitution Rules" as separate topics.

Reduction Rules

The reduction rules for built-in and user-defined procedures are the same as in the functional subset.

To model I-structure operations, we introduce a new sort of names called L-identifiers. To distinguish L-identifiers from ordinary (functional) identifiers, we will write all L-identifiers beginning with a capital L. Intuitively, these identifiers stand for I-structure locations and enable us to avoid mentioning addresses, descriptors, etc.

The reduction rule for I-structure allocations is:

$$\texttt{array(l..h)} \Rightarrow \texttt{array-l-h\{Lnew}_{l}\texttt{, Lnew}_{l+1}\texttt{, ..., Lnew}_{h}\texttt{\}}$$

(provided `l` *and* `h` *are integers and* `l` $\leq$ `h`*)*

In fact, this is a *reduction rule scheme* in that conceptually there is one such rule for every pair of integers `l` and `h` such that `l` $\leq$ `h`, and each application of the rule produces a right-hand side with new identifiers `Lnew`$_j$. Note that the identifiers `Lnew`$_j$ on the right hand side of the rewrite rule do not occur on the left hand side. Such a rewrite rule takes our language out of the class of functional languages-- cognoscenti will recognize the similarity between I-structures and first-order terms in a logic programming language such Prolog.

Here is an application of the rewrite rule:

```
let                          ⇒      let
  x = array(2..5)                     x = array-2-5{Lx, Ly, Lz, Lw}
in                                  in
  (x, x)                              (x, x)
```

The intuitive idea is that `array-2-5` is the descriptor which keeps track of the index bounds of the I-structure, and the identifiers within curly brackets are "new" L-identifiers that stand for the components of the I-structure.

We refer to forms like `array-2-5{Lx, Ly, Lz, Lw}` as *I-structure forms.*

Note that each application of the rule produces new L-identifiers, because each evaluation of `array(l..h)` allocates a new array with new component locations. Thus

```
...                          ⇒          ...
a = array(1..2) ;                       a = array-1-2{La1, La2} ;
b = array(1..2) ;                       b = array-1-2{Lb1, Lb2} ;
...                                     ...
```

where `La1`, `La2`, `Lb1` and `Lb2` are new, distinct identifiers. It would be incorrect to reduce it instead to

```
...
a = array-1-2{L1, L2} ;
b = array-1-2{L1, L2} ;
...
```

because that would indicate that the two I-structures share the same cells, *i.e.* that they are superimposed in storage.

Because of the uniqueness of L-identifiers, they may be considered to have global scope, *i.e.* extending over the entire program. All occurrences of a given L-identifier in a program refer to the same location.

Storing a value into a component of an I-structure is modeled by assignment to the L-identifier that stands for that component. Selecting a component of an I-structure can then be modeled as looking up the value associated with the L-identifier that stands for that component of the I-structure.

The reduction rule for I-structure selection is the following:

$$\texttt{array-l-h\{L}_1\texttt{, ..., L}_h\texttt{\}[i]} \Rightarrow \texttt{L}_i \qquad \textit{provided } l \leq i \leq h$$

Here, `i` matches any integer.

For example,

```
array-2-5{Lx, Ly, Lz, Lw}[3] ⇒ Ly
```

We do not introduce any new reduction rule for I-structure assignment. Given an assignment of the form

```
a[j] = e
```

`a` must be bound elsewhere to an I-structure form, so that we can substitute it to get

$$\texttt{array-l-h\{L}_1\texttt{, ..., L}_h\texttt{\}[j] = e}$$

We then reduce the left-hand side using the I-structure selection rule to get

$$\texttt{L}_j \texttt{ = e}$$

after which we can treat is as an ordinary identifier binding.

The rules for Let-Blocks are a little more complicated. We first give reduction rules for lifting

Let-Block bindings out of conditionals, applications, I-structure allocations and I-structure selections.

conditionals:

```
    if (let ... in e1) then e2 else e3
⇒
    let ... in (if e1 then e2 else e3)
```

applications:

```
    (let ... in e1) atom
⇒
    let ... in (e1 atom)
```

I-structure allocations:

```
    array((let ... in e1) .. e2)
⇒
    let ... in (array(e1 .. e2))                         (and similarly for e2)
```

I-structure selections:

```
    (let ... in e1) [e2]
⇒
    let ... in (e1[e2])                                  (and similarly for e2)
```

Notice that for conditionals, we lift Let-Block bindings out of the predicate but not the arms.

Let-Blocks may be nested in various ways, and they may be collapsed using these reduction rules:

```
let                               ⇒        let
  ...                                         ...
  x = let                                     x = e_let ;
        Bindings                              Bindings ;
      in                                      ...
        e_let ;                             in
  ...                                         e
in
  e
```

```
let                               ⇒        let
  ...                                         ...
  a[let                                       a[e_let] = e_rhs ;
      Bindings                                Bindings ;
    in                                        ...
      e_let] = e_rhs ;                      in
  ...                                         e
in
  e
```

```
let                                  ⇒        let
  . . .                                         . . .
in                                              Bindings
  let                                         in
    Bindings                                    e_let
  in
    e_let
```

In each of the Let-Block transformations, identifiers that were local to the inner scope must first be renamed to new, unique identifiers in order that they do not clash with identifiers in the outer scope. Because L-identifiers have global scope, they are *never* renamed.

Some examples of this renaming:

```
      if (let x = 3 in y < x) then x + 4 else x - 4
⇒
      let x1 = 3 in (if y < x1 then x + 4 else x - 4)
```

```
let                                  ⇒        let
  x = 2 ;                                       x = 2 ;
  y = 3 ;                                       y = 3 ;
  z = let                                       z = x1 + y ;
        x = 4                                   x1 = 4
      in                                      in
        x + y                                   x + y + z
in
  x + y + z
```

In each case, the x in the inner Let-Block has been renamed to **x1** in order that it not clash with the x in the outer scope.

As before,

- A *redex* is an expression that matches the left-hand side of a reduction rule.
- A redex can be reduced by replacing it the right-hand side of the corresponding rule, with parameter identifiers (if any) replaced by the expressions they match.

Substitution Rules

Substitution for an occurrence of an identifier is permitted only after its binding expression has been evaluated for its side-effect and reduced to a denotation of its value, *i.e.* after the binding expression has been reduced to a "Substitutable Form".

Substitutable Forms are defined in Figure 4-2.

Figure 4-2: Substitutable Forms

Constants: 0, 3.14, "Butragueno", +, true, ...

Partial applications: f sf_1 ... sf_j
(where arity of f > j)

I-structure forms: array-l-h{L_l, L_{l+1}, ..., L_h}

A *subex* (for *sub*stitutable *ex*pression) is an identifier x for which there is a binding x = sf in the query-expression Let-Block, where sf is an expression in substitutable form.

Substitution Rule: A subex x may be substituted by its binding (substitutable) expression sf.

4.3.3. Computation and Answers

To compute with an Id Kernel program, we begin with the query expression, and repeatedly rewrite it in two ways:

1. *reduction*: apply a rewrite rule to a redex (such as replacing (1 - 1) with 0) *if it is not within either arm of a conditional.*

2. *substitution*: apply the substitution rule to a subex (such as replacing x with array-1-3{La,Lb,Lc} *if it is not within either arm of a conditional.*

In the purely functional subset discussed earlier, we did not need this qualification on conditionals. There, it was only a matter of efficiency, and so we left it up to the computation rule (e.g. the Applicative Order rule) to decide whether reductions inside arms of conditionals should be performed. In Id Kernel, because of the possibility of side-effects in the arms of conditionals, this degree of freedom can lead to indeterminate answers. We therefore decree that *every* correct interpreter (*i.e.* all computation rules) must not look inside the arms of conditionals.

Again, hopefully this rewriting process terminates until the query-expression is in some sort of "simplified" form-- we call such forms *Final Forms.* Final Forms are defined in Figure 4-3.

Figure 4-3: Final Form of the Query-Expression

let
  $\texttt{x}_1$ = $\texttt{fe}_1$;
  ...
  x[y] = $\texttt{fe}_j$;
  ...
in
  $\texttt{fe}_k$

where each **fe** is a "final expression", *i.e.* an expression which *is not a redex or a subex* and is of the form:

1. Constants: **0, 3.14, "Butragueno", +, true, ...**

2. Partial applications: **f** $\texttt{fe}_1$ ... $\texttt{fe}_j$
(where arity of **f** > **j**)

3. I-structure forms: **array-l-h{**$\texttt{L}_l$, $\texttt{L}_{l+1}$, ..., $\texttt{L}_h$**}**

4. Conditionals: *if* **fe** *then* $\texttt{e}_1$ *else* $\texttt{e}_2$
(where $\texttt{e}_1$ and $\texttt{e}_2$ are arbitrary expressions)

5. I-structure selections: $\texttt{fe}_1$[$\texttt{fe}_2$]

As discussed earlier, it is possible that a programmer may assign twice to the same I-structure location. In this case, when the query-expression reaches final form, the bindings in the Let-Block will contain more than one definition for the L-identifier corresponding to that location.

Note that the following expression does not result in multiple assignments to the same L-identifier:

```
.....
if e then let Lx = 3 in ...
     else let Lx = 5 in ...
.....
```

because we were careful to prohibit any computation inside the arms of a conditional. Thus **e** must be reduced to **true** or **false** first, the conditional must be reduced to one or the other arm, and thus only one of assignments to **Lx** will remain.

We can now specify what is the answer computed by a program. *Answers* are defined in Figure 4-4.

Figure 4-4: The Answer Computed by a Program

The query-expression Let-Block must be rewritten to final form; the answer is then:

- "Error", if there are multiple assignments to any L-identifier in the bindings of the Let-Block,
- The Return-Expression of the Let-Block, otherwise.

4.3.4. The Dataflow Computation Rule

The Dataflow Computation Rule is defined in Figure 4-5.

Figure 4-5: Dataflow Computation Rule

- reduce zero or more redexes which are not inside either arm of a conditional
- substitute for zero or more subexes that are not inside either arm of a conditional

To ensure that we make progress in the computation, we will insist on some "fair-scheduling"-- no reduction or substitution can be postponed indefinitely.

The computation rule followed by a dataflow interpreter is neither applicative-order nor normal-order. A dataflow interpreter begins the evaluation of all arguments to a function before invoking the function. However, it does not wait for the evaluation of the arguments to *complete*-- instead, it can begin the execution of the body of the function before the execution of *any* of the arguments has terminated. Not surprisingly, an interpreter that follows this "dataflow rule of evaluation" must either be parallel or must simulate parallelism. Like an applicative-order interpreter, a dataflow interpreter gives special treatment to the conditional form-- it does not attempt any evaluation inside either arm of the conditional until the predicate has been evaluated. A dataflow interpreter is a safe interpreter; moreover, it evaluates the arguments in any function application exactly *once.*

4.3.5. An Example

Consider the following Id Nouveau program:

```
def table = array (1..3) ;

def fill x = let
                x[1] = 3.14 ;
                x[2] = 6.28 ;
                x[3] = 2.41
             in
                ();

let
   = fill table
in
  table[1] + table[2]
```

Transformed to Id Kernel, we have

```
def fill x = let
                x[1] = 3.14 ;
                x[2] = 6.28 ;
                x[3] = 2.41
             in
                ();

let
  table = array(1..3)
in
  let
    dummy = fill table
  in
    table[1] + table[2]
```

We rewrite the query expression as follows:

```
    let
      table = array(1..3)
    in
      let
        dummy = fill table
      in
        table[1] + table[2]

⇒                                                  (Collapse Let-Block)

    let
      table = array(1..3) ;
      dummy = fill table
    in
      table[1] + table[2]
```

⇒ *(I-structure allocation)*

```
let
  table = array-1-3{Lx, Ly, Lz} ;
  dummy = fill table
in
  table[1] + table[2]
```

⇒ *(substituting for* `table`*)*

```
let
  table = array-1-3{Lx, Ly, Lz} ;
  dummy = fill array-1-3{Lx, Ly, Lz}
in
  array-1-3{Lx, Ly, Lz}[1] + array-1-3{Lx, Ly, Lz}[2]
```

⇒ *(rewriting* `fill` *application)*

```
let
  table = array-1-3{Lx, Ly, Lz} ;
  dummy = let
              array-1-3{Lx, Ly, Lz}[1] = 3.14 ;
              array-1-3{Lx, Ly, Lz}[2] = 6.28 ;
              array-1-3{Lx, Ly, Lz}[3] = 2.41
          in
              ()
in
  array-1-3{Lx, Ly, Lz}[1] + array-1-3{Lx, Ly, Lz}[2]
```

⇒ *(collapsing Let-Blocks)*

```
let
  table = array-1-3{Lx, Ly, Lz} ;
  dummy = () ;
  array-1-3{Lx, Ly, Lz}[1] = 3.14 ;
  array-1-3{Lx, Ly, Lz}[2] = 6.28 ;
  array-1-3{Lx, Ly, Lz}[3] = 2.41
in
  array-1-3{Lx, Ly, Lz}[1] + array-1-3{Lx, Ly, Lz}[2]
```

⇒ *(I-structure selections)*

```
let
  table = array-1-3{L-x, L-y, L-z} ;
  dummy = () ;
  Lx = 3.14 ;
  Ly = 6.28 ;
  Lz = 2.41
in
  Lx + Ly
```

⇒ *(substituting for Lx, Ly)*

```
let
  table = array-1-3{L-x, L-y, L-z} ;
  dummy = () ;
  Lx = 3.14 ;
  Ly = 6.28 ;
  Lz = 2.41
in
  3.14 + 6.28
```

⇒ *(Reduction rule for +)*

```
let
  table = array-1-3{L-x, L-y, L-z} ;
  dummy = () ;
  Lx = 3.14 ;
  Ly = 6.28 ;
  Lz = 2.41
in
  9.42
```

which is in final form. The answer is thus **9.42**.

Note that at many points we had several choices of reduction/substitution for rewriting the expression. A dataflow interpreter would do some or all of them in parallel.

The striking difference between ordinary (functional) identifiers and L-identifiers can be seen in this example. The introduction of an ordinary identifier is inseparable from its definition: for example, the Let-Block

```
let
  x = 3 ;
  y = 3 + x
in
  x + y
```

introduces two identifiers **x** and **y** and gives them definitions as well. In contrast, an L-identifier can come into existence through array allocation without having any definition attached to it, and may get defined at some other point of program execution through array assignment. More operationally, an L-identifier can be considered to be a "place-holder" for a value; creating an L-identifier creates the place-holder in which a value may be written in at a later point in the program.

4.4. Translating an Id Nouveau Program into an Id Kernel Program

Although we have given an operational semantics for only the Id Kernel subset of Id Nouveau, it is simple to translate any Id Nouveau program into this subset. This is done as follows:

1. Loops are transformed into Let-Blocks with tail-recursive definitions, as outlined in Section 2.10.

2. All tuple constructs are replaced by I-structure constructs, as described in Section 4.4.1.

3. The Renaming transform described in Section 4.4.2 is applied to remove potential name ambiguities and to name the arguments of all user-defined procedures.

4. The Lambda-Lifting transform described in Section 4.4.3 is applied to close and lift all inner procedure definitions (within Blocks) to the top-level.

4.4.1. Implementing Tuples Using I-structures

All tuple constructs in an Id Nouveau program may be transformed into I-structure constructs, as follows:

Step (i) Replace all occurrences of tuple constructors of the form

```
(e1, e2, ..., en)
```

by a Let-Block of the form:

```
let
  new-id = array(1..n) ;
  new-id[1] = e1 ;
  ...
  new-id[n] = en
in
  new-id
```

where `new-id` is a new identifier that does not occur in the program.

Step (ii) Replace all tuple-structured bindings of the form

```
t1, t2, ..., tn = e
```

by a set of bindings

```
new-id = e ;
t1 = new-id[1] ;
t2 = new-id[2] ;
...
tn = new-id[n] ;
```

where `new-id` is a new identifier that does not occur in the program.

Note that since tuple constructors and tuple-stuctured bindings may be nested, these transformations may have to be repeated recursively.

An example. The expression

```
frabjous, borogoves(snark), joy
```

can be replaced by the expression

```
let
  arzoo = array(1..3) ;
  arzoo[1] = frabjous ;
  arzoo[2] = borogoves(snark) ;
  arzoo[3] = joy
in
  arzoo
```

4.4.2. Naming Copyable Subexpressions

In this step, we ensure that the arguments to all procedures are atoms (identifiers or constants). Consider any application of the form:

```
(e_0 e_1 ... e_n)
```

Replace this expression with the form:

```
let
  new-id_1 = e_1 ;
  ...
  new-id_n = e_n
in
  (e_0 new-id_1 ... new-id_n)
```

An "optimization" here is to avoid renaming an argument e_i if it is already an atom, *i.e.* it is a constant or an identifier.

4.4.3. Lambda-Lifting

In Id Nouveau, as in many languages, a procedure definition can use an identifier that is defined in some surrounding scope. For example, in

```
let
  f x = let
          g y = y + x
        in
          g
in
  (f 2) 3
```

the inner procedure **g** uses the non-local identifier **x**. **x** is also called a **Free Identifier** of procedure **g**.

A procedure with free identifiers is meaningless without an accompanying specification of what values the free identifiers represent. This "accompanying specification" is called an **Environment** for the procedure. A procedure together with its environment is called a **Closure**, and this structure truly has a meaning in the space of functions.

In languages like Pascal and Algol, the Environment for a procedure is implemented very cheaply, using mechanisms such as Displays, Static Chains, etc. But such straightforward solutions are no longer possible when procedures may be values returned as results of other procedures, as in Id Nouveau. If one thinks of an Algol-like implementation, the representation of a procedure returned as a value must not only refer to the code for the procedure, but must also capture the bindings of all its free variables currently on its static chain. Many language implementations in fact use exactly this mechanism to handle free identifiers (e.g. Scheme [13], Unix ML [7]). The run-time activities of building closures and accessing arguments from closures can be expensive. Many languages preclude procedures as values for precisely this reason.

Note that if a procedure does not contain *any* free identifiers, we needn't bother about an environment for the procedure-- it will never be looked up anyway! Such procedures are called **"Closed Procedures"**. If we could ensure that *all* procedures were closed, the implementation would become correspondingly simpler because it would *never* have to worry about building and accessing environments and closures.

Turner [15] suggested a method to transform a functional program into another equivalent functional program in which all procedure definitions were closed, so that the implementation did not have to handle environments and closures. This technique was subsequently refined in [10] and [11]. We will now outline the latter method.

"Lambda-lifting"[11] is a source-to-source transformation that achieves two effects: first, the resulting program contains *only* closed procedures, and second, since procedures are now closed, they are all lifted out to the top level, and the resulting program has no nested procedure definitions. For example, the program above could be transformed into the equivalent program:

```
let
  f x = let
          g x y = y + x
        in
          (g x)
in
  (f 2) 3
```

Here we have made x an explicit parameter in the definition of g; since g now has this extra parameter, the occurrence of g in the return-expression is converted into an application of g to x.

The function g is now closed, and so we can further transform the program into the form:

```
def g x y = y + x ;

let
  f x = (g x)
in
  (f 2) 3
```

by "lifting" the newly-closed procedure g to the top-level, without any danger of violating scope-rules.

A more complicated example (after [11]):

[11] The name alludes to the Lambda Calculus, which is the "canonical" functional programming language.

```
def a = 2 ;

def b = 4 ;

let
   c = .. ;
   f x = .. x .. g .. a .. ;
   g y = .. y .. f .. b ..
in
   .. f .. g ..
```

Here there are two nested procedure definitions f and g, and they contain free identifiers a and b respectively. Further, f and g are mutually recursive.

We first attempt to close f and g by supplying their free identifiers as parameters, and changing every use of f and g to an application of these new functions to those parameters:

```
def a = 2 ;

def b = 4 ;

let
   c = .. ;
   f a x = .. x .. (g b) .. a .. ;
   g b y = .. y .. (f a) .. b ..
in
   .. (f a) .. (g b) ..
```

But in the process, we have introduced new free identifiers: b into f and a into g! We thus repeat the process to get

```
def a = 2 ;

def b = 4 ;

let
   c = .. ;
   f b a x = .. x .. (g a b) .. a .. ;
   g a b y = .. y .. (f b a) .. b ..
in
   return .. (f b a) .. (g a b) ..
```

Now the nested procedure definitions are truly closed, and they may be lifted to the top-level:

```
def a = 2 ;

def b = 4 ;

def f b a x = .. x .. (g a b) .. a .. ;

def g a b y = .. y .. (f b a) .. b .. ;

let
   c = ..
in
   .. (f b a) .. (g a b) ..
```

A detailed algorithm to perform this transformation efficiently in one pass is given in [11].

4.5. Conclusion

Id Nouveau is not a toy language. A compiler for Id (a predecessor to Id Nouveau) has been running for over two years, generating code for the MIT Tagged-Token Dataflow Architecture (TTDA), via Dataflow Graphs. The compiler was begun by Vinod Kathail and later modified and maintained by Ken Traub. Several large scientific codes such as SIMPLE, which is a 2000 line FORTRAN program for hydrodynamic simulation, have been coded in Id and run both on a TTDA simulator and on an emulation of the TTDA on the MIT Multi-Processor Emulation Facility (MEF).

After some modifications, the compiler has begun to generate code from Id Nouveau programs. The transition from Id to Id Nouveau has been remarkably smooth-- the main effort was in handling partial applications (Id did not support partial applications and higher-order functions).

Ken Traub is in the midst of a complete rewrite of the compiler for Id Nouveau, with a new modular structure to facilitate experiments on code generation, optimization, types and type-checking, etc., and incorporating our latest understanding of dataflow graphs.

Id is supported by a interactive, integrated programming environment called "Id World". Sitting at a high-performance workstation, the programmer may

- Write programs, using an editor customized for Id.

- Selectively compile Id procedures.

- Specify a TTDA configuration (number of processors, timing parameters, performance data to be collected, etc.).

- From the user's workstation, load, execute and debug the program on any of three facilities:

   1. The local workstation.

   2. GITA (Graph Interpreter for the Tagged-Token Architecture), a multi-processor

emulation of the TTDA on the Multi-processor Emulation Facility.

3. A hardware-module level simulator running on a mainframe.

(All three facilities execute the same compiled code.)

- Analyse and plot performance data collected during the execution.

These experiments have strengthened our conviction that Id Nouveau is a good programming language for running scientific codes on parallel machines. We believe that it is also suitable for AI and database applications, but such an evaluation is difficult because of the lack of accepted benchmarks in these fields.

Appendix A. Example Programs

Here are some example of Id Nouveau programs to familiarize the reader with the programming language.

```
-- Average of two numbers

def average2 x y = (x + y) / 2 ;


-- Roots of a quadratic equation given coefficients

def roots a b c = let
                     twoa    = 2*a ;
                     x       = sqrt (b*b - 4*a*c) / twoa ;
                     bBy2a   = b / twoa
                  in
                     (- bBy2a + x, - bBy2a - x) ;


-- Absolute value of a number

def abs x = if x >= 0 then x else - x ;


-- Square Root by Newton's Method  (after Abelson/Sussman)

def sqrt x =
    let
      guess = 1 ;
      not-good-enough? guess = abs (guess * guess - x) < 0.001
    in
      while not-good-enough? guess do
        new guess = average2 guess (x / guess)
      return guess ;


-- Greatest Common Divisor of two positive numbers

def gcd x y = let
                  a, b = x, y
              in
                  while a <> b do
                    new a, new b = if a > b then (a - b), b
                                   else a, (b - a)
                  return a ;
```

```
-- Pointwise product of two vectors

def pointwiseProduct v1 v2 = let
                                  l,h = bounds v1 ;
                                  prod = array(l..h) ;
                                  = for j from l to h do
                                      prod[j] = v1[j] * v2[j]
                                    return ()
                                in
                                  prod ;

-- Inner Product of two vectors

def innerProduct v1 v2 = let
                              l,h = bounds v1 ;
                              ip = 0
                            in
                              for j from l to h do
                                new ip = ip + v1[j] * v2[j]
                              return ip ;

-- Matrix transposition

def transpose mat = let
                         n = upper mat ;
                         resultmat = array(1..n, 1..n) ;
                         = for i from 1 to n do
                             = for j from 1 to n do
                                 resultmat[i,j] = mat[j,i]
                               return ()
                           return ()
                       in
                         resultmat ;

-- Matrix multiplication: (m by n) matrix with (n by p) matrix

def matrix* m1 m2 =
    let
      m = upper m1 ;
      n = upper (m1[1]) ;
      p = upper (m2[1]) ;
      resultmat = array(1..m, 1..p) ;
      = for i from 1 to m do
          = for j from 1 to p do
              sum = 0 ;
              resultmat[i,j] = for k from 1 to n do
                                 new sum = sum + m1[i,k] * m2[k,j]
                               return sum
            return ()
        return ()
    in
      resultmat ;
```

```
-- Matrix relaxation (next value depends on four neighbours)

def relax cornerop edgeop insideop terminate? m =
    let
      s = upper m ;
      relax m m1 =
          let
            m1[1,1] = cornerop(m[1,1], m[2,1], m[1,2]) ;
            m1[1,s] = cornerop(m[1,s], m[2,s], m[1,s-1]) ;
            m1[s,1] = cornerop(m[s,1], m[s-1,1], m[s,2]) ;
            m1[s,s] = cornerop(m[s,s], m[s-1,s], m[s,s-1]) ;

            = for j from 2 to (s-1) do
                  m1[1,j] = edgeop(m[1,j], m[1,j-1], m[2,j], m[1,j+1])
              return () ;
            = for j from 2 to (s-1) do
                  m1[s,j] = edgeop(m[s,j], m[s,j-1], m[s-1,j], m[s,j+1])
              return () ;
            = for i from 2 to (s-1) do
                  m1[i,1] = edgeop(m[i,1], m[i-1,1], m[i,2], m[i+1,1])
              return () ;
            = for i from 2 to (s-1) do
                  m1[i,s] = edgeop(m[i,s], m[i-1,s], m[i,s-1], m[i+1,s])
              return () ;

            = for i from 2 to (s-1) do
                  = for j from 2 to (s-1) do
                      m1[i,j] = insideop(m[i,j],
                                         m[i-1,j], m[i+1,j],
                                         m[i,j-1], m[i,j+1])
                    return ()
              return ()
          in
            ()
    in
     while not (terminate? m) do
       m1 = array(1..s, 1..s) ;
       = relax m m1 ;
       new m = m1
     return m ;

-- LISTs

-- We can model lists (as in Lisp) by using 2-tuples and using the
-- special value nil to represent the empty list.

def cons x y = (x,y) ;
def null l   = (l = nil) ;
def car (x,y) = x ;
def cdr (x,y) = y ;
def atom x = (number x) or (boolean x) or (string x) ...    ;
def length l = if null l then 0
                 else 1 + (length (cdr l)) ;
```

```
-- reverse a list

def reverse l = let
                  rev = nil
               in
                  while not (null l) do
                     h,t = l ;
                     new rev, new l = (cons h rev), t
                  return rev ;

-- flatten a tree-structure

def flatten struc =
      let
        traverse struc leaflist =
            if null struc then leaflist
            else if atom struc then cons atom leaflist
            else let
                   hd, tl = struc
                in
                   traverse hd (traverse tl leaflist)
      in
        traverse struc nil ;

-- Some useful higher-order functions on lists

def map f l = if null l then nil
             else map (f (car l)) (cdr l) ;

def filter p l = if null l then nil
                else let
                       hd, tl = l
                    in
                       if p hd then cons hd (filter p tl)
                              else filter p tl ;

def fold f unit list = if null list then unit
                      else f (car list) (fold f unit (cdr list)) ;


-- Sum,  product and average of a list of numbers

def addup list = fold op_+ 0 list ;

def multiplyup list = fold op_* 1 list ;

def average list = let
                    accum (sum, count) x = (sum + x), count + 1 ;
                    total, number = fold accum (0,0) list
                 in
                    total / number ;
```

```
-- another definition for average of a list of numbers

def average list = let
                      total = 0 ;
                      number = 0
                   in
                      while not (null list) do
                        h,t = list ;
                        new number = number + 1 ;
                        new total = total + h ;
                        new list = t
                      return total/number ;

-- sort a list of numbers into ascending order

def mergesort list =
    let
      merge (l1,l2) = if null l1 then l2
                      else if null l2 then l1
                      else let
                             (h1,t1), (h2,t2) = l1,l2
                          in
                             if (h1 <= h2) then
                               cons h1 (merge (t1, l2))
                             else
                               cons h2 (merge (l1, t2)) ;

      divide list = if null list then nil,nil
                    else if null (cdr list) then list, nil
                    else let
                           h1,(h2,t) = list ;
                           t1,t2     = divide t
                        in
                           (cons h1 t1), (cons h2 t2) ;

      sort  list = let
                     l1, l2 = divide list
                   in
                     merge (sort l1, sort l2)
    in
      sort list ;
```

Appendix B. Id Kernel: Collected Syntax and Semantics

B.1. Id Kernel Syntax

Programs:

def $f_1\ x_{1_1}\ \ldots\ x_{1_n} = e_1$;
...
def $f_m\ x_{m_1}\ \ldots\ x_{m_n} = e_m$;
let
  ...
in
  e_0

where

- Arity(f_j) > 0
- Free Identifiers(e_j) ε $f_1, \ldots, f_m, x_{j_1}, \ldots, x_{j_n}$
- Free Identifiers(query-expression) ε $f_1, \ldots, f_m$

Expressions:

atoms : constant or identifier

`1.2, *, phony, ..`

I-structure allocations

array (e_1 .. e_2)

applications

e_f atom

conditionals

if e_1 *then* e_2 *else* e_3

I-structure selections

$e_1[e_2]$

Let-Blocks

let
  $x_1 = e_1$;
  ...
  $x_i[e_j] = e_k$;
  ...
in
  e_0

B.2. Reduction Rules

Built-in Procedures

$0 + 0 \Rightarrow 0$

$0 + 1 \Rightarrow 1$

.

.

$623 + 1062 \Rightarrow 1685$

.

.

```
16 * 3 ⇒ 48
.
.
if true    then e1  else e2 ⇒ e1
if false   then e1  else e2 ⇒ e2
.
.
```

User-defined Procedures

Each definition in the Id Kernel program:

```
def f x₁ ... xₙ = e ;
```

is treated as a reduction rule:

```
f x₁ ... xₙ ⇒ e
```

I-Structure Allocation

```
array(l..h) ⇒ array-l-h{Lnew_l, Lnew_l+1, ..., Lnew_h}
```

for every integer l and h such that $l \leq h$. Each application of the rule produces new L-identifiers Lnew_j.

I-Structure Selection

```
array-l-h{L_l, ..., L_h}[i] ⇒ L_i
```

for every integer i such that $l \leq i \leq h$.

Lifting Let-Blocks

From Conditionals:

```
if (let ... in e₁) then e₂ else e₃  ⇒  let ... in (if e₁ then e₂ else e₃)
```

Let-Blocks are *not* moved out of e_2 and e_3.

From Applications:

```
(let ... in e₁) atom  ⇒  let ... in (e₁ atom)
```

From I-structure allocations:

```
array((let ... in e₁) .. e₂)  ⇒  let ... in (array(e₁ .. e₂))
```

and similarly for e_2.

From I-structure selections:

```
(let ... in e₁) [e₂]  ⇒  let ... in (e₁[e₂])
```

and similarly for e_2.

From Let-Blocks:

```
let                          ⇒        let
  ...                                   ...
  x = let                               x = e_let ;
        Bindings                        Bindings ;
      in                                ...
        e_let ;                       in
  ...                                   e
in
  e
```

```
let                          ⇒        let
  ...                                   ...
  a[let                                 a[e_let] = e_rhs ;
      Bindings                          Bindings ;
    in                                  ...
      e_let] = e_rhs ;                in
  ...                                   e
in
  e
```

```
let                          ⇒        let
  ...                                   ...
in                                      Bindings
  let                                 in
    Bindings                            e_let
  in
    e_let
```

Before a Let-Block is lifted out, local identifiers (but not L-identifiers) are renamed to new, unique identifiers.

B.3. Redexes and Reduction

A *redex* is an expression that matches the left-hand side of a reduction rule.

A redex is *reduced* by replacing it with the right-hand side of the rule, with parameter identifiers (if any) replaced by the expressions they match.

B.4. Substitutable Forms

Constants: `0, 3.14, "Butragueno", +, true, ...`

Partial applications: $f\ sf_1\ \ldots\ sf_j$
(where arity(f) > j)

I-structure forms: $\texttt{array-l-h}\{L_l,\ L_{l+1},\ \ldots,\ L_h\}$

B.5. Subexes and Substitution

A *subex* is an identifier **x** for which there is a binding **x = sf** in the query-expression Let-Block, where **sf** is an expression in substitutable form.

Substitution Rule: A subex **x** can be substituted by its binding (substitutable) expression **sf**.

B.6. Computation

Begin with the query expression and repeatedly rewrite it by:

1. *reduction*: apply a rewrite rule to a redex *if it is not within either arm of a conditional.*
2. *substitution*: apply the substitution rule to a subex *if it is not within either arm of a conditional.*

B.7. Final Form of the Query-Expression

let
$x_1 = fe_1$;
...
$x[y] = fe_j$;
...
in
fe_k

where each **fe** is a "final expression".

Final expressions are expressions which *are not redexes or subexes* and have the form:

1. Constants: **0, 3.14, "Butragueno", +, true, ...**
2. Partial applications: $f\ fe_1\ \ldots\ fe_j$
   (where arity(**f**) > **j**)
3. I-structure forms: $\texttt{array-l-h}\{L_l,\ L_{l+1},\ \ldots,\ L_h\}$
4. Conditionals: *if* **fe** *then* e_1 *else* e_2
   (where e_1 and e_2 are arbitrary expressions)
5. I-structure selections: $fe_1[fe_2]$

B.8. The Answer Computed by a Program

The query-expression Let-Block must be rewritten to final form; the answer is then:

- "Error", if there are multiple assignments to any L-identifier in the bindings of the Let-Block,
- The Return-Expression of the Let-Block, otherwise.

B.9. Dataflow Computation Rule

- reduce zero or more redexes which are not inside either arm of a conditional
- substitute for zero or more subexes that are not inside either arm of a conditional

References

1. Arvind, and D.E.Culler. "Dataflow Architectures". CSG 294, MIT Laboratory for Computer Science, Cambridge, MA, February, 1986. To appear in Annual Reviews in Computer Science, 1986.

2. Arvind, D.E.Culler, R.A.Iannucci, V.Kathail, K.Pingali and R.E.Thomas. "The Tagged Token Dataflow Architecture". MIT Laboratory for Computer Science, Cambridge, MA, July, 1983. (Prepared for MIT Subject 6.83s).

3. Arvind, K.P.Gostelow and W.Plouffe. "An Asynchronous Programming Language and Computing Machine". 114a, Department of Information and Computer Science, University of California, Irvine, CA, December, 1978.

4. Arvind, K.Pingali and R.S.Nikhil. "I-Structures: Data Structures for Parallel Machines". CSG Memo (in preparation), MIT Laboratory for Computer Science, Cambridge, MA, 1986.

5. Arvind, and R.E.Thomas. "I-Structures: An Efficient Data Type for Functional Languages". TM-178, MIT Laboratory for Computer Science, Cambridge, MA, September, 1980.

6. Barendregt, Henk and Marc van Leeuwen. "Functional Programming and the Language TALE". TR 412, Mathematical Institute, Budapestlaan 6, 3508 TA Utrecht, The Netherlands, 1985.

7. Cardelli,L. "ML Under Unix". AT&T Bell Laboratories, 1983.

8. Heller,S.K. "An I-Structure Memory Controller". Master Th., Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA,June 1983.

9. Heller,S. and Traub,K. "Id Compiler User's Manual". CSG Memo 248, MIT Laboratory for Computer Science, Cambridge, MA, May, 1985.

10. Hughes,R.J.M. "Super-Combinators". Proc. 1982 ACM Symp. on Lisp and Functional Programming, Pittsburgh, PA, ACM, August, 1982, pp. 1-10.

11. Johnsson,T. "Lambda Lifting". Proc. Workshop on Sequential Implementations of Functional Langauges, Chalmers Institute of Technology, Goteborg, Sweden, January, 1985, pp. 13.

12. Nikhil,R.S. and Arvind. "Id/83s". MIT Laboratory for Computer Science, Cambridge, MA, July, 1985. (Prepared for MIT Subject 6.83s).

13. Steele,G.L. Jr. and Sussman,G.J. "The Revised Report on Scheme: A Dialect of LISP". AI Memo 452, MIT Artificial Intelligence Laboratory, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, January, 1978.

14. Traub,K. "Dataflow Graphs from Id Nouveau". MIT Laboratory for Computer Science, Cambridge, MA, July, 1986. (Excerpts from forthcoming Master's Thesis; prepared for MIT Subject 6.83s).

15. Turner,D.A. "A New Implementation Technique for Applicative Languages". *Software—Practice and Experience* 9, 1 (1979), 31-49.