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

Zena M. Ariola
Aiken Computational Laboratory
Harvard University

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology

November 1, 1990

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.

1 Introduction

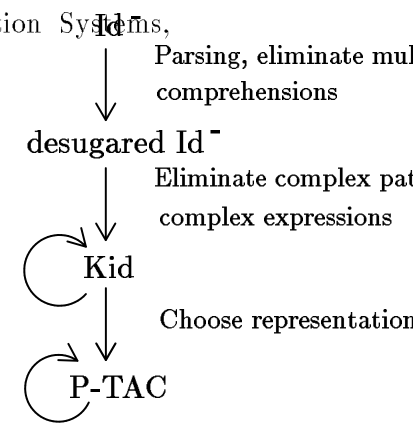


Figure 1: Compilation scheme

2.53in 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 5. 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 [7].

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 6. A translation from Kid into P-TAC is given in Section 7. We end the document by showing in Section 8 how a P-TAC program is extended with Signals [2] 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 [5], 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^-.

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

$$
\begin{array}{lll}
Variable & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \mid F \mid G \mid \cdots \\
Constant & ::= & Integer \mid Boolean \mid () \mid \textsf{Nil} \mid (\textsf{Not}) \mid \cdots \mid (+) \mid (-) \mid \cdots \mid \textsf{Make\_nD\_array} \\
UOP & ::= & \textsf{Negate} \mid \textsf{Not} \mid \textsf{nD\_Bounds} \mid \textsf{I\_nD\_array} \mid \textsf{Open\_cons} \\
BOP & ::= & + \mid - \mid * \mid \cdots \mid == \mid \textsf{Eq?} \mid < \mid \cdots \mid \textsf{And} \mid : \\
E & ::= & Variable \mid \textsf{Next}\ Variable \mid Constant \\
 & & \mid (UOP\ E) \mid (E\ BOP\ E) \mid (E\ E) \mid E[E] \mid E.\textsf{cons\_1} \mid E.\textsf{cons\_2} \\
 & & \mid (E, E) \mid (E, E, E) \mid \cdots \\
 & & \mid (\textsf{If}\ E\ \textsf{then}\ E\ \textsf{else}\ E) \mid \{\textsf{Case}\ E\ \textsf{of}\ Clause\ [\mid\ Clause]^*\ \} \\
 & & \mid \{\textsf{While}\ E\ \textsf{do}\ [Statement;]^*\ \textsf{Finally}\ E\} \\
 & & \mid \{\textsf{For}\ Variable\ \leftarrow\ E\ \textsf{to}\ E\ \textsf{step}\ E\ \textsf{do}\ [Statement;]^*\ \textsf{Finally}\ E\} \\
 & & \mid \{\textsf{Fun}\ [Variables]^*\ =\ E\} \mid \{\underline{\textsf{Fun}}\ [Variables]^*\ =\ E\} \\
 & & \mid Block \\
Block & ::= & \{[Statement;]^*\ \ \textsf{In}\ E\} \\
Statement & ::= & Binding \mid Command \\
Binding & ::= & P = E \\
Command & ::= & E[E] = E \mid E.\textsf{cons\_1} = E \mid E.\textsf{cons\_2} = E \\
Clause & ::= & P = E \\
P & ::= & \textsf{Nil} \mid Variable \mid \textsf{Next}\ Variable \mid (P, P) \mid (P, P, P) \mid \cdots \mid (P : P) \\
Program & ::= & Block
\end{array}
$$

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

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

```
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 5).

The most complex part of the de-sugaring phase is the transformation of comprehensions into loops. This is described in detail in [3]. 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. In Id we make use of "open lists" 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

3.1 Kid

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 in the operational semantics of Kid. The syntax of Kid is given in Figure 3. Every expression, except a block or λ-expression, consists of a combinator followed by the corresponding number of arguments. The translation from Id$^-$ to Kid also has the flavor of turning an applicative TRS into a functional one.

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. The 2 after the curly brace indicates that two values are to be returned by this block

$$
\begin{array}{lcl}
MV & \in & \text{Multiple Variable} \\
SE & \in & \text{Simple Expression} \\
PFi_m & \in & \text{Primitive Function with i arguments and m outputs} \\
Ap\_E_m & \in & \text{Applicative Expression with m outputs} \\
Case\_E_m & \in & \text{Case Expression with m outputs} \\
Loop\_E_m & \in & \text{Loop Expression with m outputs} \\
lambda\_E & \in & \text{Lambda Expression} \\
E\_m & \in & \text{Expression with m outputs}
\end{array}
$$

$$
\begin{array}{lcl}
Variable & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \\
MV_m & ::= & \underbrace{Variable, \cdots, Variable}_{m} \\
Constant & ::= & Integer \mid Boolean \mid () \mid (\mathsf{Not}) \mid \cdots \mid (+) \mid (-) \mid \cdots \\
 & & \mid \mathsf{Nil} \mid \mathsf{Make\_nD\_array} \mid \mathsf{Error} \mid \top \\
SE & ::= & Variable \mid Constant \\
SE_m & ::= & \underbrace{SE, \cdots, SE}_{m} \mid SE, SE_{m-1} \mid SE, SE, SE_{m-2} \mid \cdots \\
PF1_1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{nD\_Bounds} \mid \mathsf{I\_nD\_array} \mid \mathsf{Open\_cons} \mid \mathsf{Cons\_1} \mid \mathsf{Cons\_2} \\
PF1_m & ::= & \mathsf{Detuple_m} \\
PF2_1 & ::= & + \mid - \mid * \mid \cdots \mid \mathsf{Equal?} \mid \mathsf{Eq?} \mid < \mid \cdots \mid \mathsf{And} \mid \mathsf{Cons} \mid \mathsf{Apply} \\
 & & \mid \mathsf{P\_nD\_select} \mid \mathsf{Make\_tuple_2} \\
PFN_1 & ::= & \mathsf{Make\_tuple_n} \\
Ap\_E_m & ::= & \mathsf{Ap_{n,m}}\ (SE_{n+1}) \\
Case\_E_m & ::= & \mathsf{Bool\_case_m}\ (SE,\ E_m,\ E_m) \mid \mathsf{List\_case_m}\ (SE,\ E_m,\ E_m) \\
Loop\_E_m & ::= & \mathsf{WLoop_m}\ (SE_{m+3}) \mid \mathsf{FLoop_m}\ (SE_{m+4}) \\
lambda\_E & ::= & \lambda_{\mathsf{n,m}}\ (MV_n)\ .\ (E_m) \mid \underline{\lambda_{\mathsf{n,m}}}\ (MV_n)\ .\ (E_m) \\
E_1 & ::= & SE_1 \mid PF1_1\ (SE) \mid PF2_1\ (SE_2) \mid PFN_1\ (SE_n) \mid Ap\_E_1 \\
 & & \mid Case\_E_1 \mid Loop\_E_1 \mid lambda\_E \mid Block_1 \\
E_m & ::= & SE_m \mid PF1_m\ (SE) \mid Ap\_E_m \mid Case\_E_m \mid Loop\_E_m \mid Block_m \\
Block_m & ::= & \{\mathsf{m}\ [Statement;]^*\ \ \mathsf{In}\ SE_m\} \\
Statement & ::= & Binding \mid Command \\
Binding & ::= & MV_m = E_m \\
Command & ::= & \mathsf{P\_nD\_store}\ (SE,\ SE,\ SE) \mid \mathsf{Cons\_store\_1}\ (SE,\ SE) \\
 & & \mid \mathsf{Cons\_store\_2}\ (SE,\ SE) \mid \mathsf{Store\_error} \mid \top_s \\
Program & ::= & Block_1
\end{array}
$$

Figure 3: Grammar of Kid

expression. Multiple values avoid packaging values in a data structure, and they are useful in expressing some optimizations. Thus, in Kid a binding has the form $MV = E$, where MV stands for multiple variable. 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 combinator "Apply" appears as a $PF2_1$ in the grammar because all Id⁻ procedures return only one result. We also use subscripted combinators to express a family of combinators. For example, $\mathsf{Make\_tuple_n}$ stands for $\mathsf{Make\_tuple_2}$, $\mathsf{Make\_tuple_3}$, *etc.* . Subscripts in a combinator do not necessarily represent the number of values to be returned by the application of the combinator.

We will use the following conventions to minimize the use of subscripts.

$\{$	is the same as	$\{_1$
Bool_case	is the same as	$\mathsf{Bool\_case_1}$
List_case	is the same as	$\mathsf{List\_case_1}$
WLoop	is the same as	$\mathsf{WLoop_1}$
FLoop	is the same as	$\mathsf{FLoop_1}$
$\mathsf{Ap_n}$	is the same as	$\mathsf{Ap_{n,1}}$
λ_n	is the same as	$\lambda_{\mathsf{n},1}$

3.2 The Rewrite Rules of Kid

We now present a set of rewrite rules, R_{Kid}, to define the operational semantics for Kid. R_{Kid} is a Contextual Rewrite System described in [8]. We assume that a primitive function is only applied 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:

$$\begin{array}{lll} X_i,\ Z_i,\ Y_i,\ F_i,\ P,\ B,\ U,\ D & \in & \mathit{Variable} \text{ and } \mathit{Constant} \\ C & \in & \mathit{Constant} \\ S_i,\ SS_i & \in & \mathit{Statement} \\ S,\ S' & \in & [\mathit{Statement}]^* \\ E_i & \in & \mathit{Expression} \end{array}$$

It should be noted that in contextual rewriting, the statement above the line must be in the context (lexical scope) of the expression below the line. This raises some subtle free variable capture possibilities in case of function application. To avoid these problems, we will assume that all bound variables in a Kid program have been assigned unique names to begin with. When the possibility of free variable capture arises during rewriting, we will rename all bound variables of an expression to completely new variables explicitly, by applying the function RB to the expression. For example,

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

The notation $E\,[Y/X]$ means the substitution of Y for X in E. Usually this implies avoiding capture of free variables. However, due to our assumption that all variables occurring in a term to be reduced are unique, $E\,[Y/X]$ 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 $\overrightarrow{X_{n,m}}$ to stand for $(X_n, \cdots, X_m)$, $\overrightarrow{X_m}$ for $\overrightarrow{X_{1,m}}$, and $E\,[\overrightarrow{Y_n}\ /\ \overrightarrow{X_n}]$ 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.

δ rules

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

Case rules

$$\mathsf{Bool\_case_m}\ (\mathsf{True},\ E_1,\ E_2) \quad \longrightarrow \quad E_1$$

$$\mathsf{Bool\_case_m}\ (\mathsf{False},\ E_1,\ E_2) \quad \longrightarrow \quad E_2$$

$$\frac{X = \mathsf{Nil}}{\mathsf{List\_case_m}\ (X,\ E_1,\ E_2) \ \longrightarrow E_1}$$

$$\frac{X = \mathsf{Open\_cons}\ ()}{\mathsf{List\_case_m}\ (X,\ E_1,\ E_2) \ \longrightarrow E_2}$$

Arity Detection rule

$$\frac{F = \lambda_n(\overrightarrow{Z_n}).E}{\mathsf{Apply}\ (F,\ X) \ \longrightarrow \ \mathsf{Apply_1}\ (F,\ \underline{n},\ X)}$$

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

$$\frac{F_i = \mathsf{Apply_i}\ (F,\ \underline{n},\ X_i) \quad i < (n-1)}{\mathsf{Apply}\ (F_i,\ \overrightarrow{X_{i+1}}) \ \longrightarrow \ \mathsf{Apply_{i+1}}\ (F,\ \underline{n},\ \overrightarrow{X_{i+1}})}$$

$$\frac{F_{n-1} = \mathsf{Apply_{n-1}}\ (F,\ \underline{n},\ \overrightarrow{X_{n-1}}) \quad i = (n-1)}{\mathsf{Apply}\ (F_{n-1},\ \overrightarrow{X_n}) \ \longrightarrow \ \mathsf{Ap}\ (F,\ \overrightarrow{X_n})}$$

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

Application rule

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

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

Loop rules

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

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

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

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

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

Tuple rule

$$\frac{X = \mathsf{Make\_tuple_n}\ (\vec{X}_n)}{\mathsf{Detuple_n}\ (X) \longrightarrow \vec{X}_n}$$

List rules

$$\begin{array}{lll} \mathsf{Cons}\ (X,\ Y) & \longrightarrow & \{\ t = \mathsf{Open\_cons}\ (); \\ & & \quad \mathsf{Cons\_store\_1}\ (t,\ X); \\ & & \quad \mathsf{Cons\_store\_2}\ (t,\ Y) \\ & & \quad \mathsf{In}\ t\} \end{array}$$

$$\frac{\mathsf{Cons\_store\_1}\ (X,\ Y)}{\mathsf{Cons\_1}\ (X) \longrightarrow Y}$$

$$\frac{\mathsf{Cons\_store\_2}\ (X,\ Y)}{\mathsf{Cons\_2}\ (X) \quad \longrightarrow \quad Y}$$

$$\frac{\mathsf{Cons\_store\_1}\ (X,\ Y)}{\mathsf{Cons\_store\_1}\ (X,\ Y') \quad \longrightarrow \quad \top_s}$$

$$\frac{\mathsf{Cons\_store\_2}\ (X,\ Y)}{\mathsf{Cons\_store\_2}\ (X,\ Y') \quad \longrightarrow \quad \top_s}$$

Array rules

$$\frac{X = \mathsf{I\_nD\_array}\ (X_b)}{\mathsf{nD\_Bounds}\ (X) \quad \longrightarrow \quad X_b}$$

$$\frac{\mathsf{P\_nD\_store}\ (X,\ Y,\ Z)}{\mathsf{P\_nD\_select}\ (X,\ Y) \quad \longrightarrow \quad Z}$$

$$\frac{\mathsf{P\_nD\_store}\ (X,\ Y,\ Z)}{\mathsf{P\_nD\_store}\ (X,\ Y,\ Z') \quad \longrightarrow \quad \top_s}$$

Multivariable rule

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

Substitution rules

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

Block Flattening rule

$$\begin{array}{lcl} \{\mathsf{m}\ \vec{X_n} = \{\mathsf{n}\ SS_1;\ SS_2;\ \cdots & & \{\mathsf{m}\ \vec{X_n} = \vec{Y_n}; \\ \qquad\qquad \mathsf{In}\ \ \vec{Y_n}\} & \longrightarrow & \quad SS_1;\ SS_2;\ \cdots \\ \quad S_1;\ \cdots\ \ S_n & & \quad S_1;\ \cdots\ \ S_n \\ \quad \mathsf{In}\ \ \vec{Z_m}\} & & \quad \mathsf{In}\ \ \vec{Z_m}\} \end{array}$$

Propagation of $\top$

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

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

4 Translating Desugared Id$^-$ into Kid

4.1 Simplification of Expressions

We give the translation in terms of the following functions:

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

The exact syntax of Case_expression is given in Section 4.3.

We will write $\mathtt{TE}[\![E_1]\!] = E_2$, 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 "$\mathtt{TE}[\![(\mathit{UOP}\ E_1)]\!] = \{t_1 = \mathtt{TE}[\![E_1]\!];\ t = \mathtt{TO}[\![\mathit{UOP}]\!]\ (t_1);\ \mathsf{In}\ t\}$ " 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 3.2, 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 lexical 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.

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

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

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

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

$$\begin{array}{lcl}\mathtt{TE}[\![E_1[E_2]]\!] & = & \{\ t_1\ =\ \mathtt{TE}[\![E_1]\!]; \\ & & \quad t_2\ =\ \mathtt{TE}[\![E_2]\!]; \\ & & \quad t\ =\ \mathsf{Ap_2}\ (\mathsf{Select},\ t_1,\ t_2) \\ & & \quad \mathsf{In}\ \ t\}\end{array}$$

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

$$
\begin{array}{lcl}
\texttt{TE}[\![E.\mathsf{cons\_1}]\!] & = & \{\; t_1 \;=\; \texttt{TE}[\![E]\!]; \\
 & & \quad t \;=\; \mathsf{List\_case}\ (t_1,\ \mathsf{Error},\ \mathsf{Cons\_1}\ (t_1)) \\
 & & \quad\; \mathsf{In}\ \ t\}
\end{array}
$$

Similarly for E.cons_2.

$$
\begin{array}{lcl}
\texttt{TE}[\![(E_1, \cdots, E_n)]\!] & = & \{\; t_1 \;=\; \texttt{TE}[\![E_1]\!]; \\
 & & \quad\; \vdots \\
 & & \quad t_n \;=\; \texttt{TE}[\![E_n]\!]; \\
 & & \quad t \;=\; \mathsf{Make\_tuple_n}\ (\overrightarrow{t_n}) \\
 & & \quad\; \mathsf{In}\ \ t\}
\end{array}
$$

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

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

Where PM is described in the Section 4.3.

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

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.

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

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.

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

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

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

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

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

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.

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

$$
\begin{aligned}
\mathtt{TS}[\![E_1[E_2] = E_3]\!] \;=\; (\; t_1 &= \mathtt{TE}[\![E_1]\!];\\
t_2 &= \mathtt{TE}[\![E_2]\!];\\
t_3 &= \mathtt{TE}[\![E_3]\!];\\
t &= \mathsf{Ap}_3\,(\mathsf{Store},\ t_1,\ t_2,\ t_3))
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{TS}[\![E_1.\mathsf{cons\_1} = E_2]\!] \;=\; (\; t_1 &= \mathtt{TE}[\![E_1]\!];\\
t_2 &= \mathtt{TE}[\![E_2]\!];\\
t &= \mathsf{List\_case}\,(t_1,\ \{\ \mathsf{Store\_error\ In}\ ()\}, \{\ \mathsf{Cons\_store\_1}\,(t_1,\ t_2)\ \mathsf{In}\ ()\ \}\ )\ )
\end{aligned}
$$

Similarly for **cons_2**.

TB: Id⁻ Bindings ⟶ list (Kid Binding)

$$
\begin{aligned}
\mathtt{TB}[\![(P_1,\cdots,P_n) = X]\!] \;=\;& (\vec{t_n} = \mathsf{Detuple}_n\,(X);\\
& \quad \mathtt{TB}[\![P_1 = t_1]\!];\\
& \quad \vdots\\
& \quad \mathtt{TB}[\![P_n = t_n]\!]\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{TB}[\![(P_1\ :\ P_2) = X]\!] \;=\;& (t_1,\ t_2 = \mathsf{List\_case}_2\,(X,\ (\ \mathsf{Error},\ \mathsf{Error}\ ),\ (\mathsf{Cons\_1}\,(X),\ \mathsf{Cons\_2}\,(X))\ );\\
& \quad \mathtt{TB}[\![P_1 = t_1]\!];\\
& \quad \mathtt{TB}[\![P_2 = t_2]\!])
\end{aligned}
$$

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

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

TO: Id⁻ Operator ⟶ Kid Operator

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

$\mathtt{TO}[\![\mathit{BOP}]\!] =$ 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) [6]. 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 [4]. 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

$$\mathtt{PM} : \text{Case\_expression} \quad \longrightarrow \quad \text{Kid Expression}$$

where Case-expression is defined as follows

$$\begin{array}{lll} \text{Case\_expression} & ::= & \{\textsf{Case}\ \text{SE}\ \textsf{of}\ [\ |\ \text{Clause}]^*\} \\ \text{Clause} & ::= & \text{P} = \text{Kid\_Expression} \\ \text{P} & ::= & \textsf{Nil} \mid \text{Variable} \mid () \mid (\text{P, P}) \mid (\text{P, P, P}) \mid \cdots \mid (\text{P : P}) \\ \text{SE} & ::= & () \mid (\text{Variable}, \cdots, \text{Variable}\ ) \end{array}$$

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.

$$\begin{array}{lllllll} \mathtt{PM}[\![\ \{\textsf{Case} & (\overrightarrow{X_{1,i-1}}, & X_i, & \overrightarrow{X_{i+1,n}}) & \textsf{of} \\ \quad | & (\overrightarrow{P1_{1,i-1}}, & Y_1, & \overrightarrow{P1_{i+1,n}}) & = E_1 \\ & & \vdots & & \\ \quad | & (\overrightarrow{Pm_{1,i-1}}, & Y_m, & \overrightarrow{Pm_{i+1,n}}) & = E_m\}]\!] = \end{array}$$

$$\begin{array}{l} \{\ t = X_i; \\ \quad t_f = \mathtt{PM}[\![\ \{\textsf{Case}\ \ (\overrightarrow{X_{1,i-1}},\ \ \overrightarrow{X_{i+1,n}})\quad \textsf{of} \\ \qquad\qquad |\quad (\overrightarrow{P1_{1,i-1}},\ \ \overrightarrow{P1_{i+1,n}}) \quad = E_1[t/Y_1]_B \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad |\quad (\overrightarrow{Pm_{1,i-1}},\ \ \overrightarrow{Pm_{i+1,n}}) \quad = E_m[t/Y_m]_B\}]\!] \\ \quad \textsf{In}\ t_f\ \} \end{array}$$

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.

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

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

Refutable pattern rule

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

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

$$
\begin{array}{l}
\mathsf{List\_case}\quad (X_i, \\
\qquad\qquad \begin{array}{llll}
\texttt{PM}[\![\ \{\mathsf{Case} & (\overrightarrow{X_{1,i-1}}\ , & \overrightarrow{X_{i+1,n}}) & \mathsf{of} \\
\quad | & (\overrightarrow{P1_{1,i-1}}\ , & \overrightarrow{P1_{i+1,n}}) & = E_1 \\
 & & \vdots & \\
\quad | & (\overrightarrow{Pk_{1,i-1}}\ , & \overrightarrow{Pk_{i+1,n}}) & = E_k\}]\!],
\end{array} \\
\qquad\qquad \{\ t_h, t_t = \mathsf{Cons\_1}\ (X_i),\ \mathsf{Cons\_2}\ (X_i); \\
\qquad\qquad\quad \begin{array}{llllll}
t_f = \texttt{PM}[\![\ \{\mathsf{Case} & (t_h\ , & t_t\ , & \overrightarrow{X_{1,i-1}}\ , & \overrightarrow{X_{i+1,n}}) & \mathsf{of} \\
\qquad | & (Yl_1\ , & Yl_2\ , & \overrightarrow{Pl_{1,i-1}}\ , & \overrightarrow{Pl_{i+1,n}}) & = E_l \\
 & & & \vdots & & \\
\qquad | & (Ym_1\ , & Ym_2\ , & \overrightarrow{Pm_{1,i-1}}\ , & \overrightarrow{Pm_{i+1,n}}) & = E_m\}]\!]
\end{array} \\
\qquad\qquad\quad \mathsf{In}\ t_f\}\ )
\end{array}
$$

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

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

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

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:

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

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

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

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

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

5 Optimizations of Kid Programs

Following is a partial list of optimizations rules for Kid. Optimizations include all R_{Kid} rules, except the application rule. Optimizations should be performed after type checking and after all bound variables have been assigned unique names. Applicability of certain optimization rules requires some semantic check such as "$\underline{m} > 0$". We write such semantic predicates above the line but following an "&".

It is believed that all optimizations to be presented in this section preserve at least partial correctness. So far this has been proven for only a small subset of them [1].

Kid Rewrite rules

All Kid rewrite rules can be applied at compile time except for the Application rule, which can cause non-termination.

Inline Substitution

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

Partial Evaluation

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\begin{array}{lll}\mathsf{Apply}\ (F, X) & \longrightarrow & \{f = \lambda_{\mathsf{n-1,m}}\ (\vec{z_{n-1}})\ .\ ((\mathsf{RB}\ [\![E]\!])\ [\vec{z_{n-1}}\ /\ \vec{Z_{2,n}},\ X/Z_1]) \\ & & \mathsf{In}\ f\}\end{array}}$$

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

Fetch Elimination

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons\_1}\ (X)\ \longrightarrow\ X_1}$$

$$\frac{X = \mathsf{Cons}\,(X_1,\ X_2)}{\mathsf{Cons\_2}\,(X) \quad \longrightarrow \quad X_2}$$

Algebraic Identities

$$\mathsf{And}\,(\mathsf{True},\ X) \longrightarrow X$$

$$\mathsf{Or}\,(\mathsf{False},\ X) \longrightarrow X$$

$$+\,(X, 0) \longrightarrow X$$

$$*\,(X, 1) \longrightarrow X$$

$$\vdots$$

The above rules preserve total correctness, while the following rules preserve only partial correctness. Any algebraic rule that does not have a precondition can be included in the following rules.

$$\mathsf{And}\,(\mathsf{False},\ X) \longrightarrow \mathsf{False}$$

$$\mathsf{Or}\,(\mathsf{True},\ X) \longrightarrow \mathsf{True}$$

$$*\,(X, 0) \longrightarrow 0$$

$$-\,(X, X) \longrightarrow 0$$

$$\mathsf{Equal?}\,(X,\ X) \longrightarrow \mathsf{True}$$

$$\vdots$$

The following rules are also partially correct but are not confluent.

$$\frac{X = +\,(X_1, \underline{m}) \qquad \& \ \ \underline{m} > 0}{\mathsf{Less}\,(X_1,\ \ X) \quad \longrightarrow \quad \mathsf{True}}$$

$$\frac{X = +\,(X_1, \underline{m}) \qquad \& \ \ \underline{m} > 0}{\mathsf{Less}\,(X,\ \ X_1) \quad \longrightarrow \quad \mathsf{False}}$$

$$\frac{X = +(X_1, \underline{m}) \qquad \& \quad \underline{m} > 0}{\mathsf{Greater}\ (X_1, X) \quad \longrightarrow \quad \mathsf{False}}$$

$$\frac{X = +(X_1, \underline{m}) \qquad \& \quad \underline{m} > 0}{\mathsf{Equal?}\ (X_1,\ X) \quad \longrightarrow \quad \mathsf{False}}$$

$$\vdots$$

Common Subexpression Elimination

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

Primitive functions I_nD_array, Open_cons, Apply and $\mathsf{Ap}_{\mathsf{n,m}}$ are excluded from this optimization because they (may) cause side-effects.

Lift Free Expressions

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

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 $\underline{\lambda_{\mathsf{n,m}}}$. (See the restrictions in the common subexpression elimination rule).

Hoisting Code out of a Conditional

$$\frac{\&\ FE(E, (\mathsf{Boolcase_n}\ (X,\ \{_{\mathsf{n}}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{_{\mathsf{n}}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \})))}{\begin{array}{l} \mathsf{Bool\_case_n}\ (X,\ \{_{\mathsf{n}}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{_{\mathsf{n}}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \}) \quad \longrightarrow \\ \qquad \{_{\mathsf{n}}\ \ t_1 \ = \ E; \\ \qquad\quad\ \vec{t_n} \ = \ \mathsf{Bool\_case_n}\ (X, \\ \qquad\qquad\qquad\quad \{_{\mathsf{n}}\ Y = t_1;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \}, \\ \qquad\qquad\qquad\quad \{_{\mathsf{n}}\ Y' = t_1; S'\ \ \mathsf{In}\ \ \vec{X'_n}\}) \\ \qquad\quad\ \mathsf{In}\ \ \vec{t_n}\ \} \end{array}}$$

Eliminating Circulating Variables

Suppose in the 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.

$$
\frac{\begin{array}{l} P = \underline{\lambda_{\mathsf{n}}}\ (\overrightarrow{X_n})\ .\ (E) \qquad\qquad | \\ B = \underline{\lambda_{\mathsf{n,n}}}\ (\overrightarrow{X'_n})\ .\ (\{_{\mathsf{n}}\ S\ \mathsf{In}\ \ \overrightarrow{Z_{n-1}},\ X'_n\}) \end{array}}{\begin{array}{l} \mathsf{WLoop}_{\mathsf{n}}\ (P,\ B,\ \overrightarrow{Y_n},\ Y_p) \quad \longrightarrow \\ \quad \{\ p = \underline{\lambda_{\mathsf{n-1}}}\ (\overrightarrow{x_{n-1}})\ .\ (\mathtt{RB}[\![E]\!]\ [\overrightarrow{x_{n-1}}\ /\ \overrightarrow{X_{n-1}},\ Y_n/X_n]); \\ \qquad b = \underline{\lambda_{\mathsf{n-1,n-1}}}\ (\overrightarrow{x'_{n-1}})\ .\ (\mathtt{RB}[\![\{_{\mathsf{n-1}}\ S\ \mathsf{In}\ \ \overrightarrow{Z_{n-1}}\}]\!]\ [\overrightarrow{x'_{n-1}}\ /\ \overrightarrow{X_{n-1}},\ Y_n/X'_n]); \\ \qquad \overrightarrow{t_{n-1}} = \mathsf{WLoop}_{\mathsf{n-1}}\ (p,\ b,\ \overrightarrow{Y_{n-1}},\ Y_p) \\ \qquad\quad \mathsf{In}\ \ \overrightarrow{t_{n-1}},\ Y_n\} \end{array}}
$$

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.

$$\frac{\begin{array}{l} P = \underline{\lambda_{\mathsf{n}}}\ (\vec{X_n})\ .\ (E) \\ B = \underbrace{\underline{\lambda_{\mathsf{n,n}}}\ (\vec{X'_n})\ .\ (\{_{\mathsf{n}}\ S\ \mathsf{In}\ \vec{Z_n}\})}_{\rho}\ \ \&\ \ FE(Z_n, \rho) \end{array} \quad |}{\begin{array}{l} \mathsf{WLoop_n}\ (P,\ B,\ \vec{Y_n},\ Y_p) \quad \longrightarrow \\ \qquad \mathsf{Bool\_case_n}\ (Y_p, \\ \qquad\qquad \begin{array}{lcl} \{_{\mathsf{n}}\ p & = & \underline{\lambda_{\mathsf{n-1}}}\ (\vec{x_{n-1}})\ .\ (\mathtt{RB}[\![E]\!]\ [\vec{x_{n-1}}\ /\ \vec{X_{n-1}},\ t_n/X_n]); \\ \quad b & = & \underline{\lambda_{\mathsf{n-1,n}}}\ (\vec{x'_{n-1}})\ .\ (\mathtt{RB}[\![\{_{\mathsf{n-1}}\ S\ \mathsf{In}\ \vec{Z_{n-1}}\}]\!]\ [\vec{x'_{n-1}}\ /\ \vec{X'_{n-1}},\ t_n/X'_n]); \\ \quad \vec{t_n} & = & \mathsf{Ap_{n,n}}\ (B,\ \vec{Y_n}); \\ \quad t_p & = & \mathsf{Ap_{n-1}}\ (p,\ \vec{t_{n-1}}); \\ \quad \vec{t'_{n-1}} & = & \mathsf{WLoop_{n-1}}\ (p,\ b,\ \vec{t_{n-1}},\ t_p); \\ \quad\ \mathsf{In}\ \vec{t'_{n-1}},\ t_n\}, & & \\ \vec{Y_n}) & & \end{array} \end{array}}$$

Peeling the Loop once

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

Loop Body Unrolling K times

$$\frac{\& \ \text{remainder}\ ((U-X_1)/D, \underline{k}) = 0}{\begin{array}{lllll} \mathsf{FLoop_n}\ (U,\ D,\ B,\ \overrightarrow{X_n}, X_p) & \longrightarrow & \{_\mathsf{n}\ \ b & = & \underline{\lambda_{\mathsf{n,n-1}}}(\overrightarrow{x_n})\ .\ (\{_{\mathsf{n-1}}\ \begin{array}{lll} \overrightarrow{t^1_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \overrightarrow{x_n}); \\ t^1_1 & = & +\ (X_1,\ D); \\ \overrightarrow{t^2_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \overrightarrow{t^1_n}); \\ t^2_1 & = & +\ (t^1_1,\ D); \\ \vdots & & \\ \overrightarrow{t^k_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \overrightarrow{t^{k-1}_n}) \\ \multicolumn{3}{l}{\ \mathsf{In}\ \overrightarrow{t^k_{2,n}}\}\ );} \end{array} \\ & & \overrightarrow{t_n} & = & \mathsf{FLoop_n}\ (U,\ D,\ b,\ \overrightarrow{X_n},\ X_p); \\ & & \multicolumn{3}{l}{\ \mathsf{In}\ \overrightarrow{t_n}\}} \end{array}}$$

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.

6 P-TAC: Parallel Three Address Code

6.1 P-TAC

The syntax of P-TAC is given in Figure 4. 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.

6.2 Rewrite rules of P-TAC

In the following V stands for a ground value.

δ rules

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

$$
\begin{array}{lcl}
\mathit{UDF} & \in & \text{User Defined Function} \\
\mathit{GV} & \in & \text{Ground Value} \\
\\
\mathit{Integer} & ::= & 1 \mid 2 \mid \cdots \mid \underline{\mathsf{n}} \mid \cdots \\
\mathit{Boolean} & ::= & \mathsf{True} \mid \mathsf{False} \\
\mathit{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 \\
\mathit{MV}_m & ::= & \underbrace{\mathit{Variable}, \cdots, \mathit{Variable}} \\
\mathit{Label} & ::= & L \mid L1 \mid \cdots \mid L' \mid \cdots \\
\mathit{PF}1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{Allocate} \\
\mathit{PF}2 & ::= & + \mid - \mid * \mid \cdots \mid \mathsf{Less} \mid \mathsf{Equal?} \mid \mathsf{P\_select} \\
\mathit{PF}3 & ::= & \mathsf{Ack\_store} \\
\mathit{UDF} & ::= & F \mid G \mid \cdots \\
\mathit{SE} & ::= & \mathit{Variable} \mid \mathit{UDF} \mid \mathit{GV} \\
\mathit{SE}_m & ::= & \underbrace{\mathit{SE}, \cdots, \mathit{SE}}_{m} \\
\mathit{GV} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid () \mid \mathit{Label} \mid \mathsf{Error} \mid \top \\
E_1 & ::= & \mathit{SE} \mid \mathit{PF}1\ (\mathit{SE}) \mid \mathit{PF}2\ (\mathit{SE}_2) \mid \mathit{PF}3\ (\mathit{SE}_3) \mid \mathsf{Ap}_{\mathsf{n}}\ (\mathit{SE}_{n+1}) \\
 & & \mid \mathsf{Dispatch}_{\mathsf{n}}\ (\mathit{SE}, \underbrace{E, \cdots, E}_{n}) \mid \mathsf{WLoop}_{1}\ (\mathit{SE}_4) \mid \mathsf{FLoop}_{1}\ (\mathit{SE}_5) \mid \mathit{Block} \\
E_m & ::= & \mathit{SE}_m \mid \mathsf{Ap}_{\mathsf{n,m}}\ (\mathit{SE}_{n+1}) \mid \mathsf{Dispatch}_{\mathsf{n,m}}\ (\mathit{SE}, \underbrace{E_m, \cdots, E_m}_{n}) \\
 & & \mid \mathsf{WLoop}_{\mathsf{m}}\ (\mathit{SE}_{m+3}) \mid \mathsf{FLoop}_{\mathsf{m}}\ (\mathit{SE}_{m+4}) \mid \mathit{Block}_m \\
\mathit{Block}_m & ::= & \{_{\mathsf{m}}\ [\mathit{Statement};]^*\ \ \mathsf{In}\ \mathit{SE}_m\} \\
\mathit{Statement} & ::= & \mathit{Binding} \mid \mathit{Command} \mid \mathsf{Store\_Error} \\
\mathit{Command} & ::= & \mathsf{P\_store}\ (\mathit{SE}_3) \mid \top_s \\
\mathit{Binding} & ::= & \mathit{MV}_m = E_m
\end{array}
$$

Figure 4: Grammar of P-TAC

Conditional rule

$$\mathsf{Dispatch}_{\mathsf{n,m}}\ (\underline{i},\ \overrightarrow{E_{i-1}},\ E_i,\ \overrightarrow{E_{i+1,n}}) \quad \longrightarrow \quad E_i$$

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

$$\frac{F = \lambda_{\mathsf{n,m}}\ (\overrightarrow{Z_n})\ .\ (E)}{\mathsf{Ap}_{\mathsf{n,m}}\ (F,\ \overrightarrow{X_n}) \quad \longrightarrow \quad (\mathtt{RB}[\![E]\!])\ [\overrightarrow{X_n}\ /\ \overrightarrow{Z_n}]}$$

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

Loop rules

$$\begin{array}{lllll}
\mathsf{WLoop}_{\mathsf{n}}\ (P,\ B,\ \overrightarrow{X_n},\ \mathsf{True}) & \longrightarrow & \{_{\mathsf{n}}\ \overrightarrow{t_n} & = & \mathsf{Ap}_{\mathsf{n,n}}\ (B,\ \overrightarrow{X_n}); \\
 & & \quad t_p & = & \mathsf{Ap}_{\mathsf{n}}\ (P,\ \overrightarrow{t_n}); \\
 & & \quad \overrightarrow{t'_n} & = & \mathsf{WLoop}_{\mathsf{n}}\ (P,\ B,\ \overrightarrow{t_n},\ t_p) \\
 & & \quad \mathsf{In}\ \ \overrightarrow{t'_n}\} & &
\end{array}$$

$$\mathsf{WLoop}_{\mathsf{n}}\ (P,\ B,\ \overrightarrow{X_n},\ \mathsf{False}) \quad \longrightarrow \quad \overrightarrow{X_n}$$

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

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

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

$$\begin{array}{l} \{\mathsf{m}\ \vec{X_n} =\ \{\mathsf{n}\ SS_1;\ SS_2;\ \cdots \\ \qquad\qquad\qquad \mathsf{In}\ \ \vec{Y_n}\} \\ \quad S_1;\ \cdots\ S_n \\ \quad\ \mathsf{In}\ \ \vec{Z_m}\} \end{array} \longrightarrow \begin{array}{l} \{\mathsf{m}\ \vec{X_n} = \vec{Y_n}; \\ \quad SS_1;\ SS_2;\ \cdots \\ \quad S_1;\ \cdots\ S_n \\ \quad \mathsf{In}\ \ \vec{Z_m}\} \end{array}$$

Propagation of $\top$

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

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

7 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.

7.1 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.

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

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

The same holds for FLoop.

$$\begin{array}{rcll} \mathtt{TE}[\![\mathsf{Bool\_case}_{\mathsf{m}}\,(X,\ E_1,\ E_2)]\!] & = & \{_{\mathsf{m}} & t \quad = \quad \mathit{BooltoInt}\,(X); \\ & & & \vec{t_m} \quad = \quad \mathsf{Dispatch}_{2,\mathsf{m}}\,(t,\ E_1,\ E_2); \\ & & & \mathsf{In}\ \vec{t_m}\} \end{array}$$

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

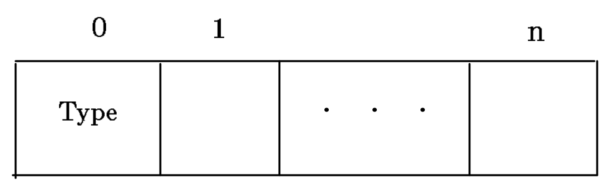
$$
\begin{aligned}
\mathtt{TE}[\![\{\mathsf{m}\ X_1 = E_1;\ \cdots\ X_n = E_n;\ S_1;\ \cdots\ S_m\ \mathsf{In}\ \vec{Y_m}\ \}]\!] \quad = \quad \{\mathsf{m}\quad & X_1 \quad = \quad \mathtt{TE}[\![E_1]\!]; \\
& \vdots \\
& X_n \quad = \quad \mathtt{TE}[\![E_n]\!]; \\
& \mathtt{TS}[\![S_1]\!]; \\
& \vdots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

7.2 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:



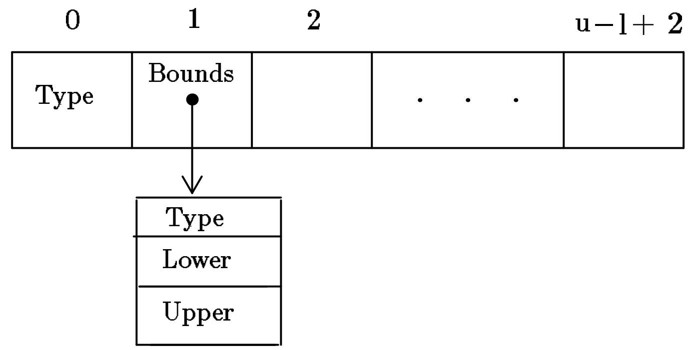
1.5in

$$
\begin{aligned}
\mathtt{TE}[\![\mathsf{Make\_tuple_n}\ (\vec{X_n})]\!] \quad = \quad & \{t = \mathsf{Allocate}\ (\underline{n+1}); \\
& \ \mathsf{P\_store}\ (t,\ \mathsf{Type},\ \text{“}n\_tuple\text{”}); \\
& \ \mathsf{P\_store}\ (t,\ 1,\ X_1); \\
& \ \vdots \\
& \ \mathsf{P\_store}\ (t,\ \underline{n},\ X_n); \\
& \quad \mathsf{In}\ t\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{TE}[\![\mathsf{Detuple}_{\mathsf{m}}\ (X)]\!] \quad = \quad \{\mathsf{m}\ \ t_1 \quad &= \quad \mathsf{P\_select}\ (X, 1); \\
&\vdots \\
t_m \quad &= \quad \mathsf{P\_select}\ (X, \underline{m}); \\
\mathsf{In}\ \ &\vec{t_m}\}
\end{aligned}
$$

1D-Arrays

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



2.5in

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

$$\mathrm{TE}\llbracket \text{nD\_Bounds}\ (X) \rrbracket = \text{P\_select}\ (X,\ \text{Bounds})$$

$$\mathrm{TE}\llbracket \text{I\_array}\ (X) \rrbracket \;=\; \{\ \begin{array}{lcl} l & = & \text{P\_select}\ (X,\ \text{Lower}); \\ u & = & \text{P\_select}\ (X,\ \text{Upper}); \\ s & = & -\ (u,\ l); \\ size & = & +\ (s,\ 3); \\ t & = & \text{Allocate}\ (size); \\ \multicolumn{3}{l}{\text{P\_store}\ (t,\ \text{Type},\ \text{``}Array\text{''});} \\ \multicolumn{3}{l}{\text{P\_store}\ (t,\ \text{Bounds},\ X);} \\ \multicolumn{3}{l}{\ \ \text{In}\ t\}} \end{array}$$

$$\mathrm{TE}\llbracket \text{P\_1D\_select}\ (X_1,\ X_2) \rrbracket \;=\; \{\ \begin{array}{lcl} t_b & = & \text{P\_select}\ (X_1,\ \text{Bounds}); \\ l & = & \text{P\_select}\ (t_b,\ \text{Lower}); \\ t_1 & = & -\ (X_2,\ l); \\ t_2 & = & +\ (t_1,\ \text{Headersize}); \\ t & = & \text{P\_select}\ (X_1,\ t_2) \\ \multicolumn{3}{l}{\ \ \text{In}\ t\}} \end{array}$$

$$\mathrm{TE}\llbracket \text{P\_1D\_store}\ (X_1,\ X_2,\ X_3) \rrbracket \;=\; \{\ \begin{array}{lcl} t_b & = & \text{P\_select}\ (X_1,\ \text{Bounds}); \\ l & = & \text{P\_select}\ (t_b,\ \text{Lower}); \\ t_1 & = & -\ (X_2,\ l); \\ t_2 & = & +\ (t_1,\ \text{Headersize}); \\ t & = & \text{P\_store}\ (X_1,\ t_2,\ X_3) \\ \multicolumn{3}{l}{\ \ \text{In}\ t\}} \end{array}$$

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".

$$\begin{array}{lcllcl} \textsf{First\_dim} & = & 1 & \textsf{Second\_dim} & = & 2 \\ \textsf{First\_index} & = & 1 & \textsf{Second\_index} & = & 2 \end{array}$$

$$\begin{array}{lcll} \texttt{TE}[\![\textsf{I\_2D\_array}\ (X)]\!] & = & \{ & d_1 = \textsf{P\_select}\ (X,\ \textsf{First\_dim}); \\ & & & d_2 = \textsf{P\_select}\ (X,\ \textsf{Second\_dim}); \\ & & & l_1 = \textsf{P\_select}\ (d_1,\ \textsf{Lower}); \\ & & & u_1 = \textsf{P\_select}\ (d_1,\ \textsf{Upper}); \\ & & & l_2 = \textsf{P\_select}\ (d_2,\ \textsf{Lower}); \\ & & & u_2 = \textsf{P\_select}\ (d_2,\ \textsf{Upper}); \\ & & & s_1 = -\ (u_1,\ l_1); \\ & & & s_2 = -\ (u_2,\ l_2); \\ & & & s'_1 = +\ (s_1,\ 1); \\ & & & s'_2 = +\ (s_2,\ 1); \\ & & & s = *\ (s'_1,\ s'_2); \\ & & & size = +\ (s,\ \textsf{Headersize}); \\ & & & t = \textsf{Allocate}\ (size); \\ & & & \textsf{P\_store}\ (t,\ \textsf{Type},\ \text{``}Array\text{''}); \\ & & & \textsf{P\_store}\ (t,\ \textsf{Bounds},\ X); \\ & & & \quad \textsf{In}\ t\} \end{array}$$

Translation for I_nD_array can be given in a similar fashion.

$$
\texttt{TE}[\![\textsf{P\_2D\_select}\ (X_1,\ X_2)]\!] \quad = \quad \{ \begin{array}{lcl}
b & = & \textsf{P\_select}\ (X_1,\ \textsf{Bounds}); \\
d_1 & = & \textsf{P\_select}\ (b,\ \textsf{First\_dim}); \\
d_2 & = & \textsf{P\_select}\ (b,\ \textsf{Second\_dim}); \\
l_1 & = & \textsf{P\_select}\ (d_1,\ \textsf{Lower}); \\
u_1 & = & \textsf{P\_select}\ (d_1,\ \textsf{Upper}); \\
l_2 & = & \textsf{P\_select}\ (d_2,\ \textsf{Lower}); \\
u_2 & = & \textsf{P\_select}\ (d_2,\ \textsf{Upper}); \\
r & = & -\ (u_2,\ l_2); \\
r' & = & +\ (r,\ 1); \\
i & = & \textsf{P\_select}\ (X_2,\ \textsf{First\_index}); \\
j & = & \textsf{P\_select}\ (X_2,\ \textsf{Second\_index}); \\
i' & = & -\ (i,\ l_1); \\
j' & = & -\ (j,\ l_2); \\
o & = & *\ (i',\ r'); \\
o' & = & +\ (o,\ 1); \\
ad & = & +\ (o,\ j'); \\
ad' & = & +\ (ad,\ \textsf{Headersize}); \\
t & = & \textsf{P\_select}\ (X_1,\ ad') \\
\multicolumn{3}{l}{\quad \textsf{In}\ t\}}
\end{array}
$$

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

Lists

The list data type can be represented as follows:

Cons :

0	1	2	3
Type	Tag	Hd	Tl

Nil :

0	1
Type	Tag

3in

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".

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

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

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

$$\mathtt{TE}[\![\mathsf{Cons\_1}\ (X)]\!] \ = \ \mathsf{P\_select}\ (X,\ \mathsf{Hd})$$

$$\mathtt{TE}[\![\mathsf{Cons\_2}\ (X)]\!] \ = \ \mathsf{P\_select}\ (X,\ \mathsf{Tl})$$

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

$$\mathtt{TS}[\![\mathsf{Cons\_store\_1}\ (X_1, X_2)]\!] \ = \ \mathsf{P\_store}\ (X_1, \mathsf{Hd}, X_2);$$

$$\mathtt{TS}[\![\mathsf{Cons\_store\_2}\ (X_1, X_2)]\!] \ = \ \mathsf{P\_store}\ (X_1, \mathsf{Tl}, X_2);$$

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

7.3 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:

2in

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.

$$
\begin{array}{llcl}
\mathsf{Make\_closure} = \underline{\lambda}\,(cl, X)\ .\ (\{ & f & = & \mathsf{P\_select}\,(cl,\ \mathsf{Funcname}); \\
 & f_{fc} & = & \mathsf{P\_select}\,(cl,\ \mathsf{Fastcallname}); \\
 & n & = & \mathsf{P\_select}\,(cl,\ \mathsf{Arity}); \\
 & ch & = & \mathsf{P\_select}\,(cl,\ \mathsf{Chain}); \\
 & cl' & = & \mathsf{Allocate}\,(\mathsf{Closure\_size}); \\
 & \multicolumn{3}{l}{\mathsf{P\_store}\,(cl',\ \mathsf{Type},\ \text{“}\mathit{Closure}\text{”});} \\
 & \multicolumn{3}{l}{\mathsf{P\_store}\,(cl',\ \mathsf{Functionname},\ f);} \\
 & \multicolumn{3}{l}{\mathsf{P\_store}\,(cl',\ \mathsf{Fastcallname},\ f_{fc});} \\
 & \multicolumn{3}{l}{\mathsf{P\_store}\,(cl',\ \mathsf{Arity},\ n');} \\
 & \multicolumn{3}{l}{\mathsf{P\_store}\,(cl',\ \mathsf{Chain},\ ch');} \\
 & n' & = & -\,(n,\ 1); \\
 & ch' & = & \mathsf{Ap}_2\,(\mathsf{Arg\_chain},\ X,\ ch); \\
 & \multicolumn{3}{l}{\quad \mathsf{In}\ cl'\}\ )}
\end{array}
$$

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

$$\mathsf{Arg\_chain} = \underline{\lambda}\,(X, Xs) \quad . \quad (\ \{xs' = \mathsf{Allocate}\ (2);\ \mathsf{P\_store}\ (xs', \mathsf{Arg}, X);\ \mathsf{P\_store}\ (xs', \mathsf{Rest}, Xs);\ \mathsf{In}\ xs'\}\ )$$

The argument chain can be destructured using the following function:

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

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.

$$\begin{array}{lll}
\mathsf{TE}[\![\mathsf{Apply}\ (F,\ X)]\!] = \{\ n & = & \mathsf{P_select}\ (F,\ \mathsf{Arity}); \\
\quad fire_b & = & \mathsf{Equal?}\ (n,\ 1); \\
\quad fire_i & = & BooltoInt(fire_b); \\
\quad res & = & \mathsf{Dispatch_{2,2}}\ (fire_i, \\
 & & \quad \{\ fun = \mathsf{P_select}\ (F,\ \mathsf{Functioname}); \\
 & & \quad\ \ as = \mathsf{P_select}\ (F,\ \mathsf{Chain}); \\
 & & \quad\ \ as' = \mathsf{Ap_2}\ (\mathsf{Arg_chain},\ X,\ as); \\
 & & \quad\ \ res' = \mathsf{Ap}\ (fun,\ as'); \\
 & & \quad\ \ \mathsf{In}\ res'\}, \\
 & & \quad \mathsf{Ap_2}\ (\mathsf{Make_closure},\ F,\ X)) \\
\quad \mathsf{In}\ res\} & &
\end{array}$$

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

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.

$$
\begin{aligned}
\mathtt{TE}[\![\lambda_{\mathsf{n}}\ (\vec{X_n})\ .\ (E)]\!] \quad = \quad & \{cl = \mathsf{Allocate}\ (\mathsf{Closure\_size});\\
& \ \mathsf{P\_store}\ (cl,\ \mathsf{Type},\ \text{“}Closure\text{”});\\
& \ \mathsf{P\_store}\ (cl,\ \mathsf{Functioname},\ \text{‘}T_c);\\
& \ \mathsf{P\_store}\ (cl,\ \mathsf{Fastcallname},\ \text{‘}T_{fc});\\
& \ \mathsf{P\_store}\ (cl,\ \mathsf{Arity},\ \underline{n});\\
& \ \mathsf{P\_store}\ (cl,\ \mathsf{Chain},\ \text{“}End\text{”});\\
& \quad \mathsf{In}\ cl\}
\end{aligned}
$$

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

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

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

8 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” .

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

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.

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

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

$$\mathsf{SE}[\![\mathsf{Ap}_{\mathsf{n,m}}\ (F,\ \vec{X_n})]\!] = \mathsf{Ap}_{\mathsf{n,m+1}}\ (F,\ \vec{X_n})$$

$$\mathsf{SC}[\![\mathsf{P\_store}\ (X,\ I\ Z)]\!] = \mathsf{Ack\_store}\ (X,\ I\ Z)$$

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:

$$
\begin{aligned}
\mathsf{WLoop'_n}\ (P,\ B,\ \vec{X}_n, S,\ \mathsf{True}) \quad \longrightarrow \quad \{_{\mathsf{n+1}}\ \vec{t}_n, S_b &= \mathsf{Ap}_{\mathsf{n,n+1}}\ (B,\ \vec{X}_n);\\
t_p, S_p &= \mathsf{Ap}_{\mathsf{n,2}}\ (P,\ \vec{t}_n);\\
S' &= \mathsf{Sync}_3\ (S,\ S_b,\ S_p);\\
\vec{t'}_n, S_l &= \mathsf{WLoop'_n}\ (P,\ B,\ \vec{t}_n,\ S',\ t_p)\\
\mathsf{In}\ \vec{t'}_n,\ S_l\}
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{Ack\_store}\ (L,\ \underline{i},\ V) \quad \longrightarrow \quad & \{t = \mathsf{Signal};\\
& \ \mathsf{P\_store}\ (L,\ \underline{i},\ V);\\
& \ \ \mathsf{In}\ t\}
\end{aligned}
$$

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

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

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] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] A. Zena Matilde and Arvind. Contextual Rewriting. 545 Technology Square, Cambridge, MA 02139, USA, 1990. In preparation.

Contents