LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Compilation of Id$^-$: a subset of Id

Computation Structures Group Memo 315

24 July 1990

Zena M. Ariola

Arvind

**** DRAFT ****

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Compilation of Id$^-$: a subset of Id

Zena Ariola
Aiken Computational Laboratory
Harvard University

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology

August 2, 1990

1 Introduction

The main motivation behind this document is to describe the compilation of Id in terms of successive transformations among different languages, each of which has a precise operational semantics. The compilation scheme is shown in Figure 1. We will concentrate on a powerful subset of Id, called Id$^-$. Id$^-$ should be seen as Id where enough parenthesis have been inserted to make operator associativity and precedence irrelevant. Id$^-$ only contains a primitive form of pattern-matching, and no user defined types or comprehensions. Most of these features will complicate the presentation but would not increase the complexity of the underlying system.

Id$^-$ is first translated into PGL, which corresponds to program graphs [3]. PGL represents the intermediate language in which compiler optimizations are expressed. PGL syntax and the translation are given in Section 3.1 and 3.2 respectively. The operational semantics of PGL is given in Section 3.4. PGL presented in this paper is not necessarily suitable to give the operational semantics of managers and other side-effect operations, like accumulators, takes and puts. Milner style type checking though not described in this document should be performed on the PGL program. Compiler optimizations are given in Section 4.

After having perfomed optimizations all nested function definitions (λ-expressions) of a PGL program are "closed", this process is known in the literature as lambda-lifting. This step is not described in this document. The next step is to translate PGL into P-TAC. P-TAC in a real sense is closer to low-level dataflow graph. P-TAC and the translation are given in Section 5.1 and 5.2 respectively. The operational semantics of P-TAC is given in Section 5.3. We end the document by showing in Section 6 how a P-TAC program is extended with Signals.

Our approach allows the formalization of questions related to correctness [1], for example, it will make sense to talk about correctness of lambda-lifting, and of the translation between PGL and P-TAC. Moreover, since there exists an independent operational semantics of Id$^-$ in terms of Kernel Id [2] it will be possible to prove the correctness of the translation between Id$^-$ and PGL. Another advantage is that we are able to delay the introduction of concepts tied up with the dataflow computation model, such as, switches and merges, until we actually generate machine code.

[0]Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988 (MIT) and N0039-88-C-0163 (Harvard).

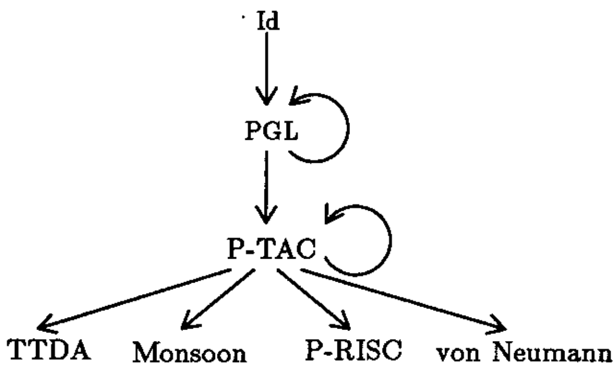

Figure 1: Compilation scheme

UOP	$\in$	Unary Operator
BOP	$\in$	Binary Operator
E	$\in$	Expression
P	$\in$	Pattern
Variable	::=	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \mid F \mid G \mid \cdots$
Integer	::=	$1 \mid 2 \mid \cdots \mid \underline{n} \mid \cdots$
Boolean	::=	True | False
Constant	::=	*Integer* | *Boolean* | Nil | Curried1_- | $\text{Curried1}_{\text{Not}}$ | $\cdots$ | Curried2_+ | Curried2_- | $\cdots$ | Make_array
UOP	::=	$-$ | Not | Bounds | I_array
BOP	::=	$+ \mid - \mid * \mid == \mid \cdots \mid$ And $\mid < \mid \cdots \mid :$
E	::=	*Variable* | Next *Variable* | *Constant* | $(UOP\ E)$ | $(E\ BOP\ E)$ | $(E\ E)$ | (E, E) | (E, E, E) | $\cdots$ | $E[E]$ | (If E then E else E) | {Case E of Nil $= E$ | $P : P = E$} | {While E do [*Statement*;]* Finally E} | *Block*
Block	::=	{[*Statement*;]* In E}
Statement	::=	*Binding* | *Command* | *Definition*
Binding	::=	$P = E$
Command	::=	$E[E] = E$
P	::=	*Variable* | Next *Variable* | (P, P) | (P, P, P) | $\cdots$ | $(P : P)$
Definition	::=	Def *Variable* $[P]^+ = E$ | Defsubst *Variable* $[P]^+ = E$
Program	::=	*Block*

Figure 2: Grammar of Id^-.

2 Id$^-$

The syntax of Id$^-$ is described in Figure 2. Id has curried versions of infix binary operators which, for example, are written by enclosing the operator in parenthesis. Thus, `(+)` represents the curried `+`, and it makes sense to write `((+) 2)` in Id. In Id$^-$ we will represent curried version of an operator `BOP` as $\texttt{Curried2}_{\texttt{BOP}}$ (rather than `(BOP)`) and treat it as a constant. For the sake of symmetry we have also included curried versions of all unary operators in PGL.

3 The first phase: from Id$^-$ to PGL

3.1 PGL

PGL, acronym for "Program Graph Language", has only uncurried operators and no complex expressions. A major subset of PGL 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 PGL. The syntax of PGL is given in Figure 3. Every expression, except a block, if, case or λ-expression, consists of a combinator followed by the corresponding number of arguments. The translation from Id$^-$ to PGL also has the flavor of turning an applicative TRS into a functional one.

An important feature of PGL 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 variable, not to be confused with a 2-tuple. Multiple values avoid packaging values in a data structure, and they are useful in expressing some optimizations. Thus, in PGL 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 correspondent syntactic category. Thus, to express the above binding we will write $MV_m = E_m$. Note that in the grammar the combinator "Apply" appears as a $PF2_1$ 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.

If	is the same as	If_1
Case	is the same as	Case_1
Loop	is the same as	Loop_1
Ap_n	is the same as	$\mathsf{Ap}_{n,1}$
λ_n	is the same as	$\lambda_{n,1}$

3.2 Translation from Id$^-$ to PGL

We give the translation in terms of the following functions:

$$
\begin{array}{lll}
\mathit{MV} & \in & \text{Multiple Variable} \\
\mathit{SE} & \in & \text{Simple Expression} \\
\mathit{PFi}_m & \in & \text{Primitive Function with i arguments and m outputs} \\
\mathit{Ap\_E}_m & \in & \text{Applicative Expression with m outputs} \\
\mathit{Cond\_E}_m & \in & \text{Conditional Expression with m outputs} \\
\mathit{Loop\_E}_m & \in & \text{Loop Expression with m outputs} \\
\lambda\_E_m & \in & \text{Definition of a function with m outputs} \\
 & & \\
\mathit{Variable} & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots \\
\mathit{MV}_m & ::= & \underbrace{\mathit{Variable}, \cdots, \mathit{Variable}}_{m} \\
\mathit{Integer} & ::= & 1 \mid 2 \mid \cdots \\
\mathit{Constant} & ::= & \mathit{Integer} \mid \mathit{Float} \mid \mathit{Boolean} \mid \cdots \mid \mathsf{Nil} \mid \mathsf{Error} \mid \top \\
\mathit{SE} & ::= & \mathit{Variable} \mid \mathit{Constant} \\
\mathit{SE}_m & ::= & \underbrace{\mathit{SE}, \cdots, \mathit{SE}}_{m} \mid \mathit{SE}, \mathit{SE}_{m-1} \mid \mathit{SE}, \mathit{SE}, \mathit{SE}_{m-2} \mid \cdots \\
\mathit{PF1}_1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{Bounds} \mid \mathsf{I\_array} \mid \mathsf{F\_array} \\
\mathit{PF1}_2 & ::= & \mathsf{Decons} \mid \mathsf{Detuple}_2 \\
\mathit{PF1}_m & ::= & \mathsf{Detuple}_m \\
\mathit{PF2}_1 & ::= & + \mid - \mid * \mid \mathsf{Equal?} \mid \cdots \mid \mathsf{And} \mid \mathsf{Cons} \mid \mathsf{Apply} \mid \mathsf{P\_select} \mid \mathsf{Make\_tuple}_2 \\
\mathit{PFN}_1 & ::= & \mathsf{Make\_tuple}_n \\
\mathit{Ap\_E}_m & ::= & \mathsf{Ap}_{n,m}\ (\mathit{Variable},\ \mathit{SE}_n) \\
\mathit{Cond\_E}_m & ::= & \mathsf{If}_m\ \mathit{SE}\ \mathsf{then}\ E_m\ \mathsf{else}\ E_m \\
\mathit{Case\_E}_m & ::= & \{\mathsf{Case}_m\ \mathit{SE}\ \mathsf{of}\ \mathsf{Nil} = E_m \mid \mathsf{Cons}\ \mathit{Variable}\ \mathit{Variable} = E_m\} \\
\mathit{Loop\_E}_m & ::= & \mathsf{Loop}_m\ (\mathit{SE}_{m+3}) \\
\mathit{lambda\_E} & ::= & \lambda_{n,m}\ (\mathit{MV}_n)\ .\ (E_m) \\
E_1 & ::= & \mathit{SE}_1 \mid \mathit{PF1}_1\ (\mathit{SE}) \mid \mathit{PF2}_1\ (\mathit{SE}_2) \mid \mathit{PFN}_1\ (\mathit{SE}_n) \mid \mathit{Ap\_E}_1 \mid \mathit{Cond\_E}_1 \mid \\
 & & \mathit{Block}_1 \mid \mathit{Loop\_E}_1 \mid \mathit{Case\_E}_1 \mid \mathit{lambda\_E} \\
E_2 & ::= & \mathit{SE}_2 \mid \mathit{PF1}_2\ (\mathit{SE}) \mid \mathit{Ap\_E}_2 \mid \mathit{Cond\_E}_2 \mid \mathit{Block}_2 \mid \mathit{Loop\_E}_2 \mid \mathit{Case\_E}_2 \\
E_m & ::= & \mathit{SE}_m \mid \mathit{PF1}_m\ (\mathit{SE}) \mid \mathit{Ap\_E}_m \mid \mathit{Cond\_E}_m \mid \mathit{Block}_m \mid \mathit{Loop\_E}_m \mid \mathit{Case\_E}_m \\
\mathit{Block}_m & ::= & \{_m\ [\mathit{Statement};]^*\ \mathsf{In}\ \mathit{SE}_m\} \\
\mathit{Statement} & ::= & \mathit{Binding} \mid \mathit{Command} \\
\mathit{Binding} & ::= & \mathit{MV}_m = E_m \\
\mathit{Command} & ::= & \mathsf{P\_store}\ (\mathit{SE},\ \mathit{SE},\ \mathit{SE}) \mid \mathsf{Store\_error} \mid \top_s \\
\mathit{Program} & ::= & \mathit{Block}
\end{array}
$$

Figure 3: Syntax of PGL

1. Translate Expression: TE: Id^- Expression $\longrightarrow$ PGL Expression

2. Translate Statement: TS: Id^- Statement $\longrightarrow$ list (PGL Statement)

3. Translate Binding: TB: Id^- Binding $\longrightarrow$ list (PGL Binding)

4. Translate Operator: TO: Id^- Operator $\longrightarrow$ (PGL Operator)

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 PGL expression. The whole translation is given in terms of syntactic categories. The proper way of reading the translation function such as "$\texttt{TE}[\![(\mathit{UOP}\ e_1)]\!] = \{t_1 = \texttt{TE}[\![e_1]\!];\ t = \texttt{TO}[\![\mathit{UOP}]\!]\ (t_1);\ \mathsf{In}\ t\}$ " is that "TE" when applied to a unary expression in Id^- produces the PGL expression on the right-hand-side.
Throughout the emphasis is on clarity of Id^- to PGL translation rather than its efficiency.

We will use the following convention for metavariables

$$\begin{array}{lcl} c & \in & \text{Constant} \\ e_i & \in & \text{Expression} \\ x_i & \in & \text{Variable} \\ s_i & \in & \text{Statement} \\ p_i & \in & \text{Pattern} \end{array}$$

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 $[\overrightarrow{y_n} / \overrightarrow{x_n}]$ for $[y_1/x_1, \cdots, y_n/x_n]$. The variables named "$t_i$" that appear in the translated expression represent new PGL variables, that is, they are not metavariables.

3.2.1 TE: Id^- Expression $\longrightarrow$ PGL Expression

$\texttt{TE}[\![x]\!] = x$

$\texttt{TE}[\![\mathsf{Next}\ x]\!] = \mathit{next}(x)$
Where "*next*" is a function on identifiers that keeps the association between a variable and its correspondent nextified version.

$\texttt{TE}[\![c]\!] = c$

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

$$\begin{array}{lcl} \texttt{TE}[\![(e_1\ \mathit{BOP}\ e_2)]\!] & = & \{\ t_1 \ = \ \texttt{TE}[\![e_1]\!]; \\ & & \quad t_2 \ = \ \texttt{TE}[\![e_2]\!]; \\ & & \quad t \ = \ \texttt{TO}[\![\mathit{BOP}]\!]\ (t_1,\ t_2); \\ & & \quad \mathsf{In}\ t\} \end{array}$$

$$
\begin{aligned}
\mathtt{TE}[\![(e_1\ e_2)]\!] \;=\; \{\ t_1 &= \mathtt{TE}[\![e_1]\!];\\
t_2 &= \mathtt{TE}[\![e_2]\!];\\
t &= \mathsf{Apply}\ (t_1,\ t_2);\\
&\mathsf{In}\ \ t\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{TE}[\![(e_1,\cdots,e_n)]\!] \;=\; \{\ t_1 &= \mathtt{TE}[\![e_1]\!];\\
&\vdots\\
t_n &= \mathtt{TE}[\![e_n]\!];\\
t &= \mathsf{Make\_tuple_n}\ (\overrightarrow{t_n});\\
&\mathsf{In}\ \ t\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{TE}[\![e_1[e_2]]\!] \;=\; \{\ t_1 &= \mathtt{TE}[\![e_1]\!];\\
t_2 &= \mathtt{TE}[\![e_2]\!];\\
t &= \mathsf{Ap_2}\ (\mathsf{Select},\ t_1,\ t_2)\\
&\mathsf{In}\ \ t\}
\end{aligned}
$$

where **Select** is a standard function definition in PGL.

$$
\begin{aligned}
\mathtt{TE}[\![(\mathsf{If}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)]\!] \;=\; \{\ t_1 &= \mathtt{TE}[\![e_1]\!];\\
t &= \mathsf{If}\ t_1\ \mathsf{then}\ \mathtt{TE}[\![e_1]\!]\ \mathsf{else}\mathtt{TE}[\![e_2]\!]\\
&\mathsf{In}\ \ t\}
\end{aligned}
$$

$$
\begin{array}{lll}
\mathtt{TE}[\![\ \mathsf{Case}\ e\ \mathsf{of} & = & \{\ t_1 = \mathtt{TE}[\![e]\!];\\
\quad \mathsf{Nil} \quad = e_1 & & \quad t_2 = \{\ \mathsf{Case}\ t_1\ \mathsf{of}\\
\quad \mid p_1 : p_2 \ = e_2]\!] & & \qquad \mathsf{Nil} \qquad\qquad = \mathtt{TE}[\![e_1]\!]\\
 & & \qquad \mid \mathsf{Cons}\ t_3\ t_4 \quad = \{\ \mathtt{TB}[\![p_1 = t_3]\!];\\
 & & \qquad\qquad\qquad\qquad\quad \mathtt{TB}[\![p_2 = t_4]\!];\\
 & & \qquad\qquad\qquad\qquad\quad t_5 = \mathtt{TE}[\![e_2]\!];\\
 & & \qquad\qquad\qquad\qquad\quad \mathsf{In}\ t_5\}\}\\
 & & \quad \mathsf{In}\ t_2\}
\end{array}
$$

$$
\begin{aligned}
\mathtt{TE}[\![\{s_1;\cdots s_n;\ \mathsf{In}\ e\}]\!] \;=\; &\{\mathtt{TS}[\![s_1]\!];\\
&\ \vdots\\
&\mathtt{TS}[\![s_n]\!];\\
&t = \mathtt{TE}[\![e]\!];\\
&\mathsf{In}\ \ t\}
\end{aligned}
$$

$$
\begin{array}{lll}
\mathtt{TE}[\![\ \{\mathsf{While}\ e\ \mathsf{do} & = & \mathtt{TSLE}[\![\ \{\mathsf{While}\ \mathtt{TE}[\![e]\!]\ \mathsf{do}\\
\quad s_1; & & \qquad \mathtt{TS}[\![s_1]\!];\\
\quad \vdots & & \qquad \vdots\\
\quad s_n; & & \qquad \mathtt{TS}[\![s_n]\!];\\
\quad \mathsf{Finally}\ e_f\ \}]\!] & & \qquad \mathsf{Finally}\ \mathtt{TE}[\![e_f]\!]\ \}]\!]
\end{array}
$$

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

$$
\begin{array}{llll}
\mathsf{TSLE}[\![\ \{\mathsf{While}\ e\ \mathsf{do} & = \{\ P & = & \underline{\lambda_n}\,(\overrightarrow{x_n})\,.\,(e); \\
\quad next(x_1) = e_1; & \quad B & = & \underline{\lambda_{n,n}}\,(\overrightarrow{x_n})\ .\ (\{_{\mathsf{n}}\ next(x_1) = e_1; \\
\quad \vdots & & & \qquad\qquad \vdots \\
\quad next(x_n) = e_n; & & & \qquad\qquad next(x_n) = e_n; \\
\quad y_1 = e_{y_1}; & & & \qquad\qquad y_1 = e_{y_1}; \\
\quad \vdots & & & \qquad\qquad \vdots \\
\quad \mathsf{Finally}\ e_f\ \}]\!] & & & \qquad\qquad \mathsf{In}\ next(x_1), \cdots, next(x_n)\ \}) \\
& \quad t_p & = & \underline{\mathsf{Ap_n}}\,(P,\ \overrightarrow{x_n}); \\
& \quad \overrightarrow{t_n} & = & \mathsf{Loop_n}\,(P,\ B,\ \overrightarrow{x_n},\ t_p); \\
& \quad t_f & = & e_f[\,\overrightarrow{t_n}\ /\ \overrightarrow{x_n}\,] \\
& \quad\ \mathsf{In}\ t_f\} & &
\end{array}
$$

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.

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

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

$$\mathtt{TS}[\![p = e]\!] \ = \ (t = \mathtt{TE}[\![e]\!];\ \mathtt{TB}[\![p = t]\!])$$

$$
\begin{array}{lll}
\mathtt{TS}[\![\mathtt{Def}\ F\ p_1\ p_2 \cdots p_n = e]\!] & = & (F = \lambda_n\,(\overrightarrow{t_n})\ .\ (\{\ \mathtt{TB}[\![p_1 = t_1]\!]; \\
& & \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\qquad \mathtt{TB}[\![p_n = t_n]\!]; \\
& & \qquad\qquad\qquad t = \mathtt{TE}[\![e]\!]; \\
& & \qquad\qquad\qquad\ \mathsf{In}\ t\ \}))
\end{array}
$$

In case of Defsubst we generate an underlined λ-expression, $\underline{\lambda}$.

$$
\begin{array}{llll}
\mathtt{TS}[\![e_1[e_2] = e_3]\!] \ = \ (& t_1 & = & \mathtt{TE}[\![e_1]\!]; \\
& t_2 & = & \mathtt{TE}[\![e_2]\!]; \\
& t_3 & = & \mathtt{TE}[\![e_3]\!]; \\
& t & = & \mathsf{Ap_3}\,(\mathsf{Store},\ t_1,\ t_2,\ t_3))
\end{array}
$$

where Store is a standard function defined in PGL

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

$$
\begin{aligned}
\mathtt{TB}[\![(p_1,\cdots,p_n) = x]\!] \;&=\; (\,\overrightarrow{t_n} = \mathsf{Detuple}_n\,(x);\\
&\qquad \mathtt{TB}[\![p_1 = t_1]\!];\\
&\qquad\quad \vdots\\
&\qquad \mathtt{TB}[\![p_n = t_n]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{TB}[\![(p_1\ :\ p_2) = x]\!] \;&=\; (t_1,\, t_2 = \mathsf{Decons}\,(x);\\
&\qquad \mathtt{TB}[\![p_1 = t_1]\!];\\
&\qquad \mathtt{TB}[\![p_2 = t_2]\!])
\end{aligned}
$$

$$\mathtt{TB}[\![y = x]\!] \;=\; (y = x)$$

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

3.2.4 TO: Id$^-$ Operator $\longrightarrow$ PGL Operator

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

$\mathtt{TO}[\![BOP]\!] =$ the corresponding PGL $PF2$

3.3 Definition of Standard Functions

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

```
Def Make_array (l,u) f = { b = (l,u);
                           a = F_array  (b);
                           i = l;
                           {  While i ≤ u  do
                                    a[i] = f i;
                                    next i = i+1 }
                              In  a };

Def Select x i = { (l,u) =  Bounds x;
                       In
                            If i > u  Or i < l  Then
                                                  Error
                                                Else
                                                  P_select x i };

Def Store x i y = { (l,u) =  Bounds x;
                        In
                             If i > u  Or i < l  Then
                                               { Store_error In () }
                                             Else
                                               {P_store x i y In () } };
```

where () represents the void value.

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

TE[[P_select x i]] = P_select (x, i)

TS[[P_store x i y]] = P_store (x, i, y)

TS[[F_array x]] = F_array (x)

Notice that F_array is used instead of I_array to facilitate type checking of functional arrays, which have a different degree of polymorphism than I_array.

3.4 The Rewrite Rules of PGL (R_{PGL})

We now present a set of rewrite rules, R_{PGL}, to define the operational semantics for PGL. R_{PGL} is a Contextual Rewrite System described in [2]. In the following, n and $\underline{n}$ represent a meta-variable and a numeral, respectively. All the variables that appear on the left-hand-side of the rules are meta-variables that range over appropriate syntactic categories. We assume that a primitive function is only applied to arguments of appropriate types, i.e., the type checking has been done statically. The new variables, t_i, that appear on the RHS of a rule represent new PGL variables. We will assume all variables have been assigned unique names and questions about lexical scopes have been resolved.

We will make use of the following metavariables

$$\begin{array}{lll} X_i, Z_i, Y & \in & \textit{Variable} \text{ and } \textit{Constant} \\ C & \in & \textit{Constant} \\ \underline{i} & \in & \textit{Integer} \\ S_i,\ SS_i & \in & \textit{Statement} \\ E & \in & \textit{Expression} \end{array}$$

As before we will use the notation $\overrightarrow{x_n}$, $\overrightarrow{X_n}$ to indicate multiple variable and metavariable respectively.

- **δ rules**

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

- **Conditional rules**

$$\begin{array}{lll} \mathsf{If_n\ True\ then}\ E1\ \mathsf{else}\ E2 & \longrightarrow & E1 \\ \mathsf{If_n\ False\ then}\ E1\ \mathsf{else}\ E2 & \longrightarrow & E2 \end{array}$$

- **Loop rules**

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

$$\mathsf{Loop_n}\ (P,\ B,\ \overrightarrow{X_n},\ \mathsf{False}) \longrightarrow \overrightarrow{X_n}$$

- **Tuple rules**

$$\frac{X = \mathsf{Make\_tuple_n}\ (\overrightarrow{X_n})}{\mathsf{Detuple_n}\ (X) \longrightarrow \overrightarrow{X_n}}$$

- **List rules**

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

$$\mathsf{Decons}\ (\mathsf{Nil}) \longrightarrow \mathsf{Error}, \mathsf{Error}$$

$$\{\mathsf{Case_n}\ \mathsf{Nil\ of}\ \ \mathsf{Nil} = E1 \ \mid\ \mathsf{Cons}\ Y_1\ Y_2 = E2\} \longrightarrow E1$$

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\{\mathsf{Case_n}\ X\ \mathsf{of}\ \ \mathsf{Nil}{=}E1\ \mid\ \mathsf{Cons}\ Y_1\ Y_2{=}E2\} \longrightarrow E2[(X_1/Y_1),(X_2/Y_2)]}$$

where $e[X/Y]$ represents naive substitution.

- **Array rules**

$$\mathsf{F\_array}\ (X) \longrightarrow \mathsf{I\_array}\ (X)$$

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

$$\frac{\begin{array}{lcll} X_b & = & \mathsf{Make\_tuple}\ (\underline{l},\ \underline{u}) & \mid \\ X & = & \mathsf{I\_array}\ (X_b) & \mid \\ \multicolumn{4}{l}{\mathsf{P\_store}\ (X,\ \underline{i},\ Y)} \end{array}}{\mathsf{P\_select}\ (X, \underline{i}) \longrightarrow Y}$$

$$\frac{\begin{array}{lcl} X_b & = & \mathsf{Make\_tuple}\ (\underline{l},\ \underline{u}) \quad | \\ X & = & \mathsf{I\_array}\ (X_b) \quad | \\ \multicolumn{3}{l}{\mathsf{P\_store}\ (X,\ \underline{i},\ Y)} \end{array}}{\mathsf{P\_store}\ (X, \underline{i}, Y') \longrightarrow \top_s}$$

- **Arity Detection rules**

$$\frac{\begin{array}{lcl} F & = & \lambda_{\mathsf{n}}\ (\overrightarrow{Z_n})\ .\ (E) \quad | \\ F_1 & = & \mathsf{Apply}\ (F,\ X_1) \quad | \\ F_2 & = & \mathsf{Apply}\ (F_1,\ X_2) \quad | \\ \vdots & & \\ F_{n-1} & = & \mathsf{Apply}\ (F_{n-2},\ X_{n-1}) \end{array}}{\mathsf{Apply}\ (F_{n-1}, X_n) \longrightarrow \mathsf{Ap_n}\ (F, \overrightarrow{X_n})}$$

If the corresponding λ-expression is underlined then we generate "$\underline{\mathsf{Ap_n}}\ (F,\ \overrightarrow{X_n})$".

- **Application rules**

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

This substitution will require renaming of bound variables.

- **Multivariable rule**

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

- **Substitution rules**

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

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

- **Block Flattening rules**

$$\begin{array}{lcl} \{\ \overrightarrow{X_n} = \{SS_1;\ SS_2;\ \cdots & & \{\overrightarrow{X_n} = \overrightarrow{Y_n}; \\ \qquad \mathsf{In}\ \ \overrightarrow{Y_n}\} & \longrightarrow & SS_1';\ SS_2';\ \cdots\ ; \\ S_1;\ \cdots\ ;\ S_n\ ; & & S_1;\ \cdots\ ;\ S_n\ ; \\ \mathsf{In}\ \ Z\} & & \mathsf{In}\ \ Z\} \end{array}$$

- **Propagation of $\top$**

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

4 The second phase: optimizations on PGL programs

Following is a partial list of optimizations rules for PGL. Optimizations include most of the R_{PGL} rules. Optimizations should be performed after type checking.

- **Common Subexpression Elimination rule**

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

Primitive functions F_array, I_array, Apply and $\mathsf{Ap}_{n,m}$ are excluded from this optimization.

- **Fetch Elimination rules**

$$\frac{\mathsf{Store}\ (X_1,\ X_2,\ X_3)}{\mathsf{Select}\ (X_1,\ X_2) \longrightarrow X_3}$$

- **Algebraic Identity rules**

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

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

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

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

$$\vdots$$

The above rules preserve total correctness.

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

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

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

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

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

$$\vdots$$

These rules preserve only partial correctness. Any algebraic rule that does not have a precondition can be included.

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

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

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

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

$$\vdots$$

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

- **Partial Evaluation**

$$\frac{F = \lambda_{n,m}\ (\overrightarrow{Z_n})\ .\ E}{\textsf{Apply}\ (F, X) \ \longrightarrow\ \begin{array}{l}\{F' = \lambda_{n-1,m}\ (\overrightarrow{Z_{2,n}})\ .\ E[X/Z_1] \\ \quad \textsf{In}\ F'\}\end{array}}$$

- **Lift free expressions**

$$\frac{\&\ FE(E, \lambda_{\mathsf{n,m}}\ (\overrightarrow{Z_n})\ .\ (\{Y = E;\ S_1;\ \textsf{In}\ X\})}{\lambda_{\mathsf{n,m}}\ (\overrightarrow{Z_n})\ .\ (\{Y = E;\ S_1;\ \textsf{In}\ X\}) \ \longrightarrow\ \begin{array}{rcl}\{\ t_1 & = & E; \\ t & = & \lambda_{\mathsf{n,m}}\ (\overrightarrow{Z_n})\ .\ (\{Y = t_1;\ S_1;\ \textsf{In}\ X\}) \\ \textsf{In}\ t\} & & \end{array}}$$

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.

- **Hoisting code out of a conditional**

$$\frac{\&\ FE(E, (\textsf{If}_{\mathsf{n}}\ X\ \textsf{then}\ \{X = E;\ S\}\ \textsf{else}\ \{X' = E; S'\}))}{\textsf{If}_{\mathsf{n}}\ X\ \textsf{then}\ \{X = E;\ S\}\ \textsf{else}\ \{X' = E; S'\} \ \longrightarrow\ \begin{array}{rcl}\{\ t_1 & = & E; \\ t & = & \textsf{If}_{\mathsf{n}}\ X\ \textsf{then}\ \{X = t_1;\ S;\}\ \textsf{else}\ \{X' = t_1; S';\} \\ \textsf{In}\ t\} & & \end{array}}$$

- **Eliminating circulating constants**

In case the nextified variable is a constant among the different iterations, that is, in the loop body there exists an expression like "`next(x) = x` ", then the variable `x` can become a free variable of the loop body avoiding the circulation of `x`. Without loss of generality we assume that the nextified variable to be eliminated is the last one.

$$\frac{\begin{array}{l} X_p = \mathsf{Ap_n}\,(P,\ \overrightarrow{X_n}) \\ \overrightarrow{X'_n} = \mathsf{Loop_n}\,(P,\ B,\ \overrightarrow{X_n},\ X_p) \end{array} \quad \begin{array}{l} P = \lambda_{\mathsf{n}}\,(\overrightarrow{X_n})\,.\,(E) \\ B = \lambda_{\mathsf{n,n}}\,(\overrightarrow{X_n})\,.\,(\{S \mathrel{\mathsf{In}} next(X_1), \cdots, next(X_n),\ X_n\}) \end{array}}{} \longrightarrow$$

$$\begin{array}{lcl} P' & = & \lambda_{\mathsf{n-1}}\,(\overrightarrow{X_{n-1}})\,.\,(E[X_n/next(X_n)]); \\ B' & = & \lambda_{\mathsf{n-1}}\,(\overrightarrow{X_{n-1}})\,.\,(\{S \mathrel{\mathsf{In}} next(X_1), \cdots, next(X_{n-1})\}[X_n/next(X_n)]); \\ X_p & = & \mathsf{Ap_{n-1}}\,(P',\ \overrightarrow{X_{n-1}}) \\ \overrightarrow{X'_{n-1}} & = & \mathsf{Loop_{n-1}}\,(P',\ B',\ \overrightarrow{X_{n-1}},\ X_p);\ \ X'_n = X_n; \end{array}$$

- **Eliminating circulating variables**

  Consider the following example

```
x = 0;
y = 5;
in
  {while p(x) do
            next x = a + b;
            next y = x + 1;
           Finally x + y};
```

after having applied the "lift free expression" transformation we will obtain

```
x = 0;
y = 5;
t = a + b;
in
  {while p(x) do
     next x = t;
     next y = x + 1;
   Finally x + y};
```

Notice that in the first iteration the nextified variable "x" takes value "0". However, from the second iteration on the value of "x" is constant. Thus, we can avoid circulating the nextified variable "x" by unrolling the loop once.

```
if p(x) then
  { x1 = t;
    y  = x + 1;
   in
     {while p(x1) do
        next y = x1 + 1;
      finally x1+ y};
   }
  else x + y;
```

$$P = \lambda_{\mathsf{n}}\ (\overrightarrow{X_n})\ .\ (E) \qquad |$$
$$B = \underbrace{\lambda_{\mathsf{n,n}}\ (\overrightarrow{X_n})\ .\ (\{S\ \mathsf{In}\ next(X_1), \cdots, next(X_{n-1}), Z_n\})}_{\rho}\ \ \&\ \ FE(Z_n, \rho)$$

$$\overrightarrow{X'_n} = \mathsf{Loop_n}\ (P,\ B,\ \overrightarrow{X_n},\ X) \longrightarrow$$
$$\begin{array}{l}\overrightarrow{X'_n} = \mathsf{If}\ X\ \mathsf{then}\ \{\ \begin{array}{lcl} P' & = & \lambda_{\mathsf{n-1}}\ (\overrightarrow{X_{n-1}})\ .\ (E[t_n/next(X_n)]); \\ B' & = & \lambda_{\mathsf{n-1,n}}\ (\overrightarrow{X_{n-1}})\ .\ (\{S\ \mathsf{In}\ next(X_1), \cdots, next(X_n)\}[t_n/next(X_n)]) \\ \overrightarrow{t_n} & = & \mathsf{Ap_{n,n}}\ (B,\ \overrightarrow{X_n}); \\ p & = & \mathsf{Ap_{n-1}}\ (P',\ \overrightarrow{t_{n-1}}); \\ \overrightarrow{t'_{n-1}} & = & \mathsf{Loop_{n-1}}\ (P',\ B',\ \overrightarrow{t_{n-1}},\ p); \\ \multicolumn{3}{l}{\mathsf{In}\ \overrightarrow{t'_{n-1},\ t_n}\}} \end{array} \\ \quad \mathsf{else}\ \ \overrightarrow{X_n} \end{array}$$

The following optimizations are only applicable to for loops.

- **Peeling the loop once**

$$\mathsf{Loop_n}\ (P,\ B,\ \overrightarrow{X_n},\ X) \longrightarrow \begin{array}{l}\mathsf{If}\ X\ \mathsf{then}\ \{\ \begin{array}{lcl} \overrightarrow{t_n} & = & \mathsf{Ap_{n,n}}\ (B,\ \overrightarrow{X_n}); \\ p & = & \mathsf{Ap_n}\ (P,\ \overrightarrow{t_n}); \\ \overrightarrow{t'_n} & = & \mathsf{Loop_n}\ (P,\ B,\ \overrightarrow{t_n},\ p); \\ \multicolumn{3}{l}{\mathsf{In}\ \ \overrightarrow{t'_n}\ \}} \end{array} \\ \mathsf{else}\ \overrightarrow{X_n} \end{array}$$

- **Loop body unrolling K times**

Without loss of generality we assume that the index variable is the first one.

$$P = \lambda_{\mathsf{n}}\ (X_1)\ .\ (X_1\ \leq\ \underline{m})\ \ |$$
$$B = \lambda_{\mathsf{n,n}}\ (\overrightarrow{X_n})\ .\ \{next(X_1) = X_1 + 1;\ S_1; \cdots;\ \mathsf{In}\ next(X_1), \cdots, next(X_n)\}$$
$$\&\ \ \mathrm{remainder}(\underline{m}, \underline{k}) = 0$$

$$\mathsf{Loop_n}\ (P,\ B,\ \overrightarrow{X_n}, X_p) \longrightarrow \{\ B' = \lambda_{\mathsf{n,n}}(\overrightarrow{X_n})\ .\ \{\ \begin{array}{lcl} \overrightarrow{t_n^1} & = & \underline{\mathsf{Ap_{n,n}}}\ (B,\ \overrightarrow{X_n}); \\ \overrightarrow{t_n^2} & = & \underline{\mathsf{Ap_{n,n}}}\ (B,\ \overrightarrow{t_n^1}); \\ & \vdots & \\ \overrightarrow{t_n^k} & = & \underline{\mathsf{Ap_{n,n}}}\ (B,\ \overrightarrow{t_n^{k-1}}) \\ \multicolumn{3}{l}{\mathsf{In}\ \ \overrightarrow{t_n^k}\ \}} \end{array}$$
$$\begin{array}{lcl} \overrightarrow{t_n} & = & \mathsf{Loop_n}\ (P,\ B',\ \overrightarrow{X_n}, X_p); \\ \multicolumn{3}{l}{\mathsf{In}\ \ \overrightarrow{t_n}\ \}} \end{array}$$

$$
\begin{array}{lll}
\mathit{Integer} & ::= & 1 \mid 2 \mid \cdots \mid \underline{n} \mid \cdots \\
\mathit{Boolean} & ::= & \mathsf{True} \mid \mathsf{False} \\
\mathit{Variable} & ::= & x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid x_2 \mid \cdots \\
\mathit{MV}_m & ::= & \underbrace{\mathit{Variable}, \cdots, \mathit{Variable}} \\
\mathit{Label} & ::= & L \mid L1 \mid \cdots \mid L' \mid \cdots \\
\mathit{PF}1 & ::= & \mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{Allocate} \\
\mathit{PF}2 & ::= & + \mid - \mid * \mid \cdots \mid \mathsf{Less} \mid \mathsf{Equal?} \mid \mathsf{P\_select} \\
\mathit{PF}3 & ::= & \mathsf{Ack\_store} \\
\mathit{UDF} & ::= & F \mid G \mid \cdots \\
\mathit{SE} & ::= & \mathit{Variable} \mid \mathit{UDF} \mid \mathit{GV} \\
\mathit{SE}_m & ::= & \underbrace{\mathit{SE}, \cdots, \mathit{SE}}_{m} \\
\mathit{GV} & ::= & \mathit{Integer} \mid \mathit{Boolean} \mid \mathit{Label} \mid \mathsf{Error} \mid \top \\
E_1 & ::= & \mathit{SE} \mid \mathit{PF}1\ (\mathit{SE}) \mid \mathit{PF}2\ (\mathit{SE}_2) \mid \mathit{PF}3\ (\mathit{SE}_3) \mid \mathsf{Ap_n}\ (\mathit{SE}_{n+1}) \mid \\
 & & \mathsf{Dispatch_n}\ (\mathit{SE}, \underbrace{\mathit{SE}, \cdots, \mathit{SE}}_{n}) \mid \mathsf{Loop_1}\ (\mathit{SE}_4) \mid \mathit{Block} \\
E_m & ::= & \mathit{SE}_m \mid \mathsf{Ap_{n,m}}\ (\mathit{SE}_{n+1}) \mid \mathsf{Dispatch_{n,m}}\ (\mathit{SE}, \underbrace{\mathit{SE}_m, \cdots, \mathit{SE}_m}_{n}) \mid \mathsf{Loop_m}\ (\mathit{SE}_{m+3}) \mid \mathit{Block}_m \\
\mathit{Block}_m & ::= & \{[\mathit{Statement};]^*\ \mathsf{In}\ \mathit{SE}_m\} \\
\mathit{Statement} & ::= & \mathit{Binding} \mid \mathit{Command} \mid \mathsf{Store\_Error} \\
\mathit{Command} & ::= & \mathsf{P\_store}\ (\mathit{SE}_3) \mid \top_s \\
\mathit{Binding} & ::= & \mathit{MV}_m = E_m
\end{array}
$$

Figure 4: The Grammar of P-TAC.

The condition that the remainder, call it r, of $\underline{m}/\underline{k}$ equals zero can be easily removed by executing the loop r times first.

5 The third phase: from PGL to P-TAC

5.1 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 constant that can be freely substituted.

5.2 Translation of PGL to P-TAC

Prior to translating PGL to P-TAC, λ-lifting has to be performed. A PGL program after λ-lifting only contains closed λ-expressions. The translator, given a PGL program, produces the corresponding P-TAC program and a set, "D" , of definitions. The set D is initialized with

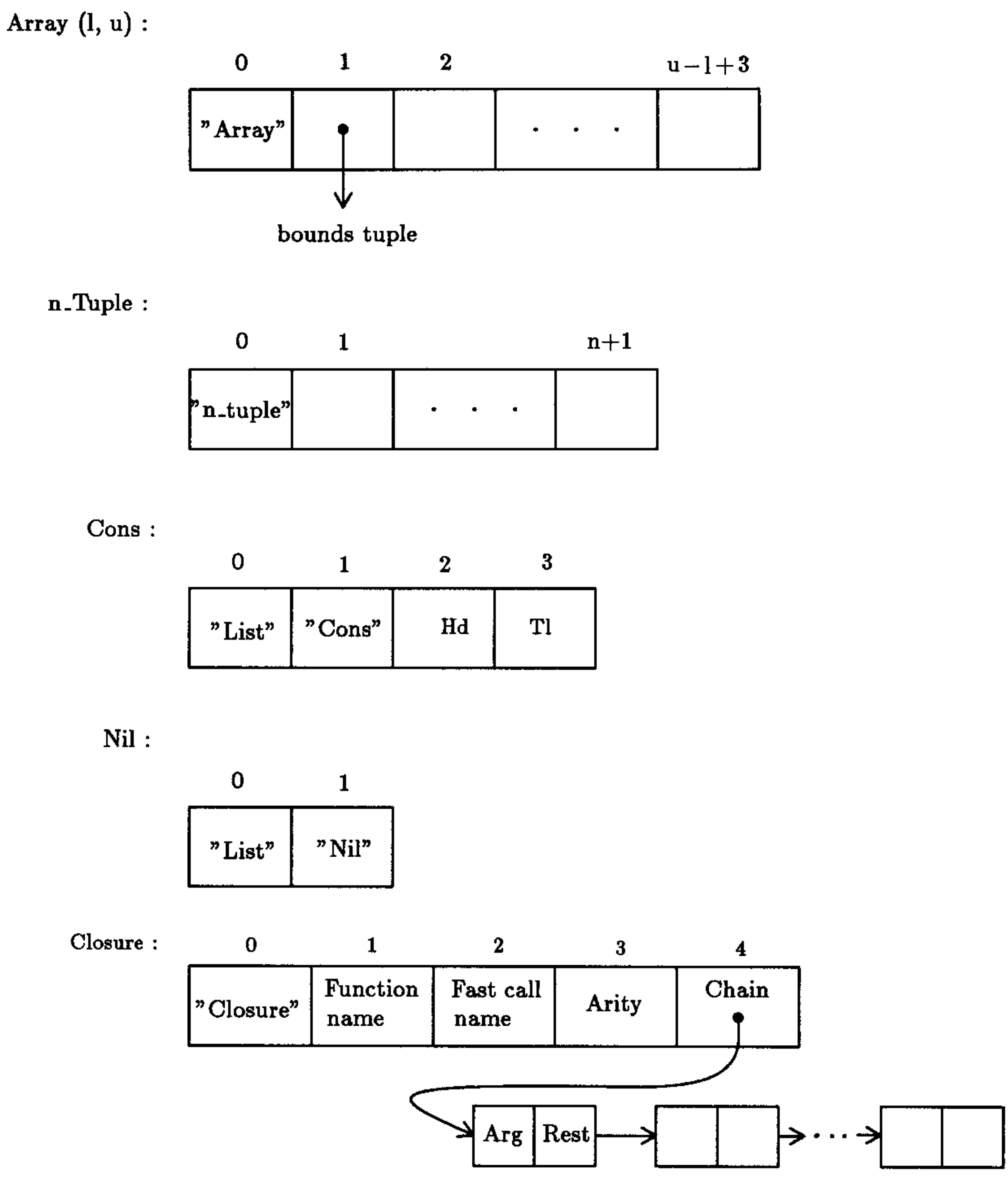

Figure 5: Representation of data structures

constants that are introduced by the translator. Data structures and closure are represented as shown in Figure 5.

5.2.1 TE: PGL Expression $\longrightarrow$ P-TAC Expression

$\mathrm{TE}[\![x]\!] = x$

$\mathrm{TE}[\![c]\!] = c$

$\mathrm{TE}[\![\mathsf{Negate}\ x]\!] = \mathsf{Negate}\ x$

The same holds for **Not**.

$\mathrm{TE}[\![\mathsf{Bounds}\ x]\!] = \mathsf{P\_select}\ (x,\ Bounds)$

$$\begin{aligned}
\mathrm{TE}[\![\mathsf{I_array}\ (x)]\!] \;=\; \{\ & l &&= \mathsf{P_select}\ (x,\ Lower); \\
& u &&= \mathsf{P_select}\ (x,\ Upper); \\
& s &&= u - l; \\
& size &&= s + 3; \\
& t &&= \mathsf{Allocate}\ (size); \\
& \mathsf{P_store}\ (t,\ Type,\ \text{``}Array\text{''}); \\
& \mathsf{P_store}\ (t,\ Bounds,\ x); \\
& \ \mathsf{In}\ t\}
\end{aligned}$$

$\mathrm{TE}[\![\mathsf{F\_array}\ (x)]\!] \;=\; \mathrm{TE}[\![\mathsf{I\_array}\ (x)]\!]$

$$\begin{aligned}
\mathrm{TE}[\![\mathsf{Cons}\ (x_1, x_2)]\!] \;=\; \{\ & t = \mathsf{Allocate}\ (Cons_size); \\
& \mathsf{P_store}\ (t,\ Type,\ \text{``}List\text{''}); \\
& \mathsf{P_store}\ (t,\ Tag,\ \text{``}Cons\text{''}); \\
& \mathsf{P_store}\ (t,\ Hd,\ x_1); \\
& \mathsf{P_store}\ (t,\ Tl,\ x_2); \\
& \ \mathsf{In}\ t\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{TE}[\![\mathsf{Make_tuple}_n\ (\overrightarrow{x_n})]\!] \;=\; \{& t = \mathsf{Allocate}\ (n+1); \\
& \mathsf{P_store}\ (t,\ Type,\ \text{``}n_tuple\text{''}); \\
& \mathsf{P_store}\ (t,\ 1,\ x_1); \\
& \vdots \\
& \mathsf{P_store}\ (t,\ \underline{n},\ x_n); \\
& \ \mathsf{In}\ t\}
\end{aligned}$$

$$
\mathrm{TE}[\![\mathsf{Apply}\ (f,\ x)]\!] \;=\; \{ \begin{array}{lcl}
n & = & \mathsf{P\_select}\ (f,\ Arity); \\
fun & = & \mathsf{P\_select}\ (f,\ Functionname); \\
as & = & \mathsf{P\_select}\ (f,\ Chain); \\
as' & = & \mathsf{Ap}\ (\mathsf{Arg\_chain},\ x,\ as); \\
cl' & = & \mathsf{Ap}\ (\mathsf{Make\_closure},\ f,\ x); \\
fire_b & = & \mathsf{Equal?}\ n\ 1; \\
fire_i & = & BooltoInt(fire_b); \\
t_1,\ t_2 & = & \mathsf{Dispatch}_{2,2}\ (fire,\ (fun,\ as'), (I, cl')); \\
res & = & \mathsf{Ap}\ (t_1,\ t_2); \\
\multicolumn{3}{l}{\quad \mathsf{In}\ res\}}
\end{array}
$$

Booltoint is a coercion function which converts **True** to 1 and **False** to 2.

$$
\mathrm{TE}[\![\mathsf{Ap}_{\mathsf{n,m}}\ (f,\ \overrightarrow{x})]\!] \;=\; \{ \begin{array}{lcl}
f' & = & \mathsf{P\_select}\ (f, Fastcallname); \\
\overrightarrow{t_m} & = & \mathsf{Ap}_{\mathsf{n,m}}\ (f',\ \overrightarrow{x}); \\
\multicolumn{3}{l}{\quad \mathsf{In}\ \overrightarrow{t_m}\ \}}
\end{array}
$$

$$
\mathrm{TE}[\![+\ (x,\ y)]\!] \;=\; +\ (x,\ y)
$$

The same holds for all strict operators in $PF2_1$.

$$
\mathrm{TE}[\![\{x_1 = e_1;\ \cdots\ x_n = e_n;\ c_1;\ \cdots\ c_m\ \mathsf{In}\ x\}]\!] \;=\; \{ \begin{array}{lcl}
x_1 & = & \mathrm{TE}[\![e_1]\!]; \\
\vdots & & \\
x_n & = & \mathrm{TE}[\![e_n]\!]; \\
c_1; & & \\
\vdots & & \\
c_m; & & \\
\mathsf{In}\ \ x\} & &
\end{array}
$$

Notice that the commands do not need any translation.

$$
\mathrm{TE}[\![\lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ (e)]\!] \;=\; \begin{array}{l}
\{cl = \mathsf{Allocate}\ (Closure\_size); \\
\ \mathsf{P\_store}\ (cl,\ Type,\ \text{"}Closure\text{"}); \\
\ \mathsf{P\_store}\ (cl,\ Functionname,\ \text{'}T_c); \\
\ \mathsf{P\_store}\ (cl,\ Fastcallname,\ \text{'}T_{fc}); \\
\ \mathsf{P\_store}\ (cl,\ Arity,\ \underline{n}); \\
\ \mathsf{P\_store}\ (cl,\ Chain,\ \text{"}End\text{"}); \\
\quad \mathsf{In}\ cl\}
\end{array}
$$

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

$$\begin{aligned} T_c = \lambda_{1,\mathsf{m}}\ (xs)\ .\ \ &\{\overrightarrow{x_n} = \mathsf{Ap}_{1,\mathsf{n}}\ (\mathsf{Args_n},\ xs);\\ &\ \ \overrightarrow{t_m} = \mathtt{TE}[\![e]\!];\\ &\ \ \mathsf{In}\ \ \overrightarrow{t_m}\};\\ T_{fc} = \lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ \ &\mathtt{TE}[\![e]\!]; \end{aligned}$$

'T_c indicates the name T_c and not the value associated to T_c. Note that $\mathtt{TE}[\![e]\!]$ can be computed only once.

$$\begin{aligned} \mathtt{TE}[\![\mathsf{Detuple_m}\ x]\!] \;=\; \{\ t_1 \;&=\; \mathsf{P\_select}\ (x, 1);\\ &\vdots\\ t_m \;&=\; \mathsf{P\_select}\ (x, \underline{m});\\ \mathsf{In}\ \ &\overrightarrow{t_m}\} \end{aligned}$$

$$\begin{aligned} \mathtt{TE}[\![\mathsf{Decons}\ x]\!] \;=\; \mathtt{TE}[\![\ \{t = \{\ &\mathsf{Case_2}\ x\ \mathsf{of}\\ &\ \ \mathsf{Nil} \;=\; \mathsf{Error}, \mathsf{Error}\\ &\mid \mathsf{Cons}\ x_1\ x_2 \;=\; x_1, x_2\}\\ \mathsf{In}\ \ t\}]\!] & \end{aligned}$$

$$\begin{aligned} \mathtt{TE}[\![\mathsf{If_n}\ x\ \mathsf{then}\ e1\ \mathsf{else}\ e2]\!] \;=\; \{\ t \;&=\; BooltoInt(x);\\ f \;&=\; \mathsf{Dispatch_2}\ (t,\ `T1_{fc},\ T2_{fc});\\ \overrightarrow{t_m} \;&=\; \mathsf{Ap_{n,n}}\ (f,\ \overrightarrow{x_n});\\ \mathsf{In}\ \ &\overrightarrow{t_m}\} \end{aligned}$$

and the following two definitions are included in D.

$$\begin{aligned} T1_{fc} &= \lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ \mathtt{TE}[\![e_1]\!];\\ T2_{fc} &= \lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ \mathtt{TE}[\![e_2]\!] \end{aligned}$$

Where $\overrightarrow{x_n} \ \in\ FV(e_1) \cup FV(e_2)$. If both e_1 and e_2 do not have any free variable then a dummy variable is used.

$$\begin{aligned} \mathtt{TE}[\![\ \mathsf{Case_m}\ x\ \mathsf{of}\ \ \ \mathsf{Nil} = e1\ \mid \mathsf{Cons}\ y_1\ y_2 = e2]\!] \;=\; \{\ t_t \;&=\; \mathsf{P\_select}\ (x,\ Tag);\\ f \;&=\; \mathsf{Dispatch_2}\ (t_t,\ `T1_{fc},\ `T2_{fc});\\ \overrightarrow{t_m} \;&=\; \mathsf{Ap_{n,n}}\ (f,\ \overrightarrow{x_n});\\ \mathsf{In}\ \ &\overrightarrow{t_m}\} \end{aligned}$$

and the following two definitions are included in D.

$$\begin{aligned} T1_{fc} &= \lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ \mathtt{TE}[\![e_1]\!];\\ T2_{fc} &= \lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n})\ .\ \{\ y_1 \;=\; \mathsf{P\_select}\ (x, Hd);\\ &\qquad\qquad\qquad\quad\ y_2 \;=\; \mathsf{P\_select}\ (x, Tl);\\ &\qquad\qquad\qquad\quad\ t \;=\; \mathtt{TE}[\![e2];\\ &\qquad\qquad\qquad\quad\ \mathsf{In}\ t\} \end{aligned}$$

where $\overrightarrow{x_n} \;\in\; FV(e_1) \cup (FV(e_2) - \{y_1, y_2\}) \cup \{x\}$

5.2.2 Standard identifiers

The set "D" is initialized with the following constant definitions.

$Lower = 1$	$Functioname = 1$	$Cons\_size = 4$	$Arg = 0$
$Upper = 2$	$Fastcallname = 2$	$Nil\_size = 2$	$Rest = 1$
$Type = 0$	$Arity = 3$	$Closure\_size = 5$	$Hd = 2$
$Cons = 1$	$Chain = 4$	$Bounds = 1$	$Tl = 3$
$Tag = 1$			

$\mathsf{I} = \lambda\,(x)\ .\ x$

$$\begin{aligned}\mathsf{Nil} = \ \{ & t = \mathsf{Allocate}\,(Nilsize);\\ & \mathsf{P\_store}\,(t,\ Type,\ \text{“}List\text{”});\\ & \mathsf{P\_store}\,(t,\ Tag,\ \text{“}Nil\text{”});\\ & \ \mathsf{In}\ t\}\end{aligned}$$

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

$$\begin{aligned}\mathsf{Args_n} = \lambda\,(x)\ .\ \{\ t_1 &= \mathsf{P\_select}\,(x,\ Chain);\\ a_n &= \mathsf{P\_select}\,(t_1,\ Arg);\\ t_2 &= \mathsf{P\_select}\,(t_1,\ Rest);\\ a_{n-1} &= \mathsf{P\_select}\,(t_2,\ Arg);\\ t_3 &= \mathsf{P\_select}\,(t_2,\ Rest);\\ &\ \vdots\\ a_1 &= \mathsf{P\_select}\,(t_{n-1},\ Arg);\\ \mathsf{In}\ \ &\overrightarrow{a_n}\}\end{aligned}$$

$$
\begin{aligned}
\mathsf{Make\_closure} = \lambda\,(cl, x)\ .\ \{\ & f &&= \mathsf{P\_select}\ (cl,\ Funcname);\\
& f_{fc} &&= \mathsf{P\_select}\ (cl,\ Fastcallname);\\
& n &&= \mathsf{P\_select}\ (cl,\ Arity);\\
& ch &&= \mathsf{P\_select}\ (cl,\ Chain);\\
& cl' &&= \mathsf{Allocate}\ (Closure\_size);\\
& \mathsf{P\_store}\ (cl',\ Type,\ \text{``}Closure\text{''});\\
& \mathsf{P\_store}\ (cl',\ Functionname,\ f);\\
& \mathsf{P\_store}\ (cl',\ Fastcallname,\ f_{fc});\\
& \mathsf{P\_store}\ (cl',\ Arity,\ n');\\
& \mathsf{P\_store}\ (cl',\ Chain,\ ch');\\
& n' &&= n - 1;\\
& ch' &&= \mathsf{Ap}\ (\mathsf{Arg\_chain},\ x,\ ch);\\
& \quad \mathsf{In}\ cl'\}
\end{aligned}
$$

5.3 Rewrite rules of P-TAC

In the following V stands for a ground value.

- **δ rules**
- **Conditional rules**

$$\mathsf{Dispatch}\ (\underline{i},\ \overrightarrow{X_{i-1}},\ V_i,\ \overrightarrow{X_{i+1,n}}) \longrightarrow V_i$$

- **I_structure rules**

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

$$\frac{\mathsf{P\text{-}store}\ (L,\ \underline{i},\ V)}{\mathsf{P\text{-}select}\ (L,\ \underline{i}) \longrightarrow V}$$

$$\frac{\mathsf{P\text{-}store}\ (L,\ \underline{i},\ V)}{\mathsf{P\_store}\ (L,\ \underline{i},\ V') \longrightarrow \top_s}$$

The following rules are the same as the corresponding rules in PGL.

- **Loop rules**

$$\text{Loopn}\ (P,\ B,\ \overrightarrow{X_n},\ \text{True}) \longrightarrow \begin{array}{l}\{\text{n}\ \overrightarrow{t_n} = \text{Apn,n}\ (B,\ \overrightarrow{X_n});\\ \quad t_p = \text{Apn}\ (P,\ \overrightarrow{t_n});\\ \quad \overrightarrow{t'_n} = \text{Loopn}\ (P,\ B,\ \overrightarrow{t_n},\ t_p)\\ \quad \text{In}\ \overrightarrow{t'_n}\ \}\end{array}$$

$$\text{Loopn}\ (P,\ B,\ \overrightarrow{X_n},\ \text{False}) \longrightarrow \overrightarrow{X_n}$$

- **Block Flattening rules**

$$\begin{array}{l}\{\ \overrightarrow{X_n} = \{SS_1;\ SS_2;\ \cdots\\ \qquad \text{In}\ \overrightarrow{Y_n}\}\\ S_1;\ \cdots\ ;\ S_n\ ;\\ \text{In}\ Z\}\end{array} \longrightarrow \begin{array}{l}\{\overrightarrow{X_n} = \overrightarrow{Y_n}\ ;\\ SS'_1;\ SS'_2;\ \cdots\ ;\\ S_1;\ \cdots\ ;\ S_n\ ;\\ \text{In}\ Z\}\end{array}$$

- **Propagation of** $\top$

$$\{X = \top;\ S_1 \cdots S_n\ \text{In}\ Z\} \longrightarrow \top$$

$$\{\top_s;\ S_1 \cdots S_n\ \text{In}\ Z\} \longrightarrow \top$$

- **Multivariable rule**

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

- **Substitution rules**

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

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

- **Application rules**

$$\frac{F = \lambda\text{n,m}\ (\overrightarrow{Z_n})\ .\ E}{\text{Apn,m}\ (F,\ \overrightarrow{X_n}) \longrightarrow E'[\overrightarrow{X_n}\ /\ \overrightarrow{Z_n}]}$$

6 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" .

$$
\begin{array}{llll}
\mathsf{S}[\![\lambda_{\mathsf{n,m}}\ (\overrightarrow{x_n}).\{\ y_1 = se_1 & = \lambda_{\mathsf{n,m+1}}\ (\overrightarrow{x_n})\ .\ \{\ y_1 & = & se_1 \\
\quad \vdots & \quad \vdots & & \\
\quad y_n = se_n & \quad y_n & = & se_n \\
\quad y_{n+1} = nse_1 & \quad y_{n+1},\ S_1 & = & \mathsf{SE}[\![nse_1]\!] \\
\quad \vdots & \quad \vdots & & \\
\quad y_{n+m} = nse_m & \quad y_{n+m},\ S_m & = & \mathsf{SE}[\![nse_m]\!] \\
\quad\quad c_1 & \quad S_{m+1} & = & \mathsf{SC}[\![c_1]\!]; \\
\quad\quad \vdots & \quad \vdots & & \\
\quad\quad c_k & \quad S_{m+k} & = & \mathsf{SC}[\![c_k]\!]; \\
\quad \mathsf{In}\ \overrightarrow{r_m}\}]\!] & \quad S' & = & \mathsf{Sync}_{\mathsf{m+k+i}}\ (Deadvariables,\ \overrightarrow{S_{m+k}}); \\
 & \quad\quad \mathsf{In}\ \overrightarrow{r_m},\ S'\} & &
\end{array}
$$

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

$$\mathsf{SE}[\![\mathsf{Loop_n}\ (P, B, \overrightarrow{y_n}, y)]\!] = \mathsf{Loop'_n}\ (P, B, \overrightarrow{y_n}, S_p, y)$$

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

$$\mathsf{SE}[\![\mathsf{Ap_{n,m}}\ (f,\ \overrightarrow{x_n})]\!] = \mathsf{Ap_{n,m+1}}\ (f,\ \overrightarrow{x_n})$$
$$\mathsf{SC}[\![\mathsf{P\_store}\ (x,\ i\ z)]\!] = \mathsf{Ack\_store}\ (x,\ i\ z)$$

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:

$$
\begin{array}{lllll}
\mathsf{Loop'_n}\ (P,\ B,\ \overrightarrow{X_n}, S,\ \mathsf{True}) & \longrightarrow & \{_{\mathsf{n+1}}\ \overrightarrow{t_n}, S_b & = & \mathsf{Ap_{n,n+1}}\ (B,\ \overrightarrow{X_n}); \\
 & & t_p, S_p & = & \mathsf{Ap_{n,2}}\ (P,\ \overrightarrow{t_n}); \\
 & & S' & = & \mathsf{Sync_3}\ (S,\ S_b,\ S_p); \\
 & & \overrightarrow{t'_n}, S_l & = & \mathsf{Loop'_n}\ (P,\ B,\ \overrightarrow{t_n},\ S',\ t_p) \\
 & & \mathsf{In}\ \overrightarrow{t'_n},\ S_l\} & &
\end{array}
$$

$$\mathsf{Loop'_n}\ (P,\ B,\ \overrightarrow{X_n},\ S,\ \mathsf{False}) \longrightarrow \overrightarrow{X_n},\ S$$

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

$$\mathsf{Sync_n}\ (\overrightarrow{V_n}) \longrightarrow ()$$

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

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

[3] K. R. Traub. 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.

Contents