LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Compilation of Id

Computation Structures Group Memo 341
23 September 1991

Zena M. Ariola
Arvind

To appear in the Proceedings of the Fourth Worskshop on Languages and Compilers for Parallel Computing Semantics Based Program Manipulation, August 1991, Santa Clara, California.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988 (MIT) and N00039-88-C-0163 (Harvard).

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Compilation of Id

Zena M. Ariola
Aiken Computational Laboratory
Harvard University

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology

September 1991

Abstract

In this paper we illustrate, using the Id language, that both the operational semantics of a language and its compilation process can be formalized together. Id is a higher-order non-strict functional language augmented with I-structures and M-structures. The operational semantics of Id is given in terms of a smaller *kernel* language, called Kid. Kid is also the intermediate form used by the compiler to perform type checking and optimizations. Optimizations are described as extensions of Kid operational semantics. A criteria for correctness of optimizations is presented. P-TAC, a lower-level language, is introduced to capture some efficiency issues related to code generation. The salient features of translating Kid into P-TAC are presented.

1 Introduction

Modern (functional) languages are too complex to be given direct operational semantics. It is usually better to translate the source language into a simpler and smaller *kernel*

language in order to explain its meaning precisely [11]. A program is said to be *well-formed* if it can be translated into the kernel language, and if it satisfies certain other constraints such as type correctness. Operational or dynamic semantics is concerned only with well-formed programs.

All compilers do a similar translation into an intermediate form in the process of generating code for a machine. A compiler performs type checking and optimizations on this intermediate form before generating machine code. In this paper we will show that the intermediate form can actually be the kernel language. In fact, we may translate the kernel language into still lower-level language(s), where more machine oriented or efficiency related concerns can be expressed directly. Furthermore, compiler optimizations may be expressed as source-to-source transformations on an intermediate language. The semantics of well-formed programs in these intermediate languages is important if we want to show the correctness of these optimizations or the translation process. Thus, each module of the compiler does one of the following three things:

- translate a language L_i into language L_{i+1}; or
- optimize, *i.e.*, source-to-source transformation in language L_i; or
- annotate a program in language L_i with some properties, such as types, scopes or source line numbers.

An advantage of formalizing the compiler modules in these terms is that it gives flexibility to the compiler writer in choosing data structures for various modules. For example, our presentation does not take a position regarding the representation of terms in our kernel language. The compiler may choose different data structures, such as, parse trees or graphs, for terms in different modules. The idea of viewing intermediate forms as languages is not new in the functional language community [6, 9, 14], but it is still rare in the Fortran community (Pingali's work being a notable exception [15]).

In this paper we will show certain aspects of the process of compiling Id, an implicit parallel language [13]. Id is a higher-order functional language augmented with

I-structures [5] and M-structures [7]. I-structures add a flavor of logic variables, while M-structures add side-effects and non-determinism to Id. Id, like most modern functional languages, has a Hindley-Milner type system and non-strict semantics. Id has been in use at MIT as the language for programming dataflow machines. In the last few years interest has grown in compiling Id for workstations and stock parallel machines [16, 17]. We have recently started a project to write the Id compiler in Id, which will embody the strategy outlined in this paper.

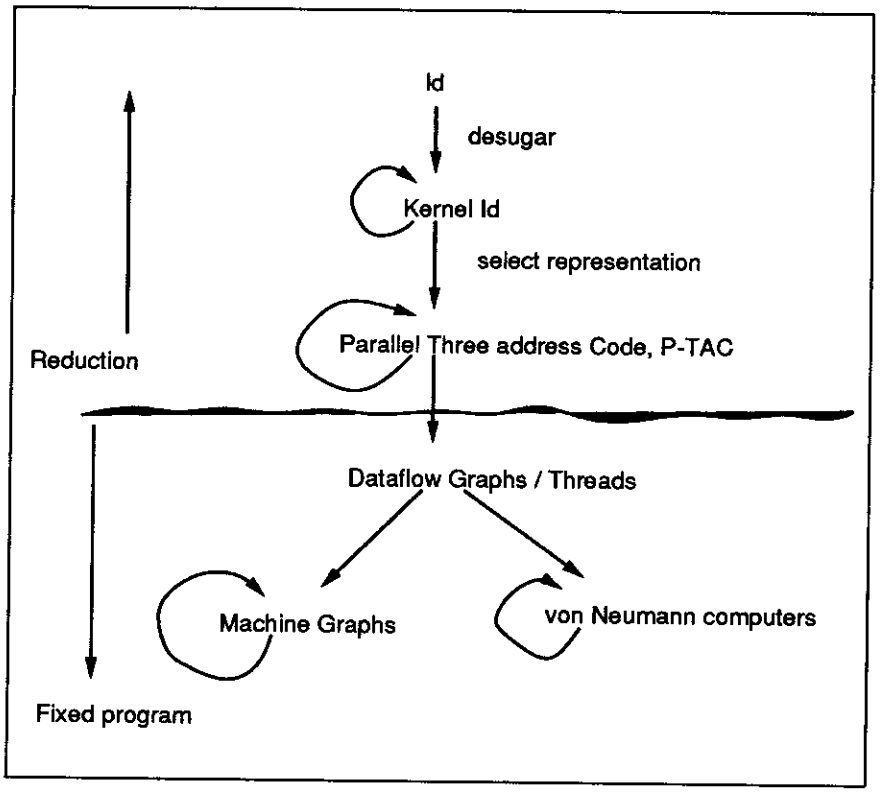


Figure 1: Compilation sheme of Id

Figure 1 shows the high-level Id compilation process, while Figure 2 shows some of the steps in going from Id to Kid, the kernel Id language. The circular arrows in Figure 1 refer to optimizations. The steps in Figure 2 together with the operational semantics of Kid are needed to formalize the operational semantics of Id. The compiler needs to do parsing and scope analysis in order to actually carry out the steps shown in Figure 2; we

have not shown these phases explicitly.

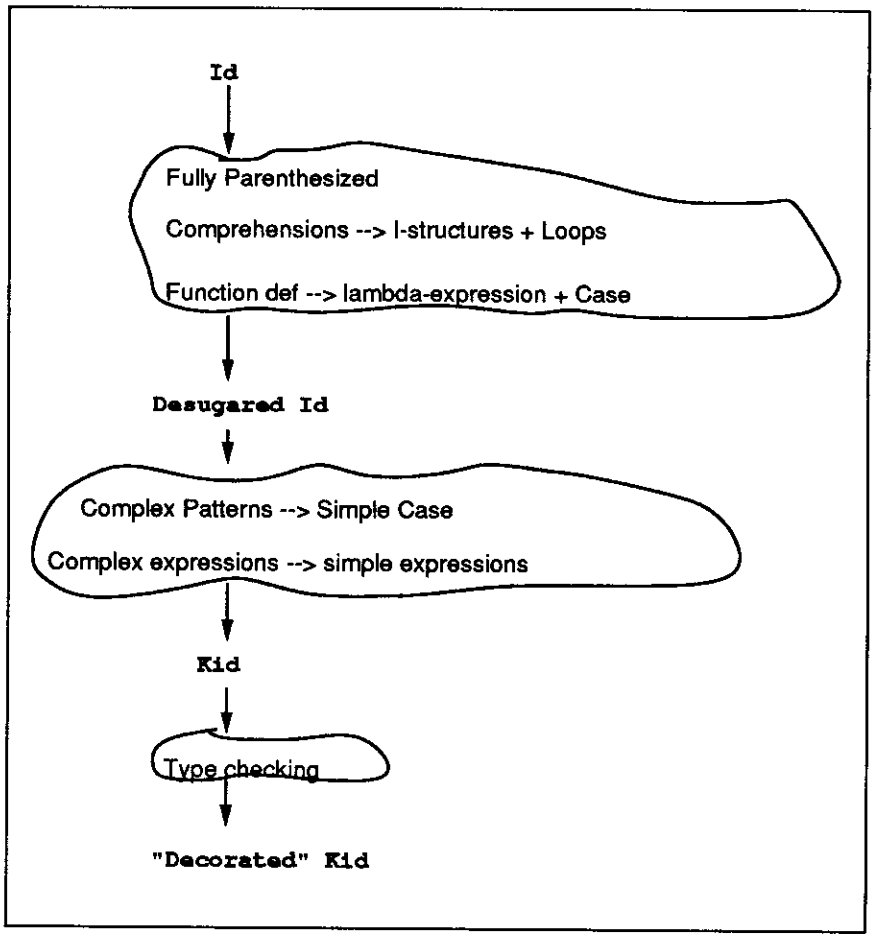


Figure 2: Operational semantics of Id

As an example of the desugaring process, consider the following function which maps f to each element of a list.

```
Def map f Nil   = Nil
 |  map f x:xs = (f x) : map f xs;
```

The multi-clause definition gets turned into a case expression and the function definition into a λ-expression as follows:

```
map =  { Fun t1 t2 = { Case (t1, t2)   of
                     |  (f,  Nil)   = Nil
                     |  (f,  x:xs)  = (f x) : map f xs } }
```

where t1 and t2 are new variables. The pattern matching module simplifies the case-

expression into the following simple case expression:

```
map =  { Fun t1 t2 = { Case t2   of
                     |  Nil  = Nil
                     |  x:xs = (t1 x) : map t1 xs } }
```

In Section 2 we will describe Kid, some aspects of translating desugared Id into Kid, and some optimizations on Kid programs. The Kid to P-TAC translation, which involves choosing representations for data structures and higher-order functions, is discussed in Section 3. The tone of the paper is informal throughout; for a more comprehensive description of the compiler the reader may refer to [3]. However, even in [3], M-structures are not discussed.

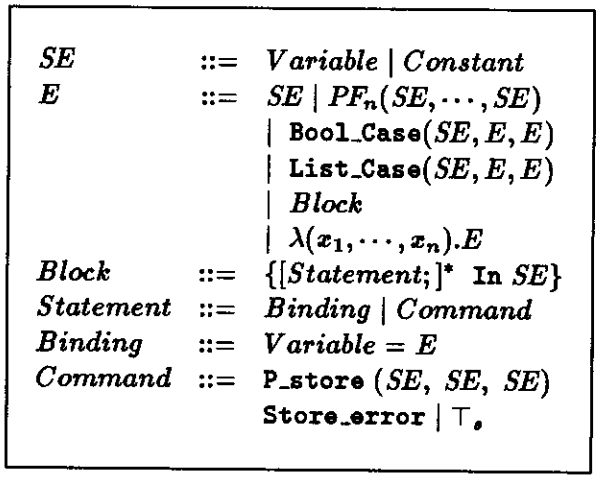
2 Kid: The Kernel Id language

In Kid every expression, except a block, a case or a λ-expression, is of the form

$$PF_n(SE_1, \cdots, SE_n)$$

where PF_n is the name of a *primitive function* of arity n, and SE stands for *simple expression* which is either a constant or a variable. Some examples of primitive functions are the + operator, the application operator `Apply`, and the array constructor `I_array`. A simplified description of the language is given in Figure 3; for a complete understanding of Kid the reader may refer to [4]. As a strongly typed language, Kid needs a different case-expression and a set of selector and constructor operators for each algebraic type. Since a discussion of user defined types will complicate our presentation without necessarily providing additional insight, we have only included the operators for the list type.

Kid can be seen as the λ-calculus with constants and let-blocks. However, unlike other functional languages [14], let-blocks play a fundamental role in the operational semantics of Kid. Our let-block semantics precisely defines how arguments are shared; an essential feature for Id extended with I-structures and M-structures. Sharing is expressed by giving a name to each subexpression and by allowing substitution of values and variables



SE	::=	*Variable* | *Constant*
E	::=	$SE \mid PF_n(SE, \cdots, SE)$
		| `Bool_Case`(SE, E, E)
		| `List_Case`(SE, E, E)
		| *Block*
		| $\lambda(x_1, \cdots, x_n).E$
Block	::=	$\{[Statement;]^*$ `In` $SE\}$
Statement	::=	*Binding* | *Command*
Binding	::=	*Variable* $= E$
Command	::=	`P_store` $(SE,\ SE,\ SE)$
		`Store_error` | $\top$.

Figure 3: Grammar of Kid

only. This idea can be formalized in a *Contextual Rewriting Systems* (CRS) [2, 4] by the following *Substitution rules*:

$$\frac{\mathtt{X} = \mathtt{V}}{\mathtt{X} \longrightarrow \mathtt{V}} \qquad \frac{\mathtt{X} = \mathtt{Y}}{\mathtt{X} \longrightarrow \mathtt{Y}}$$

where `V` is either an Integer or a Boolean or an `Error`. Intuitively, the above rule should be read as follows: occurrences of `X` in a program M can be rewritten to `V`, only if the binding `X` $=$ `V` appears in the context of `X` in M. A context represents a term with "holes", that is, a term with some unspecified subexpressions or bindings. If we substitute `V` for `X`, or `Y` for `X`, everywhere then the corresponding binding can be deleted from the term.

The following *Block Flattening rule* is essential in all CRS's.

$$\begin{array}{l} \{_{\mathtt{m}}\ \vec{X}_n = \{_{\mathtt{n}}\ SS_1;\ SS_2;\ \cdots \\ \qquad\qquad \mathtt{In}\ \ \vec{Y}_n\} \\ \quad S_1;\ \cdots\ S_n \\ \quad \mathtt{In}\ \ \vec{Z}_m\} \end{array} \quad \longrightarrow \quad \begin{array}{l} \{_{\mathtt{m}}\ \vec{X}_n = \vec{Y}_n; \\ \quad SS_1;\ SS_2;\ \cdots \\ \quad S_1;\ \cdots\ S_n \\ \quad \mathtt{In}\ \ \vec{Z}_m\} \end{array}$$

where $\vec{X}_n$ represents *multiple variables*. Multiple variables allow expressions in Kid to return multiple values without the necessity of packaging them into a data structure. A reader familiar with dataflow graphs may think of multiple values as a graph with multiple input or output arcs. Multiple values require the following CRS rule:

$$\vec{X_n} = \vec{Y_n} \longrightarrow (X_1 = Y_1; \cdots X_n = Y_n)$$

2.1 Translating Id into Kid

We will use `foldl`, the Id definition of the fold-from-left function to demonstrate the translation from an Id program to a Kid program. The pictorial description of `foldl` is given in Figure 4.

```
Def foldl f s A = { (l,u) = Bounds A;
                    In
                     { for i<- l to u do
                           next s = f A[i] s;
                       finally s }}
```

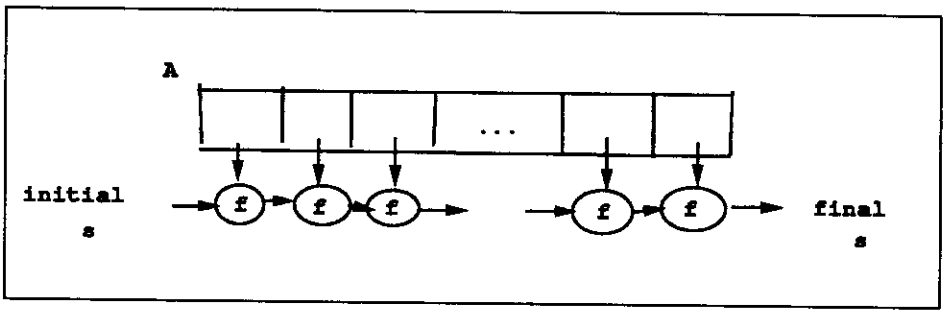


Figure 4: An application of the `foldl` function to array A

Some of the uses of the `foldl` are:

```
foldl (+) 0;     % sums elements of an array
foldl (≤) ∞;     % computes the minimum element in an array
foldl (:) Nil;   % converts an array into a list
```

The translation into Kid introduces explicit `Apply` operators and converts the loop into a λ-expression and `FLoop` combinator:

```
foldl = λ(f,s,A).{ t1 = Bounds A;
                   l = P-select(t1,1);
                   u = P-select(t1,2);
                   tf1 = { b = λ(i,s).{ t1 = Select A i;
                                        t2 = Apply(f,t1);
                                        next-s = Apply(t2,s);
                                         In next-s };
                           tp = ≤(l,u);
                           tf = FLoop(u,1,b,l,s,tp);
                            In tf}
                    In tf1}
```

Now we discuss some salient aspects of this translation process.

Array selection

In Id there are only two operations defined on arrays. *Selection, i.e.* `A[i]`, which returns the value of the i^{th} slot of array `A` assuming that `i` is within the bounds, and (`Bounds A`), which returns the tuple containing the bounds. The selection `A[i]` is translated into the Kid expression (`Select A i`). However, `Select` is not a primitive operator in Kid, due to the complexity of bound checking. Kid uses `P_select` as a primitive selection operator whose semantics does not require bounds checking. Thus, `Select` may be expressed in terms of `P_select` as follows:

```
Select = λ(x,i).{ (l,u) =  Bounds x;
                   In If (i > u Or i < l)  then
                         Error
                          else
                         P_select(x,i) }
```

Notice that the translation of the tuple pattern in the `foldl` program is given using the primitive select operator because type checking guarantees that the indices will be within bounds.

The definition of array elements in Id are given either by array comprehension expressions or by using I-structure assignment rules. In either case, the definition of an array element translates into a `P_store` command. The behavior of `P_select` can be explained using the following rule:

$$\frac{\texttt{P\_store(X,Y,Z)}}{\texttt{P\_select(X,Y)} \longrightarrow \texttt{Z}}$$

The above CRS rule says that the expression `P_select(X,Y)` in a program M, can be rewritten to `Z`, if `P_store(X,Y,Z)` appears in its context in M. Notationally a context is represented by $C[\Box]$. C[`P_store(X,Y,Z)`] then represents the program obtained by filling the hole with `P_store(X,Y,Z)`. Thus, we can say that if M is the program C[`P_store(X,Y,Z)`,`P_select(X,Y)`], then we can *rewrite* M to the program C[`P_store(X,Y,Z)`,`Z`]. Notice that the precondition is not affected by the rewriting; it only enables the rewriting. Furthermore, the precondition does not imply any semantic check; it involves only syntactic pattern matching.

The rule for the **Bounds** operator is as follows:

$$\frac{\texttt{X} = \texttt{I\_array}(\texttt{X}_\texttt{b})}{\texttt{Bounds(X)} \longrightarrow \texttt{X}_\texttt{b}}$$

I-structure semantics prohibits multiple definitions for an array element. This condition can not be checked statically at compile time. Thus, we need the following rules for generating and propagating inconsistent state represented by $\top$ or $\top_s$. These rules are needed to guarantee the confluence of Id.

$$\frac{\texttt{P\_store(X, i, V)}}{\texttt{P\_store(X, i, V')} \longrightarrow \top_s}$$

$$\{_\texttt{m}\ \texttt{X} = \top;\ \texttt{S}_1;\cdots \texttt{S}_\texttt{n}\ \texttt{In}\ \vec{\texttt{Z}}_\texttt{m}\ \} \longrightarrow \top$$

$$\{_\texttt{m}\ \top_s;\ \texttt{S}_1;\cdots \texttt{S}_\texttt{n}\ \texttt{In}\ \vec{\texttt{Z}}_\texttt{m}\ \} \longrightarrow \top$$

Higher-order functions

Another part of the translation deals with higher-order functions. All functions in Id have an associated *arity*. For example, the arity of `foldl` is 3. When all the three arguments

for a `fold` application are available the `Apply` rewrites to an $\mathtt{Ap}_n$ operator, and then the following *full application* rule is used:

$$\frac{\mathtt{F} = \lambda_{n,m}\,(\vec{\mathtt{Z}_n})\,.\,\mathtt{E}}{\mathtt{Ap}_{n,m}(\mathtt{F},\,\vec{\mathtt{X}_n}) \;\longrightarrow\; (\mathcal{RB}[\![\mathtt{E}]\!])\,[\vec{\mathtt{X}_n}\,/\,\vec{\mathtt{Z}_n}]}$$

where $\mathcal{RB}$ is a function which renames the bound variables of `E` to avoid name conflicts. $\mathcal{RB}$ corresponds to the allocation of a *frame* in a stack-based implementation. The notation $[\vec{\mathtt{X}_n}\,/\,\vec{\mathtt{Z}_n}]$ stands for the substitution of $\mathtt{X}_1 \cdots \mathtt{X}_n$ for each occurrence of $\mathtt{Z}_1 \cdots \mathtt{Z}_n$.

Exactly how does an `Apply` become an `Ap` operator? Since the function being applied is not necessarily known at compile time, we need to compile applications for all contingencies. Notice that the Id expression (`f A[i] s`) is a legal expression even if the arity of `f` is greater than 2 or less than 2. We compile each application as a function of one argument. Thus, the Id expression (`f A[i] s`) is translated into the following Kid expression:

```
{ t1 = Select A i;
  t2 = Apply(f,t1);
  t3 = Apply(t2,s);
   In t3 }
```

The `Apply` operator can now be described as follows:

$$\frac{\begin{array}{lcl} \mathtt{F} & = & \lambda_n(\vec{\mathtt{Z}_n})\,.\,\mathtt{E} \\ \mathtt{F}_1 & = & \mathtt{Apply}(\mathtt{F},\,\mathtt{X}_1) \\ \mathtt{F}_2 & = & \mathtt{Apply}(\mathtt{F}_1,\,\mathtt{X}_2) \\ & \vdots & \\ \mathtt{F}_{n-1} & = & \mathtt{Apply}(\mathtt{F}_{n-2},\mathtt{X}_{n-1}) \end{array}}{\mathtt{Apply}(\mathtt{F}_{n-1},\mathtt{X}_n) \;\longrightarrow\; \mathtt{Ap}_n(\mathtt{F},\vec{\mathtt{X}_n})}$$

The above rule says that in order to fire a function `f` of arity n, we need to collect n arguments. Thus, for example, the expression `Apply(f,x)` will not get subjected to any rewriting. An alternative and easier way of expressing the above rule is:

$$\frac{F = \lambda_n(\vec{Z_n}) \,.\, E}{\texttt{Apply}(F,X) \longrightarrow \texttt{Apply}_1(F,\underline{n},X)}$$

$$\frac{F' = \texttt{Apply}_i(F,\underline{n},\vec{X_i}) \quad i < (n-1)}{\texttt{Apply}(F',X_{i+1}) \longrightarrow \texttt{Apply}_{i+1}(F,\underline{n},\vec{X_{i+1}})}$$

$$\frac{F' = \texttt{Apply}_i(F,\underline{n},\vec{X_i}) \quad i = (n-1)}{\texttt{Apply}(F',X_{i+1}) \longrightarrow \texttt{Ap}_n(F,\vec{X_n})}$$

where each operator $\texttt{Apply}_i$ remembers the i arguments collected so far.

Loops

The left-hand-side variable of a binding in a loop gets a new value in each iteration of the loop. When a variable in a loop binding is preceded by the keyword **next**, (for example, **next x** = ...), **x** and **next x** are two distinct variables which can exist simultaneously. One should not think of variable **x** as being updated in a loop as one does in an imperative language. A natural way of expressing this flow of information is by translating the loop body into a function. For example, the body of the following loop in the `foldl` definition:

```
{ for i<- 1 to u do
    next s = f A[i] s;
  finally s }}
```

is translated into the following function:

```
λ(i,s).{ t1 = Select A i;
         t2 = Apply(f,t1);
         next-s = Apply(t2,s);
          In next-s};
```

which returns the nextified variable. The `FLoop` operator invokes this function under the appropriate conditions. The paramenters of the `FLoop` operator correspond to the upper bound, the step (1 in our example), the loop-body function, the nextified variables (including the index variable), and finally, the predicate. The rule for the `FLoop` operator is as follows:

$$
\begin{array}{lcl}
\mathtt{FLoop_n}(\mathtt{U},\mathtt{D},\mathtt{B},\vec{\mathtt{X}}_\mathtt{n},\mathtt{True}) & \longrightarrow & \{_\mathtt{n}\ \vec{\mathtt{t}}_{\mathtt{2,n}} = \mathtt{Ap_{n,n-1}}(\mathtt{B},\vec{\mathtt{X}}_\mathtt{n});\\
& & \quad\ \mathtt{t_1} = +(\mathtt{X_1},\mathtt{D});\\
& & \quad\ \mathtt{t_p} = <(\mathtt{t_1},\mathtt{U});\\
& & \quad\ \vec{\mathtt{t}}'_\mathtt{n} = \mathtt{FLoop_n}(\mathtt{U},\mathtt{D},\mathtt{B},\vec{\mathtt{t}}_\mathtt{n},\mathtt{t_p})\\
& & \quad\ \mathtt{In}\ \vec{\mathtt{t}}'_\mathtt{n}\}\\
\mathtt{FLoop_n}(\mathtt{U},\mathtt{D},\mathtt{B},\vec{\mathtt{X}}_\mathtt{n},\mathtt{False}) & \longrightarrow & \vec{\mathtt{X}}_\mathtt{n}
\end{array}
$$

If the index variable is within the loop bounds, that is, the value of the last argument of `FLoop` is `True`, then function `b` is invoked, index $\mathtt{X_1}$ is incremented and checked to see if $\mathtt{X_1}$ is still less than the upper bound, `U`, and finally, a new instance of the loop is generated. In case the index variable is outside the loop bounds, that is, the value of the last argument of `FLoop` is `False`, the values of the nextified variables are returned.

The reason for introducing special combinators for for-loops and while-loops is to facilitate loop optimizations that a compiler may perform.

Lists

For the sake of completeness, we also give the rules for the algebraic type list.

$$
\frac{\mathtt{Z} = \mathtt{Cons}(\mathtt{X},\mathtt{Y})}{\mathtt{Cons\_1}(\mathtt{Z}) \longrightarrow \mathtt{X}}
\qquad
\frac{\mathtt{Z} = \mathtt{Cons}(\mathtt{X},\mathtt{Y})}{\mathtt{Cons\_2}(\mathtt{Z}) \longrightarrow \mathtt{Y}}
$$

$$
\mathtt{List\_case_m}(\mathtt{Nil},\mathtt{E_1},\mathtt{E_2}) \longrightarrow \mathtt{E_1}
\qquad
\frac{\mathtt{Z} = \mathtt{Cons}(\mathtt{X},\mathtt{Y})}{\mathtt{List\_case_m}(\mathtt{Z},\mathtt{E_1},\mathtt{E_2}) \longrightarrow \mathtt{E_2}}
$$

2.2 Optimizations on Kid

There are many situations where a Kid rewrite rule can be applied at compile time. Take for example, the following Kid program:

```
Def f x = { i = +(2,3); P-store(x,i,v); ... a = P-select(x,5); ... }
```

The compiler could clearly replace the expression +(2,3) by 5 and substitute 5 for i everywhere. These optimizations are often referred to as *constant folding* and *constant propagation* [1]. Moreover, knowing that the 5^{th} element of array x is v, the compiler can rewrite the P_select(x,i) to v. This optimization is called *fetch-elimination*, and is performed in hardware by many supercomputers! The Id compiler also does many optimizations which are not directly derived from the rewrite rules used for giving the operational semantics of Id. A partial list of these other optimizations is:

- inline substitution;
- partial evaluation;
- algebraic rules;
- eliminating circulating variables and constants;
- loop peeling and unrolling;
- common subexpression elimination;
- lift free expressions (loop invariants);
- loop variable induction analysis;
- dead code elimination.

A very interesting fact is that all the above optimizations, except the last two, can be expressed as Kid rewrite rules. For example, *common subexpression elimination* (cse) and the *algebraic rules* may be specified in terms of the following rules:

Cse rule:
$$\frac{Z_1 = +(X,Y)}{Z_2 = +(X,Y) \longrightarrow Z_2 = Z_1}$$

Algebraic rules:
$$*(X,0) \longrightarrow 0$$
$$\texttt{Equal}(X,X) \longrightarrow \texttt{True}$$
$$\frac{X = +(Y,3)}{< (Y,X) \longrightarrow \texttt{True}}$$

Reduction Strategy for Optimizations

The example at the beginning of this section shows that an optimization can trigger another one. Furthermore, optimizations can be applied in many different orders. Thus, interesting questions arise regarding the uniqueness of the final program, and regarding the termination of the optimization process. Since we have expressed the optimizations as rewrite rules, the problem of termination and the effect of reduction strategies can be stated in terms of *strongly normalizing* rules and *confluence*, respectively [10].

Most optimization rules are indeed *confluent.* A few that do destroy confluence do so only in programs with deadlocks. The confluence of optimization rules gives us some flexibility in choosing the order in which rules are to be applied. A powerful and efficient strategy is to apply the first five rules along with all the normal Kid rules in an *outside-in* manner. After that, common subexpression elimination and the lifting of free expressions can be done in an *inside-out* manner. Unfortunately, inside-out rules can trigger more outside-in optimizations, and thus, the whole process needs to be repeated until the expression stops changing.

The only optimization rules that can cause non-termination are the partial evaluation and the inline substitution rules. This problem of termination has been studied extensively in the partial evaluation literature which an interested reader may refer [19]. The deadcode elimination can be done at any stage but must be done once more in the end to pick up maximum dead code. Though we are not sure, we think that the loop variable induction analysis should be done as late as possible, so that maximum information about array subscripts and loop index variables can be used.

The Correctness of Optimizations

Optimizations extend the operational semantics of Kid, and therefore, it is interesting to study if they preserve the *meaning* of a term. Questions about meanings are usually addressed in denotational semantics. However, we think that a great deal of knowledge

can be drawn from just the syntactic structure of terms, and thus, we prefer to formulate the correctness problem from a more syntactic or operational point of view. We define several different notions of equality for Kid terms. The first and the least troublesome equality is based on the notion of *convertibility*, which is induced by replacing the arrow in the rules by the equality sign. For example, constant folding and inline substitution can easily be shown to preserve convertibility and thus, correctness.

Suppose we consider a slightly more complicated optimization, like the common subexpression elimination, it is easy to see that convertibility will not suffice for correctness. Consider the following example:

```
M = { x = +(a,b);   cse        M1 = { x = +(a,b);
      y = +(a,b);   --->              y = x;
      z = +(x,y);                     z = +(x,y);
       In z }                          In z}
```

M is not convertible to M1. It is, however, easy to see that the *unravelled* version of M, +(+(a,b),+(a,b)), is equal to the unravelled version of M1. In fact, M and M1 are said to be *tree equivalent*. It is possible to show that the tree equivalence preserves semantics, that is, no context will be able to distinguish between two terms that are tree equivalent. Thus, we can conclude that the cse rule is correct.

Convertibility and tree equivalence are still very syntactic notions to show the equivalence of the following terms related by an algebraic rule:

```
M = { y = *(x,0);   --->      M1 = 0
      In y }
```

A still harder question arises when we consider "infinite terms" (such as the infinite list of 1's shown below), that can be found in any non-strict functional language.

```
M = { x = 1:x;                M1 = { Def f x = x:(f x);
      In x}                          In f 1}
```

At this point we need to ask exactly what external behavior of a program we want to preserve. In defining the concept of observable behavior or *answer*, we first introduce the notion of a *printable value* associated with a term. Intuitively, the printable value of a term corresponds to the stable or fixed part of the term, that is, the part which will not be subjected to further reduction. For example, we could say that the printable value of

{ x = +(2,3); In x} is Ω, which means no useful information is printable. Only after we reduce +(2,3) to 5, we can say that the printable value of the term is 5. This concept of printable value is related to the notion of *instant semantics* introduced by Welch [18]. The answer is then defined in terms of the maximum information that can be extracted by reducing that term. Levy [12] and Ariola [2] have shown that by picking a suitable definition of print, the domain of answers becomes a term model for the language. Suppose $\mathcal{A}(M)$ denotes the answer associated with M. Then we will say that a rule r is correct if it preserves the answer, that is, for all programs M, we have:

$$M \xrightarrow{r} M1 \Longrightarrow \mathcal{A}(M) = \mathcal{A}(M1)$$

This notion is strong enough to show the correctness of all optimizations.

3 P-TAC: Parallel Three Address Code

If we were only interested in giving operational semantics to Kid then we could stop at the Kid level; however, we want to generate code for a specific machine. Rather than trying to generate code for a specific machine directly, we introduce a level of detail in the translation process that will be relevant to (almost) all machines. We call this next lower level language P-TAC. In P-TAC we have only one data structure called an I-structure array. I-structures have the following rules associated with them:

$$\texttt{Allocate}(\underline{\texttt{n}}) \longrightarrow \texttt{L}$$

where L is a brand new label. Labels are treated like other scalar values such as integers and booleans, and can be freely substituted and stored. The rest of the rules for I-structures are very similar to Kid rules and are as follows:

$$\frac{\texttt{P\_store}(\texttt{L},\underline{\texttt{i}},\texttt{V})}{\texttt{P\_select}(\texttt{L},\underline{\texttt{i}}) \longrightarrow \texttt{V}}$$

$$\frac{\texttt{P\_store}(\texttt{L},\underline{\texttt{i}},\texttt{V})}{\texttt{P\_store}(\texttt{L},\underline{\texttt{i}},\texttt{V}') \longrightarrow \top_s}$$

where V is either an Integer or a Boolean or a Label or Error.

$$\{_{\mathtt{m}}\ \mathtt{X} = \top\ \mathtt{S_1}; \cdots \mathtt{S_n}\ \mathtt{In}\ \vec{\mathtt{Z}}_{\mathtt{m}}\ \} \longrightarrow \top$$

$$\{_{\mathtt{m}}\ \top_{\mathtt{s}};\ \mathtt{S_1}; \cdots \mathtt{S_n}\ \mathtt{In}\ \vec{\mathtt{Z}}_{\mathtt{m}}\ \} \longrightarrow \top$$

All composite objects, that is, tuples, arrays, higher-dimension arrays, all algebraic types, and closures are represented using I-structure arrays. There are usually several reasonable ways to represent each data structure in terms of I-structures. For each type we give one representation, though not necessarily the most efficient one. We have included a "type" tag field for all composite objects, even though it is not needed by the P-TAC interpreter since Id is a statically typed language. However, we might need type information for other reasons, such as garbage collection, and for printing values in a partially executed program.

The grammar of P-TAC is given in Figure 5. Notice that in P-TAC, case expressions for all algebraic types are translated into a single untyped `dispatch` operator.

$$\mathtt{Dispatch}_{\mathtt{n,m}}(\mathtt{i}, \mathtt{E_1}, \cdots, \mathtt{E_i}, \cdots \mathtt{E_n}) \longrightarrow \mathtt{E_i}$$

Prior to translating Kid into P-TAC, all nested λ-expressions are lifted to the top level by a

SE	::=	*Variable* | *Constant*
PF_1	::=	`Allocate`
PF_2	::=	`P_select`
E	::=	$SE \mid PF_n(SE, \cdots, SE)$
		| $\mathtt{Dispatch}_n(SE, E_1, \cdots, E_n)$
		| $\mathtt{FLoop}(SE, SE, SE, SE, SE)$
		| *Block*
Block	::=	{[*Statement*;]* In *SE*}
Statement	::=	*Binding* | *Command*
Binding	::=	*Variable* = *E*
Command	::=	`P_store` (*SE*, *SE*, *SE*)
		`Store_error` | $\top_s$

Figure 5: Grammar of P-TAC

process known as λ-lifting [8]. A Kid program after λ-lifting only contains closed λ-expressions.

The translator, given a Kid program, produces the corresponding P-TAC program and a set,"D", of definitions. The set D is initialized with constants that are introduced by the translator.

3.1 Translation from Kid into P-TAC

In the following we will only discuss how arrays, lists and functions are represented in P-TAC; the interested reader may refer [3] for more details.

Arrays

The representation of `Array(l,u)` is given in Figure 6. The constant definitions for `Headersize`, `Upper`, `Lower`, *etc.* should be included in D.

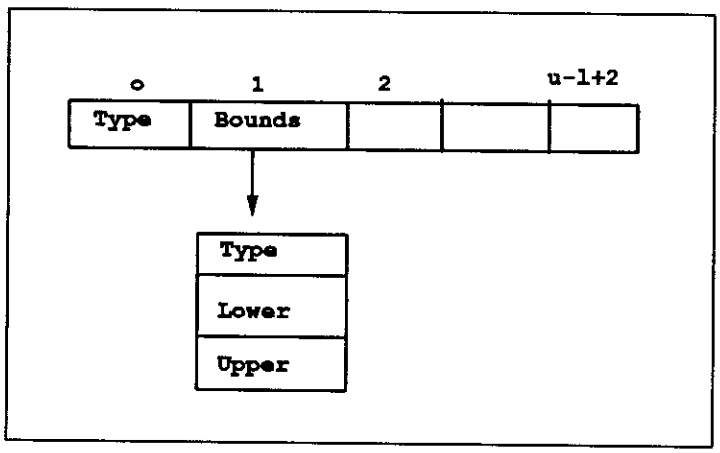


Figure 6: Representation of 1D-array

```
TE[[Bounds(X)]]   =  P_select(X,Bounds)

TE[[I_array(X)]]  =  { l      =  P_select(X,Lower);
                       u      =  P_select(X,Upper);
                       s      =  -(u,l);
                       size   =  +(s,3);
                       t      =  Allocate(size);
                       P_store(t,Type,"Array");
                       P_store(t,Bounds,X);
                         In t}
```

$$
\begin{aligned}
\mathtt{TE}[\![\mathtt{P\_select}(\mathtt{X}_1,\mathtt{X}_2)]\!] \;=\; \{\; & t_b \;=\; \mathtt{P\_select}(\mathtt{X}_1,\mathtt{Bounds});\\
& l \;=\; \mathtt{P\_select}(t_b,\mathtt{Lower});\\
& t_1 \;=\; -(\mathtt{X}_2,l);\\
& t_2 \;=\; +(t_1,\mathtt{Headersize});\\
& t \;=\; \mathtt{P\_select}(\mathtt{X}_1,t_2)\\
& \mathtt{In}\ t\}\\
\mathtt{TE}[\![\mathtt{P\_store}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3)]\!] \;=\; \{\; & t_b \;=\; \mathtt{P\_select}(\mathtt{X}_1,\mathtt{Bounds});\\
& l \;=\; \mathtt{P\_select}(t_b,\mathtt{Lower});\\
& t_1 \;=\; -(\mathtt{X}_2,l);\\
& t_2 \;=\; +(t_1,\mathtt{Headersize});\\
& t \;=\; \mathtt{P\_store}(\mathtt{X}_1,t_2,\mathtt{X}_3)\\
& \mathtt{In}\ t\}
\end{aligned}
$$

A representation that may be more efficient for computing slot addresses would store l and u values redundantly in two additional fields in the array.

Lists

The representation of the list data type is shown in Figure 7. The constant definitions for `Cons_size`, `Hd`, `Tl`, *etc.* should be included in D.

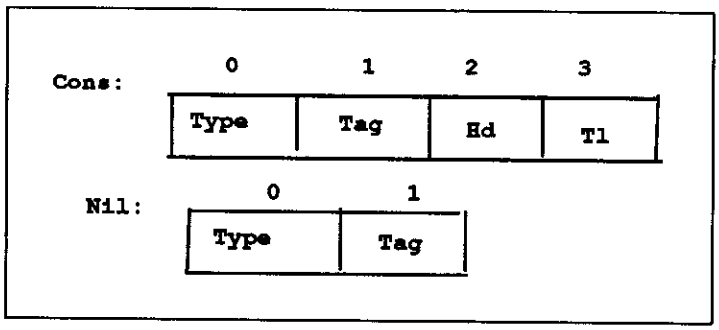


Figure 7: Representation of lists

$$
\begin{aligned}
\mathtt{TE[\![Cons\_1(X)]\!]} &= \mathtt{P\_select(X, Hd)} \\
\\
\mathtt{TE[\![Cons\_2(X)]\!]} &= \mathtt{P\_select(X, Tl)} \\
\\
\mathtt{TE[\![Cons(X_1, X_2)]\!]} &= \{\ t = \mathsf{Allocate(Cons\_size)}; \\
&\qquad \mathtt{P\_store(t, Type, "List")}; \\
&\qquad \mathtt{P\_store(t, Tag, Cons\_Tag)}; \\
&\qquad \mathtt{P\_store(t, Hd, X_1)}; \\
&\qquad \mathtt{P\_store(t, Tl, X_2)}; \\
&\qquad \mathtt{In\ t}\} \\
\mathtt{TE[\![List\_case_m(X, E_1, E_2)]\!]} &= \{_{\mathtt{m}}\ \mathtt{t} = \mathtt{P\_select(X, Tag)}; \\
&\qquad \vec{\mathtt{t}}_{\mathtt{m}} = \mathtt{Dispatch_{2,m}(t, E_1, E_2)}; \\
&\qquad \mathtt{In}\ \vec{\mathtt{t}}_{\mathtt{m}}\}
\end{aligned}
$$

Function Calls and Closures

At the machine level, the apply operator checks if the arity of the function has been satisfied. If the arity has not been satisfied, it stores the argument in a data structure called a *closure*. There is a wide range of representations of closures and associated function calling conventions. In fact, it is possible for a function to be compiled using several different calling conventions; the compiler can pick the most appropriate one for a given application. A representation for the closure data type is shown in Figure 8. The constant definitions for `Closure_size`, *etc.* should be included in D.

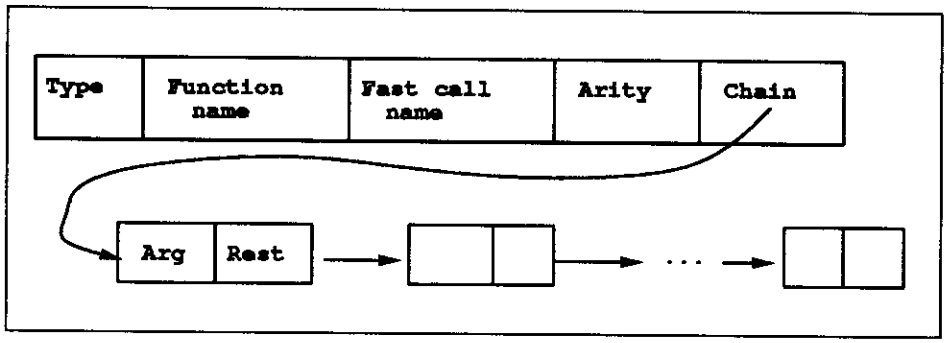


Figure 8: Representation of a closure

We begin by describing a procedure that builds a new closure given an old closure and an argument.

$$
\begin{aligned}
\texttt{Make\_closure} = \lambda\,(\texttt{cl},\texttt{X})\,.\,\{\ \texttt{f} &= \texttt{P\_select}(\texttt{cl},\texttt{Funcname});\\
\texttt{f}_{\texttt{fc}} &= \texttt{P\_select}(\texttt{cl},\texttt{Fastcallname});\\
\texttt{n} &= \texttt{P\_select}(\texttt{cl},\texttt{Arity});\\
\texttt{ch} &= \texttt{P\_select}(\texttt{cl},\texttt{Chain});\\
\texttt{cl}' &= \texttt{Allocate}(\texttt{Closure\_size});\\
&\texttt{P\_store}(\texttt{cl}',\texttt{Type},\text{“}\texttt{Closure}\text{”});\\
&\texttt{P\_store}(\texttt{cl}',\texttt{Functionname},\texttt{f});\\
&\texttt{P\_store}(\texttt{cl}',\texttt{Fastcallname},\texttt{f}_{\texttt{fc}});\\
&\texttt{P\_store}(\texttt{cl}',\texttt{Arity},\texttt{n}');\\
&\texttt{P\_store}(\texttt{cl}',\texttt{Chain},\texttt{ch}');\\
\texttt{n}' &= -(\texttt{n},1);\\
\texttt{ch}' &= \texttt{Ap}_2(\texttt{Arg\_chain},\texttt{X},\texttt{ch});\\
&\ \texttt{In cl}'\}
\end{aligned}
$$

where the function to build argument chains is defined as follows:

$$
\begin{aligned}
\texttt{Arg\_chain} = \lambda\,(\texttt{X},\texttt{Xs})\,.\ \ &\{\texttt{xs}' = \texttt{Allocate}(2);\\
&\texttt{P\_store}(\texttt{xs}',\texttt{Arg},\texttt{X});\\
&\texttt{P\_store}(\texttt{xs}',\texttt{Rest},\texttt{Xs});\\
&\ \texttt{In xs}'\}
\end{aligned}
$$

The argument chain can be destructured using the following function:

$$
\begin{aligned}
\texttt{Args}_{\texttt{n}} = \lambda\,(\texttt{X})\,.\,\{_{\texttt{n}}\ \ \texttt{t}_1 &= \texttt{P\_select}(\texttt{X},\texttt{Chain});\\
\texttt{a}_{\texttt{n}} &= \texttt{P\_select}(\texttt{t}_1,\texttt{Arg});\\
\texttt{t}_2 &= \texttt{P\_select}(\texttt{t}_1,\texttt{Rest});\\
\texttt{a}_{\texttt{n}-1} &= \texttt{P\_select}(\texttt{t}_2,\texttt{Arg});\\
\texttt{t}_3 &= \texttt{P\_select}(\texttt{t}_2,\texttt{Rest});\\
&\ \vdots\\
\texttt{a}_1 &= \texttt{P\_select}(\texttt{t}_{\texttt{n}-1},\texttt{Arg});\\
&\ \texttt{In}\ \vec{\texttt{a}_{\texttt{n}}}\}
\end{aligned}
$$

These three definitions must be included in D.

Now we can give the translation for the apply operator. As stated earlier, the apply basically checks to see if the arity is satisfied and either makes a new closure or calls Ap.

```
TE⟦Apply(F,X)⟧ = { n       = P_select(F,Arity);
                   fire_b  = Equal?(n,1);
                   fire_i  = BooltoInt(fire_b);
                   res     = Dispatch_{2,2}(fire_i,
                               { fun  = P_select(F,Functionname);
                                 as   = P_select(F,Chain);
                                 as'  = Ap_2(Arg_chain,X,as);
                                 res' = Ap(fun,as');
                                  In res'},
                               Ap_2(Make_closure,F,X))
                    In res}
```

Notice that the building of closures corresponds to the $\texttt{Apply}_i$ combinators. The first closure for a function, which corresponds to the $\texttt{Apply}_1$ combinator, is processed during the translation of λ-expressions, as follows:

```
TE⟦λ_n (X_n→) . E⟧ = {cl = Allocate(Closure_size);
                      P_store(cl,Type,"Closure");
                      P_store(cl,Functioname,'T_c);
                      P_store(cl,Fastcallname,'T_fc);
                      P_store(cl,Arity,n);
                      P_store(cl,Chain,"End");
                       In cl}
```

The following two function definitions are included in D.

$$\texttt{T}_c = \lambda_1\ (\texttt{Xs})\ .\ \{\ \vec{\texttt{X}}_n\ =\ \texttt{Ap}_{1,n}(\texttt{Args}_n,\texttt{Xs});\quad \texttt{t}\ =\ \texttt{TE}[\![\texttt{E}]\!];\quad \texttt{In t}\}$$

$$\texttt{T}_{fc} = \lambda_n\ (\vec{\texttt{X}}_n)\ .\ \texttt{TE}[\![\texttt{E}]\!]$$

'T_c indicates the name T_c and not the value associated with T_c. Note that $\texttt{TE}[\![E]\!]$ can be computed once and shared between the curried and the fastcall version of the function.

3.2 Signals generation

In P-TAC we can express some more low-level machine concerns, such as the generation of a signal to indicate that a function call has terminated. Signals are needed, for example, for deallocating frame storage. Signals are also needed to express sequencing in the presence of

M-structures in Id. Before introducing signals, the P-TAC program is canonicalized, that is, all blocks are flattened and variables and values are substituted. Furthermore, dead code should be eliminated. We add signals only to non-strict combinators, and to combinators that produce side-effects, such as `P_store`. The output of a strict operator can be interpreted as a signal that the instruction has indeed fired. We give the signal transformation using the translation functions S, SE and SC. The transformation is also applied to each constant definition in D.

$$
\begin{array}{l}
\mathbf{S}[\![\lambda_{n,m}\,(\vec{X_n}).\{_m\ Y_1 = Se_1 = \lambda_{n,m+1}\,(\vec{X_n})\,. \\
\quad\vdots \qquad\qquad\qquad\quad (\{_{m+1}\ Y_1 = Se_1 \\
\quad Y_n = Se_n \qquad\qquad\qquad\quad \vdots \\
\quad Y_{n+1} = Nse_1 \qquad\qquad\quad Y_n = Se_n \\
\quad\vdots \qquad\qquad\qquad\quad Y_{n+1},\ S_1 = SE[\![Nse_1]\!] \\
\quad Y_{n+m} = Nse_m \qquad\qquad\quad \vdots \\
\qquad C_1 \qquad\qquad\qquad\quad Y_{n+m},\ S_m = SE[\![Nse_m]\!] \\
\qquad\vdots \qquad\qquad\qquad\quad S_{m+1} = SC[\![C_1]\!]; \\
\qquad C_k \qquad\qquad\qquad\quad \vdots \\
\quad \mathrm{In}\ \vec{R_m}\}]\!] \qquad\qquad\quad S_{m+k} = SC[\![C_k]\!]; \\
\qquad\qquad\qquad\qquad\qquad\quad S' = Sync_{m+k+i}\,(Deadvar,\ \vec{S_{m+k}}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathrm{In}\ \vec{R_m},\ S'\})
\end{array}
$$

Where Se_i stands for an expression involving strict operators, and Nse_i stands for either an applicative or a loop expression. `Deadvar` are the parameters that are not being used in the body of the function.

$$SE[\![WLoop_n(P,B,\vec{Y_n},Y)]\!] = WLoop'_n(P,B,\vec{Y_n},S_p,Y)$$

where S_p is the signal associated with the invocation of the loop predicate.

$$SE[\![Ap_{n,m}(F,\vec{X_n})]\!] = Ap_{n,m+1}(F,\vec{X_n})$$
$$SC[\![P\_store(X,I,Z)]\!] = Ack\_store(X,I,Z)$$

where `Ack_store` is a new P-TAC function symbol of arity 3, which generates a `Signal` when the store actually takes place, that is, when `X`, `I` and `Z` all become values.
The new rewrite rules are:

$$
\begin{array}{lll}
\texttt{WLoop}'_n(\texttt{P},\texttt{B},\vec{\texttt{X}_n},\texttt{S},\texttt{True}) & \longrightarrow & \{_{n+1}\ \vec{\texttt{t}_n},\texttt{S}_b = \texttt{Ap}_{n,n+1}(\texttt{B},\vec{\texttt{X}_n});\\
 & & \quad \texttt{t}_p,\texttt{S}_p = \texttt{Ap}_{n,2}(\texttt{P},\vec{\texttt{t}_n});\\
 & & \quad \texttt{S}' = \texttt{Sync}_3(\texttt{S},\texttt{S}_b,\texttt{S}_p);\\
 & & \quad \vec{\texttt{t}'_n},\texttt{S}_1 = \texttt{WLoop}'_n(\texttt{P},\texttt{B},\vec{\texttt{t}_n},\texttt{S}',\texttt{t}_p)\\
 & & \quad \texttt{In}\ \vec{\texttt{t}'_n},\ \texttt{S}_1\}\\
\texttt{WLoop}'_n(\texttt{P},\texttt{B},\vec{\texttt{X}_n},\texttt{S},\texttt{False}) & \longrightarrow & \vec{\texttt{X}_n},\ \texttt{S}\\
\texttt{Ack\_store}(\texttt{L},\underline{\texttt{i}},\texttt{V}) & \longrightarrow & \{\texttt{t} = \texttt{Signal};\\
 & & \quad \texttt{P\_store}(\texttt{L},\underline{\texttt{i}},\texttt{V});\\
 & & \quad \texttt{In t}\}\\
\texttt{Sync}_n(\vec{\texttt{V}_n}) & \longrightarrow & ()
\end{array}
$$

Sync produces a void value when all the signals are received.

4 Conclusions

This paper has informally described the compilation process of Id. Following these lines a project "Id-in-Id compiler" led by Shail Aditya and Yuli Zhou is under progress. Besides easing the portability of Id, the Id compiler in Id will allow us to study the implicit parallelism in the Id compiler.

We have not discussed the operational semantics of M-structures, even though we think it is quite straightforward. However, the definition of a *term model* in the presence of non-determinism is a difficult issue, and has not been investigated yet.

We also plan to formalize the compilation process beyond P-TAC, for both parallel and sequential machines. For example, the notion of a *frame* to hold the temporary variables for a function application or loop iteration, can be abstracted in a useful way for most machines. We think this will facilitate the study of issues related to reuse of frames, storing of loop constants, and pre-allocation of multiple frames for parallel execution. Similarly, the analysis required for detecting sequential threads can also be performed in a machine independent manner.

Acknowledgments

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988 (MIT) and N00039-88-C-0163 (Harvard). This paper has benefitted by informal discussions with the members of Computation Structures Group. In particular, we would like to thank Id compiler gurus Jamey Hicks and Shail Aditya Gupta. Thanks to Paul Barth, Steve Glim, Derek Chiou and Boon Ang for proof reading the final draft of this paper.

References

[1] A. Aho, J. Ullman, and R. Sethi. *Compilers: Principles, Techniques, Tools.* London, Addison-Wesley, 1986.

[2] Z. M. Ariola. Orthogonal Graph Rewriting Systems. Technical Report CSG Memo 323, MIT Laboratory for Computer Science, 1991.

[3] Z. M. Ariola and Arvind. Compilation of Id$^-$: a Subset of Id. Technical Report CSG Memo 315, MIT Laboratory for Computer Science, November 1990.

[4] Z. M. Ariola and Arvind. A Syntactic Approach to Program Transformations. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, Yale University*, 1991.

[5] Arvind, R. Nikhil, and K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, Springer-Verlag LNCS 279*, pages 336–369, September/October 1987.

[6] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Chalmers University of Technology, Department of Computer Science, 1987.

[7] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a Parallel, Non-strict, Functional Language with state. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge*, 1991.

[8] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. Conf. on Functional Programming Languages and Computer Architecture, Nancy, France*, September 1985.

[9] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, Department of Computer Science, 1987.

[10] J. Klop. Term Rewriting Systems. Course Notes, Summer course organized by Corrado Boehm, Ustica, Italy, September 1985.

[11] P. Landin. A Correspondence between Algol60 and Church's Lambda notation. *Communications ACM*, 8, 1965.

[12] J.-J. Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul.* Ph.D. thesis, Université Paris VII, October 1978.

[13] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[14] S. L. Peyton Jones. *The implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, N.J., 1987.

[15] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In *Proceedings of the 18th ACM Symposium on Principle of programming Languages*, pages 67–78, January 1991.

[16] K. Schauser, D. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge*, 1991.

[17] K. Traub. Compilation as Partitioning: A New Approach to Compiling Non-strict Functional Languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge*, 1991.

[18] P. Welch. Continuous Semantics and Inside-out Reductions. In *λ-Calculus and Computer Schience Theory, Italy (Springer-Verlag Lecture Notes in Computer Science 37)*, March 1975.

[19] *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, Yale University, New Haven, CN.* June 1991.