January 27, 1995

Compilation of Id$^-$: A Subset of Id *

Zena M. Ariola
Aiken Computational Laboratory
Harvard University

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology

January 27, 1995

Abstract

Compilation of Id, a higher-order non-strict functional language augmented with I-structures, is described in terms of two languages. The first language is called Kid, which is a Kernel language for Id. Kid is further translated into a language where all Kid data structures and functions are represented using only one data structure. This second language is called P-TAC for Parallel Three Address Code. The operational semantics of both Kid and P-TAC are presented in terms of two *contextual reduction systems*. Formalization of many commonly known optimizations is also presented in the contextual reduction framework.

Keywords and phrases: Term Rewriting Systems, Contextual Reduction Systems, Dataflow, Functional Languages, I-structures, Compiler Optimizations.

*This technical report is created by merging CGS memo 315 (dated November 1 1990) and CSG memo 322 (dated December 10 1990)

1 Introduction

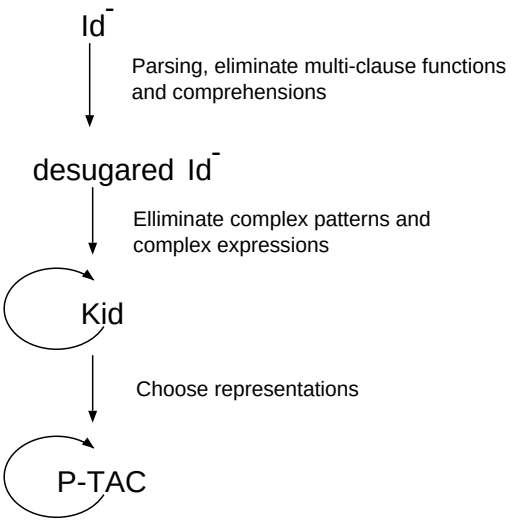


Figure 1: Compilation scheme

We describe the compilation of Id$^-$, a subset of the Id language, in terms of successive translations into different languages. Each of the successive languages has a precise operational semantics, which is given in terms of rewrite rules. The compilation scheme is shown in Figure 1. Id$^-$ does not contain the following features of Id:

abstract types and algebraic types except lists;

overloading;

mutable structures.

Id$^-$ is first parsed into de-sugared Id$^-$, where enough parentheses are inserted to make operator associativity and precedence irrelevant. Desugared Id$^-$ also does not contain patterns in function definitions. During the de-sugaring phase list-comprehensions and array-comprehensions are transformed into nested loops. We will informally describe this phase in the next Section.

In the next phase de-sugared Id$^-$ is translated into Kid, the Kernel language for Id. Kid is a subset of Id except for the notion of *multiple values* which have been inspired by dataflow graphs. Essentially a Kid program contains only primitive patterns and simple expressions. The Kid syntax and operational semantics are given in Section 3. The translation from de-sugared Id$^-$ to Kid, including the compilation of complex patterns into primitive patterns, is described in Section 4.

Kid serves several purposes. Since Id is too complex to be given direct operational semantics, it is preferable first to give a translation of Id into a smaller language such as Kid. Thus, Id is defined indirectly in terms of the rewrite rule semantics of Kid. Kid is also the intermediate language in which a large number of architecture independent compiler optimizations are performed. These optimizations are described in Section 7.1. We think that many other types of analyses such as Milner style type checking, or abstract interpretation for storage reclamation should be performed at the Kid level. These analyses are not described in this document.

An implementation method that is preferred by many researchers eliminates explicit environments containing the value of free variables of a function. This requires applying a program transformation known as lambda-lifting. Lambda-lifting involves two steps: first, all nested function definitions (λ-expressions) of a Kid program are "closed" by turning their free variables into parameters. In the second step, all closed definitions are lifted to the top level. Lambda-lifting should be done after optimizations have been performed on Kid programs. There are many algorithms for doing lambda-lifting and we do not describe any of them in this document. Interested reader is referred to Johnsson's paper [14].

The next step is to translate a lambda-lifted Kid program into P-TAC [1], which is a much smaller language than Kid. This translation calls for choosing a machine representation for each type of Kid value and thus, involves many low level efficiency issues. P-TAC and its operational semantics are given in Section 8.1. A translation from Kid into P-TAC is given in Section 9. We end the document by showing in Section 10 how a P-TAC program is extended with Signals [7] which are needed to detect termination and

for resource management.

Our compiling approach allows the formalization of questions related to correctness [1]. For example, it will make sense to talk about the correctness of lambda-lifting, or the correctness of the translation from Kid to P-TAC. Moreover, since there exists an independent operational semantics of Id$^-$ in terms of a even smaller subset of kernel Id [12], it may be possible to prove the correctness of the translation between de-sugared Id$^-$ and Kid. Another advantage of our approach is that we are able to delay the introduction of concepts tied to specific machines or even to the dataflow computation model until we actually generate machine code. However, in this document we do not go below the P-TAC level.

2 Desugared Id$^-$

The syntax of de-sugared Id$^-$ is described in Figure 2. As can be seen from the grammar, de-sugared Id is fully parenthesized and thus has no operator precedence. Thus, an Id expression like

```
(f x) : map f xs
```

becomes

```
((f x):((map f) xs))
```

Id has curried versions of infix binary operators which are written by enclosing the operator in parenthesis. Thus, `(+)` represents the curried `+`, and it makes sense to write `((+) 2)` in Id. In de-sugared Id$^-$ we treat the curried version of an operator as a constant, which has a standard function definition associated with it. For the sake of symmetry we have also included curried versions of all unary operators in de-sugared Id$^-$.

During the de-sugaring phase multi-clause functions, are turned into equivalent case expressions, as shown in the example below

$$
\begin{array}{lcl}
\mathit{UOP} & \in & \text{Unary Operator} \\
\mathit{BOP} & \in & \text{Binary Operator} \\
E & \in & \text{Expression} \\
P & \in & \text{Pattern}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Variable} & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \mid F \mid G \mid \cdots \\
\mathit{Constant} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid () \mid \mathsf{Nil} \mid (\mathsf{Not}) \mid \cdots \mid (+) \mid (-) \mid \cdots \mid \mathsf{Make_nD_array} \\
\mathit{UOP} & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{nD_Bounds} \mid \mathsf{I_nD_array} \mid \mathsf{Open_cons} \\
\mathit{BOP} & ::= & + \mid - \mid * \mid \cdots \mid == \mid \mathsf{Eq?} \mid < \mid \cdots \mid \mathsf{And} \mid : \\
E & ::= & \mathit{Variable} \mid \mathsf{Next}\ \mathit{Variable} \mid \mathit{Constant} \\
 & & \mid (\mathit{UOP}\ E) \mid (E\ \mathit{BOP}\ E) \mid (E\ E) \mid E[E] \mid E.\mathsf{cons_1} \mid E.\mathsf{cons_2} \\
 & & \mid (E, E) \mid (E, E, E) \mid \cdots \\
 & & \mid (\mathsf{If}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E) \mid \{\mathsf{Case}\ E\ \mathsf{of}\ \mathit{Clause}\ [\mid\ \mathit{Clause}]^*\ \} \\
 & & \mid \{\mathsf{While}\ E\ \mathsf{do}\ [\mathit{Statement};]^*\ \mathsf{Finally}\ E\} \\
 & & \mid \{\mathsf{For}\ \mathit{Variable} \leftarrow E\ \mathsf{to}\ E\ \mathsf{step}\ E\ \mathsf{do}\ [\mathit{Statement};]^*\ \mathsf{Finally}\ E\} \\
 & & \mid \{\mathsf{Fun}\ [\mathit{Variables}]^* = E\} \mid \{\underline{\mathsf{Fun}}\ [\mathit{Variables}]^* = E\} \\
 & & \mid \mathit{Block} \\
\mathit{Block} & ::= & \{[\mathit{Statement};]^*\ \mathsf{In}\ E\} \\
\mathit{Statement} & ::= & \mathit{Binding} \mid \mathit{Command} \\
\mathit{Binding} & ::= & P = E \\
\mathit{Command} & ::= & E[E] = E \mid E.\mathsf{cons_1} = E \mid E.\mathsf{cons_2} = E \\
\mathit{Clause} & ::= & P = E \\
P & ::= & \mathsf{Nil} \mid \mathit{Variable} \mid \mathsf{Next}\ \mathit{Variable} \mid (P, P) \mid (P, P, P) \mid \cdots \mid (P : P) \\
\mathit{Program} & ::= & \mathit{Block}
\end{array}
$$

Figure 2: Grammar of de-sugared Id$^-$

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

becomes

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

where t1 and t2 are new variables. Furthermore, all function definitions are turned into λ - expressions.

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

If the Id function is defined as substitutable (Defsubst) then the corresponding λ-expression (Fun) is underlined. The "underlined" λ-expressions are then expanded in line during optimizations (see Section 7.1).

The most complex part of the de-sugaring phase is the transformation of comprehensions into loops. This is described in detail in [9]. Here we only illustrate the idea using the following example:

```
{: e || x <- xs ; y <- ys}
```

A typical translation of this list-comprehension is given in terms of nested map-list operations followed by a list flattening operation [13]. In Id we make use of "open lists", a type

of I-structure, to generate a tail recursive program. Basically, in the following program, a open list (signified by `h` in the inner loop) is generated for each element of `xs` and then these open lists are "glued" together in the outer loop.

```
{ h1  =  Open_cons ();
  hn  = { For x <- xs do
            Next h1 =
               { Case ys of
                   | Nil = h1
                   | y:yss =
                        { h =  Open_cons ();
                          h1.cons_2 = h.cons_2;
                         In { For y <- ys do
                                t  = Open_cons ();
                                t.cons_1 = e;
                                h.cons_2 = t;
                                Next h = t;
                              Finally h }}};
          Finally h1 };
   hn.cons_2 = Nil;
  In h1.cons_2 }
```

The above translation is correct if we treat open lists and lists as the same type. However, there are several subtle issues regarding type checking that are still not resolved.

3 Kid : The Kernel Id Language

Id is a high level functional language [10] augmented with a novel data structuring facility known as I-structures [3]. Id, like most modern functional languages, is non-strict, and has higher-order functions and a Milner-style type system. It also has a fairly large syntax to express currying, loops, list and array comprehensions, and pattern matching for all algebraic types. We have found it difficult to give direct operational semantics of Id because of its syntactic complexities. Kid, a kernel language for Id, has been developed to give precise (though indirect) operational semantics of Id. Kid has also proved to be an extremely useful intermediate language for the Id compiler [2]. Within the compiler,

$MV \in$ Multiple Variable

$SE \in$ Simple Expression

$PFi_m \in$ Primitive Function with i arguments and m outputs

$Ap_E_m \in$ Applicative Expression with m outputs

$Case_E_m \in$ Case Expression with m outputs

$Loop_E_m \in$ Loop Expression with m outputs

$lambda_E \in$ Lambda Expression

$E_m \in$ Expression with m outputs

$$
\begin{array}{lcl}
Variable & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \\
MV_m & ::= & \underbrace{Variable, \cdots, Variable}_{m} \\
Constant & ::= & Integer \mid Boolean \mid () \mid (\mathsf{Not}) \mid \cdots \mid (+) \mid (-) \mid \cdots \\
 & & \mid \mathsf{Nil} \mid \mathsf{Make_nD_array} \mid \mathsf{Error} \mid \top \\
SE & ::= & Variable \mid Constant \\
SE_m & ::= & \underbrace{SE, \cdots, SE}_{m} \mid SE, SE_{m-1} \mid SE, SE, SE_{m-2} \mid \cdots \\
PF1_1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{nD_Bounds} \mid \mathsf{I_nD_array} \mid \mathsf{Open_cons} \mid \mathsf{Cons_1} \mid \mathsf{Cons_2} \\
PF1_m & ::= & \mathsf{Detuple_m} \\
PF2_1 & ::= & + \mid - \mid * \mid \cdots \mid \mathsf{Equal?} \mid \mathsf{Eq?} \mid < \mid \cdots \mid \mathsf{And} \mid \mathsf{Cons} \mid \mathsf{Apply} \\
 & & \mid \mathsf{P_nD_select} \mid \mathsf{Make_tuple_2} \\
PFN_1 & ::= & \mathsf{Make_tuple_n} \mid \mathsf{Apply_{n-2}} \\
Ap_E_m & ::= & \mathsf{Ap_{n,m}}\ (SE_{n+1}) \\
Case_E_m & ::= & \mathsf{Bool_case_m}\ (SE,\ E_m,\ E_m) \mid \mathsf{List_case_m}\ (SE,\ E_m,\ E_m) \\
Loop_E_m & ::= & \mathsf{WLoop_m}\ (SE_{m+3}) \mid \mathsf{FLoop_m}\ (SE_{m+4}) \\
lambda_E & ::= & \lambda_{\mathsf{n,m}}\ (MV_n)\ .\ (E_m) \mid \underline{\lambda_{\mathsf{n,m}}}\ (MV_n)\ .\ (E_m) \\
E_1 & ::= & SE_1 \mid PF1_1\ (SE) \mid PF2_1\ (SE_2) \mid PFN_1\ (SE_n) \mid Ap_E_1 \\
 & & \mid Case_E_1 \mid Loop_E_1 \mid lambda_E \mid Block_1 \\
E_m & ::= & SE_m \mid PF1_m\ (SE) \mid Ap_E_m \mid Case_E_m \mid Loop_E_m \mid Block_m \\
Block_m & ::= & \{\mathsf{m}\ [Statement;]^*\ \ \mathsf{In}\ SE_m\} \\
Statement & ::= & Binding \mid Command \\
Binding & ::= & MV_m = E_m \\
Command & ::= & \mathsf{P_nD_store}\ (SE,\ SE,\ SE) \mid \mathsf{Cons_store_1}\ (SE,\ SE) \\
 & & \mid \mathsf{Cons_store_2}\ (SE,\ SE) \mid \mathsf{Store_error} \mid \top_s \\
Program & ::= & Block_1
\end{array}
$$

Figure 3: Grammar of Kid

all machine independent optimizations are expressed as source-to-source program transformations on Kid. We believe I-structures are an essential feature of Kid, and are needed to express the semantics of the purely functional subset of Id.

Kid has only uncurried operators and no complex expressions. A major subset of Kid is simply the λ-calculus with constants and let-blocks. However, unlike other functional languages, let-blocks play a fundamental role, that is, they can not be eliminated by systematic substitution of variables. The syntax of Kid is given in Figure 3. Every expression, except a block or λ-expression, consists of a function symbol followed by the corresponding number of arguments. *The ordering of statements within a block is irrelevant.*

An important feature of Kid is the concept of *multiple values*. The expression

$$x,\ y = \{_2\ a = \cdots;\ b = \cdots;\ \mathsf{In}\ a,\ b\}$$

is a well-formed-expression, where "$x,\ y$" indicates multiple variables, not to be confused with a 2-tuple in Id [10]. The 2 after the curly brace indicates that two values are to be returned by this block expression. Multiple values avoid packaging values in a data structure, and are useful in expressing some optimizations. Thus, in Kid a binding has the form $MV = E$, where MV stands for multiple variables. Suppose we have m variables on the left-hand-side, then the expression E on the right-hand-side must return m values. In the sequel we capture the number of values that an expression produces by subscripting the corresponding syntactic category. Thus, to express the above binding we will write $MV_m = E_m$. Note that the function symbol Apply appears as a $PF2_1$ in the grammar because we assume that all user-defined procedures return only one result. We also use subscripted function symbols to express a family of functions. For example, $\mathsf{Make_tuple_n}$ stands for $\mathsf{Make_tuple_2}$, $\mathsf{Make_tuple_3}$, *etc.* . Subscripts in a function symbol do not necessarily represent the number of values to be returned by the application of the function. By convention, we drop the subscript when its value is one.

Kid also contains While-loops and For-loops because these play a significant role in program transformations. These constructs also play a major role when Kid is further

translated into P-TAC.

4 Translating Desugared Id$^-$ into Kid

4.1 Simplification of Expressions

We give the translation in terms of the following functions:

Translate Expression,	`TE` :	Id$^-$ Expression	$\longrightarrow$	Kid Expression
Translate Statement,	`TS` :	Id$^-$ Statement	$\longrightarrow$	list (Kid Statement)
Translate Binding,	`TB` :	Id$^-$ Binding	$\longrightarrow$	list (Kid Binding)
Translate Operator,	`TO` :	Id$^-$ Operator	$\longrightarrow$	Kid Operator
Pattern Matching,	`PM` :	Case_expression	$\longrightarrow$	Kid Expression

The exact syntax of Case_expression is given in Section 4.3.

We will write $\texttt{TE}[\![E_1]\!] = E_2$, where the expression enclosed in double brackets represents an Id$^-$ expression and E_2 is the corresponding Kid expression. The whole translation is given in terms of syntactic categories. The proper way of reading a translation function such as "$\texttt{TE}[\![(UOP\ E_1)]\!] = \{t_1 = \texttt{TE}[\![E_1]\!];\ t = \texttt{TO}[\![UOP]\!]\ (t_1);\ \texttt{In}\ t\}$ " is that "`TE`" when applied to a unary expression in Id$^-$ produces the Kid expression on the right-hand-side.

Throughout, the emphasis is on clarity of Id$^-$ to Kid translation rather than its efficiency. We will use the same conventions introduced in Section 6, with the addition of meta-variable P_i that ranges over Patterns. As before $\vec{X}_n$ indicates multiple meta-variables. The lower case variables, such as t_i, x_i, that appear in the translated expression represent new Kid variables. The translation procedure given below does not require lex-

ical scope analysis for variables. However, after the translation is complete a procedure to make all bound variables unique has to be applied before any rewriting can be done.

A common situation in the translation procedure is the need to replace variable X by a new variable t in some Id expression E. This idea can be expressed by addding another block around E as in $\{\ X = t;\ t' = E;\ \mathsf{In}\ t'\ \}$. To avoid clutter, we will write $E\ [t/X]_B$ as a shortand for $\{\ X = t;\ t' = E;\ \mathsf{In}\ t'\ \}$. (As an aside, it should be noted that since we do not assume unicity of variables during translation, we can not use the notion for naive substitution. Conversely, the use of $E\ [\vec{X}_n\ /\ \vec{Z}_n]_B$ in the application rewrite rule, given earlier, would have been incorrect, because it could lead to duplication of bound variables $\vec{Z}_n$, while expanding two different applications of the same function).

TE: Id$^-$ Expression $\longrightarrow$ Kid Expression

$$\mathtt{TE}[\![X]\!] = X$$

$$\mathtt{TE}[\![\mathsf{Next}\ X]\!] = next(X)$$

Where "$next$" is a function on identifiers that keeps the association between a variable and its corresponding nextified version during the loop translation phase. After the translation is complete, $next$(X) is treated like an ordinary identifier which is different from X.

$$\mathtt{TE}[\![C]\!] = C$$

$$\begin{array}{lcl}
\mathtt{TE}[\![(UOP\ E)]\!] & = & \{\ t_1 = \mathtt{TE}[\![E]\!]; \\
 & & \quad t = \mathtt{TO}[\![UOP]\!]\ (t_1) \\
 & & \quad \mathsf{In}\ t\}
\end{array}$$

$$\begin{array}{lcl}
\mathtt{TE}[\![(E_1\ BOP\ E_2)]\!] & = & \{\ t_1 = \mathtt{TE}[\![E_1]\!]; \\
 & & \quad t_2 = \mathtt{TE}[\![E_2]\!]; \\
 & & \quad t = \mathtt{TO}[\![BOP]\!]\ (t_1,\ t_2) \\
 & & \quad \mathsf{In}\ t\}
\end{array}$$

$$
\begin{array}{lcl}
\mathtt{TE}[\![(E_1\ E_2)]\!] & = & \{\ t_1 \ = \ \mathtt{TE}[\![E_1]\!]; \\
& & \ \ t_2 \ = \ \mathtt{TE}[\![E_2]\!]; \\
& & \ \ t \ = \ \mathsf{Apply}\ (t_1,\ t_2) \\
& & \ \ \mathsf{In}\ \ t\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{TE}[\![E_1[E_2]]\!] & = & \{\ t_1 \ = \ \mathtt{TE}[\![E_1]\!]; \\
& & \ \ t_2 \ = \ \mathtt{TE}[\![E_2]\!]; \\
& & \ \ t \ = \ \mathsf{Ap}_2\ (\mathsf{Select},\ t_1,\ t_2) \\
& & \ \ \mathsf{In}\ \ t\}
\end{array}
$$

where Select is a standard function definition in Kid (see Section 4.2).

$$
\begin{array}{lcl}
\mathtt{TE}[\![E.\mathsf{cons_1}]\!] & = & \{\ t_1 \ = \ \mathtt{TE}[\![E]\!]; \\
& & \ \ t \ = \ \mathsf{List_case}\ (t_1,\ \mathsf{Error},\ \mathsf{Cons_1}\ (t_1)) \\
& & \ \ \mathsf{In}\ \ t\}
\end{array}
$$

Similarly for E.cons_2.

$$
\begin{array}{lcl}
\mathtt{TE}[\![(E_1, \cdots, E_n)]\!] & = & \{\ t_1 \ = \ \mathtt{TE}[\![E_1]\!]; \\
& & \ \ \vdots \\
& & \ \ t_n \ = \ \mathtt{TE}[\![E_n]\!]; \\
& & \ \ t \ = \ \mathsf{Make_tuple_n}\ (\vec{t_n}) \\
& & \ \ \mathsf{In}\ \ t\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{TE}[\![(\mathsf{If}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3)]\!] & = & \{\ t_1 \ = \ \mathtt{TE}[\![E_1]\!]; \\
& & \ \ t \ = \ \mathsf{Bool_case}\ (t_1,\ \mathtt{TE}[\![E_2]\!],\ \mathtt{TE}[\![E_3]\!]) \\
& & \ \ \mathsf{In}\ \ t\}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{TE}[\![\{\ \mathsf{Case}\ E\ \mathsf{of} & = & \{\ t_1 = \texttt{TE}[\![E]\!]; \\
\quad \mid\ P_1 = E_1 & & \quad t = \texttt{PM}[\![\{\ \mathsf{Case}\ (t_1)\ \mathsf{of} \\
\quad\ \vdots & & \qquad \mid\ P_1 = \texttt{TE}[\![E_1]\!] \\
\quad \mid\ P_n = E_n\ \}]\!] & & \qquad\ \vdots \\
 & & \qquad \mid\ P_n = \texttt{TE}[\![E_n]\!]\ \}]\!] \\
 & & \mathsf{In}\ \ t\}
\end{array}
$$

Where PM is described in the Section 4.3.

$$
\begin{array}{lcl}
\texttt{TE}[\![\{\ \mathsf{While}\ E\ \mathsf{do} & = & \texttt{TSLE}[\![\{\ \mathsf{While}\ \texttt{TE}[\![E]\!]\ \mathsf{do} \\
\quad S_1; & & \qquad \texttt{TS}[\![S_1]\!]; \\
\quad\ \vdots & & \qquad\ \vdots \\
\quad S_n; & & \qquad \texttt{TS}[\![S_n]\!]; \\
\quad \mathsf{Finally}\ E_f\ \}]\!] & & \qquad \mathsf{Finally}\ \texttt{TE}[\![E_f]\!]\ \}]\!]
\end{array}
$$

Note that TE uses an auxiliary function TSLE which stands for "Translate simple loop expression". However this is only done for clarity of exposition. In fact, we are slightly abusing our notation because the expression inside TSLE[\![ ]\!] is a mixture of Id$^-$ and Kid syntax.

$$
\begin{array}{lcl}
\texttt{TSLE}[\![\{\ \mathsf{While}\ E\ \mathsf{do} & = & \{\ p = \underline{\lambda_n}\ (\vec{X_n})\ .\ (E); \\
\quad next(X_1) = E_1; & & \quad b = \underline{\lambda_{n,n}}\ (\vec{X_n})\ .\ (\{\mathsf{n}\ next(X_1) = E_1; \\
\quad\ \vdots & & \qquad\ \vdots \\
\quad next(X_n) = E_n; & & \qquad next(X_n) = E_n; \\
\quad Y_1 = E_{y_1}; & & \qquad Y_1 = E_{y_1}; \\
\quad\ \vdots & & \qquad\ \vdots \\
\quad \mathsf{Finally}\ E_f\ \}]\!] & & \qquad \mathsf{In}\ next(X_1), \cdots, next(X_n)\ \}); \\
 & & t_p = \mathsf{Ap_n}\ (p,\ \vec{X_n}); \\
 & & \vec{t_n} = \mathsf{WLoop_n}\ (p,\ b,\ \vec{X_n},\ t_p); \\
 & & t_f = E_f\ [\vec{t_n}\ /\ \vec{X_n}]_B \\
 & & \mathsf{In}\ t_f\}
\end{array}
$$

Notice that the correspondence between the formal parameters of the procedure "b" and the multiple value returned is not accidental. It will be wrong to have $(X_1, \cdots, X_n)$ as

input and $(next(X_n), \cdots, next(X_1))$ as output, because the values of the nextified variables come either from the surrounding scope or from the previous iteration. The λ-expressions corresponding to the predicate and loop body are underlined indicating the fact that they can be inlined at compile time.

$$
\begin{array}{l}
\mathtt{TE}[\![\{\ \mathsf{For}\ X\ \leftarrow\ E_i\ \mathsf{to}\ E_b\ \mathsf{step}\ Es\ \mathsf{do} \\
\qquad S_1; \\
\qquad \vdots \\
\qquad S_n; \\
\qquad \mathsf{Finally}\ E_f\ \}]\!] \\
\end{array}
\quad = \quad
\begin{array}{l}
\mathtt{TSLE}[\![\{\ \mathsf{For}\ X\ \leftarrow\ \mathtt{TE}[\![E_i]\!]\ \mathsf{to}\ \mathtt{TE}[\![E_b]\!]\ \mathsf{step}\ \mathtt{TE}[\![E_s]\!]\ \mathsf{do} \\
\qquad \mathtt{TS}[\![S_1]\!]; \\
\qquad \vdots \\
\qquad \mathtt{TS}[\![S_n]\!]; \\
\qquad \mathsf{Finally}\ \mathtt{TE}[\![E_f]\!]\ \}]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathtt{TSLE}[\![\ \{\mathsf{For}\ X_1\ \leftarrow\ E_i\ \mathsf{to}\ E_b\ \mathsf{step}\ E_s\ \mathsf{do} \\
\qquad next(X_2) = E_2; \\
\qquad \vdots \\
\qquad next(X_n) = E_n; \\
\qquad Y_1 = E_{y_1}; \\
\qquad \vdots \\
\qquad \mathsf{Finally}\ E_f\ \}]\!]
\end{array}
\quad = \quad
\begin{array}{lcl}
\{\ b & = & \underline{\lambda_{n,n-1}}\ (\vec{X_n})\ . \\
 & & \quad (\{_{\mathsf{n-1}}\ next(X_2) = E_2; \\
 & & \qquad \vdots \\
 & & \qquad next(X_n) = E_n; \\
 & & \qquad Y_1 = E_{y_1}; \\
 & & \qquad \vdots \\
 & & \qquad \mathsf{In}\ next(X_2), \cdots, next(X_n)\ \}); \\
t_i & = & E_i; \\
t_b & = & E_b; \\
t_s & = & E_s; \\
t_p & = & \leq\ (t_i,\ t_b); \\
\vec{t_n} & = & \mathsf{FLoop_n}\ (t_b,\ t_s,\ b,\ t_i,\ \vec{X_{2,n}},\ t_p); \\
t_f & = & E_f\ [\vec{t_n}\ /\ \vec{X_n}]_B \\
 & & \mathsf{In}\ t_f\}
\end{array}
$$

$$
\mathtt{TE}[\![\{\mathsf{Fun}\ \vec{X_n} = E\}]\!] \quad = \quad \lambda_{\mathsf{n}}\ (\vec{X_n})\ .\ (\mathtt{TE}[\![E]\!])
$$

$$
\mathtt{TE}[\![\{\underline{\mathsf{Fun}}\ \vec{X_n} = E\}]\!] \quad = \quad \underline{\lambda_{\mathsf{n}}}\ (\vec{X_n})\ .\ (\mathtt{TE}[\![E]\!])
$$

$$\begin{array}{lcl}
\texttt{TE}[\![\{S_1;\cdots S_n;\ \mathsf{In}\ E\}]\!] & = & \{\ \texttt{TS}[\![S_1]\!]; \\
 & & \quad\vdots \\
 & & \texttt{TS}[\![S_n]\!]; \\
 & & t = \texttt{TE}[\![E]\!] \\
 & & \mathsf{In}\ \ t\ \}
\end{array}$$

TS: Id$^-$ Statement $\longrightarrow$ list (Kid Statement)

Often an Id$^-$ statement translates into a group of Kid statements. We will enclose the translated statement or statements within parenthesis even though parenthesis are not part of Kid syntax. These parenthesis do not introduce a new lexical scope.

$$\texttt{TS}[\![P = E]\!] \ = \ (t = \texttt{TE}[\![E]\!];\ \texttt{TB}[\![P = t]\!])$$

$$\begin{array}{lcll}
\texttt{TS}[\![E_1[E_2] = E_3]\!] & = & (\ t_1 & = \ \texttt{TE}[\![E_1]\!]; \\
 & & \quad t_2 & = \ \texttt{TE}[\![E_2]\!]; \\
 & & \quad t_3 & = \ \texttt{TE}[\![E_3]\!]; \\
 & & \quad t & = \ \mathsf{Ap_3}\ (\mathsf{Store},\ t_1,\ t_2,\ t_3))
\end{array}$$

where Store is a standard function defined in Kid (see Section 4.2).

$$\begin{array}{lcll}
\texttt{TS}[\![E_1.\mathsf{cons_1} = E_2]\!] & = & (\ t_1 & = \ \texttt{TE}[\![E_1]\!]; \\
 & & \quad t_2 & = \ \texttt{TE}[\![E_2]\!]; \\
 & & \quad t & = \ \mathsf{List_case}\ (t_1,\ \{\ \mathsf{Store_error\ In}\ ()\},\{\ \mathsf{Cons_store_1}\ (t_1,\ t_2)\ \mathsf{In}\ ()\ \}\ )\ )
\end{array}$$

Similarly for cons_2.

TB: Id$^-$ Bindings $\longrightarrow$ list (Kid Binding)

$$\begin{array}{lcl}
\texttt{TB}[\![(P_1,\cdots,P_n) = X]\!] & = & (\vec{t_n} = \mathsf{Detuple_n}\ (X); \\
 & & \quad \texttt{TB}[\![P_1 = t_1]\!]; \\
 & & \quad\vdots \\
 & & \quad \texttt{TB}[\![P_n = t_n]\!]\ )
\end{array}$$

$$\texttt{TB}[\![(P_1 \ : \ P_2) = X]\!] \ = \ (t_1,\ t_2 = \mathsf{List_case}_2\ (X,\ (\ \mathsf{Error},\ \mathsf{Error}\ ),\ (\mathsf{Cons_1}\ (X),\ \mathsf{Cons_2}\ (X))\ );$$
$$\texttt{TB}[\![P_1 = t_1]\!];$$
$$\texttt{TB}[\![P_2 = t_2]\!])$$

$$\texttt{TB}[\![Y = X]\!] \ = \ (Y = X)$$

$$\texttt{TB}[\![\mathsf{Next}\ Y = X]\!] \ = \ (next(Y) = X)$$

TO: Id$^-$ Operator $\longrightarrow$ Kid Operator

$\texttt{TO}[\![UOP]\!] =$ the corresponding Kid $PF1$

$\texttt{TO}[\![BOP]\!] =$ the corresponding Kid $PF2$

The Kid operator corresponding to Id$^-$ ":" is Cons and for "==" is Equal?.

4.2 Definition of Standard Functions

In our translation from Id$^-$ to Kid we have introduced two new functions Select and Store. These are not primitive operators in Kid, therefore we give their definitions. Similarly we give a definition for Make_1D_array. For the sake of brevity we give these definitions in Id$^-$. These definitions and Make_1D_array can be written as follows

```
Make_1D_array = { Fun b f = { (l,u) = b ;
                              a = I_array b;
                              i = l;
                              d = { While i ≤ u  do
                                     a[i] = f i;
                                     Next i = i+1
                                   Finally () }
                              In a } }
```

Like lists we are treating I-arrays and functional arrays as the same type. However, functional arrays have a different degree of polymorphism than I_nD_array. We are ignoring

these subtle issues in the above translation.

```
Select = { Fun x i = { (l,u) =  1D_Bounds x;
                       In
                         If (i > u Or i < l)  then
                                              Error
                                                 else
                                              P_1D_select x i } }

Store = { Fun x i y = { (l,u) =  1D_Bounds x;
                        In
                          If (i > u Or i < l)  then
                                              { Store_error In () }
                                                   else
                                              { P_1D_store x i y In () } } }
```

The above Id$^-$ definitions can be translated into Kid by adding the following rules:

TE[[P_nD_select x i]] = P_nD_select $(x,\ i)$

TS[[P_nD_store x i y]] = P_nD_store $(x,\ i,\ y)$

TE[[Error]] = Error

TE[[Store_error]] = Store_error

4.3 Pattern Matching: Elimination of Complex Patterns

Pattern matching in Id follows a different philosophy than other functional languages such as Miranda and Haskell. In both these languages clauses in patterns are examined from top to bottom and patterns in a clause are examined from left to right. Within the limitations of the above rule, pattern matching does not force the evaluation of a pattern whose type is irrefutable (tuples are examples of irrefutable patterns) [13]. Miranda in addition allows repeated occurrences of a variable in a clause.

Id pattern matching is designed to be order-insensitive and is maximally non-strict for sequential implementations. In Id, patterns must not overlap to allow order-insensitivity.

Id also does not permit repeated variables.

Determination of which variables are necessary to evaluate to resolve patterns is quite tricky as illustrated by the following example (due to Lennart Augustsson):

```
{ Case (x, y, z) of
    |   (1, 0, w) = 1
    |   (0, w, 1) = 2
    |   (w, 1, 0) = 3 }
```

First of all notice that the patterns are non-overlapping, i.e., only one can be true for a given `x`, `y` and `z`. Second, notice that top-to-bottom and left-to-right rule will evaluate `x` and then evaluate `y` or `z` depending upon the value of `x`. Thus, if `x` does not terminate no answer will ever be produced. Notice if `y = 1` and `z = 0` one could demand the answer `3` regardless of the `x`. Traversal of patterns in a different order will force evaluation of different variable.

Id pattern matching rules will force the evaluation of all three variables in this example because no unique sequential order exists. (In Miranda and Haskell, all three variables will be evaluated only if `y` turns out to be `1` or `z` turns out to be `0`).

Now we informally describe a simplified version of the pattern matching algorithm used by the Id compiler [11]. For Id$^-$ we take into consideration lists and tuples only. In the following we describe various cases that arise during pattern matching.

The signature of the pattern matching function is

$$\texttt{PM} : \text{Case_expression} \longrightarrow \text{Kid Expression}$$

where Case-expression is defined as follows

Case_expression	::=	{Case SE of [| Clause]*}
Clause	::=	P = Kid_Expression
P	::=	Nil | Variable | () | (P, P) | (P, P, P) | $\cdots$ | (P : P)
SE	::=	() | (Variable, $\cdots$, Variable)

If the pattern is a variable, say "X", then "X" is equivalent to "(X)".

As was stated earlier, no scope analysis of variable names is required for the translation from Id to Kid. Thus, we will not assume that all bound variable names are unique in the Kid program that is generated by the translation process.

Variable rule

The column of patterns corresponding to a case variable consists only of variables.

$$
\begin{array}{l}
\texttt{PM}[\![\ \{\textsf{Case}\ \ (\vec{X}_{1,i-1},\ \ X_i,\ \ \vec{X}_{i+1,n})\ \ \textsf{of} \\
\qquad\quad |\ \ (\vec{P1}_{1,i-1},\ \ Y_1,\ \ \vec{P1}_{i+1,n})\ \ = E_1 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\quad |\ \ (\vec{Pm}_{1,i-1},\ \ Y_m,\ \ \vec{Pm}_{i+1,n})\ \ = E_m\}]\!] = \\
\\
\quad \{\ t = X_i; \\
\qquad t_f = \texttt{PM}[\![\ \{\textsf{Case}\ \ (\vec{X}_{1,i-1},\ \ \vec{X}_{i+1,n})\ \ \textsf{of} \\
\qquad\qquad\qquad |\ \ (\vec{P1}_{1,i-1},\ \ \vec{P1}_{i+1,n})\ \ = E_1[t/Y_1]_B \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad |\ \ (\vec{Pm}_{1,i-1},\ \ \vec{Pm}_{i+1,n})\ \ = E_m[t/Y_m]_B\}]\!] \\
\qquad \textsf{In}\ t_f\ \}
\end{array}
$$

Note that meta-variables $Y_1 \cdots Y_m$ may be distinct.

Irrefutable pattern rule

The column of patterns corresponding to a case variable consists of at least one tuple and zero or more variables. Without loss of generality in the following rule we assume the tuple to be a 2-tuple.

$$
\begin{array}{lllll}
\texttt{PM}[\![\ \{\mathsf{Case} & (\overrightarrow{X_{1,i-1}}\ , & X_i\ , & \overrightarrow{X_{i+1,n}}) & \mathsf{of} \\
\quad | & (\overrightarrow{P1_{1,i-1}}\ , & Y_1\ , & \overrightarrow{P1_{i+1,n}}) & = E_1 \\
 & & \vdots & & \\
\quad | & (\overrightarrow{Pk_{1,i-1}}\ , & Y_k\ , & \overrightarrow{Pk_{i+1,n}}) & = E_k \\
\quad | & (\overrightarrow{Pl_{l,i-1}}\ , & (Yl_1\ , Yl_2)\ , & \overrightarrow{Pl_{i+1,n}}) & = E_l \\
 & & \vdots & & \\
\quad | & (\overrightarrow{Pm_{1,i-1}}\ , & (Ym_1\ , Ym_2)\ , & \overrightarrow{Pm_{i+1,n}}) & = E_m\}]\!] =
\end{array}
$$

$$
\begin{array}{lll}
\{t_1, t_2 & = & \mathsf{Detuple}_2\ (X_i); \\
t & = & X_i; \\
t_f & = & \texttt{PM}[\![\ \{\mathsf{Case}\ (\overrightarrow{X_{1,i-1}}\ , \quad t_1\ , \quad t_2\ , \quad \overrightarrow{X_{i+1,n}}) \quad \mathsf{of} \\
 & & \quad | \quad (\overrightarrow{P1_{1,i-1}}\ , \quad t_{1,1}\ , \quad t_{1,2}\ , \quad \overrightarrow{P1_{i+1,n}}) \quad = E_1[t/Y_1]_B \\
 & & \qquad\qquad\qquad\qquad \vdots \\
 & & \quad | \quad (\overrightarrow{Pk_{1,i-1}}\ , \quad t_{k,1}\ , \quad t_{k,2}\ , \quad \overrightarrow{Pk_{i+1,n}}) \quad = E_k[t/Y_k]_B \\
 & & \quad | \quad (\overrightarrow{Pl_{l,i-1}}\ , \quad Yl_1\ , \quad Yl_2\ , \quad \overrightarrow{Pl_{i+1,n}}) \quad = E_l \\
 & & \qquad\qquad\qquad\qquad \vdots \\
 & & \quad | \quad (\overrightarrow{Pm_{1,i-1}}\ , \quad Ym_1\ , \quad Ym_2\ , \quad \overrightarrow{Pm_{i+1,n}}) \quad = E_m\}]\!] \\
 & & \mathsf{In}\ t_f\}
\end{array}
$$

Refutable pattern rule

The column of patterns corresponding to a case variable consists of only Nil's and $(Y_1 : Y_2)$'s.

$$
\begin{array}{lllll}
\texttt{PM}[\![\ \{\mathsf{Case} & (\overrightarrow{X_{1,i-1}}\ , & X_i\ , & \overrightarrow{X_{i+1,n}}) & \mathsf{of} \\
\quad | & (\overrightarrow{P1_{1,i-1}}\ , & \mathsf{Nil}\ , & \overrightarrow{P1_{i+1,n}}) & = E_1 \\
 & & \vdots & & \\
\quad | & (\overrightarrow{Pk_{1,i-1}}\ , & \mathsf{Nil}\ , & \overrightarrow{Pk_{i+1,n}}) & = E_k \\
\quad | & (\overrightarrow{Pl_{l,i-1}}\ , & Yl_1 : Yl_2\ , & \overrightarrow{Pl_{i+1,n}}) & = E_l \\
 & & \vdots & & \\
\quad | & (\overrightarrow{Pm_{1,i-1}}\ , & Ym_1 : Ym_2\ , & \overrightarrow{Pm_{i+1,n}}) & = E_m\}]\!] =
\end{array}
$$

$$
\begin{array}{l}
\mathsf{List_case}\ \ (X_i, \\
\qquad \texttt{PM}[\![\ \{\mathsf{Case}\ \ (\overrightarrow{X_{1,i-1}}\ , \quad \overrightarrow{X_{i+1,n}}) \quad \mathsf{of} \\
\qquad\quad | \quad (\overrightarrow{P1_{1,i-1}}\ , \quad \overrightarrow{P1_{i+1,n}}) \quad = E_1 \\
\qquad\qquad\qquad \vdots \\
\qquad\quad | \quad (\overrightarrow{Pk_{1,i-1}}\ , \quad \overrightarrow{Pk_{i+1,n}}) \quad = E_k\}]\!], \\
\\
\qquad \{\ t_h, t_t = \mathsf{Cons_1}\ (X_i),\ \mathsf{Cons_2}\ (X_i); \\
\qquad\quad t_f = \texttt{PM}[\![\ \{\mathsf{Case}\ \ (t_h\ , \quad t_t\ , \quad \overrightarrow{X_{1,i-1}}\ , \quad \overrightarrow{X_{i+1,n}}) \quad \mathsf{of} \\
\qquad\qquad\quad | \quad (Yl_1\ , \quad Yl_2\ , \quad \overrightarrow{Pl_{1,i-1}}\ , \quad \overrightarrow{Pl_{i+1,n}}) \quad = E_l \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\quad | \quad (Ym_1\ , \quad Ym_2\ , \quad \overrightarrow{Pm_{1,i-1}}\ , \quad \overrightarrow{Pm_{i+1,n}}) \quad = E_m\}]\!] \\
\qquad\quad \mathsf{In}\ t_f\}\ )
\end{array}
$$

Note that if the case expression is not exhaustive then an alternative that will raise a run-time error is generated. As, for example:

$$
\begin{array}{lllll}
\texttt{PM}[\![\ \{\mathsf{Case} & X & \mathsf{of} & = & \texttt{PM}[\![\ \{\mathsf{Case} \quad X \quad \mathsf{of} \\
\quad | & Y_1 : Y_2 & = E\}]\!] & & \quad | \quad Y_1 : Y_2 \quad = E \\
 & & & & \quad | \quad \mathsf{Nil} \quad = \mathsf{Error}\}]\!]
\end{array}
$$

$$
\begin{array}{llll}
\texttt{PM}[\![\ \{\mathsf{Case} & X & \mathsf{of} & \\
\quad | & \mathsf{Nil} & = E\}]\!] & =
\end{array}
\quad
\begin{array}{llll}
\texttt{PM}[\![\ \{\mathsf{Case} & X & \mathsf{of} & \\
\quad | & \mathsf{Nil} & = E & \\
\quad | & t_1 : t_2 & = \mathsf{Error}\}]\!] &
\end{array}
$$

Mixed variable-refutable patterns

The column of patterns corresponding to a case variable consists of at least one refutable pattern. In such a case each variable is replaced by all the possible alternatives, as shown below:

$$
\begin{array}{llllll}
\texttt{PM}[\![\ \{\mathsf{Case} & (X_1, & X_2, & X_3 &) & \mathsf{of} \\
\quad | & (\mathsf{Nil}\ , & Y_{1,1} : Y_{1,2}\ , & Y_1 &) & = E_1 \\
\quad | & (Y_{2,1} : Y_{2,2}\ , & Y_2\ , & \mathsf{Nil} &) & = E_2 \\
\quad | & (Y_3\ , & \mathsf{Nil}\ , & Y_{3,1} : Y_{3,2} &) & = E_3\}]\!] =
\end{array}
$$

$$
\begin{array}{llllllll}
\{\ \vec{t_3} & = & \vec{X_3}; & & & & & \\
\quad t_f & = & \texttt{PM}[\![\ \{\mathsf{Case} & (X_1, & X_2, & X_3 &) & \mathsf{of} \\
& & \quad | & (\mathsf{Nil}\ , & Y_{1,1} : Y_{1,2}\ , & \mathsf{Nil} &) & = E_1[t_3/Y_1]_B \\
& & \quad | & (\mathsf{Nil}\ , & Y_{1,1} : Y_{1,2}\ , & t_{1,1} : t_{1,2} &) & = E_1[t_3/Y_1]_B \\
& & \quad | & (Y_{2,1} : Y_{2,2}\ , & \mathsf{Nil}\ , & \mathsf{Nil} &) & = E_2[t_2/Y_2]_B \\
& & \quad | & (Y_{2,1} : Y_{2,2}\ , & t_{2,1} : t_{2,2}\ , & \mathsf{Nil} &) & = E_2[t_2/Y_2]_B \\
& & \quad | & (\mathsf{Nil}\ , & \mathsf{Nil}\ , & Y_{3,1} : Y_{3,2} &) & = E_3[t_1/Y_3]_B \\
& & \quad | & (t_{3,1} : t_{3,2}\ , & \mathsf{Nil}\ , & Y_{3,1} : Y_{3,2} &) & = E_3[t_1/Y_3]_B\}]\!] \\
\quad \mathsf{In}\ t_f\} & & & & & & &
\end{array}
$$

This rule can be applied only after checking that none of other rules apply. It should be noted that patterns can be tested in many different orders, each giving rise to a correct program though non necessarily the same program. The above rules cover all legal cases. If we ever get tuple and list patterns in one column, the compiler will flag it as a type violation.

Empty variable rule

$$
\begin{array}{lllll}
\mathtt{PM}[\![\ \{\mathsf{Case} & () & \mathsf{of} & = & \textbf{Compiler Error ``overlapping patterns''} \\
\quad | & () & = E_1 & & \\
 & \vdots & & & \\
\quad | & () & = E_m\}]\!] & &
\end{array}
$$

$$
\begin{array}{llll}
\mathtt{PM}[\![\ \{\mathsf{Case} & () & \mathsf{of} \quad = & E \\
\quad | & () & = E\}]\!] &
\end{array}
$$

5 Contextual Reduction Systems

A Contextual Reduction System is a pair (A(F, S), R) , where A is a set of terms, and R is a set of context sensitive rules. F includes at least the $\mathsf{Ap_{n,m}}$ primitive. The saliant syntactic features of the terms in A can be seen by the grammar given in Figure 4. This grammar can be related to the Kid grammar by defining F to be +, Apply, WLoop_m, P_select, *etc.* and S to be P_store, Cons_store, *etc.* respectively.

5.1 Basic Definitions

The applicability of a context sensitive rule depends upon both the structure of the term and a precondition on the context in which the term occurs. For example, the context sensitive rule corresponding to the well known rule " $\mathsf{Cons_1}\ (\mathsf{Cons}\ X_1\ X_2) \longrightarrow X_1$" is

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons_1}\ (X) \quad \longrightarrow \quad X_1}$$

where the binding $X = \mathsf{Cons}\ (X_1,\ X_2)$ over the line denotes a precondition, and X, X_1, and X_2 denote meta-variables. A way of reading the above rule is that $\mathsf{Cons_1}\ (X)$ can be rewritten to X_1 if the binding $X = \mathsf{Cons}\ (X_1,\ X_2)$ occurs in the context of $\mathsf{Cons_1}\ (X)$.

A *context* is a term with an arbitrary number of $\Box_E$, expression holes, and $\Box_S$, statement holes. Formally contexts are terms generated by the grammar of Figure 4, where $\Box_E$ and $\Box_S$ are added to *SE* and *Statement* categories, respectively. Thus, given a context

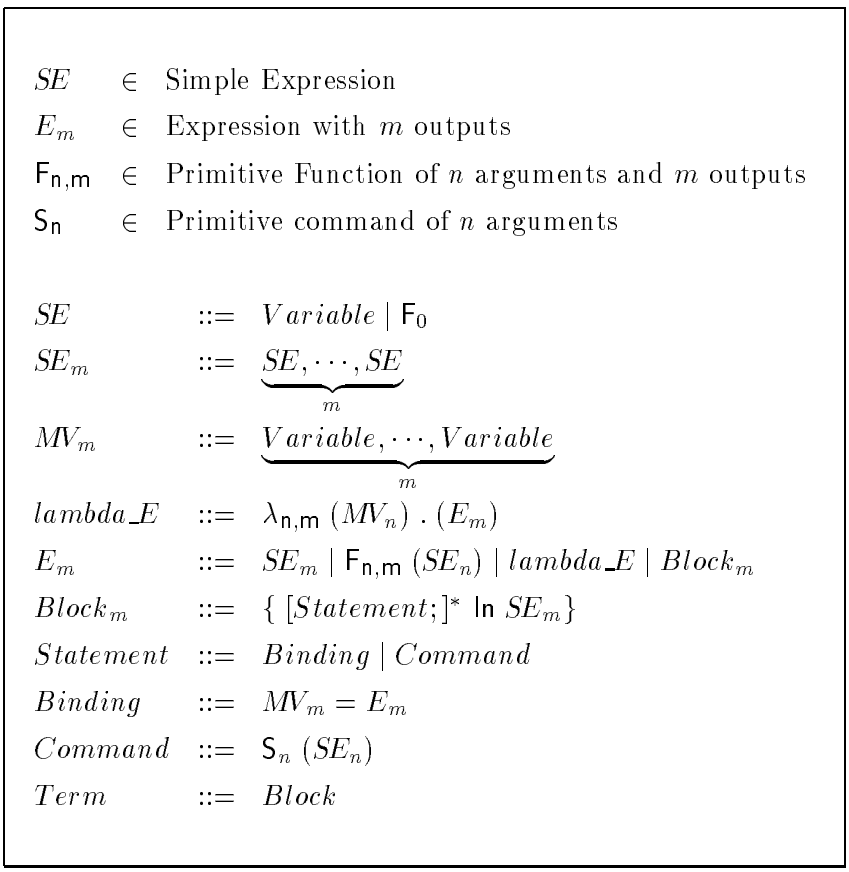


Figure 4: Syntax of terms of a CRS

$C[\Box_S,\ \Box_E]$, and a *substitution*, σ, for the meta-variables X, X_1 and X_2 such that

$$C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ (\mathsf{Cons_1}\ (X))^\sigma]$$

is a term, we say

$$C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ (\mathsf{Cons_1}\ (X))^\sigma] \longrightarrow C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ X_1^\sigma] \qquad (1)$$

where $\longrightarrow$ stands for "rewrites to". From (1) we can observe that the precondition of the rule is not affected by the rewriting, however, the precondition usually affects the outcome of the rewriting.

Definition 5.1 (Context sensitive rule) *A rule is an ordered set of preconditions, $P_1 \cdots P_n$, and a left-hand-side, l, and a right-hand-side, r, and is written as*

$$\frac{P_1 \mid \cdots \mid P_n}{l \longrightarrow r}$$

where P_i is a statement and l and r are expressions.

Definition 5.2 (Redex) *Given a rule* $\dfrac{P_1 \mid \cdots \mid P_n}{l \longrightarrow r}$ *and a term $M \equiv C[S_1, \cdots, S_n, E]$, E is said to be a* **redex** *iff*

(1) $E \equiv l^\sigma$;

(2) $S_j \equiv P_j^\sigma \quad \forall\ j,\ 1 \leq j \leq n$;

where σ is a substitution.

Definition 5.3 (Reduction Relations) *Given terms M, $N \in A$ and a rule $\tau \in R$, M reduces to N in one step ($M \underset{\tau}{\longrightarrow} N$), iff $M \equiv C[P_1^\sigma,\ \cdots, P_n^\sigma,\ l^\sigma]$, and $N \equiv C[P_1^\sigma,\ \cdots, P_n^\sigma,\ r^\sigma]$.*

The one-step rewriting relation $\underset{R}{\longrightarrow} = \bigcup_{\tau \in R}(\underset{\tau}{\longrightarrow})$.

The transitive reflexive closure of $\underset{R}{\longrightarrow}$ is written as $\underset{R}{\twoheadrightarrow}$.

5.2 Basic Rules of Contextual Reduction System's ($R_{\mathbf{CRS}}$)

One has to deal with the problem of free-variable capture in any formal system with bound variables. In the λ-calculus, the problem is usually solved either by making some

variable convention, as for example, assuming that all free variables are different from the bound variables, or by adopting a *variable-free notation* such as that of DeBruijn [6]. According to the variable convention given in [4], the term $(\lambda x.xx)(\lambda x.xx)$ is a legal term, while the term $(\lambda y.(\lambda x.+(x,y))x$ is not a legal term, because the variable x appears both free and bound. The convention allows one to express application in terms of a *naive* form of substitution, which does not require α-renaming at execution time. In a system with let-blocks, in order to avoid the free-variable capture as well as a to allow naive substitution, we have to adopt a even more stringent convention: *all* bound variables in an expression must be unique and different from variables free in the whole program. In order to maintain this invariant we will occasionally rename all bound variables of a subterm to completely new variables explicitly, by applying the function $\mathcal{RB}$ to the subterm. For example,

$$\mathcal{RB}\ [\![\{x = +\ (a,\ 1)\ \mathsf{In}\ x\}]\!] = \{x' = +\ (a,\ 1)\ \mathsf{In}\ x'\}$$

An example of the use of $\mathcal{RB}$ function arises in the use of the application rule (see Section 6.1) as shown below

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Ap_{n,m}}\ (F,\ \vec{X_n}) \ \longrightarrow\ (\mathcal{RB}[\![E]\!])\ [\vec{X_n}\ /\ \vec{Z_n}]}$$

All contextual reduction systems have the following rules, named R_{CRS}.

Substitution rules

$$\frac{X = Y}{X \longrightarrow Y}$$
$$\frac{X = C}{X \longrightarrow C}$$

where meta-variable C stands for a constant, and X and Y stand for distinct variables.

Block Flattening rule

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

Multivariable rule

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

Lemma 5.4 *R_{CRS} is Strongly Normalizing and Confluent.*
Proof: Since there are only a finite number of blocks and occurrences of a variable in a term M, R_{CRS} is strongly normalizing. It is easy to see that the block flattening rules are *locally confluent*. Since variable substitution rules within a flattened block are also *locally confluent*, the uniqueness of the normal form of M follows trivially from Newman's Lemma [4]. ■

5.3 Canonical Forms of terms in a CRS

The notion of α-equivalence between terms as defined in the λ-calculus, makes too fine a distinction in a CRS calculus, for example, the following two Kid terms have apparently different syntactic structure.

$$\{x = 8;\ t' = \{y = x;\ t = x + y\ \mathsf{In}\ t\}\ \mathsf{In}\ t'\} \quad \text{and} \quad \{x = 8;\ y = x;\ t = x + y\ \mathsf{In}\ t\}$$

However, we consider the difference between the above two terms merely "syntactic noise". Eliminating this syntactic noise is the motivation behind the following definitions.

Definition 5.5 (Canonical form) *Let N be the normal form of a term M with respect to R_{CRS}. $\overline{M}$, the canonical form of M, is the term obtained by deleting from N all bindings of the form $x = y$ (where x and y are distinct variables) or $x = v$ (where v is a constant).*

Definition 5.6 (α-equivalence) *Two terms M and N are said to be α-equivalent, if $\overline{M}$ and $\overline{N}$ can be transformed into each other by a consistent renaming of bound variables.*

Lemma 5.7 *Each term has a unique canonical form.*

6 Contextual Reduction System of Kid

We now present a set of rewrite rules, R_{Kid}, which give an intuitive understanding of how Kid terms may get evaluated. If we call A the set of Kid terms generated by the grammar of Figure 3, then the structure $\langle A, R_{Kid} \rangle$ is a Contextual Reduction System. We assume that a primitive function is applied only to arguments of appropriate types, i.e., the type checking has been done statically.

All the variables that appear on the left-hand-side of the rules are meta-variables that range over appropriate syntactic categories. By convention, we use capital letters for meta-variables and small letters for Kid variables. All variables that appear on the right-hand-side of the rules are either meta-variables or "new" Kid variables. We will make use of the following convention regarding meta-variables:

$X_i,\ Z_i,\ Y_i,\ F_i,\ P,\ B,\ U,\ D$	$\in$	Variable and Constant
C	$\in$	Constant
$S_i,\ SS_i,\ S'$	$\in$	Statement
E_i	$\in$	Expression

The notation $E\ [Y/X]$ means the substitution of Y for X in E. Due to our assumptions $E\ [Y/X]$ will simply indicate naive substitution, that is, substitution where no danger of free-variable-capture exists and where X can be replaced by Y without regards to scope. Moreover, we will use the notation $\vec{X}_{n,m}$ to stand for $(X_n, \cdots, X_m)$, $\vec{X}_m$ for $\vec{X}_{1,m}$, and $E\ [\vec{Y}_n\ /\ \vec{X}_n]$ for $E\ [Y_1/X_1, \cdots, Y_n/X_n]$, which is the same as $(\cdots((E\ [Y_1/X_1])\ [Y_2/X_2])\cdots)\ [Y_n/X_n])$.

In the following, $\underline{n}$ represents a numeral.

6.1 Kid Rewrite Rules

δ rules

$$
\begin{array}{lcll}
+\ (\underline{m},\ \underline{n}) & \underset{\delta}{\longrightarrow} & \underline{+(m,\ n)} & \\
 & \vdots & & \\
\mathsf{Equal?}\ (\underline{n},\ \underline{n}) & \underset{\delta}{\longrightarrow} & \mathsf{True} & \\
\mathsf{Equal?}\ (\underline{m},\ \underline{n}) & \underset{\delta}{\longrightarrow} & \mathsf{False} & (\text{if m} \neq n)
\end{array}
$$

Case rules

$$
\begin{array}{lcl}
\mathsf{Bool_case_m}\ (\mathsf{True},\ E_1,\ E_2) & \longrightarrow & E_1 \\
\mathsf{Bool_case_m}\ (\mathsf{False},\ E_1,\ E_2) & \longrightarrow & E_2 \\
\mathsf{List_case_m}\ (\mathsf{Nil},\ E_1,\ E_2) & \longrightarrow & E_1
\end{array}
$$

$$
\frac{X = \mathsf{Open_cons}\ ()}{\mathsf{List_case_m}\ (X,\ E_1,\ E_2) \longrightarrow E_2}
$$

Arity Detection rule

$$
\frac{F = \lambda(Z).E}{\mathsf{Apply}\ (F,\ X) \longrightarrow \mathsf{Ap}\ (F,\ X)}
$$

$$
\frac{F = \lambda_n(\vec{Z}_n).E}{\mathsf{Apply}\ (F,\ X) \longrightarrow \mathsf{Apply_1}\ (F,\ \underline{n},\ X)} \quad n > 1
$$

Similar rules apply for $\underline{\lambda_{\mathsf{n}}}$.

$$
\frac{F_i = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X}_i)}{\mathsf{Apply}\ (F_i,\ X) \longrightarrow \mathsf{Apply_{i+1}}\ (F,\ \underline{n},\ \vec{X}_{i+1})} \quad i < (n-1)
$$

$$
\frac{F_i = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X}_i)}{\mathsf{Apply}\ (F_i,\ X) \longrightarrow \mathsf{Ap_n}\ (F,\ \vec{X}_n)} \quad i = (n-1)
$$

Application rule

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Ap_{n,m}}\ (F,\ \vec{X_n}) \ \longrightarrow\ (\mathcal{RB}[\![E]\!])\ [\vec{X_n}\ /\ \vec{Z_n}]}$$

A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$.

Loop rules

$$\begin{array}{lcl}
\mathsf{WLoop_n}\ (P,\ B,\ \vec{X_n},\ \mathsf{True}) & \longrightarrow & \{_{\mathsf{n}}\ \vec{t_n} = \mathsf{Ap_{n,n}}\ (B,\ \vec{X_n}); \\
 & & \quad t_p = \mathsf{Ap_n}\ (P,\ \vec{t_n}); \\
 & & \quad \vec{t'_n} = \mathsf{WLoop_n}\ (P,\ B,\ \vec{t_n},\ t_p) \\
 & & \quad \mathsf{In}\ \vec{t'_n}\} \\
\mathsf{WLoop_n}\ (P,\ B,\ \vec{X_n},\ \mathsf{False}) & \longrightarrow & \vec{X_n}
\end{array}$$

In the following two rules we assume that the index variable is the first variable in $\vec{X_n}$.

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

Tuple rule

$$\frac{X = \mathsf{Make_tuple_n}\ (\vec{X_n})}{\mathsf{Detuple_n}\ (X) \ \longrightarrow\ \vec{X_n}}$$

List rules

$$\begin{array}{lcl} \mathsf{Cons}\ (X,\ Y) & \longrightarrow & \{\ t\ =\ \mathsf{Open_cons}\ ();\\ & & \quad \mathsf{Cons_store_1}\ (t,\ X);\\ & & \quad \mathsf{Cons_store_2}\ (t,\ Y)\\ & & \quad\ \mathsf{In}\ t\} \end{array}$$

$$\frac{\mathsf{Cons_store_1}\ (X,\ Y)}{\mathsf{Cons_1}\ (X) \longrightarrow Y}$$

$$\frac{\mathsf{Cons_store_2}\ (X,\ Y)}{\mathsf{Cons_2}\ (X) \longrightarrow Y}$$

$$\frac{\mathsf{Cons_store_1}\ (X,\ Y)}{\mathsf{Cons_store_1}\ (X,\ Y') \longrightarrow \top_s}$$

$$\frac{\mathsf{Cons_store_2}\ (X,\ Y)}{\mathsf{Cons_store_2}\ (X,\ Y') \longrightarrow \top_s}$$

Array rules

$$\frac{X = \mathsf{I_array}\ (X_b)}{\mathsf{Bounds}\ (X) \longrightarrow X_b}$$

$$\frac{\mathsf{P_store}\ (X,\ Y,\ Z)}{\mathsf{P_select}\ (X,\ Y) \longrightarrow Z}$$

$$\frac{\mathsf{P_store}\ (X,\ Y,\ Z)}{\mathsf{P_store}\ (X,\ Y,\ Z') \longrightarrow \top_s}$$

Propagation of $\top$

$$\begin{array}{lcl} \{\mathsf{m}\ X = \top;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} & \longrightarrow & \top \\ \{\mathsf{m}\ \top_s;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} & \longrightarrow & \top \end{array}$$

The rules for propagating $\top$ were motivated by a discussion with Vinod Kathail.

Theorem 6.1 *Kid is Confluent upto α-renaming on canonical terms.*
Proof: See [1]. ∎

6.2 Printable Values and Answer of a Kid Term

We now define the printable information associated with a term. The grammar for printable values for Kid is given in Figure 5. A precise notion of printable values is essential to develop an interpreter for Kid as well as to discuss the correctness of optimizations (see Sections 6.3 and 7.3, respectively).

$$
\begin{array}{lcl}
\mathit{Atoms} & ::= & \mathit{Integers} \mid \mathit{Booleans} \mid \mathsf{Error} \mid \text{“}\mathsf{Function}\text{”} \mid \Omega \\
\mathit{List} & ::= & \langle \mathsf{List}\ \mathit{PV}\ \mathit{List} \rangle \mid \mathsf{Nil} \\
\mathit{Tuple} & ::= & \langle \mathsf{2_Tuple}\ \mathit{PV}\ \mathit{PV} \rangle \mid \langle \mathsf{3_Tuple}\ \mathit{PV}\ \mathit{PV}\ \mathit{PV} \rangle \mid \cdots \\
\mathit{Array} & ::= & \langle \mathsf{n_Array}\ \mathit{Tuple}\ \mathit{PV} \cdots \mathit{PV} \rangle \\
\mathit{PV} & ::= & \mathit{Atoms} \mid \mathit{List} \mid \mathit{Tuple} \mid \mathit{Array} \mid \top
\end{array}
$$

Figure 5: Grammar of Printable Values

The following procedure, $\mathcal{P}$, produces the printable value associated with a term. ρ and σ represent, respectively, the list of bindings that have as RHS either a λ-expression or an allocator, *i.e.* Make_tuple, I_array, Open_cons, and the list of store commands, *i.e.* the I-structure store. The procedure $\mathcal{L}$ is used to lookup the value of a variable or a location in ρ and σ, respectively. Given a program, *i.e.* a closed term, M, the $\mathcal{P}rint$ procedure is invoked as follows:

$$\mathcal{P}rint\ (M) = \mathcal{P}\ [\![M]\!]\ \mathsf{Nil}\ \mathsf{Nil}$$

where $\mathcal{P}$ is:

$\mathcal{P}[\![\{S_s;\ A_s;\ B_s\ \mathsf{In}\ X\}]\!]\ \rho\ \sigma = \mathcal{P}[\![X]\!]\ (A_s : \rho)\ (S_s : \sigma)$

$\mathcal{P}[\![\underline{n}]\!]\ \rho\ \sigma = \underline{n}$

$\mathcal{P}[\![\mathsf{True}]\!]\ \rho\ \sigma = \mathsf{True}$

$\mathcal{P}[\![\mathsf{False}]\!]\ \rho\ \sigma = \mathsf{False}$

$\mathcal{P}[\![\top]\!]\ \rho\ \sigma = \top$

$\mathcal{P}[\![\mathsf{Nil}]\!]\ \rho\ \sigma = \mathsf{Nil}$

$$\mathcal{P}[\![X]\!]\,\rho\,\sigma = \begin{cases} \langle \mathsf{n_Tuple}\ \ (\mathcal{P}[\![X_1]\!]\,\rho\,\sigma)\cdots(\mathcal{P}[\![X_n]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \mathsf{Make_tuple_n}(X_1,\cdots,X_n) \\ \langle \mathsf{List}\ \ (\mathcal{P}[\![X.1]\!]\,\rho\,\sigma)\ (\mathcal{P}[\![X.2]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \mathsf{Open_cons}() \\ \langle \mathsf{Array}\ \langle \mathsf{2_Tuple}\ \underline{l}\ \underline{u}\rangle\ (\mathcal{P}[\![X.\underline{l}]\!]\,\rho\,\sigma)\cdots(\mathcal{P}[\![X.\underline{u}]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \mathsf{I_array}(Y) \\ & \text{and } \mathcal{L}(\mathrm{Y},\,\rho) = \mathsf{Make_tuple}\ (\underline{l},\ \underline{u}) \\ \text{“Function”} & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \lambda\ \vec{X_n}\ .(E)\ \text{ or} \\ & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i < (n-1) \\ \Omega & \text{Otherwise} \end{cases}$$

$$\mathcal{P}[\![X.i]\!]\,\rho\,\sigma = \begin{cases} \mathcal{P}[\![Y]\!]\,\rho\,\sigma & \text{if } \mathcal{L}(\mathrm{X.i},\,\sigma) = Y \\ \Omega & \text{if } \mathcal{L}(\mathrm{X.i},\,\sigma) = \text{not found} \end{cases}$$

Notice that the printable value of a finite term can be an infinite tree. For example, the $\mathcal{P}$rint of :

$$\begin{array}{l} \{x = \mathsf{Open_cons}\ (); \\ \ \mathsf{Cons_store_1}\ (x,\ 1); \\ \ \mathsf{Cons_store_2}\ (x,\ x); \\ \ \ \mathsf{In}\ x\} \end{array}$$

is $\langle\ \mathsf{List}\ 1\ \langle\ \mathsf{List}\ 1\ \langle\cdots\rangle\cdots\rangle\ \rangle$, that is, an infinite list of 1's.

We want to define the answer associated with a term in terms of printable values. Furthermore, we want to describe the answer independently of any interpreter, *i.e.* method of computing it. We need to define an ordering on printable values.

Definition 6.2 (Partial Order on Printable Values) *Let a and b be Printable Values; $a \sqsubseteq b$ iff*

(i) $a = \Omega$; or

(ii) $b = \top$; or

(iii) both a and b are Atoms and $a = b$; or

(iv) $a = \langle \mathsf{n_Tuple}\ a_1 \cdots a_n\rangle$ and $b = \langle \mathsf{n_Tuple}\ b_1 \cdots b_n\rangle$ and $a_i \sqsubseteq b_i\ \forall\ 1 \le i \le n$; or

(v) $a = \langle \mathsf{List}\ a_1\ a_2\rangle$ and $b = \langle \mathsf{List}\ b_1\ b_2\rangle$ and $a_1 \sqsubseteq b_1$ and $a_2 \sqsubseteq b_2$; or

(vi) $a = \langle \mathsf{n_Array}\ abounds\ a_1 \cdots a_n\rangle$ and $b = \langle \mathsf{n_Tuple}\ bbounds\ b_1 \cdots b_n\rangle$ and $abounds = bbounds$ and $a_i \sqsubseteq b_i\ \forall\ 1 \le i \le n$.

Theorem 6.3 *PV is a complete partial order with respect to $\sqsubseteq$.*

For the proof see [5].

Lemma 6.4 *Given a term M,* $M \longrightarrow N \implies \mathcal{P}rint\ (M) \sqsubseteq \mathcal{P}rint\ (N)$.

Pictorially:

$$\begin{array}{lllll} M & \longrightarrow & M_1 & \longrightarrow & M_2 \quad \cdots \\ \mathcal{P}rint(M) & \sqsubseteq & \mathcal{P}rint(M_1) & \sqsubseteq & \mathcal{P}rint(M_2) \quad \cdots \end{array}$$

Intuitevely, the answer of a term may be defined as the limit of this chain, that is, the answer is the maximum printable value that can be obtained by reducing a term. We need the following definitions to give a precise definition of "Answer".

Definition 6.5 (Reduction Graph of a term) *Given a term M, the reduction graph of M,* $\mathcal{G}(M)$*, is defined as*

$$\{N \mid M \longrightarrow\!\!\!\!\!\rightarrow N\}.$$

Definition 6.6 (Printable Reduction Graph of a term) *Given a term M, the printable reduction graph of M,* $\mathcal{PG}(M)$*, is defined as*

$$\{\mathcal{P}rint\ (N) \mid M \longrightarrow\!\!\!\!\!\rightarrow N\}.$$

Definition 6.7 (Answer) *Given a term M, the Answer of M,* $\mathcal{P}rint^*(M)$*, is defined as*

$$\sqcup\, \mathcal{PG}(M).$$

In order for the above definition to make sense we need to show that the limit of the printable reduction graph exists and is unique.

Theorem 6.8 $\forall$ *M,* $\mathcal{PG}(M)$ *is directed.*
Proof: Given $a, b \in \mathcal{PG}(\mathrm{M})$, we have:

$$\exists M_1, M_2 \text{ such that } a = \mathcal{P}rint(M_1),\ M \longrightarrow\!\!\!\!\!\rightarrow M_1 \ \wedge\ b = \mathcal{P}rint(M_2),\ M \longrightarrow\!\!\!\!\!\rightarrow M_2.$$

By confluence, $\exists\ M_3$ such that

$$M_1 \longrightarrow\!\!\!\!\!\rightarrow M_3 \ \wedge\ M_2 \longrightarrow\!\!\!\!\!\rightarrow M_3$$

By lemma 6.4,

$$a \sqsubseteq c \ \wedge\ b \sqsubseteq c,\ c = \mathcal{P}rint(M_3)$$

■

Since $\mathcal{PG}(\mathrm{M}) \subseteq PV$ is directed, by theorem 6.3 we know that the limit exists and is unique.

6.3 An Interpreter to Compute the Answer of a Kid term

The set of rewrite rules given in Section 6 per se do not define the operational semantics of Kid, they define a calculus to prove properties of programs. From a computational point of view we need to specify a *strategy* to apply these rules. Furthermore, the reduction strategy should be normalizing with respect to the definition of the answer. Intuitevely the interpreter should stop when it is known that no more printable information can be gotten by further reductions.

The strategy consists in evaluating the outermost redexes first, and in case of a block, evaluating all the right-hand-side redexes in parallel. We will represent a block as $\{Ss;\ As;\ Bs\ \mathsf{In}\ X\}$, where

- Ss represents a set of store commands in the block
- As represents a set of λ-bindings and allocator bindings in the block
- Bs represents a set of bindings, excluding λ-binding and allocators, in the block

The interpreter $\mathcal{E}$ keeps track of store commands it encounters in σ, an I-structure store. It also keeps track of λ-bindings and allocator bindings encountered in ρ. At the start of the evaluation, both ρ and σ are considered to be empty.

The rules for propagating $\top$ imply that whenever new store commands are added to the store a check has to be made for inconsistencies, that is, multiple writes into the same location. We will assume this function is performed by $\mathcal{CC}$. Thus, $\mathcal{CC}\ (\rho \cup Ss)$ returns either a new consistent store or $\top_s$. While evaluating a block, we need to substitute variables and constants and eliminate the corresponding binding. We also need to flatten blocks. Though all nested blocks can be flattened in one go, we need to merge in one step only the block expressions on the RHS of bindings in the top-level block. Any substitution or block flattening can create new redexes on the RHS of bindings in a block. According to the $\top$ propagation rules, if a binding of the form x = $\top$ exists in any block then that block goes to $\top$. This can also be done during the substitution process. We will assume a function $\mathcal{F}\&\mathcal{S}$ (for flatten and substitute) that accomplishes all this. In addition to the

new block, it returns a flag showing if any substitutions or flattening was done. Thus,

$$\begin{aligned}\mathcal{F}\&\mathcal{S}[\![\{X = 1; Y = \{\mathsf{P_store}\ (W,\ I,\ Z);\ Z = X + 3\ \mathsf{In}\ Z\}\ \mathsf{In}\ Y\}]\!] \\ = \ \{\mathsf{P_store}\ (W,\ I,\ Z);\ Z = 1 + 3;\ \mathsf{In}\ Z\}, \mathsf{True}\end{aligned}$$

We will also assume the existence of a function $\mathcal{ERHS}$ which is only applied to a block expression. It may be described informally as

$$\begin{aligned}\mathcal{ERHS}[\![\{Ss;\ As;\ X_1 = E_1; \cdots X_n = E_n\ \mathsf{In}\ X_i\}]\!]\ \rho\ \sigma = \\ \{Ss;\ As;\ X_1 = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma; \cdots X_n = \mathcal{E}[\![E_n]\!]\ \rho\ \sigma\ \mathsf{In}\ X_i\}\end{aligned}$$

Given a program M, the $\mathcal{E}$ procedure is invoked as follows:

$$\mathcal{E}val\ [\![M]\!] = \mathcal{E}[\![M]\!]\ \mathsf{Nil}\ \mathsf{Nil}$$

The Interpreter:

$\mathcal{E}[\![+\ (\underline{m},\ \underline{n})]\!]\ \rho\ \sigma = \underline{+(m,\ n)}$

$\mathcal{E}[\![\mathsf{Equal?}\ (\underline{n},\ \underline{n})]\!]\ \rho\ \sigma = \mathsf{True}$

$\mathcal{E}[\![\mathsf{Equal?}\ (\underline{n},\ \underline{m})]\!]\ \rho\ \sigma = \mathsf{False}$

$\mathcal{E}[\![\mathsf{Bool_case_m}\ (\mathsf{True},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{Bool_case_m}\ (\mathsf{False},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_2]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{List_case_m}\ (\mathsf{Nil},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{List_case_m}\ (X,\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_2]\!]\ \rho\ \sigma \qquad \text{if } \mathcal{L}\ (X, \rho) = \mathsf{Open_cons}\ ()$

$$\mathcal{E}\ [\![\mathsf{Apply}\ (F,\ X)]\!]\ \rho\ \sigma = \begin{cases} \mathcal{E}[\![\mathsf{Ap}\ (F,\ X)]\!]\ \rho\ \sigma & \text{if } \mathcal{L}\ (F,\ \rho) = \lambda Z\ .\ (E) \\ \mathsf{Apply_1}\ (F,\ \underline{n},\ X) & \text{if } \mathcal{L}\ (F,\ \rho) = \lambda_n(\vec{Z_n}).(E)\ \wedge\ n > 1 \\ \mathsf{Apply_{i+1}}\ (F,\ \underline{n},\ \vec{X_{i+1}}) & \text{if } \mathcal{L}\ (F,\ \rho) = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i < (n-1) \\ \mathcal{E}[\![\mathsf{Ap_n}\ (F,\ \vec{X_n})]\!]\ \rho\ \sigma & \text{if } \mathcal{L}\ (F,\ \rho) = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i = (n-1) \end{cases}$$

$\mathcal{E}[\![\mathsf{Ap_{n,m}}\ (F,\ \vec{X_n})]\!]\ \rho\ \sigma = \mathcal{E}[\![(\mathcal{RB}\ [\![E]\!])\ [\vec{X_n}\ /\ \vec{Z_n}]]\!]\ \rho\ \sigma$ where $\mathcal{L}\ (F,\ \rho) = \lambda_{n,m}(\vec{Z_n}).E$

$$\begin{aligned}\mathcal{E}[\![\mathsf{WLoop_n}\ (P,\ B,\ \vec{X_n},\ \mathsf{True})]\!]\ \rho\ \sigma = \mathcal{E}[\![\{\mathsf{n}\ \vec{t_n} &= \mathsf{Ap_{n,n}}\ (B,\ \vec{X_n}); \\ t_p &= \mathsf{Ap_n}\ (P,\ \vec{t_n}); \\ \vec{t'_n} &= \mathsf{WLoop_n}\ (P,\ B,\ \vec{t_n},\ t_p) \\ &\quad \mathsf{In}\ \vec{t'_n}\}]\!]\ \rho\ \sigma\end{aligned}$$

$\mathcal{E}[\![\mathsf{WLoop_n}\ (P,\ B,\ \vec{X_n},\ \mathsf{False})]\!]\ \rho\ \sigma = \vec{X_n}$

$\mathcal{E}\,[\![\mathsf{Detuple_n}\ (X)]\!]\ \rho\ \sigma = \vec{X}_n \qquad$ if $\mathcal{L}\ (X, \rho) = \mathsf{Make_tuple}\ \vec{X}_n$

$\mathcal{E}\,[\![\mathsf{Cons}\ (X,\ Y)]\!]\ \rho\ \sigma = \mathcal{E}[\![\{t = \mathsf{Open_cons}\ ();\ \mathsf{Cons_store_1}\ (t,\ X);\ \mathsf{Cons_store_2}\ (t,\ Y)\ \ \mathsf{In}\ t\}]\!]\ \rho\ \sigma$

$\mathcal{E}\,[\![\mathsf{Cons_1}\ (X)]\!]\ \rho\ \sigma = Y \qquad$ if $\mathcal{L}\ (X, \sigma) = \mathsf{Cons_store_1}\ (X,\ Y)$

$\mathcal{E}\,[\![\mathsf{Cons_2}\ (X)]\!]\ \rho\ \sigma = Y \qquad$ if $\mathcal{L}\ (X, \sigma) = \mathsf{Cons_store_2}\ (X,\ Y)$

$\mathcal{E}[\![\mathsf{Bounds}\ (X)]\!]\ \rho\ \sigma = X_b \qquad$ if $\mathcal{L}\ (X, \rho) = \mathsf{I_array}\ (X_b)$

$\mathcal{E}[\![\mathsf{P_select}\ (X,\ Y)]\!]\ \rho\ \sigma = Z \qquad$ if $\mathcal{L}\ (X, \sigma) = \mathsf{P_store}\ (X,\ Y,\ Z)$

$\mathcal{E}[\![\{Ss;\ As;\ Bs\ \mathsf{In}\ X\}]\!]\ \rho\ \sigma = e$

where e is obtained by the execution of the following program written in pseudo-Id

$\{\ blk, - = \mathcal{F}\&\mathcal{S}(\{Ss;\ As;\ Bs\ \mathsf{In}\ X\});$

$flag = \mathsf{True};$

$\mathsf{In\ If}\ blk\ = \top\ \mathsf{then}\ \top$

$\mathsf{else}\quad \{\mathsf{While}\ flag\ \mathsf{do}$

Suppose blk is $\{Ss;\ As;\ Bs\ In\ X\}$

$is = \mathcal{CC}(\sigma \cup Ss);$

$fs = \rho \cup As;$

$\mathsf{next}\ blk, \mathsf{next}\ flag = \ \mathsf{If}\ is = \top_\mathsf{s}\ \mathsf{then}\ (\top, \mathsf{False})$

$\mathsf{else}\ \mathcal{F}\&\mathcal{S}(\mathcal{E}_\mathcal{RHS}(blk, is, fs))$

$\mathsf{finally}\ blk\}\}$

If none of the above clauses apply, then

$\mathcal{E}[\![E]\!]\ \rho\ \sigma = E$

Theorem 6.9 (Soundness) *Given a term M,*

$$\mathcal{E}val[\![M]\!] = N \Longrightarrow \mathcal{P}rint^*(M) = \mathcal{P}rint(N)$$

Theorem 6.10 (Normalization Theorem) *Given a term M,*

$$\mathcal{P}rint^*(M)\ \textit{is finite}\ \Longrightarrow \mathcal{P}rint(\mathcal{E}[\![M]\!]) = \mathcal{P}rint^*(M).$$

7 Optimizations of Kid Programs

7.1 Optimizations as Rewrite rules ($\mathbf{R_{opt}}$)

Following is a partial list of optimization rules or source-to-source transformations that can be expressed as rules in the CRS for Kid. Optimizations should be performed after type checking and after all bound variables have been assigned unique names. A strategy for applying optimizations is discussed in the next section. Applicability of certain rules requires some semantic check such as "$\underline{n} > 1$". We write such semantic predicates above the line but following an "&".

$\mathbf{R_{opt}}$ is as follows:

$$\mathbf{R_{opt}} = \mathbf{R_{Kid}} \cup \mathbf{R_{oi}} \cup \mathbf{R_{io}} - \{\mathsf{Ap_{n,m}} - \text{rule}, \ \mathsf{Cons} - \text{rule}\}$$

where $\mathbf{R_{oi}}$ and $\mathbf{R_{io}}$ are defined below.

$\mathbf{R_{Kid}}$

All Kid rewrite rules, except the Cons-rule, can be applied at compile time. However, care needs to be excercised in applying some rules such as the *Application rule,* because these can cause non-termination. A similar problem exists with WLoop and FLoop. We will assume that the user will *annotate* the program to indicate when these rules are safely applicable.

$\mathbf{R_{oi}}$:

$\mathbf{R_{oi}}$ rules are applied "*outside in*", as will be shown in the next section.

Inline Substitution

$$\frac{F = \underline{\lambda_{\mathsf{n,m}}} \ (\vec{Z}_n) \ . \ (E)}{\mathsf{Ap_{n,m}} \ (F, \ \vec{X}_n) \ \longrightarrow \ (\mathcal{RB} \ [\![E]\!]) \ [\vec{X}_n \ / \ \vec{Z}_n]}$$

An underlined λ corresponds to the Defsubst annotation in Id. It indicates that the function is substitutable at compiler time. Notice, if the user underlines a recursive procedure, the above rule will cause non-termination. This mechanism of underlining is often used in term rewriting systems to turn a set of rules into an equivalent strongly normalizing set of rules [8]. This mechanism also allows us to control the applicability of the loop rules. If the λ's corresponding to predicate (P) and loop body (B) of a loop expression are *not underlined*, then the loop rule can not be applicable more than once. If the user wants to unfold loops more than once at compiler time then, he must annotate the loop to indicate this. When a loop in Id is annotated to be unfolded, then the translation from Id to Kid will generate underlined λ's for P and B.

Partial Evaluation

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Apply}_{pe}\ (F, X) \quad \longrightarrow \quad \{f = \lambda_{\mathsf{n-1,m}}\ (\vec{z_{n-1}})\ .\ ((\mathcal{RB}\ [\![E]\!])\ [\vec{z_{n-1}}\ /\ \vec{Z_{2,n}},\ X/Z_1]) \\ \mathsf{In}\ f\}}$$

$$\frac{F_i = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Apply}_{pe}\ (F_i,\ X) \quad \longrightarrow \quad \{f = \lambda_{\mathsf{n-i-1,m}}\ (\vec{z_{n-i-1}})\ .\ ((\mathcal{RB}\ [\![E]\!])\ [\vec{z_{n-i-1}}\ /\ \vec{Z_{i+2,n}},\ \vec{X_{i+1}}\ /\ \vec{Z_{i+1}}]) \\ \mathsf{In}\ f\}}$$

A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$.

Apply_{pe} indicates an Apply which the user has annotated for partial evaluation. Notice the difference with the previous rule. Instead of underlining the λ, the information is associated with the application. The rational being that we want to avoid the generation of too many specialized functions. This rule can also cause non termination, if an application in a recursive definition is annotated for partial evaluation. A more sophisticated strategy, that we have not explored yet, would try to compute some fixpoint in case of such an annotation.

Fetch Elimination

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons_1}\ (X) \quad \longrightarrow \quad X_1}$$

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons_2}\ (X) \quad \longrightarrow \quad X_2}$$

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{List_case}\ (X,\ E_1,\ E_2) \quad \longrightarrow \quad E_2}$$

The above rules are useful because we do not reduce Cons $(X_1,\ X_2)$ into Open_cons at compile time.

Algebraic Identities

$\mathbf{Alg_1}$			$\mathbf{Alg_2}$			$\mathbf{Alg_3}$		
And (True, X)	$\longrightarrow$	X	And (False, X)	$\longrightarrow$	False	$X = +\ (X_1, \underline{m})$	& $\underline{m} > 0$	
Or (False, X)	$\longrightarrow$	X	Or (True, X)	$\longrightarrow$	True	Less $(X_1,\ X)$	$\longrightarrow$	True
$+\ (X, 0)$	$\longrightarrow$	X	$*\ (X, 0)$	$\longrightarrow$	0	$X = +\ (X_1, \underline{m})$	& $\underline{m} > 0$	
$*\ (X, 1)$	$\longrightarrow$	X	$-\ (X, X)$	$\longrightarrow$	0	Less $(X,\ X_1)$	$\longrightarrow$	False
$\vdots$			Equal? $(X,\ X)$	$\longrightarrow$	True	$X = +\ (X_1, \underline{m})$	& $\underline{m} > 0$	
			$\vdots$			Greater (X_1, X)	$\longrightarrow$	False
						$X = +\ (X_1, \underline{m})$	& $\underline{m} > 0$	
						Equal? $(X_1,\ X)$	$\longrightarrow$	False
						$\vdots$		

Eliminating Circulating Variables

Suppose in the Id loop body of an Id program there exists an expression like "`Next x = x`" , then the variable `x` can be made into a free variable of the loop and its circulation can be avoided. Without loss of generality we assume that the nextified variable to be eliminated is the last one.

$$\frac{\begin{array}{l} P = \lambda_{\mathsf{n}}\ (\vec{X}_n)\ .\ (E) \\ B = \lambda_{\mathsf{n,n}}\ (\vec{X'_n})\ .\ (\{_{\mathsf{n}}\ S\ \mathsf{In}\ \vec{Z}_{n-1},\ X'_n\}) \end{array}}{\begin{array}{l} \mathsf{WLoop_n}\ (P,\ B,\ \vec{Y}_n,\ Y_p) \quad \longrightarrow \\ \quad \{\ p = \lambda_{\mathsf{n}-1}\ (\vec{x}_{n-1})\ .\ (\mathcal{RB}[\![E]\!]\ [\vec{x}_{n-1}\ /\ \vec{X}_{n-1},\ Y_n/X_n]); \\ \quad\ \ b = \lambda_{\mathsf{n}-1,\mathsf{n}-1}\ (\vec{x'}_{n-1})\ .\ (\mathcal{RB}[\![\{_{\mathsf{n}-1}\ S\ \mathsf{In}\ \vec{Z}_{n-1}\}]\!]\ [\vec{x'}_{n-1}\ /\ \vec{X}_{n-1},\ Y_n/X'_n]); \\ \quad\ \ \vec{t}_{n-1} = \mathsf{WLoop_{n-1}}\ (p,\ b,\ \vec{Y}_{n-1},\ Y_p) \\ \quad\quad \mathsf{In}\ \vec{t}_{n-1},\ Y_n\} \end{array}}$$

A similar optimization applies to for-loops.

Eliminating Circulating Constants

Suppose in the loop body there exists an expression like "`Next x = t `", where the variable `t` is a free variable of the loop body then its circulation can be avoided. Such situations may arise as a consequence of lifting invariants from a loop. The following example illustrates this transformation:

```
{ While (p x y) do
    Next x = t;
    Next y = f x y;
  Finally  y}
```

This may be transformed as follows:

```
If (p x y) then
       { y1 = f x y;
        In
          { While (p t y1) do
              Next y1 = f t y1;
            Finally  y1} }
          else
        y
```

Notice that it is only after the first iteration that the value of "x" is t. Thus, to avoid the circulation of the nextified variable "x", the loop has to be peeled once. This rule can be expressed as follows. Please note that we could have also written Z_n instead of t_n on the right-hand-side.

$$P = \lambda_{\mathsf{n}}\ (\vec{X_n})\ .\ (E) \qquad |$$
$$B = \underbrace{\lambda_{\mathsf{n,n}}\ (\vec{X'_n})\ .\ (\{_{\mathsf{n}}\ S\ \mathsf{In}\ \vec{Z_n}\})}_{\rho}\ \ \&\ \ FE(Z_n, \rho)$$

$$\mathsf{WLoop_n}\ (P,\ B,\ \vec{Y_n},\ Y_p) \longrightarrow$$
$$\mathsf{Bool_case_n}\ (Y_p,$$
$$\{_{\mathsf{n}}\ p = \lambda_{\mathsf{n-1}}\ (\vec{x_{n-1}})\ .\ (\mathcal{RB}[\![E]\!]\ [\vec{x_{n-1}}\ /\ \vec{X_{n-1}},\ t_n/X_n]);$$
$$b = \lambda_{\mathsf{n-1,n}}\ (\vec{x'_{n-1}})\ .\ (\mathcal{RB}[\![\{_{\mathsf{n-1}}\ S\ \mathsf{In}\ \vec{Z_{n-1}}\}]\!]\ [\vec{x'_{n-1}}\ /\ \vec{X'_{n-1}},\ t_n/X'_n]);$$
$$\vec{t_n} = \mathsf{Ap_{n,n}}\ (B,\ \vec{Y_n});$$
$$t_p = \mathsf{Ap_{n-1}}\ (p,\ \vec{t_{n-1}});$$
$$\vec{t'_{n-1}} = \mathsf{WLoop_{n-1}}\ (p,\ b,\ \vec{t_{n-1}},\ t_p);$$
$$\mathsf{In}\ \vec{t'_{n-1}},\ t_n\},$$
$$\vec{Y_n})$$

Peeling the Loop

$$\mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{X_n},\ X) \longrightarrow \mathsf{Bool_case_n}\ (X,\ \{_{\mathsf{n}}\ \vec{t_{2,n}} = \mathsf{Ap_{n,n-1}}\ (B,\ \vec{X_n});$$
$$t_1 = +\ (X_1,\ D);$$
$$t_p = <\ (t_1,\ U);$$
$$\vec{t'_n} = \mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{t_n},\ t_p)$$
$$\mathsf{In}\ \vec{t'_n}\},$$
$$\vec{X_n})$$

Notice that the above rule is again applicable to the Floop generated on the RHS. To avoid unbounded number of applications of this rule the user will have to indicate how many times the loop peeling should be performed.

Loop Body Unrolling K times

$$\frac{\&\ \text{remainder}\ ((U - X_1)/D, \underline{k}) = 0}{\begin{array}{lcl}
\mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{X}_n, X_p) & \longrightarrow & \{\mathsf{n}\ \ b \ = \ \lambda_{\mathsf{n,n-1}}(\vec{x_n})\ .\ (\{_{\mathsf{n-1}}\ t^{\vec{1}}_{2,n} = \mathsf{Ap_{n,n-1}}\ (B,\ \vec{x_n});\\
& & \qquad\qquad t^1_1 = +\ (X_1,\ D);\\
& & \qquad\qquad t^{\vec{2}}_{2,n} = \mathsf{Ap_{n,n-1}}\ (B,\ \vec{t^1_n});\\
& & \qquad\qquad t^2_1 = +\ (t^1_1,\ D);\\
& & \qquad\qquad \vdots\\
& & \qquad\qquad t^{\vec{k}}_{2,n} = \mathsf{Ap_{n,n-1}}\ (B,\ \vec{t^{k-1}_n})\\
& & \qquad\qquad \mathsf{In}\ \vec{t^k_{2,n}}\}\ );\\
& & \quad \vec{t_n} = \mathsf{FLoop_n}\ (U,\ D,\ b,\ \vec{X}_n,\ X_p);\\
& & \quad \mathsf{In}\ \vec{t_n}\}
\end{array}}$$

Suppose $r = remainder((U - X_1)/D, \underline{k})$, and r is not zero. We can still apply the above transformation by first peeling the loop r times. Notice, k has to be supplied by the user.

$\mathbf{R_{io}}$:

These optimizations need to be applied "inside-out", because that way bigger and bigger common expressions or expressions to be lifted can be formed. Furthermore these optimizations are only applied to side-effect-free expressions (sef's) defined below:

Definition 7.1 (Side-effect-free expression) *(1) Variables and constants are sef;*

(2) all $PF_{n,m}\ (\vec{X}_n)$ *, except* I_array, Open_cons, Apply, WLoop and FLoop, *are sef;*

(3) $\mathsf{Bool_Case_m}\ (X,\ E_1,\ E_2)$ *is sef if* E_1 *and* E_2 *are sef; similarly for* List_Case.

(4) $\lambda_{\mathsf{n,m}}x.E$ *is sef if* E *is sef;*

(5) $\mathsf{Ap_{n,m}}\ (F,\ \vec{X}_n)$ *is sef if* F *is bound to a sef* λ*-abstraction and* $\vec{X}_n$ *are bound to sef's;*

(6) $\mathsf{WLoop_m}\ (P, B, \vec{X}_n,\ Y)$ *is sef if* P *and* B *are bound to sef* λ*-abstractions; similarly for* FLoop*;*

(7) Blocks .....

Common Subexpression Elimination

$$\frac{\vec{Y_m} = PFN_m\ (\vec{X_n})}{PFN_m\ (\vec{X_n}) \quad \longrightarrow \quad \vec{Y_m}}$$

Lift Free Expressions

$$\frac{\&\ FE(E, \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{_{\mathsf{m}}\ Y = E;\ S\ \mathsf{In}\ \ \vec{X_m}\ \}))}{\lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{_{\mathsf{m}}\ Y = E;\ S\ \mathsf{In}\ \ \vec{X_m}\ \}) \quad \longrightarrow \quad \{_{\mathsf{m}}\ \ t_1 \ = \ E; \\ t \ = \ \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{_{\mathsf{m}}\ Y = t_1;\ S\ \mathsf{In}\ \ \vec{X_m}\ \}) \\ \mathsf{In}\ t\ \}}$$

Where $FE(e, e')$ return true if the expression e is free in e'. This optimization allows us to deal with loop invariants, that is, expressions that do not depend on the nextified variables. A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$. This rule in conjunction with the loop rules will cause loop invariants to be lifted from loops.

Hoisting Code out of a Conditional

$$\frac{\&\ FE(E, (\mathsf{Bool_case}_{\mathsf{n}}\ (X,\ \{_{\mathsf{n}}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{_{\mathsf{n}}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \})))}{\mathsf{Bool_case}_{\mathsf{n}}\ (X,\ \{_{\mathsf{n}}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{_{\mathsf{n}}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \}) \quad \longrightarrow \\ \{_{\mathsf{n}}\ \ t_1 \ = \ E; \\ \vec{t_n} \ = \ \mathsf{Bool_case}_{\mathsf{n}}\ (X, \\ \{_{\mathsf{n}}\ Y = t_1;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \}, \\ \{_{\mathsf{n}}\ Y' = t_1; S'\ \ \mathsf{In}\ \ \vec{X'_n}\}) \\ \mathsf{In}\ \ \vec{t_n}\ \}$$

7.2 A Partial Evaluator for Kid

Applying $\mathbf{R_{kid}}$ rules and $\mathbf{R_{oi}}$ rules is like *partial evaluation* of a Kid program. In the following we present an efficient strategy for applying $\mathbf{R_{kid}}$ and $\mathbf{R_{oi}}$ rules using the function $\mathcal{EV}$.

However, space does not permit us to give a full interpreter. Therefore, we illustrate our strategy using a few carefully chosen rule, *i.e.* $R_{CRS} \cup \{\delta$ rules, $Case$ rules for booleans, P_select rule $\}$. The extension of $\mathcal{EV}$ to other rules of $\mathbf{R_{kid}}$ and $\mathbf{R_{oi}}$ is straightforward. As usual, initially ρ and σ would be empty.

$\mathcal{EV}[\![+ (\underline{m}, \underline{n})]\!]\ \rho\ \sigma = \underline{+(m, n)}$

$\mathcal{EV}[\![\mathsf{Bool_case_m}\ (\mathsf{True}, E_1, E_2)]\!]\ \rho\ \sigma = \mathcal{EV}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{EV}[\![\mathsf{Bool_case_m}\ (\mathsf{False}, E_1, E_2)]\!]\ \rho\ \sigma = \mathcal{EV}[\![E_2]\!]\ \rho\ \sigma$

$\mathcal{EV}[\![\mathsf{Bool_case_m}\ (X, E_1, E_2)]\!]\ \rho\ \sigma = \mathsf{Bool_case_m}\ (X, (\mathcal{EV}[\![E_1]\!]\ \rho\ \sigma), (\mathcal{EV}[\![E_2]\ \rho\ \sigma)])$

$\mathcal{EV}[\![\mathsf{P_select}\ (X, Y)]\!]\ \rho\ \sigma = Z$ if $(\mathcal{L}\ (X, \sigma)) = \mathsf{P_store}\ (X, Y, Z)$

$\mathcal{EV}[\![\{Ss;\ As;\ Bs\ \mathsf{In}\ X\}]\!]\ \rho\ \sigma = e$

where e is obtained by the execution of the following program written in pseudo-Id

$\{\ blk, _ = \mathcal{F}\&\mathcal{S}(\{Ss;\ As;\ Bs\ \mathsf{In}\ X\});$

$flag = \mathsf{True};$

$\mathsf{If}\ blk = \top\ \mathsf{then}\ \top$

else

$\{\ blk = \{\ \mathsf{While}\ flag\ \mathsf{do}$

Suppose blk is $\{Ss;\ As;\ Bs\ In\ X\}$

$is = \mathcal{CC}(\sigma \cup Ss);$

$fs = \rho \cup As;$

$\mathsf{next}\ blk, \mathsf{next}\ flag = \ \mathsf{If}\ is = \top_{\mathsf{s}}\ \mathsf{then}\ (\top, \mathsf{False})$

$\mathsf{else}\ \mathcal{F}\&\mathcal{S}(\mathcal{EV}'_\mathcal{RHS}(blk, is, fs))$

$\mathsf{finally}\ blk\}$

Suppose blk' is $\{Ss';\ As;'\ Bs'\ \mathsf{In}\ X'\}$

$is = \mathcal{CC}(\sigma \cup Ss);$

$fs = \rho \cup As;$

$blk'' = \mathsf{If}\ is = \top_{\mathsf{s}}\ \mathsf{then}\ (\top, \mathsf{False})$

$\mathsf{else}\ (\mathcal{EV}_\mathcal{RHS}(blk'', is, fs))$

$\mathsf{In}\ blk''\}\}$

where

$$\mathcal{EV}'_\mathcal{RHS}([\![Ss;\ As;\ X_1 = E_1; \cdots X_n = E_n\ \mathsf{In}\ X_i]\!]\ \rho\ \sigma = \\ \{Ss;\ As;\ X_1 = \mathcal{EV}'(E_1,\ \rho\ \sigma); \cdots X_n = \mathcal{EV}'(E_n,\ \rho\ \sigma)\ \mathsf{In}\ X_i\}$$

and the function $\mathcal{EV}'$ given below unlike $\mathcal{EV}$ does not go inside encapsulators.

$$\mathcal{EV}'[\![+\ (\underline{m},\ \underline{n})]\!]\ \rho\ \sigma = \underline{+(m,\ n)}$$
$$\mathcal{EV}'[\![\mathsf{Bool_case_m}\ (\mathsf{True},\ E_1,\ E_2)]\!]\ \rho\ \sigma = E_1$$
$$\mathcal{EV}'[\![\mathsf{Bool_case_m}\ (\mathsf{False},\ E_1,\ E_2)]\!]\ \rho\ \sigma = E_2$$
$$\mathcal{EV}'[\![\mathsf{P_select}\ (X,\ Y)]\!]\ \rho\ \sigma = Z,\ \mathsf{True} \qquad \text{if } (\mathcal{L}\ (\ X, \sigma)) = \mathsf{P_store}\ (X,\ Y,\ Z)$$
else
$$\mathcal{EV}'[\![E]\!]\ \rho\ \sigma = E$$

else
$\mathcal{EV}[\![E]\!]\ \rho\ \sigma = E$

Theorem 7.2 (Normalization Theorem) *Given terms M and N, and let* $R = R_{CRS} \cup R_{kid} \cup R_{oi}$

$$M \underset{R}{\twoheadrightarrow} N \implies \mathcal{EV}[\![M]\!]\ \mathsf{Nil}\ \mathsf{Nil} = N.$$

We do not describe the strategy for $\mathbf{R_{io}}$ rules because it requires choosing a data structure for Kid programs. These rules should be applied from inside out. Unfortunately $\mathbf{R_{io}}$ rules can trigger some $\mathbf{R_{oi}}$ optimizations. For example, the cse-rule may trigger an algebraic rule, such as, the equality rule.

Briefly the overall strategy for optimizations is as follows:

(1) apply $\mathbf{R_{Kid}} \cup \mathbf{R_{oi}}$ - $\{\mathsf{Ap_{n,m}} - \text{rule},\ \mathsf{Cons} - \text{rule}\ \}$, from outside in, as explained in Section 7.2;
(2) apply $\mathbf{R_{io}}$ rules from inside out;
(3) if there is any change in the expression in step (2) go to step (1), else stop.

We beleive that the above strategy is normalizing with respect to $\mathbf{R_{opt}}$.

An other interesting question about optimizations is whether $\mathbf{R_{opt}}$ is confluent. We have shown in [1] that $\mathbf{Alg_3}$ rules cause $\mathbf{R_{opt}}$ to be non confluent. However, this is not a serious drawback because the cases where the confluence is lost are the ones where the unoptimized program would have produced no information .

7.3 Correctness of Optimizations

Definition 7.3 (Observational Equivalence) *Given two terms M and N, M and N are said to be Observationally Equivalent iff*

$$\mathcal{P}rint^*\ (M) \equiv \mathcal{P}rint^*\ (N).$$

For correctness, observational equivalence is not enough. It has to be shown that no context can distinguish between optimized and unoptimized term.

Definition 7.4 (Observational Congruence) *Given two terms M and N, M and N are said to be Observationally Congruent iff*

$$\forall\ C[\Box],\ \ \mathcal{P}rint^*\ (C[M]) \equiv \mathcal{P}rint^*\ (C[N])\ .$$

Definition 7.5 (Correctness) *An optimizer (A,R_{opt}) of (A,R) is correct iff*

$$\forall\ M \in A, M \twoheadrightarrow_{R_{opt}} N \Longrightarrow \begin{array}{l} (i)\ \ \mathcal{P}rint^*(M) \sqsubseteq \mathcal{P}rint^*(N)\ \textit{and} \\ (ii)\ \ \mathcal{P}rint^*(N) = \top \Longrightarrow \mathcal{P}rint^*(M) = \top. \end{array}$$

Notice that condition (ii) is needed because $\top$ is higher than all print values and we do not want the optimizer to take all programs to $\top$.

It is believed that all optimizations presented in Section 7.1 preserve correctness, though this has been proven for only a small subset of them so far [1]. In general, correctness of an optimization is difficult to prove. However, some optimizations can be proven correct easily because they are *derived* rules.

Definition 7.6 (Derived rule) *A rule r is said to be derived in R iff*

$$M \xrightarrow{r} M_1 \Longrightarrow \exists\ M_2,\ \ M \twoheadrightarrow_R M_2\ \ \wedge\ \ M_1 \twoheadrightarrow_R M_2$$

Corollary 7.7 *Given an optimizer (A,R_{opt}) of (A,R),*

$$\forall\, r \in R_{opt}, r \text{ is derived } \implies (A, R_{opt}) \text{ is correct .}$$

Lemma 7.8 *All* $\mathbf{R_{Kid}}$ *rules, Inline Substitution and Fetch Elimination are derived rules, and hence correct.*

Unfortunately, it is not always possible to mimic inside R_{Kid} what an optimizations does. Therefore, we introduce a notion of *graph equivalence*, which is useful for proving the correctness of *cse*-like rules. The terms of a graph may be described by the grammar of Figure 3 by changing the definition of the *SE* syntactic category with $SE = E$. Graphs for a Kid term can be generated by applying a new substitution rule, called the E_substitution rule.

E_substitution rule

$$\frac{X = E}{X \longrightarrow E}$$

where E can be any *side-effect free expression.*

Suppose $\mathbf{R'_{CRS}}$ be {E_substitution rule, Block flattening rule, Multivariable rule}.

Definition 7.9 (Unravelled form) *Let N be the normal form of a term M with respect to* R'_{CRS}*. U(M), the unravelling form of a term M, is the term obtained by deleting from N all bindings of the form X = E (where E is a side-effect free expression).*

Notice that the unravelled version of a term is not necessarily a tree.

Definition 7.10 (graph-equivalence $\equiv_g$) *Two terms M and N are said to be graph-equivalent, if*

$$U(M) \equiv_\alpha U(N).$$

Corollary 7.11 *Given an optimizer (A,R_{opt}) of (A,R),*

$$\forall\, M,\ N \in A,\ [M \longrightarrow_{opt} N \implies M \equiv_g N] \implies (A, R_{opt}) \text{ is correct .}$$

Lemma 7.12 *Let R be the set {Common subexpression elimination rule, lift free expressions rule, Hoisting code out of a conditional rule }.*

$$M \longrightarrow_R N \implies M \equiv_g N.$$

Hence *cse*-like rules are correct.

8 P-TAC: Parallel Three Address Code

8.1 P-TAC

The syntax of P-TAC is given in Figure 6. In P-TAC, I-structure Storage is modelled in greater detail which requires the notion of Labels. All composite objects, that is, data structures and closures are stored in I-structure store and assigned unique labels, which are treated as constants that can be substituted freely.

8.2 Rewrite rules of P-TAC

In the following V stands for a ground value.

δ rules

$$+ (\underline{m},\ \underline{n}) \underset{\delta}{\longrightarrow} \underline{+(m,\ n)}$$
$$\vdots$$

Conditional rule

$$\mathsf{Dispatch_{n,m}}\ (\underline{i},\ \vec{E_{i-1}},\ E_i,\ \vec{E_{i+1,n}}) \longrightarrow E_i$$

$UDF \in$ User Defined Function

$GV \in$ Ground Value

$$
\begin{array}{lll}
Integer & ::= & 1 \mid 2 \mid \cdots \mid \underline{\mathsf{n}} \mid \cdots \\
Boolean & ::= & \mathsf{True} \mid \mathsf{False} \\
Variable & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid x_2 \mid \cdots \\
MV_m & ::= & \underbrace{Variable, \cdots, Variable} \\
Label & ::= & L \mid L1 \mid \cdots \mid L' \mid \cdots \\
PF1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{Allocate} \\
PF2 & ::= & + \mid - \mid * \mid \cdots \mid \mathsf{Less} \mid \mathsf{Equal?} \mid \mathsf{P_select} \\
PF3 & ::= & \mathsf{Ack_store} \\
UDF & ::= & F \mid G \mid \cdots \\
SE & ::= & Variable \mid UDF \mid GV \\
SE_m & ::= & \underbrace{SE, \cdots, SE}_{m} \\
GV & ::= & Integer \mid Boolean \mid () \mid Label \mid \mathsf{Error} \mid \top \\
E_1 & ::= & SE \mid PF1\ (SE) \mid PF2\ (SE_2) \mid PF3\ (SE_3) \mid \mathsf{Ap_n}\ (SE_{n+1}) \\
 & & \mid \mathsf{Dispatch_n}\ (SE, \underbrace{E, \cdots, E}_{n}) \mid \mathsf{WLoop}_1\ (SE_4) \mid \mathsf{FLoop}_1\ (SE_5) \mid Block \\
E_m & ::= & SE_m \mid \mathsf{Ap_{n,m}}\ (SE_{n+1}) \mid \mathsf{Dispatch_{n,m}}\ (SE, \underbrace{E_m, \cdots, E_m}_{n}) \\
 & & \mid \mathsf{WLoop_m}\ (SE_{m+3}) \mid \mathsf{FLoop_m}\ (SE_{m+4}) \mid Block_m \\
Block_m & ::= & \{\mathsf{m}\ [Statement;]^*\ \mathsf{In}\ SE_m\} \\
Statement & ::= & Binding \mid Command \mid \mathsf{Store_Error} \\
Command & ::= & \mathsf{P_store}\ (SE_3) \mid \top_s \\
Binding & ::= & MV_m = E_m
\end{array}
$$

Figure 6: Grammar of P-TAC

I_structure rules

$$\mathsf{Allocate}\ (\underline{n}) \quad \longrightarrow \quad L$$

where L is a brand new label.

$$\frac{\mathsf{P_store}\ (L,\ \underline{i},\ V)}{\mathsf{P_select}\ (L,\ \underline{i}) \quad \longrightarrow \quad V}$$

$$\frac{\mathsf{P_store}\ (L,\ \underline{i},\ V)}{\mathsf{P_store}\ (L,\ \underline{i},\ V') \quad \longrightarrow \quad \top_{\mathsf{s}}}$$

where V is either an Integer or a Boolean or a Label or Error.
The following rules are the same as the corresponding rules in Kid.

Application rule

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\vec{Z}_n)\ .\ (E)}{\mathsf{Ap_{n,m}}\ (F,\ \vec{X}_n) \quad \longrightarrow \quad (\mathtt{RB}[\![E]\!])\ [\vec{X}_n\ /\ \vec{Z}_n]}$$

A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$.

Loop rules

$$\begin{array}{lll} \mathsf{WLoop_n}\ (P,\ B,\ \vec{X}_n,\ \mathsf{True}) & \longrightarrow & \{\mathsf{n}\ \vec{t}_n \ = \ \mathsf{Ap_{n,n}}\ (B,\ \vec{X}_n); \\ & & \quad\ t_p \ = \ \mathsf{Ap_n}\ (P,\ \vec{t}_n); \\ & & \quad\ \vec{t'}_n \ = \ \mathsf{WLoop_n}\ (P,\ B,\ \vec{t}_n,\ t_p) \\ & & \quad\ \mathsf{In}\ \vec{t'}_n\} \end{array}$$

$$\mathsf{WLoop_n}\ (P,\ B,\ \vec{X}_n,\ \mathsf{False}) \quad \longrightarrow \quad \vec{X}_n$$

$$\mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{X}_n,\ \mathsf{True}) \longrightarrow \{\mathsf{n}\ \vec{t}_{2,n} = \mathsf{Ap_{n,n-1}}\ (B,\ \vec{X}_n);\ t_1 = +\ (X_1,\ D);\ t_p = <\ (t_1,\ U);\ \vec{t'}_n = \mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{t}_n,\ t_p)\ \mathsf{In}\ \vec{t'}_n\}$$

$$\mathsf{FLoop_n}\ (U,\ D,\ B,\ \vec{X}_n,\ \mathsf{False}) \longrightarrow \vec{X}_n$$

Multivariable rule

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

Substitution rules

$$\frac{X = Y}{X \longrightarrow Y}$$

$$\frac{X = V}{X \longrightarrow V}$$

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

Block Flattening rule

$$\{\mathsf{m}\ \vec{X}_n = \{\mathsf{n}\ SS_1;\ SS_2;\ \cdots\ \mathsf{In}\ \vec{Y}_n\}\ S_1;\ \cdots\ S_n\ \mathsf{In}\ \vec{Z}_m\} \longrightarrow \{\mathsf{m}\ \vec{X}_n = \vec{Y}_n;\ SS_1;\ SS_2;\ \cdots\ S_1;\ \cdots\ S_n\ \mathsf{In}\ \vec{Z}_m\}$$

Propagation of $\top$

$$\{_{\mathsf{m}}\ X = \top;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} \quad \longrightarrow \quad \top$$

$$\{_{\mathsf{m}}\ \top_s;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} \quad \longrightarrow \quad \top$$

9 Translation of Kid into P-TAC

Prior to translating Kid to P-TAC, λ-lifting is performed. 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.

9.1 Simple Kid Expressions

$\mathtt{TE}[\![X]\!] = X$

$\mathtt{TE}[\![C]\!] = C$

where C ranges over Integers and Booleans.

$\mathtt{TE}[\![\mathsf{Negate}\ (X)]\!] = \mathsf{Negate}\ (X)$

The same holds for Not.

$\mathtt{TE}[\![+\ (X,\ Y)]\!] \quad = \quad +\ (X,\ Y)$

$\mathtt{TE}[\![\mathsf{WLoop_m}\ (P,\ B,\ \vec{X_m},\ X_b)]\!] \quad = \quad \mathsf{WLoop_m}\ (P,\ B,\ \vec{X_m},\ X_b)$

The same holds for FLoop.

$$\begin{array}{lcll} \texttt{TE}[\![\mathsf{Bool_case_m}\ (X,\ E_1,\ E_2)]\!] & = & \{\mathsf{m} & t = \mathit{BooltoInt}\ (X); \\ & & & \vec{t_m} = \mathsf{Dispatch_{2,m}}\ (t,\ E_1,\ E_2); \\ & & & \mathsf{In}\ \vec{t_m}\} \end{array}$$

BooltoInt is a coercion function which converts True to 0 and False to 1.

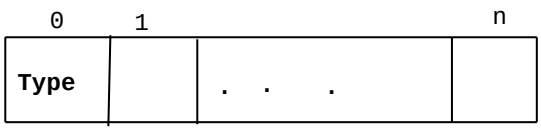
$$\begin{array}{lcll} \texttt{TE}[\![\{\mathsf{m}\ X_1 = E_1;\ \cdots\ X_n = E_n;\ S_1;\ \cdots\ S_m\ \mathsf{In}\ \vec{Y_m}\ \}]\!] & = & \{\mathsf{m} & X_1 = \texttt{TE}[\![E_1]\!]; \\ & & & \vdots \\ & & & X_n = \texttt{TE}[\![E_n]\!]; \\ & & & \texttt{TS}[\![S_1]\!]; \\ & & & \vdots \\ & & & \texttt{TS}[\![S_m]\!]; \\ & & & \mathsf{In}\ \vec{Y_m}\ \} \end{array}$$

9.2 Data Structure Representations

There are usually several reasonable ways to represent each data structure in terms of a P-TAC array. For each type we present 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. We might need types information for other reasons, such as, garbage collection, and printing values in a partially executed program.

Tuples

All n-tuple data types may be represented as follows:

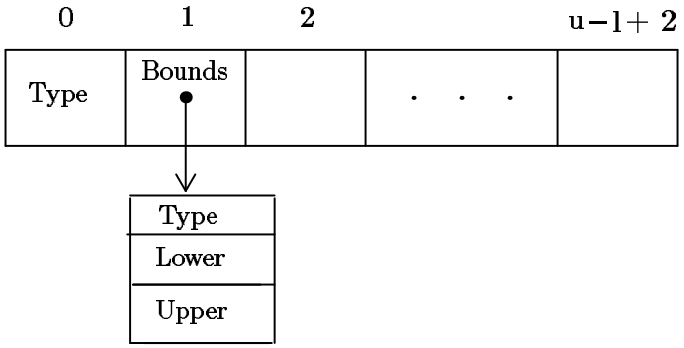


$$
\begin{aligned}
\texttt{TE}[\![\mathsf{Make_tuple_n}\ (\vec{X}_n)]\!] \quad = \quad & \{t = \mathsf{Allocate}\ (\underline{n+1}); \\
& \quad \mathsf{P_store}\ (t,\ \mathsf{Type},\ \text{“}n_tuple\text{”}); \\
& \quad \mathsf{P_store}\ (t,\ 1,\ X_1); \\
& \quad \vdots \\
& \quad \mathsf{P_store}\ (t,\ \underline{n},\ X_n); \\
& \quad \mathsf{In}\ t\}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{TE}[\![\mathsf{Detuple_m}\ (X)]\!] \quad = \quad & \{\mathsf{m}\ t_1 \quad = \quad \mathsf{P_select}\ (X, 1); \\
& \qquad\qquad \vdots \\
& \qquad t_m \quad = \quad \mathsf{P_select}\ (X, \underline{m}); \\
& \quad \mathsf{In}\ \vec{t}_m\}
\end{aligned}
$$

1D-Arrays

1D-arrays "Array (l, u)" may be represented as follows:



The constant definition Headersize = 2 should be included in the set "D".

TE⟦nD_Bounds (X)⟧ = P_select $(X,$ Bounds$)$

$$
\begin{aligned}
\text{TE}[\![\text{l_array } (X)]\!] \;=\; \{\; & l \;=\; \text{P_select } (X,\ \text{Lower});\\
& u \;=\; \text{P_select } (X,\ \text{Upper});\\
& s \;=\; -\ (u,\ l);\\
& size \;=\; +\ (s,\ 3);\\
& t \;=\; \text{Allocate } (size);\\
& \text{P_store } (t,\ \text{Type},\ \text{“}Array\text{”});\\
& \text{P_store } (t,\ \text{Bounds},\ X);\\
& \text{In } t\}
\end{aligned}
$$

$$
\begin{aligned}
\text{TE}[\![\text{P_1D_select } (X_1,\ X_2)]\!] \;=\; \{\; & t_b \;=\; \text{P_select } (X_1,\ \text{Bounds});\\
& l \;=\; \text{P_select } (t_b,\ \text{Lower});\\
& t_1 \;=\; -\ (X_2,\ l);\\
& t_2 \;=\; +\ (t_1,\ \text{Headersize});\\
& t \;=\; \text{P_select } (X_1,\ t_2)\\
& \text{In } t\}
\end{aligned}
$$

$$
\begin{array}{lcl}
\texttt{TE}[\![\mathsf{P_1D_store}\ (X_1,\ X_2,\ X_3)]\!] & = & \{\ t_b \ = \ \mathsf{P_select}\ (X_1,\ \mathsf{Bounds});\\
& & \quad\ l \ = \ \mathsf{P_select}\ (t_b,\ \mathsf{Lower});\\
& & \quad\ t_1 \ = \ -\ (X_2,\ l);\\
& & \quad\ t_2 \ = \ +\ (t_1,\ \mathsf{Headersize});\\
& & \quad\ t \ = \ \mathsf{P_store}\ (X_1,\ t_2,\ X_3)\\
& & \quad\ \mathsf{In}\ t\}
\end{array}
$$

A representation that will be more efficient for computing the slot address may want to store l and u values redundantly in two additional fields.

2D-Arrays

The translation given below assumes that the matrix is stored in the row major order. The following constant definitions should be included in the set "D".

$$
\begin{array}{llllll}
\mathsf{First_dim} & = & 1 & \mathsf{Second_dim} & = & 2\\
\mathsf{First_index} & = & 1 & \mathsf{Second_index} & = & 2
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{TE}[\![\mathsf{I_2D_array}\ (X)]\!] & = & \{\ d_1 \ = \ \mathsf{P_select}\ (X,\ \mathsf{First_dim});\\
& & \quad\ d_2 \ = \ \mathsf{P_select}\ (X,\ \mathsf{Second_dim});\\
& & \quad\ l_1 \ = \ \mathsf{P_select}\ (d_1,\ \mathsf{Lower});\\
& & \quad\ u_1 \ = \ \mathsf{P_select}\ (d_1,\ \mathsf{Upper});\\
& & \quad\ l_2 \ = \ \mathsf{P_select}\ (d_2,\ \mathsf{Lower});\\
& & \quad\ u_2 \ = \ \mathsf{P_select}\ (d_2,\ \mathsf{Upper});\\
& & \quad\ s_1 \ = \ -\ (u_1,\ l_1);\\
& & \quad\ s_2 \ = \ -\ (u_2,\ l_2);\\
& & \quad\ s'_1 \ = \ +\ (s_1,\ 1);\\
& & \quad\ s'_2 \ = \ +\ (s_2,\ 1);\\
& & \quad\ s \ = \ *\ (s'_1,\ s'_2);\\
& & \quad\ size \ = \ +\ (s,\ \mathsf{Headersize});\\
& & \quad\ t \ = \ \mathsf{Allocate}\ (size);\\
& & \quad\ \mathsf{P_store}\ (t,\ \mathsf{Type},\ \text{"}Array\text{"});\\
& & \quad\ \mathsf{P_store}\ (t,\ \mathsf{Bounds},\ X);\\
& & \quad\ \mathsf{In}\ t\}
\end{array}
$$

Translation for I_nD_array can be given in a similar fashion.

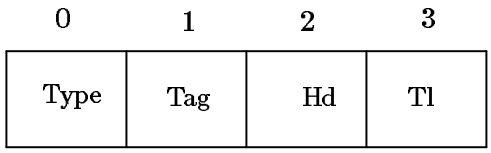
$$
\begin{array}{lcll}
\texttt{TE}[\![\text{P_2D_select } (X_1,\ X_2)]\!] & = & \{\ b & = & \text{P_select } (X_1,\ \text{Bounds}); \\
 & & d_1 & = & \text{P_select } (b,\ \text{First_dim}); \\
 & & d_2 & = & \text{P_select } (b,\ \text{Second_dim}); \\
 & & l_1 & = & \text{P_select } (d_1,\ \text{Lower}); \\
 & & u_1 & = & \text{P_select } (d_1,\ \text{Upper}); \\
 & & l_2 & = & \text{P_select } (d_2,\ \text{Lower}); \\
 & & u_2 & = & \text{P_select } (d_2,\ \text{Upper}); \\
 & & r & = & -\ (u_2,\ l_2); \\
 & & r' & = & +\ (r,\ 1); \\
 & & i & = & \text{P_select } (X_2,\ \text{First_index}); \\
 & & j & = & \text{P_select } (X_2,\ \text{Second_index}); \\
 & & i' & = & -\ (i,\ l_1); \\
 & & j' & = & -\ (j,\ l_2); \\
 & & o & = & *\ (i',\ r'); \\
 & & o' & = & +\ (o,\ 1); \\
 & & ad & = & +\ (o,\ j'); \\
 & & ad' & = & +\ (ad,\ \text{Headersize}); \\
 & & t & = & \text{P_select } (X_1,\ ad') \\
 & & \text{In } t\} & &
\end{array}
$$

A similar rule applies for P_2D_store, and higher dimensional arrays.

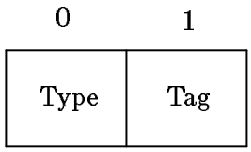
Lists

The list data type can be represented as follows:

Cons :



Nil :



It is often possible to store niladic constructors using much less space by combining them with pointers. We won't discuss such machine dependent representations in this paper.

The following constant definitions should be included in the set "D".

$$\begin{aligned} \mathsf{Nil_size} &= 2 \\ \mathsf{Cons_size} &= 4 \\ \mathsf{Nil_Tag} &= 0 \\ \mathsf{Cons_Tag} &= 1 \end{aligned}$$

$$\begin{aligned} \mathtt{TE}[\![\mathsf{Nil}]\!] = \ & \{t = \mathsf{Allocate}\ (\mathsf{Nilsize}); \\ & \mathsf{P_store}\ (t,\ \mathsf{Type},\ \text{“}List\text{”}); \\ & \mathsf{P_store}\ (t,\ \mathsf{Tag},\ \mathsf{Nil_Tag}); \\ & \ \mathsf{In}\ t\} \end{aligned}$$

$$\begin{aligned} \mathtt{TE}[\![\mathsf{Open_cons}\ ()]\!] \ = \ & \{\ t = \mathsf{Allocate}\ (\mathsf{Cons_size}); \\ & \quad \mathsf{P_store}\ (t,\ \mathsf{Type},\ \text{“}List\text{”}); \\ & \quad \mathsf{P_store}\ (t,\ \mathsf{Tag},\ \mathsf{Cons_Tag}); \\ & \quad \ \mathsf{In}\ t\} \end{aligned}$$

$$\mathtt{TE}[\![\mathsf{Cons_1}\ (X)]\!] \ = \ \mathsf{P_select}\ (X,\ \mathsf{Hd})$$

$$\mathtt{TE}[\![\mathsf{Cons_2}\ (X)]\!] \ = \ \mathsf{P_select}\ (X,\ \mathsf{Tl})$$

$$
\begin{aligned}
\mathtt{TE}[\![\mathsf{Cons}\ (X_1, X_2)]\!] \;=\; \{\ & t = \mathsf{Allocate}\ (\mathsf{Cons_size});\\
& \mathsf{P_store}\ (t,\ \mathsf{Type},\ \text{“}List\text{”});\\
& \mathsf{P_store}\ (t,\ \mathsf{Tag},\ \mathsf{Cons_Tag});\\
& \mathsf{P_store}\ (t,\ \mathsf{Hd},\ X_1);\\
& \mathsf{P_store}\ (t,\ \mathsf{Tl},\ X_2);\\
& \mathsf{In}\ t\}
\end{aligned}
$$

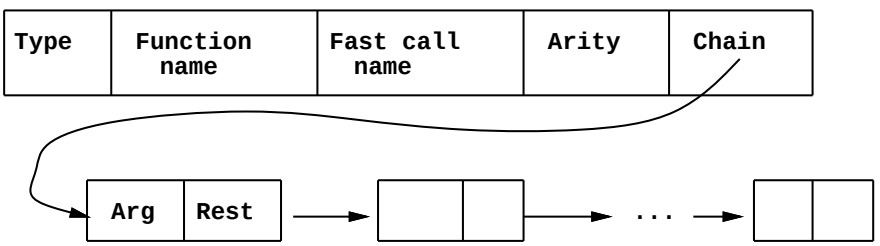
$$
\mathtt{TS}[\![\mathsf{Cons_store_1}\ (X_1, X_2)]\!] \;=\; \mathsf{P_store}\ (X_1, \mathsf{Hd}, X_2);
$$

$$
\mathtt{TS}[\![\mathsf{Cons_store_2}\ (X_1, X_2)]\!] \;=\; \mathsf{P_store}\ (X_1, \mathsf{Tl}, X_2);
$$

$$
\begin{aligned}
\mathtt{TE}[\![\mathsf{List_case_m}\ (X,\ E_1,\ E_2)]\!] \;=\; \{_{\mathsf{m}}\ & t \;=\; \mathsf{P_select}\ (X,\ \mathsf{Tag});\\
& \vec{t_m} \;=\; \mathsf{Dispatch}_{2,\mathsf{m}}\ (t,\ E_1,\ E_2);\\
& \mathsf{In}\ \vec{t_m}\}
\end{aligned}
$$

9.3 Function Calls and Closures

At the machine level, the apply operator checks if the arity of the function has been satisfied or not, and in case the arity has not been satisfied, it stores the argument in a data stucture called a closure. There is great choice in representing closures and associated function calling conventions. In fact, a function can be compiled using several different calling conventions and the compiler can pick up the most appropriate one for a given application. As an illustration we chose the following representation for the closure data type:



The constant definition $\mathsf{Closure_size} = 5$ should be included in the set “D”.

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

$$
\begin{array}{lll}
\mathsf{Make_closure} = \underline{\lambda}\,(cl, X)\ .\ (\{\ f & = & \mathsf{P_select}\ (cl,\ \mathsf{Funcname});\\
f_{fc} & = & \mathsf{P_select}\ (cl,\ \mathsf{Fastcallname});\\
n & = & \mathsf{P_select}\ (cl,\ \mathsf{Arity});\\
ch & = & \mathsf{P_select}\ (cl,\ \mathsf{Chain});\\
cl' & = & \mathsf{Allocate}\ (\mathsf{Closure_size});\\
\multicolumn{3}{l}{\mathsf{P_store}\ (cl',\ \mathsf{Type},\ \text{“}Closure\text{”});}\\
\multicolumn{3}{l}{\mathsf{P_store}\ (cl',\ \mathsf{Functionname},\ f);}\\
\multicolumn{3}{l}{\mathsf{P_store}\ (cl',\ \mathsf{Fastcallname},\ f_{fc});}\\
\multicolumn{3}{l}{\mathsf{P_store}\ (cl',\ \mathsf{Arity},\ n');}\\
\multicolumn{3}{l}{\mathsf{P_store}\ (cl',\ \mathsf{Chain},\ ch');}\\
n' & = & -\ (n,\ 1);\\
ch' & = & \mathsf{Ap}_2\ (\mathsf{Arg_chain},\ X,\ ch);\\
\multicolumn{3}{l}{\quad \mathsf{In}\ cl'\}\ )}
\end{array}
$$

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

$$
\begin{array}{lll}
\mathsf{Arg_chain} = \underline{\lambda}\,(X, Xs)\ .\ (\ \{xs' = \mathsf{Allocate}\ (2);\\
\multicolumn{3}{l}{\quad \mathsf{P_store}\ (xs', \mathsf{Arg}, X);}\\
\multicolumn{3}{l}{\quad \mathsf{P_store}\ (xs', \mathsf{Rest}, Xs);}\\
\multicolumn{3}{l}{\quad \mathsf{In}\ xs'\}\ )}
\end{array}
$$

The argument chain can be destructured using the following function:

$$
\begin{array}{lll}
\mathsf{Args_n} = \underline{\lambda}\,(X)\ .\ (\{\mathsf{n} & t_1 & = \ \mathsf{P_select}\ (X,\ Chain);\\
& a_n & = \ \mathsf{P_select}\ (t_1,\ \mathsf{Arg});\\
& t_2 & = \ \mathsf{P_select}\ (t_1,\ \mathsf{Rest});\\
& a_{n-1} & = \ \mathsf{P_select}\ (t_2,\ \mathsf{Arg});\\
& t_3 & = \ \mathsf{P_select}\ (t_2,\ \mathsf{Rest});\\
& \vdots & \\
& a_1 & = \ \mathsf{P_select}\ (t_{n-1},\ \mathsf{Arg});\\
& \mathsf{In}\ \vec{a_n}\}\ ) &
\end{array}
$$

These three definitions must be included in the "D" set.

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.

$$
\begin{array}{lll}
\mathtt{TE}[\![\mathsf{Apply}\ (F,\ X)]\!] & = & \{\ n = \mathsf{P_select}\ (F,\ \mathsf{Arity});\\
& & \quad fire_b = \mathsf{Equal?}\ (n,\ 1);\\
& & \quad fire_i = BooltoInt(fire_b);\\
& & \quad res = \mathsf{Dispatch}_{2,2}\ (fire_i,\\
& & \qquad \{\ fun = \mathsf{P_select}\ (F,\ \mathsf{Functionname});\\
& & \qquad\ \ as = \mathsf{P_select}\ (F,\ \mathsf{Chain});\\
& & \qquad\ \ as' = \mathsf{Ap}_2\ (\mathsf{Arg_chain},\ X,\ as);\\
& & \qquad\ \ res' = \mathsf{Ap}\ (fun,\ as');\\
& & \qquad\quad \mathsf{In}\ res'\},\\
& & \qquad \mathsf{Ap}_2\ (\mathsf{Make_closure},\ F,\ X))\\
& & \mathsf{In}\ res\}
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{TE}[\![\mathsf{Ap_{n,m}}\ (F,\ \vec{X})]\!] & = & \{\mathsf{m}\ \ f' = \mathsf{P_select}\ (F, \mathsf{FastcallName});\\
& & \qquad \vec{t_m} = \mathsf{Ap_{n,m}}\ (f', \vec{X});\\
& & \quad \mathsf{In}\ \vec{t_m}\ \}
\end{array}
$$

The only thing that remains to be described is the creation of the first closure for a function. It is built as a consequence of translating a λ-expression.

$$
\begin{aligned}
\texttt{TE}[\![\lambda_{\mathsf{n}}\ (\vec{X_n})\ .\ (E)]\!] \quad = \quad & \{cl = \mathsf{Allocate}\ (\mathsf{Closure_size});\\
& \mathsf{P_store}\ (cl,\ \mathsf{Type},\ \text{“}Closure\text{”});\\
& \mathsf{P_store}\ (cl,\ \mathsf{Functionname},\ `T_c);\\
& \mathsf{P_store}\ (cl,\ \mathsf{Fastcallname},\ `T_{fc});\\
& \mathsf{P_store}\ (cl,\ \mathsf{Arity},\ \underline{n});\\
& \mathsf{P_store}\ (cl,\ \mathsf{Chain},\ \text{“}End\text{”});\\
& \ \mathsf{In}\ cl\}
\end{aligned}
$$

The following two function definitions are included in the set D.

$$
\begin{aligned}
T_c = \lambda_1\ (Xs)\ .\ \{\ \vec{X_n} \ &= \ \mathsf{Ap}_{1,\mathsf{n}}\ (\mathsf{Args}_{\mathsf{n}},\ Xs);\\
t \ &= \ \texttt{TE}[\![E]\!];\\
& \ \mathsf{In}\ t\}\\
T_{fc} = \lambda_{\mathsf{n}}\ (\vec{X_n})\ .\ & \texttt{TE}[\![E]\!]
\end{aligned}
$$

‘T_c indicates the name T_c and not the value associated to T_c. Note that $\texttt{TE}[\![E]\!]$ can be computed once and shared between the curried aabd the fastcall version of the function. The same translation rule applies for $\underline{\lambda_{\mathsf{n}}}$.

10 Signals

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-effect, such as P_store. The output of a strict operator can be interpreted as the signal that the instruction has indeed fired. We give the signal transformation using the translation functions S, SE and SC. The transformation is applied also to each constant definition in “D” .

$$
\begin{array}{lllll}
\mathtt{S}[\![\lambda_{\mathsf{n,m}}\ (\vec{X}_n).\{_{\mathsf{m}} & Y_1 & = Se_1 & = \lambda_{\mathsf{n,m+1}}\ (\vec{X}_n)\ . & \\
 & \vdots & & (\{_{\mathsf{m+1}} & Y_1 = Se_1 \\
 & Y_n & = Se_n & & \vdots \\
 & Y_{n+1} & = Nse_1 & & Y_n = Se_n \\
 & \vdots & & & Y_{n+1},\ S_1 = \mathtt{SE}[\![Nse_1]\!] \\
 & Y_{n+m} & = Nse_m & & \vdots \\
 & & C_1 & & Y_{n+m},\ S_m = \mathtt{SE}[\![Nse_m]\!] \\
 & & \vdots & & S_{m+1} = \mathtt{SC}[\![C_1]\!]; \\
 & & C_k & & \vdots \\
 & \mathsf{In}\ \vec{R}_m\}]\!] & & & S_{m+k} = \mathtt{SC}[\![C_k]\!]; \\
 & & & & S' = \mathsf{Sync}_{\mathsf{m+k+i}}\ (Deadvariables,\ \vec{S}_{m+k}) \\
 & & & & \quad \mathsf{In}\ \vec{R}_m,\ S'\})
\end{array}
$$

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

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

Where S_p is the signal associated with the invocation of the predicate.

$$\mathtt{SE}[\![\mathsf{Ap_{n,m}}\ (F,\ \vec{X}_n)]\!] = \mathsf{Ap_{n,m+1}}\ (F,\ \vec{X}_n)$$

$$\mathtt{SC}[\![\mathsf{P_store}\ (X,\ I\ Z)]\!] = \mathsf{Ack_store}\ (X,\ I\ Z)$$

Where Ack_store is a new P-TAC function symbol of arity 3, which generates a Signal when the store actually takes place.

The new rewrite rules are:

$$\mathsf{WLoop'_n}\ (P,\ B,\ \vec{X}_n, S,\ \mathsf{True}) \longrightarrow \{_{\mathsf{n+1}}\ \vec{t}_n, S_b = \mathsf{Ap_{n,n+1}}\ (B,\ \vec{X}_n);\ t_p, S_p = \mathsf{Ap_{n,2}}\ (P,\ \vec{t}_n);\ S' = \mathsf{Sync_3}\ (S,\ S_b,\ S_p);\ \vec{t'}_n, S_l = \mathsf{WLoop'_n}\ (P,\ B,\ \vec{t}_n,\ S',\ t_p)\ \mathsf{In}\ \vec{t'}_n,\ S_l\}$$

$$\mathsf{WLoop'_n}\ (P,\ B,\ \vec{X}_n,\ S,\ \mathsf{False}) \longrightarrow \vec{X}_n,\ S$$

$$\mathsf{Ack_store}\ (L,\ \underline{i},\ V) \longrightarrow \{t = \mathsf{Signal};\ \mathsf{P_store}\ (L,\ \underline{i},\ V);\ \mathsf{In}\ t\}$$

$$\mathsf{Sync_n}\ (\vec{V}_n) \longrightarrow ()$$

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

11 Conclusions

Kid goes a long way towards expressing many machine independent implementation concerns. Since Kid has a proper calculus associated with it, correctness issues can be handled at an abstract level. Kid is central to the current restructuring of the Id compiler which is already being used by at least 4 or 5 groups to generate code for their machines by just changing back end of the compiler. The major deficiency of Kid is the inability to express storage reuse. We believe a language with a proper calculus which does not obscure parallelism and which can express storage reuse would be extremely useful in practical compilers for functional and other declarative languages.

Some of the important optimizations that we currently perform in the Id compiler but have not included in R_{opt} are *dead code elimination*, *loop variable induction* to hoist array bound checking, and *array subscript analyses*. These optimizations do not fit in the CRS model very well. A better formalization of these optimizations would be very useful.

Acknowledgments

We would like to thank Shail Aditya for reading several earlier drafts of this paper and suggesting changes for better readability. This paper has benefitted a lot by informal discussions with the members of Computation Structures Group.

References

[1] Z. M. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, London*, 1989. Also: CSG Memo 295, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[2] Z. M. Ariola and Arvind. Compilation of Id$^-$: a subset of Id. Technical Report CSG Memo 315, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990. Revised 1 November.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989. An earlier version appeared in Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, September/October 1986, Springer-Verlag LNCS 279, pages 336-369.

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[5] B. Courcelle. *Fundamentals Properties of Infinite Trees.* D. Reidel Publishing Company, 1982.

[6] N. de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. In *Koninkijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences*, 1972.

[7] T. Kenneth R. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[8] J. Klop. Term rewriting systems. Course Notes, Summer course organized by Corrado Boehm, Ustica,Italy, September 1985.

[9] R. S. Nikhil. Notes on Translating List Comprehensions in Id. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, January 1988.

[10] 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.

[11] R. S. Nikhil and Arvind. Notes on Pattern Matching Algorithm. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, February 1988.

[12] R. S. Nikhil and Arvind. *Programming in Id: a parallel programming language*. 1990. (book in preparation).

[13] P. J. Simon L. *The implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, N.J., 1987.

[14] J. Thomas. Lambda lifting: Transforming programs to recursive equations. In *Springer-Verlag LNCS 201 (Proc. Functional Programming Languages and Computer Architecture, Nancy, France)*, September 1985.

Contents

1 Introduction **3**

2 Desugared Id$^-$ **5**

3 Kid : The Kernel Id Language **8**

4 Translating Desugared Id$^-$ into Kid **11**

4.1 Simplification of Expressions 11

4.2 Definition of Standard Functions 17

4.3 Pattern Matching: Elimination of Complex Patterns 18

5 Contextual Reduction Systems **24**

5.1 Basic Definitions 24

5.2 Basic Rules of Contextual Reduction System's (R_{CRS}) 26

5.3 Canonical Forms of terms in a CRS 28

6 Contextual Reduction System of Kid **29**

6.1 Kid Rewrite Rules 30

6.2 Printable Values and Answer of a Kid Term 33

6.3 An Interpreter to Compute the Answer of a Kid term 36

7 Optimizations of Kid Programs **39**

7.1 Optimizations as Rewrite rules ($\mathbf{R_{opt}}$) 39

7.2 A Partial Evaluator for Kid 45

7.3 Correctness of Optimizations 48

8 P-TAC: Parallel Three Address Code **50**

8.1 P-TAC 50

8.2 Rewrite rules of P-TAC 50

9 Translation of Kid into P-TAC **54**

9.1 Simple Kid Expressions 54

9.2 Data Structure Representations 55

9.3 Function Calls and Closures 61

10 Signals **64**

11 Conclusions **66**