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.
Compilation of \(\text{Id}^-\): a subset of \(\text{Id}\)

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 \(\text{Id}\) 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 \(\text{Id}\), called \(\text{Id}^-\). \(\text{Id}^-\) should be seen as \(\text{Id}\) where enough parenthesis have been inserted to make operator associativity and precedence irrelevant. \(\text{Id}^-\) 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.

\(\text{Id}^-\) 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, makes 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 performed optimizations all nested function definitions (\(\lambda\)-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 \(\text{Id}^-\) in terms of Kernel \(\text{Id}\) [2] it will be possible to prove the correctness of the translation between \(\text{Id}^-\) 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.

\textsuperscript{6}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 N00014-88-C-0163 (Harvard).
Figure 1: Compilation scheme

![Diagram of compilation scheme]

Figure 2: Grammar of Id⁻.

```
UOP ∈ Unary Operator
BOP ∈ Binary Operator
E ∈ Expression
P ∈ Pattern

Variable ::= x | y | z | ⋯ | a | b | ⋯ | f | ⋯ | x₁ | ⋯ | F | G | ⋯
Integer ::= 1 | 2 | ⋯ | n | ⋯
Boolean ::= True | False
Constant ::= Integer | Boolean | Nil | Curried₁_ | Curried₁_Not | ⋯ | Curried₂_ | Curried₂_Not | ⋯ | Make_array
UOP ::= − | Not | Bounds | L_array
BOP ::= + | − | * | == | ⋯ | And | < | ⋯ | :
E ::= Variable | Next Variable | Constant | (UOP E) | (E BOP E) | (E E) | (E, E) | (E, E, E) | ⋯ | E[E] | (if E then E else E) |
    | {Case E of Nil = E | P : P = E} | {While E do [Statement;⁎ In 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) | ⋯ | (P : P)
Definition ::= Def Variable [P]+ = E | Defsubst Variable [P]+ = E
Program ::= Block
```
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) 1) in Id. In Id- we will represent curried version of an operator BOP as Curried2$bop$ (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 $\lambda$-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 $\lambda$-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

$$x, y = \{\{a = \cdots; b = \cdots; \ln a, b\}$$

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 $PF_2$, because all Id- procedures return only one result. We also use subscripted combinators to express a family of combinators. For example, Make_tuple$_n$ stands for Make_tuple$_2$, 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 $I_if_1$ |
| $Case$ | is the same as $Case_{1}$ |
| $Loop$ | is the same as $Loop_{1}$ |
| $Ap_n$ | is the same as $Ap_{n,1}$ |
| $\lambda_n$ | 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{align*}
MV & \in \text{Multiple Variable} \\
SE & \in \text{Simple Expression} \\
Pfi_m & \in \text{Primitive Function with } i \text{ arguments and } m \text{ outputs} \\
Ap.E_m & \in \text{Applicative Expression with } m \text{ outputs} \\
Cond.E_m & \in \text{Conditional Expression with } m \text{ outputs} \\
Loop.E_m & \in \text{Loop Expression with } m \text{ outputs} \\
\lambda.E_m & \in \text{Definition of a function with } m \text{ outputs} \\
Variable & ::= \{ x | y | z | \cdots | a | b | \cdots | f | \cdots | z_1 | \cdots \} \\
MV_m & ::= \underbrace{Variable, \cdots, Variable}_m \\
\text{Integer} & ::= \{ 1 \mid 2 \mid \cdots \} \\
\text{Constant} & ::= \{ \text{Integer} \mid \text{Float} \mid \text{Boolean} \mid \cdots \mid \text{Nil} \mid \text{Error} \mid T \} \\
SE & ::= \text{Variable} \mid \text{Constant} \\
SE_m & ::= \underbrace{SE_1, \cdots, SE}_m \mid SE, SE_{m-1} \mid SE, SE, SE_{m-2} \mid \cdots \\
PFI_1 & ::= \{ \text{Negate} \mid \text{Not} \mid \text{Bounds} \mid \text{Larray} \mid \text{F.array} \} \\
PFI_2 & ::= \{ \text{Decons} \mid \text{Detuple}_2 \} \\
PFI_m & ::= \{ \text{Detuple}_1 \} \\
PFI \_2 & ::= \{ + \mid - \mid * \mid \text{Equal?} \mid \cdots \mid \text{And} \mid \text{Cons} \mid \text{Apply} \mid \text{P.select} \mid \text{Make.tuple}_2 \} \\
PFN_1 & ::= \{ \text{Make.tuple}_1 \} \\
Ap.E_m & ::= Ap_n.m(Variable, SE_n) \\
\text{Cond.E.m} & ::= If_m(SE) \text{ then } E_m \text{ else } E_m \\
\text{Case.E.m} & ::= \{ \text{Case}_m(SE \text{ of } \text{Nil} = E_m \mid \text{Cons Variable } Variable = E_m) \} \\
\text{Loop.E.m} & ::= \text{Loop}_m(SE_{m+1}) \\
\text{lambda.E} & ::= \lambda_n.m(MV_m) \cdot (E_m) \\
E_1 & ::= \{ SE_1 \mid PFI_1(SE) \mid PFI_1(SE) \mid PFI_1(SE) \mid PFI_1(SE) \mid PFI_1(SE) \mid \text{Ap.E}_1 \mid \text{Cond.E}_1 \mid \text{Block}_1 \mid \text{Loop.E}_1 \mid \text{Case.E}_1 \} \\
E_2 & ::= \{ SE_2 \mid PFI_2(SE) \mid \text{Ap.E}_2 \mid \text{Cond.E}_2 \mid \text{Block}_2 \mid \text{Loop.E}_2 \mid \text{Case.E}_2 \} \\
E_m & ::= \{ SE_m \mid PFI_m(SE) \mid \text{Ap.E}_m \mid \text{Cond.E}_m \mid \text{Block}_m \mid \text{Loop.E}_m \mid \text{Case.E}_m \} \\
\text{Block}_m & ::= \{ m \text{[Statement;]} \ast \text{in } SE_m \} \\
\text{Statement} & ::= \text{Binding} \mid \text{Command} \\
\text{Binding} & ::= MV_m = E_m \\
\text{Command} & ::= P\text{-store}(SE, SE, SE) \mid \text{Store.error} \mid T, \\
\text{Program} & ::= \text{Block} \\
\end{align*}
\]

Figure 3: Syntax of PGL
1. Translate Expression: $\text{TE}: \text{Id}^{-} \ \text{Expression} \rightarrow \text{PGL} \ \text{Expression}$

2. Translate Statement: $\text{TS}: \text{Id}^{-} \ \text{Statement} \rightarrow \text{list} (\text{PGL} \ \text{Statement})$

3. Translate Binding: $\text{TB}: \text{Id}^{-} \ \text{Binding} \rightarrow \text{list} (\text{PGL} \ \text{Binding})$

4. Translate Operator: $\text{TO}: \text{Id}^{-} \ \text{Operator} \rightarrow (\text{PGL} \ \text{Operator})$

We will write $\text{TE}[e_1] = e_2$, 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 $\text{TE}[(\text{UOP} \ e_1)] = \{ t_1 = \text{TE}[e_1]; \ t = \text{TO}[\text{UOP}] (t_1); \ \text{ln} \ t \}$ is that $\text{TE}$ 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

\[
\begin{align*}
\ c & \in \text{Constant} \\
\ e_i & \in \text{Expression} \\
\ x_i & \in \text{Variable} \\
\ s_i & \in \text{Statement} \\
\ p_i & \in \text{Pattern}
\end{align*}
\]

Moreover, we will use the notation $\overrightarrow{x_{n,m}}$ to stand for $(x_n, \cdots, x_m)$, $\overrightarrow{x_m}$ for $(x_{1,m})$, 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 \hspace{0.5cm} \text{TE}: \text{Id}^{-} \ \text{Expression} \rightarrow \text{PGL} \ \text{Expression}

$\text{TE}[x] = x$

$\text{TE}[\text{Next} \ x] = \text{next}(x)$

Where "next" is a function on identifiers that keeps the association between a variable and its correspondent nextified version.

$\text{TE}[c] = c$

$\text{TE}[(\text{UOP} \ e)] = \{ \ t_1 = \text{TE}[e]; \ t = \text{TO}[\text{UOP}] (t_1); \ \text{ln} \ t \}$

$\text{TE}[(e_1 \ \text{BOP} \ e_2)] = \{ \ t_1 = \text{TE}[e_1]; \ t_2 = \text{TE}[e_2]; \ t = \text{TO}[\text{BOP}] (t_1, t_2); \ \text{ln} \ t \}$
\[
\begin{align*}
\text{TE}(e_1, e_2) & = \{ t_1 = \text{TE}(e_1); \\
& \quad t_2 = \text{TE}(e_2); \\
& \quad t = \text{Apply}(t_1, t_2); \\
& \quad \ln t \} \\
\text{TE}(e_1, \ldots, e_n) & = \{ t_1 = \text{TE}(e_1); \\
& \quad \vdots \\
& \quad t_n = \text{TE}(e_n); \\
& \quad t = \text{MakeTuple}_n(t_n); \\
& \quad \ln t \} \\
\text{TE}(e_1, e_2) & = \{ t_1 = \text{TE}(e_1); \\
& \quad t_2 = \text{TE}(e_2); \\
& \quad t = \text{Ap}_2(\text{Select}, t_1, t_2) \\
& \quad \ln t \} \\
\end{align*}
\]

where Select is a standard function definition in PGL.

\[
\begin{align*}
\text{TE}(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) & = \{ t_1 = \text{TE}(e_1); \\
& \quad t = \text{if } t_1 \text{ then } \text{TE}(e_1) \text{ else } \text{TE}(e_2) \\
& \quad \ln t \} \\
\text{TE}(\text{Case } e \text{ of } c) & = \{ t_1 = \text{TE}(c); \\
& \quad t_2 = \{ \text{Case } t_1 \text{ of } \\
& \quad \quad \text{Nil} = e_1 \\
& \quad \quad \text{p1 : p2 = e}_2 \} \\
& \quad \quad \text{Nil} = \text{TE}(e_1) \\
& \quad \quad \text{Cons } t_3 t_4 = \{ \text{TE}(p_1 = t_3); \\
& \quad \quad \text{TE}(p_2 = t_4); \\
& \quad \quad t_5 = \text{TE}(e_2); \\
& \quad \quad \ln t_5 \} \\
& \quad \ln t_2 \} \\
\text{TE}(s_1; \ldots; s_n; \ln e) & = \{ \text{TS}(s_1); \\
& \quad \vdots \\
& \quad \text{TS}(s_n); \\
& \quad t = \text{TE}(e); \\
& \quad \ln t \} \\
\text{TE}(\text{While } e \text{ do } s_1; \ldots; s_n; \text{ Finally } e_f) & = \{ \text{TSLE}(\text{While } e \text{ do } s_1; \ldots; s_n; \text{ Finally } e_f) \} \\
\end{align*}
\]

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 PGL syntax.
\textbf{TSLE}$\{ \text{While } e \text{ do }$
\begin{align*}
\text{next}(x_1) &= e_1; \\
\vdots \\
\text{next}(x_n) &= e_n; \quad \text{next}(x_1) = e_1; \\
\vdots \\
\text{Finally } e_f \\}
\}
\text{In } \text{next}(x_1), \ldots, \text{next}(x_n) \}
\begin{align*}
    t_p &= \text{Ap}_n(P, \overline{x_n}); \\
    \overline{t_n} &= \text{Loop}_n(P, B, \overline{x_n}, t_p); \\
    t_f &= e_f[\overline{t_n} / \overline{x_n}] \\
\end{align*}

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 \((x_1, \ldots, x_n)\) as input and \((\text{next}(x_n), \ldots, \text{next}(x_1))\) as output, because the values of the nextified variables come either from the surrounding scope or from the previous iteration. The \(\lambda\)-expressions corresponding to the predicate and loop body are underlined indicating the fact that they can be inlined at compile time.

\subsection{TS: Id$^-\text{ Statement} \rightarrow \text{list (PGL Statement)}$}

Often an Id$^-\text{ 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.}

\begin{align*}
\text{TS}[p = e] &= (t = \text{TE}[e]; \text{TB}[p = t]) \\
\text{TS[Def P_1 P_2 \ldots P_n = e]} &= (F = \lambda_n(\overline{t_n}) . \{( \text{TB}[P_1 = t_1]; \\
&\quad \vdots \\
&\quad \text{TB}[P_n = t_n]; \\
&\quad t = \text{TE}[e]; \\
&\quad \text{ln } t \}))
\end{align*}

In case of DefSubst we generate an underlined \(\lambda\)-expression, \(\Delta\).

\begin{align*}
\text{TS}[e_1[e_2] = e_3] &= (t_1 = \text{TE}[e_1]; \\
&\quad t_2 = \text{TE}[e_2]; \\
&\quad t_3 = \text{TE}[e_3]; \\
&\quad t = \text{Ap}_3(\text{Store}, t_1, t_2, t_3))
\end{align*}

where \text{Store} is a standard function defined in PGL.
3.2.3 \( \text{TB: Id}^- \text{ Bindings} \rightarrow \text{list (PGL Binding)} \)

\[
\text{TB}[(p_1, \cdots, p_n) = z] = (t_n = \text{Dieten}(x); \\
\quad \text{TB}[p_1 = t_1]; \\
\quad \vdots \\
\quad \text{TB}[p_n = t_n])
\]

\[
\text{TB}[(p_1 : p_2) = z] = (t_1, t_2 = \text{Decons}(z); \\
\quad \text{TB}[p_1 = t_1]; \\
\quad \text{TB}[p_2 = t_2])
\]

\[
\text{TB}[y = z] = (y = z)
\]

\[
\text{TB}[\text{next } y = z] = (\text{next}(y) = z)
\]

3.2.4 \( \text{TO: Id}^- \text{ Operator} \rightarrow \text{PGL Operator} \)

\( \text{TO}[UOP] = \) the corresponding PGL \text{PF1} \\
\( \text{TO}[BOP] = \) the corresponding PGL \text{PF2} \\

3.3 Definition of Standard Functions

In our translation from \( \text{Id}^- \) 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\text{array}. For the sake of brevity we give these definitions in \( \text{Id}^- \). These definitions and Make\text{array} can be written as follows

```
Def Make\text{array} (1,u) f = { b = (1,u); \\
\quad a = \text{F.array} \ (b); \\
\quad i = 1; \\
\quad \{ \text{While } i \leq u \text{ do} \ \\
\quad \quad a[i] = f \ i; \\
\quad \quad \text{next } i = i+1 \} \\
\quad \text{In } a \};
```

```
Def Select x i = { (1,u) = \text{Bounds } x; \\
\quad \text{In} \\
\quad \quad \text{If } i > u \text{ Or } i < 1 \text{ Then} \\
\quad \quad \quad \text{Error} \\
\quad \quad \quad \text{Else} \\
\quad \quad \quad \text{P.select } x \ i \};
```

```
Def Store x i y = { (1,u) = \text{Bounds } x; \\
\quad \text{In} \\
\quad \quad \text{If } i > u \text{ Or } i < 1 \text{ Then} \\
\quad \quad \quad \{ \text{Store.error In } () \} \\
\quad \quad \quad \text{Else} \\
\quad \quad \quad \{\text{P.store } x \ i \ y \ \text{In } () \} \};
```
where () represents the void value.

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

\[ T E[P\_select\ x\ i] = P\_select\ (x,\ i) \]
\[ T S[P\_store\ x\ i\ y] = P\_store\ (x,\ i,\ y) \]
\[ T S[F\_array\ x] = F\_array\ (x) \]

Notice that \( F\_array \) is used instead of \( L\_array \) to facilitate type checking of functional arrays, which have a different degree of polymorphism than \( L\_array \).

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 \( \underline{m} \) 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

\[ X_i, Z_i, Y \in Variable \hspace{1em} C \in Constant \]
\[ i \in Integer \]
\[ S_i, SS_i \in Statement \]
\[ E \in Expression \]

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

- \( \delta \) rules

\[ \delta \]
\[ +\ (m,\ n) \quad \xrightarrow{i} \quad +\ (m,\ n) \]
\[ \vdots \]
\[ \text{Equal?} \ (n,\ n) \quad \xrightarrow{i} \quad \text{True} \quad \text{(if } m \neq n \text{)} \]
\[ \text{Equal?} \ (m,\ n) \quad \xrightarrow{i} \quad \text{False} \quad \text{(if } m \neq n \text{)} \]

- Conditional rules

\[ \text{If}_{n} \hspace{1em} \text{True} \hspace{1em} \text{then} \hspace{1em} E_1 \hspace{1em} \text{else} \hspace{1em} E_2 \quad \xrightarrow{} \quad E_1 \]
\[ \text{If}_{n} \hspace{1em} \text{False} \hspace{1em} \text{then} \hspace{1em} E_1 \hspace{1em} \text{else} \hspace{1em} E_2 \quad \xrightarrow{} \quad E_2 \]
• Loop rules

\[
\text{Loop}_n (P, B, \overline{X_n}, \text{True}) \rightarrow \begin{cases} \overline{t_n} = \text{App}_n (B, \overline{X_n}); \\
\overline{t_p} = \text{App}_n (P, \overline{t_n}); \\
\overline{t'_n} = \text{Loop}_n (P, B, \overline{t_n}, \overline{t_p}) \end{cases}
\]

\[
\text{Loop}_n (P, B, \overline{X_n}, \text{False}) \rightarrow \overline{X_n}
\]

• Tuple rules

\[
\begin{align*}
X &= \text{Make\_tuple}_n (\overline{X_n}) \\
\text{Detuple}_n (X) &\rightarrow X_n
\end{align*}
\]

• List rules

\[
\begin{align*}
X &= \text{Cons} (X_1, X_2) \\
\text{Decons} (X) &\rightarrow X_1, X_2 \\
\text{Decons} (\text{Nil}) &\rightarrow \text{Error}, \text{Error}
\end{align*}
\]

\[
\begin{cases} \text{Case}_n \text{ Nil of Nil = E1} \mid \text{Cons} Y_1, Y_2 = E2 \end{cases} \rightarrow E1
\]

\[
\begin{cases} x = \text{Cons} (X_1, X_2) \\
\text{Case}_n x \text{ of Nil = E1} \mid \text{Cons} Y_1, Y_2 = E2 \end{cases} \rightarrow E2 ((X_1/Y_1), (X_2/Y_2))
\]

where \( e[X/Y] \) represents naive substitution.

• Array rules

\[
\begin{align*}
\text{F\_array} (X) &\rightarrow \text{l\_array} (X) \\
x &= \text{l\_Array} (X_b) \\
\text{Bounds} (X) &\rightarrow X_b \\
x_b &= \text{Make\_tuple} (l, u) \mid \\
x &= \text{l\_Array} (X_b) \mid \\
\text{P\_store} (X, i, Y) &\rightarrow \text{P\_select} (X, i, Y) \rightarrow Y
\end{align*}
\]
$X_b = \text{Make\_tuple}\, (i, u) \quad | \\
X = \text{l\_array}\, (X_b) \quad | \\
P\_\text{store}\, (X, i, Y) \\
P\_\text{store}\, (X, i, Y') \longrightarrow \tau$.

- **Arity Detection rules**

\[
\begin{align*}
F &= \lambda_n\,(z_n)\cdot (E) \\
F_1 &= \text{Apply}\,(F, X_1) \\
F_2 &= \text{Apply}\,(F_1, X_2) \\
\vdots \\
F_{n-1} &= \text{Apply}\,(F_{n-2}, X_{n-1}) \\
\text{Apply}\,(F_{n-1}, x_n) &\longrightarrow A_{P\_n}\,(F, x_n)
\end{align*}
\]

If the corresponding $\lambda$-expression is underlined then we generate $"A_{P\_n}\,(F, \overline{x_n})"$.

- **Application rules**

\[
\frac{F = \lambda_{n+m}\,(z_n)\cdot (E)}{A_{P\_n,m}\,(F, x_n) \longrightarrow F'[x_n/z_n]}
\]

This substitution will require renaming of bound variables.

- **Multivariable rule**

\[
\overline{x_n} = Y_n \longrightarrow (X_1 = Y_1; \cdots; X_n = Y_n)
\]

- **Substitution rules**

\[
\begin{align*}
\frac{X = Y}{x = y} \\
\frac{X = C}{x = \overline{c}}
\end{align*}
\]

- **Block Flattening rules**

\[
\begin{align*}
\{ \overline{x_n} = \{SS_1; SS_2; \cdots \} \quad &\longrightarrow \quad SS_1; SS_2; \cdots \} \\
\quad \text{ln}\ Y_n \} \\
S_1; \cdots; S_n \}; \quad &\longrightarrow \quad S_1; \cdots; S_n \}; \\
\quad \text{ln}\ Z \} \\
\quad \text{ln}\ Z \}
\end{align*}
\]

- **Propagation of $\top$**

\[
\begin{align*}
\{X = \top; S_1; \cdots; S_n \text{ln}\ Z\} \longrightarrow \top \\
\{\top; S_1; \cdots; S_n \text{ln}\ Z\} \longrightarrow \top
\end{align*}
\]

11
4 The second phase: optimizations on PGL programs

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

- **Common Subexpression Elimination rule**

\[
\frac{Y_m = P\text{FN}_m (X_n)}{P\text{FN}_m (X_n) \rightarrow Y_m}
\]

Primitive functions \(\text{F}\_\text{array}, \text{L}\_\text{array}, \text{Apply}\) and \(A_{p,n,m}\) are excluded from this optimization.

- **Fetch Elimination rules**

\[
\frac{\text{Store} (X_1, X_2, X_3)}{\text{Select} (X_1, X_2) \rightarrow X_3}
\]

- **Algebraic Identity rules**

\[
\begin{align*}
\text{And} (\text{True}, X) & \rightarrow X \\
\text{Or} (\text{False}, X) & \rightarrow X \\
+ (X, 0) & \rightarrow X \\
* (X, 1) & \rightarrow X \\
\vdots
\end{align*}
\]

The above rules preserve total correctness.

\[
\begin{align*}
\text{And} (\text{False}, X) & \rightarrow \text{False} \\
\text{Or} (\text{True}, X) & \rightarrow \text{True} \\
* (X, 0) & \rightarrow 0 \\
- (X, X) & \rightarrow 0 \\
\text{Equal?} (X, X) & \rightarrow \text{True} \\
\vdots
\end{align*}
\]

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

\[
\frac{X \equiv + (X_1, m) \quad \& \quad m > 0}{\text{Less} (X_1, X) \rightarrow \text{True}}
\]

12
\[
\begin{align*}
X &= + (X_1, m) \quad & \text{& } m > 0 \\
\text{Less } (X, X_1) &\rightarrow \text{ False} \\
X &= + (X_1, m) \quad & \text{& } m > 0 \\
\text{Greater } (X_1, X) &\rightarrow \text{ False} \\
X &= + (X_1, m) \quad & \text{& } m > 0 \\
\text{Equal? } (X_1, X) &\rightarrow \text{ False} \\
\end{align*}
\]

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

- **Partial Evaluation**

\[
\begin{align*}
F &= \lambda_n,m (\overline{Z_n}) \cdot E \\
\text{Apply } (F, X) &\rightarrow \{ F' = \lambda_{n-1,m} (\overline{Z_{2,n}}) \cdot E[X/Z_1] \} \\
\text{In } F'
\end{align*}
\]

- **Lift free expressions**

\[
\begin{align*}
&\text{& } \text{FE}(E, \lambda_n,m (\overline{Z_n}) \cdot \text{(\{Y = E; S_i; \text{ln } X\}))} \\
\lambda_n,m (\overline{Z_n}) \cdot \text{(\{Y = E; S_i; \text{ln } X\})} &\rightarrow \{ t_1 = E; \\
& t = \lambda_n,m (\overline{Z_n}) \cdot \text{(\{Y = t_i; S_i; \text{ln } X\})} \\
\text{ln } t\}
\end{align*}
\]

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**

\[
\begin{align*}
&\text{& } \text{FE}(E, (\text{if}_n X \text{ then } \{ X = E; S \} \text{ else } \{ X' = E; S' \})) \\
\text{if}_n X \text{ then } \{ X = E; S \} \text{ else } \{ X' = E; S' \} &\rightarrow \{ t_1 = E; \\
& t = \text{if}_n X \text{ then } \{ X = t_1; S \} \text{ else } \{ X' = t_1; S' \} \\
\text{ln } t\}
\end{align*}
\]

- **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 \( \text{next}(x) = x \) \text{, then the variable } x \text{ can become a free variable of the loop body avoiding the circulation of } x \text{. Without loss of generality we assume that the nextified variable to be eliminated is the last one.}
\[ P = \lambda_n (X_n) \cdot (E) \]
\[ B = \lambda_{n,n} (X_n) \cdot ([S \in \text{next}(X_1), \ldots, \text{next}(X_n), X_n]) \]

\[ X_p = \text{Ap}_{n} (P, X_n) \]
\[ X_{n-1}' = \text{Loop}_{n} (P, B, X_n, X_p) \]

\[ P' = \lambda_{n-1} (X_{n-1}) \cdot (E[X_n/\text{next}(X_n)]); \]
\[ B' = \lambda_{n-1} (X_{n-1}) \cdot ([S \in \text{next}(X_1), \ldots, \text{next}(X_{n-1})][X_n/\text{next}(X_n)]); \]
\[ X_{p'} = \text{Ap}_{n-1} (P', X_{n-1}) \]
\[ X_{n-1}' = \text{Loop}_{n-1} (P', B', X_{n-1}, X_p); \quad X_n' = X_n; \]

- **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;
  ```
\[ P = \lambda_n \left( X_n \right) \cdot \left( E \right) \]
\[ B = \lambda_{n,n} \left( X_n \right) \cdot \left\{ \left\{ S \in \text{next}(X_1), \ldots, \text{next}(X_{n-1}), Z_n \right\} \right\} \quad \& \quad \text{FE}(Z_n, \rho) \]

\[ \overline{X_n'} = \text{Loop}_n \left( P, B, \overline{X_n}, X \right) \quad \rightarrow \quad \overline{X_n'} = \text{if } X \text{ then } \{ \begin{array}{l}
\overline{P'} = \lambda_{n-1} \left( X_{n-1} \right) \cdot \left( E[t_n/\text{next}(X_n)] \right); \\
\overline{B'} = \lambda_{n-1,n} \left( X_{n-1} \right) \cdot \left( \left\{ S \in \text{next}(X_1), \ldots, \text{next}(X_{n-1}) \right\}[t_n/\text{next}(X_n)] \right) \\
\overline{t_n} = \overline{\text{Ap}_{n,n} (B, X_n)}; \\
p = \overline{\text{Ap}_{n-1} (P', t_{n-1})}; \\
\overline{t'_{n-1}} = \text{Loop}_{n-1} \left( P', B', t_{n-1}, p \right); \\
\ln \overline{t'_{n-1}} \quad \text{else } \overline{X_n} \}
\]

The following optimizations are only applicable to for loops.

- **Peeling the loop once**

  \[ \text{Loop}_n \left( P, B, \overline{X_n}, X \right) \quad \rightarrow \quad \text{if } X \text{ then } \{ \begin{array}{l}
\overline{t_n} = \overline{\text{Ap}_{n,n} (B, X_n)}; \\
p = \overline{\text{Ap}_{n} (P, t_n)}; \\
\overline{t'_{n}} = \text{Loop}_n \left( P, B, \overline{t_n}, p \right); \\
\ln \overline{t'_{n}} \quad \text{else } \overline{X_n} \}
\]

- **Loop body unrolling K times**

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

\[ P = \lambda_n \left( X_1 \right) \cdot \left( X_1 \leq m \right) \]
\[ B = \lambda_{n,n} \left( X_n \right) \cdot \left\{ \text{next}(X_1) = X_1 + 1; S_1; \ldots; \text{next}(X_1), \ldots, \text{next}(X_n) \right\} \]
\[ \& \quad \text{remainder}(m, k) = 0 \]

\[ \text{Loop}_n \left( P, B, X_n, X_p \right) \quad \rightarrow \quad \{ \begin{array}{l}
\overline{B'} = \lambda_{n,n} \left( X_n \right) \cdot \left\{ \begin{array}{l}
\overline{t_1} = \overline{\text{Ap}_{n,n} (B, X_n)}; \\
\overline{t_2} = \overline{\text{Ap}_{n,n} (B, t_1)}; \\
\vdots \\
\overline{t_n} = \overline{\text{Ap}_{n,n} (B, t_{n-1})} \\
\ln \overline{t_n} \quad \text{else } \overline{X_n} \}
\end{array} \right\}
\]

\[ \overline{t_n} = \text{Loop}_n \left( P, B', X_n, X_p \right); \\
\ln \overline{t_n} \} \]
Figure 4: The Grammar of P-TAC.

The condition that the remainder, call it r, of m/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
Array \((l, u)\):

\[
\begin{array}{cccc}
0 & 1 & 2 & u-l+3 \\
"Array" & & \\
\downarrow \\
bounds\ tuple
\end{array}
\]

\(n\)-Tuple:

\[
\begin{array}{ccc}
0 & 1 & n+1 \\
"n\text{.tuple}" & & \\
\end{array}
\]

Cons:

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
"List" & "Cons" & Hd & Tl
\end{array}
\]

Nil:

\[
\begin{array}{cc}
0 & 1 \\
"List" & "Nil"
\end{array}
\]

Closure:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
"Closure" & Function name & Fast\ call\ name & Arity & Chain
\end{array}
\]

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 → P-TAC Expression

\[ \text{TE}[x] = x \]

\[ \text{TE}[c] = c \]

\[ \text{TE}[\text{Negate } x] = \text{Negate } x \]

The same holds for Not.

\[ \text{TE}[\text{Bounds } x] = \text{P.select}(x, \text{Bounds}) \]

\[ \text{TE}[\text{Array } (x)] = \{ \]
\[ l = \text{P.select}(x, \text{Lower}); \]
\[ u = \text{P.select}(x, \text{Upper}); \]
\[ s = u - l; \]
\[ \text{size} = s + 3; \]
\[ t = \text{Allocate}(\text{size}); \]
\[ \text{P.store}(t, \text{Type}, \text{"Array");} \]
\[ \text{P.store}(t, \text{Bounds}, x); \]
\[ \text{ln } t \} \]

\[ \text{TE}[\text{F.array } (x)] = \text{TE}[\text{L.array } (x)] \]

\[ \text{TE}[\text{Cons } (x_1, x_2)] = \{ \]
\[ t = \text{Allocate}(\text{Cons.size}); \]
\[ \text{P.store}(t, \text{Type}, \text{"List");} \]
\[ \text{P.store}(t, \text{Tag}, \text{"Cons");} \]
\[ \text{P.store}(t, \text{Hd}, x_1); \]
\[ \text{P.store}(t, \text{Tl}, x_2); \]
\[ \text{ln } t \} \]

\[ \text{TE}[\text{Make.tuple}_n (\overline{x_n})] = \{ \]
\[ t = \text{Allocate}(n + 1); \]
\[ \text{P.store}(t, \text{Type}, \text{"n.tuple");} \]
\[ \text{P.store}(t, 1, x_1); \]
\[ : \]
\[ \text{P.store}(t, n, x_n); \]
\[ \text{ln } t \} \]
\[ \text{TE}[\text{Apply} (f, x)] = \{ n = \text{P.select} (f, \text{Arity}); \]
\[ \text{fun} = \text{P.select} (f, \text{Functionname}); \]
\[ \text{as} = \text{P.select} (f, \text{Chain}); \]
\[ \text{as'} = \text{Ap} (\text{Arg.chain}, z, as); \]
\[ cl' = \text{Ap} (\text{Make.closure}, f, z); \]
\[ \text{fire}_b = \text{Equal?} n 1; \]
\[ \text{fire}_i = \text{BooltoInt} (\text{fire}_b); \]
\[ t_1, t_2 = \text{Dispatch}_{2,2} (\text{fire}, (\text{fun}, \text{as'}), (I, cl')); \]
\[ \text{res} = \text{Ap} (t_1, t_2); \]
\[ \text{In res} \} \]

\textit{BooltoInt} is a coercion function which converts True to 1 and False to 2.

\[ \text{TE}[\text{Ap}_{n,m} (f, z')] = \{ f' = \text{P.select} (f, \text{Fastcallname}); \]
\[ t_m = \text{Ap}_{n,m} (f', z'); \]
\[ \text{ln t}_m \} \]

\[ \text{TE}[+ (z, y)] = + (z, y) \]

The same holds for all strict operators in PF2_1.

\[ \text{TE}\{z_1 = e_1; \ldots z_n = e_n; c_1; \ldots c_m \text{ ln } z\} = \{ z_1 = \text{TE}[e_1]; \]
\[ \vdots \]
\[ z_n = \text{TE}[e_n]; \]
\[ c_1; \]
\[ \vdots \]
\[ c_m; \]
\[ \text{ln } z \} \]

Notice that the commands do not need any translation.

\[ \text{TE}[\lambda_{n,m} (z'_n) \cdot (e)] = \{ cl = \text{Allocate} (\text{Closure.size}); \]
\[ \text{P.store} (cl, \text{Type, "Closure"}); \]
\[ \text{P.store} (cl, \text{Functionname}, 'T_e); \]
\[ \text{P.store} (cl, \text{Fastcallname}, 'T_0); \]
\[ \text{P.store} (cl, \text{Arity}, n); \]
\[ \text{P.store} (cl, \text{Chain, "End"}); \]
\[ \text{ln cl} \} \]

The following two function definitions are included in the set D.
\[ T_c = \lambda_{1,m}(zs). \begin{cases} x_n = \text{Ap}_{1,n}(\text{Args}_n, zs); \\ t_m = \text{TE}[e]; \\ \ln t_m \end{cases} \]

\[ T_{fc} = \lambda_{n,m}(\overline{x_n}) . \text{TE}[e]; \]

\( T_c \) indicates the name \( T_c \) and not the value associated to \( T_c \). Note that \( \text{TE}[e] \) can be computed only once.

\[ \text{TE}[	ext{Detuple}_m z] = \{ t_1 = \text{P.select}(z, 1); \]
\[ \vdots \]
\[ t_m = \text{P.select}(z, m); \]
\[ \ln t_m \} \]

\[ \text{TE}[	ext{Decons} z] = \text{TE}\{ t = \{ \text{Case}_2 z \ of \]
\[ \text{Nil} = \text{Error}, \text{Error} \]
\[ \mid \text{Cons} x_1 x_2 = x_1, x_2 \} \]
\[ \ln t \} \]

\[ \text{TE}[	ext{If}_n z \text{ then } e_1 \text{ else } e_2] = \{ t = \text{BooInt}(z); \]
\[ f = \text{Dispatch}_2(t, 'T_{1fc}, T_{2fc}); \]
\[ t_m = \text{Ap}_{n,n}(f, \overline{x_n}); \]
\[ \ln t_m \} \]

and the following two definitions are included in D.

\[ T_{1fc} = \lambda_{n,m}(\overline{x_n}) . \text{TE}[e_1]; \]
\[ T_{2fc} = \lambda_{n,m}(\overline{x_n}) . \text{TE}[e_2] \]

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

\[ \text{TE}[	ext{Case}_m z \ of] = \{ t_t = \text{P.select}(z, \text{Tag}); \]
\[ \text{Nil} = e_1 \]
\[ f = \text{Dispatch}_2(t_t, 'T_{1fc}, 'T_{2fc}); \]
\[ t_m = \text{Ap}_{n,n}(f, \overline{x_n}); \]
\[ \ln t_m \} \]

and the following two definitions are included in D.

\[ T_{1fc} = \lambda_{n,m}(\overline{x_n}) . \text{TE}[e_1]; \]
\[ T_{2fc} = \lambda_{n,m}(\overline{x_n}) . \{ y_1 = \text{P.select}(z, \text{Hd}); \]
\[ y_2 = \text{P.select}(z, Tl); \]
\[ t = \text{TE}[e_2]; \]
\[ \ln t \} \]

20
where $\overline{z}_n \in FV(e_1) \cup (FV(e_2) - \{y_1, y_2\}) \cup \{z\}$

5.2.2 Standard identifiers

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

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

$l = \lambda (x) \cdot x$

$Nil = \{t = \text{Allocate}(Nil\_size);$
- $P\_store(t, Type, "List");$
- $P\_store(t, Tag, "Nil");$
- $ln t\}$

$Arg\_chain = \lambda (x, zs) \cdot \{zs' = \text{Allocate}(2);$
- $P\_store(zs', Arg, x);$  $P\_store(zs', Rest, zs);$  $ln zs'\}$

$Args_n = \lambda (z) \cdot \{t_1 = P\_select(z, Chain);$  $a_n = P\_select(t_1, Arg);$  $t_2 = P\_select(t_1, Rest);$  $a_{n-1} = P\_select(t_2, Arg);$  $t_3 = P\_select(t_2, Rest);$  $\vdots$
- $a_1 = P\_select(t_{n-1}, Arg);$  $ln a_n\}$
\[
\text{Make\_closure} = \lambda (cl, z). \{ \begin{align*}
  f &= \text{P\_select} (cl, \text{Funcname}); \\
  f_{fc} &= \text{P\_select} (cl, \text{Fastcallname}); \\
  n &= \text{P\_select} (cl, \text{Arity}); \\
  ch &= \text{P\_select} (cl, \text{Chain}); \\
  cl' &= \text{Allocate} (\text{Closure\_size}); \\
  \text{P\_store} (cl', \text{Type}, \text{"Closure"}); \\
  \text{P\_store} (cl', \text{Functionname}, f); \\
  \text{P\_store} (cl', \text{Fastcallname}, f_{fc}); \\
  \text{P\_store} (cl'_n, \text{Arity}, n'); \\
  \text{P\_store} (cl'_n, \text{Chain}, ch'); \\
  n' &= n - 1; \\
  ch' &= \text{Ap} (\text{Arg\_chain}, z, ch); \\
  \text{ln} cl'
\end{align*} \}
\]

5.3 Rewrite rules of P-TAC

In the following \( V \) stands for a ground value.

- \( \delta \) rules

- Conditional rules

\[
\text{Dispatch} \ (i, X_{i-1}, V_i, X_{i+1}, n) \rightarrow V_i
\]

- \text{Allocation rules}

\[
\text{Allocate} (n) \rightarrow L
\]

\[
\frac{
\text{P\_store} (L, i, V) \\
\text{P\_select} (L, i) \rightarrow V
}{
\text{P\_store} (L, i, V) \\
\text{P\_store} (L, i, V') \rightarrow T_s
}
\]

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

\[ \text{Loop}_n (P, B, X_n, \text{True}) \implies \{ n \overset{t_n}{\longrightarrow} = A\text{p}_n.n (B, X_n); \]
\[ t_B = A\text{p}_n (P, t_n); \]
\[ t'_B = \text{Loop}_n (P, B, t_n, t_p) \]
\[ \text{ln } t'_B \} \]

\[ \text{Loop}_n (P, B, X_n, \text{False}) \implies X_n \]

• Block Flattening rules

\[ \{ \overset{X_n}{\longrightarrow} = \{ SS_1; SS_2; \cdots \} \]
\[ \text{ln } Y_n \} \implies \{ SS_1; SS_2; \cdots ; S_1; \cdots ; S_n \; \text{ln } Z \} \]

• Propagation of \( T \)

\[ \{ X = T; S_1 \cdots S_n \; \text{ln } Z \} \implies T \]

\[ \{ T; S_1 \cdots S_n \; \text{ln } Z \} \implies T \]

• Multivariable rule

\[ \overset{X_n}{\longrightarrow} = Y_n \implies (X_1 = Y_1; \cdots ; X_n = Y_n) \]

• Substitution rules

\[ \overset{x=y}{\longrightarrow} \]

\[ \overset{x=y}{\longrightarrow} \]

• Application rules

\[ A\text{p}_n.m (\overset{F}{\longrightarrow} \; X_n). E \]

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”.
\[ S[\lambda_n, m \ (x_n^m)] \cdot \{ y_1 = se_1 = \lambda_n, m+1 \ (x_n^m) \cdot \{ y_1 = se_1 \}
\vdots
\]
\[ y_n = se_n \]
\[ y_{n+1} = nse_1 \]
\[ \vdots \]
\[ y_{n+m} = nse_m \]
\[ c_1 \]
\[ \vdots \]
\[ c_k \]
\[ \ln \overline{r_m} \} \] \[ S_{m+1} = SC[c_1] \]
\[ y_{n+m}, S_m = SE[nse_m] \]
\[ S_{m+k} = SC[c_k] \]
\[ S' = \text{Sync}_{m+k+i} \ (\text{Deadvariables}, S_{m+k}) \]
\[ \ln \overline{r_m}, S' \}

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

\[ SE[\text{Loop}_n \ (P, B, \overline{y_n}, y)] = \text{Loop}_n \ (P, B, \overline{y_n}, S, y) \]

Where \(S_p\) is the signal associated with the invocation of the predicate.

\[ SE[\text{Ap}_{n,m} \ (f, \overline{x_n})] = \text{Ap}_{n, m+1} \ (f, \overline{x_n}) \]
\[ SC[\text{P}\_\text{store} \ (x, i z)] = \text{Ack}\_\text{store} \ (x, i z) \]

Where \(\text{Ack}\_\text{store}\) 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:

\[ \text{Loop}'_n \ (P, B, \overline{X_n}, S, \text{True}) \rightarrow \{ n+1 \ \overline{t_n}, S_b \} = \text{Ap}_{n, n+1} \ (B, \overline{X_n}); \]
\[ t_p, S_p = \text{Ap}_{n, 2} \ (P, \overline{t_n}); \]
\[ S' = \text{Sync}_{3} \ (S, S_b, S_p); \]
\[ \overline{t_n'}, S_i = \text{Loop}'_n \ (P, B, \overline{t_n}, S', t_p) \]
\[ \ln \overline{t_n'}, S_i \}

\[ \text{Loop}'_n \ (P, B, \overline{X_n}, S, \text{False}) \rightarrow \overline{X_n}, S \]
\begin{align*}
\text{Ack\_store}\ (L, \hat{i}, V) & \rightarrow \{ t = \text{Signal}; \\
& \quad \text{P\_store}\ (L, \hat{i}, V); \\
& \quad \text{ln}\ t\} \\
\text{Sync\_n}\ (V_n) & \rightarrow ()
\end{align*}

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

References


# Contents

1 Introduction .......................... 1

2 Id^- .................................. 3

3 The first phase: from Id^- to PGL .......................... 3
   3.1 PGL .................................. 3
   3.2 Translation from Id^- to PGL .......................... 3
      3.2.1 TE: Id^- Expression → PGL Expression .................. 5
      3.2.2 TS: Id^- Statement → list (PGL Statement) ................ 7
      3.2.3 TB: Id^- Bindings → list (PGL Binding) .................. 8
      3.2.4 TO: Id^- Operator → PGL Operator .................... 8
   3.3 Definition of Standard Functions ..................... 8
   3.4 The Rewrite Rules of PGL (R_PGL) ...................... 9

4 The second phase: optimizations on PGL programs ................ 12

5 The third phase: from PGL to P-TAC ................................ 16
   5.1 P-TAC ................................ 16
   5.2 Translation of PGL to P-TAC .................... 16
      5.2.1 TE: PGL Expression → P-TAC Expression ............... 18
      5.2.2 Standard identifiers .................................. 21
   5.3 Rewrite rules of P-TAC .......................... 22

6 Signals .................................. 23