Compilation of $\text{Id}^-$: a subset of $\text{Id}$

Computation Structures Group Memo 315
24 July 1990
Revised 1 November 1990

Zena M. Ariola
Arvind

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1986 (MIT) and N0039-88-C-0163 (Harvard).
## Contents

1 Introduction ................................................. 1  
2 Desugared \text{Id}^\text{−} ............................................. 3  
3 \text{Kid} : The Kernel \text{Id} Language .......................... 6  
   3.1 \text{Kid} ......................................................... 6  
   3.2 The Rewrite Rules of \text{Kid} ................................. 8  
4 Translating Desugared \text{Id}^\text{−} into \text{Kid} .......... 13  
   4.1 Simplification of Expressions ............................... 13  
   4.2 Definition of Standard Functions .......................... 18  
   4.3 Pattern Matching: Elimination of Complex Patterns .. 19  
5 Optimizations of \text{Kid} Programs ......................... 24  
6 \text{P-TAC}: Parallel Three Address Code ................. 29  
   6.1 \text{P-TAC} ................................................... 29  
   6.2 Rewrite rules of \text{P-TAC} ................................. 29  
7 Translation of \text{Kid} into \text{P-TAC} .......................... 33  
   7.1 Simple \text{Kid} Expressions ................................. 33  
   7.2 Data Structure Representations ............................ 34  
   7.3 Function Calls and Closures ............................... 38  
8 Signals ......................................................... 41
Compilation of $\text{Id}^{-}$: A Subset of $\text{Id}$

Zena M. Ariola  
Aiken Computational Laboratory  
Harvard University

Arvind  
Laboratory for Computer Science  
Massachusetts Institute of Technology

November 1, 1990

Abstract

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

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

1 Introduction

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

- abstract types and algebraic types except lists;
- overloading;
Id\textsuperscript{−} Parsing, eliminate multi-clause functions and comprehensions
desugared Id\textsuperscript{−} Eliminate complex patterns and complex expressions

\text{Kid} Choose representation
\text{P-TAC}

Figure 1: Compilation scheme

mutable structures.

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

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

Kid serves several purposes. Since Id is too complex to be given direct operational semantics, it is preferable first to give a translation of Id into a smaller language such as Kid. Thus, Id is defined indirectly in terms of the rewrite rule semantics of Kid. Kid is also the intermediate language in which a large number of architecture independent compiler optimizations are performed. These optimizations are described in Section 5. We think that many other types of analyses such as Milner style type checking, or abstract interpretation for storage reclamation should be performed at the Kid level. These analyses are not described in this document.

An implementation method that is preferred by many researchers eliminates explicit environments containing the value of free variables of a function. This requires applying a program transformation known as lambda-lifting. Lambda-lifting involves two steps: first,
all nested function definitions (λ-expressions) of a Kid program are "closed" by turning their free variables into parameters. In the second step, all closed definitions are lifted to the top level. Lambda-lifting should be done after optimizations have been performed on Kid programs. There are many algorithms for doing lambda-lifting and we do not describe any of them in this document. Interested reader is referred to Johnsson's paper [3].

The next step is to translate a lambda-lifted Kid program into P-TAC [2], which is a much smaller language than Kid. This translation calls for choosing a machine representation for each type of Kid value and thus, involves many low level efficiency issues. P-TAC and its operational semantics are given in Section 6. A translation from Kid into P-TAC is given in Section 7. We end the document by showing in Section 8 how a P-TAC program is extended with Signals [8] which are needed to detect termination and for resource management.

Our compiling approach allows the formalization of questions related to correctness [2]. For example, it will make sense to talk about the correctness of lambda-lifting, or the correctness of the translation from Kid to P-TAC. Moreover, since there exists an independent operational semantics of \( \text{Id}^- \) in terms of a even smaller subset of kernel \( \text{Id} \) [6], it may be possible to prove the correctness of the translation between de-sugared \( \text{Id}^- \) and Kid. Another advantage of our approach is that we are able to delay the introduction of concepts tied to specific machines or even to the dataflow computation model until we actually generate machine code. However, in this document we do not go below the P-TAC level.

2 Desugared \( \text{Id}^- \)

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

\[(f\ x)\ :\ \text{map}\ f\ xs\]

becomes

\[(f\ x):((\text{map}\ f\ xs))\]

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

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

\[
\text{Def } \text{map } f \text{ Nil } = \text{Nil} \\
| \text{map } f \, x:x:xs = (f \, x) : \text{map } f \, xs;
\]

becomes

\[
\text{Def } \text{map } t1 \, t2 = \{ \text{Case } (t1, t2) \text{ of} \\
| (f, \text{ Nil}) = \text{Nil} \\
| (f, x:x) = (f \, x) : \text{map } f \, xs \}
\]

where \( t1 \) and \( t2 \) are new variables. Furthermore, all function definitions are turned into \( \lambda \)-expressions.

\[
\text{map } = \{ \text{Fun } t1 \, t2 = \{ \text{Case } (t1, t2) \text{ of} \\
| (f, \text{ Nil}) = \text{Nil} \\
| (f, x:x) = (f \, x) : \text{map } f \, xs \} \}
\]

If the Id function is defined as substitutable (Defsubst) then the corresponding \( \lambda \)-expression (Fun) is underlined. The “underlined” \( \lambda \)-expressions are then expanded in line during optimizations (see Section 5).

The most complex part of the de-sugaring phase is the transformation of comprehensions into loops. This is described in detail in [4]. Here we only illustrate the idea using the following example:

\[
\{ : = \mid | x \leftarrow xs ; y \leftarrow ys \}
\]

A typical translation of this list-comprehension is given in terms of nested map-list operations followed by a list flattening operation. In Id we make use of “open lists” to generate a tail recursive program. Basically, in the following program, a open list (signified by h in the inner loop) is generated for each element of \( xs \) and then these open lists are “glued” together in the outer loop.

5
The above translation is correct if we treat open lists and lists as the same type. However, there are several subtle issues regarding type checking that are still not resolved.

3 Kid: The Kernel Id Language

3.1 Kid

Kid has only uncurried operators and no complex expressions. A major subset of Kid is simply the λ-calculus with constants and let blocks. However, unlike other functional languages, let - blocks play a fundamental role in the operational semantics of Kid. The syntax of Kid is given in Figure 3. Every expression, except a block or λ-expression, consists of a combinator followed by the corresponding number of arguments. The translation from Id− to Kid also has the flavor of turning an applicative TRS into a functional one.

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

\[ x, y = \{ \, a = \ldots ; b = \ldots \, \} \]

is a well-formed-expression, where "x, y" indicates multiple variables, not to be confused with a 2-tuple. The 2 after the curly brace indicates that two values are to be returned by this block expression. Multiple values avoid packaging values in a data structure, and they are useful in expressing some optimizations. Thus, in Kid a binding has the form \( MV = E \), where \( MV \) stands for multiple variable. Suppose we have \( m \) variables on the left-hand-side, then the expression \( E \) on the right-hand-side must return \( m \) values. In the sequel we capture the number of values that an expression produces by subscripting
Figure 3: Grammar of Kid
the corresponding syntactic category. Thus, to express the above binding we will write \( MV_m = E_m \). Note that the combinator “Apply” appears as a \( PF_2 \) in the grammar because all \( Id^- \) procedures return only one result. We also use subscripted combinators to express a family of combinators. For example, \( \text{Make\_tuple}_n \) stands for \( \text{Make\_tuple}_2, \text{Make\_tuple}_3, \) 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.

\[
\begin{align*}
\{ & \text{ is the same as } \{_1 \\
\text{Bool\_case} & \text{ is the same as } \text{Bool\_case}_1 \\
\text{List\_case} & \text{ is the same as } \text{List\_case}_1 \\
\text{WLoop} & \text{ is the same as } \text{WLoop}_1 \\
\text{FLoop} & \text{ is the same as } \text{FLoop}_1 \\
\text{Ap}_n & \text{ is the same as } \text{Ap}_{n,1} \\
\text{\lambda}_n & \text{ is the same as } \text{\lambda}_{n,1}
\end{align*}
\]

3.2 The Rewrite Rules of Kid

We now present a set of rewrite rules, \( R_{Kld} \), to define the operational semantics for Kid. \( R_{Kld} \) is a Contextual Rewrite System described in [1]. We assume that a primitive function is only applied to arguments of appropriate types, i.e., the type checking has been done statically.

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

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

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

\[
\text{RB} \left[ \{ x = + (a, 1) \ln z \} \right] = \{ x' = + (a, 1) \ln z' \}
\]

The notation \( E [Y/X] \) means the substitution of \( Y \) for \( X \) in \( E \). Usually this implies avoiding capture of free variables. However, due to our assumption that all variables occurring in a term to be reduced are unique, \( E [Y/X] \) will simply indicate naive substitution, that is, substitution where no danger of free variable capture exists and where \( X \) can be replaced by \( Y \) without regards to scope. Moreover, we will use the notation \( 
\overrightarrow{x_{n,m}} \)

for \( (x_n, \ldots, x_m) \), \( x_m \) for \( \overrightarrow{x_{1,m}} \), and \( E [Y_n / X] \) for \( E [Y_1 / X_1, \ldots, Y_n / X_n] \), which is the same as \( \cdots ((E [Y_1 / X_1]) [Y_2 / X_2]) \cdots ) [Y_n / X_n] \).

In the following, \( n \) represents a numeral.

\[ \delta \text{ rules} \]

\[
\begin{align*}
+ (m, n) & \xrightarrow{\delta} + (m, n) \\
\vdots & \\
\text{Equal?} (m, n) & \xrightarrow{\delta} \text{True} \\
\text{Equal?} (m, n) & \xrightarrow{\delta} \text{False} \quad \text{(if } m \neq n \text{)}
\end{align*}
\]

\[ \text{Case rules} \]

\[
\begin{align*}
\text{Bool.casem} (\text{True}, E_1, E_2) & \rightarrow E_1 \\
\text{Bool.casem} (\text{False}, E_1, E_2) & \rightarrow E_2 \\
X = \text{Nil} & \\
\text{List.casem} (X, E_1, E_2) & \rightarrow E_1 \\
X = \text{Open.cons} () & \\
\text{List.casem} (X, E_1, E_2) & \rightarrow E_2
\end{align*}
\]
Aritity Detection rule

\[
F = \lambda_n (\overrightarrow{Z_n}).(E) \\
F_1 = \text{Apply} (F, X_1) \\
F_2 = \text{Apply} (F_1, X_2) \\
\vdots \\
F_{n-1} = \text{Apply} (F_{n-2}, X_{n-1}) \\
\text{Apply} (F_{n-1}, X_n) \rightarrow \text{Ap}_n (F, X_n)
\]

A similar rule applies for \( \lambda_n \).

Application rule

\[
F = \lambda_{n,m} (\overrightarrow{Z_n}) \cdot (E) \\
\text{Ap}_{n,m} (F, X_n) \rightarrow (\text{RB}[E]) [X_n / \overrightarrow{Z_n}]
\]

A similar rule applies for \( \lambda_{n,m} \).

Loop rules

\[
\text{WLoop}_n (P, B, \overrightarrow{X_n}, \text{True}) \rightarrow \{ n \overrightarrow{t_n} = \text{Ap}_{n,n} (B, \overrightarrow{X_n}); \\
\quad \overrightarrow{t_p} = \text{Ap}_n (P, \overrightarrow{t_n}); \\
\quad \overrightarrow{t_n'} = \text{WLoop}_n (P, B, \overrightarrow{t_n}, \overrightarrow{t_p}) \\
\quad \ln \overrightarrow{t_n'} \}
\]

\[
\text{WLoop}_n (P, B, \overrightarrow{X_n}, \text{False}) \rightarrow \overrightarrow{X_n}
\]

In the following two rules we assume that the index variable is the first variable in \( \overrightarrow{X_n} \).

\[
\text{FLoop}_n (U, D, B, \overrightarrow{X_n}, \text{True}) \rightarrow \{ n \overrightarrow{t_2} = \text{Ap}_{n,n-1} (B, \overrightarrow{X_n}); \\
\quad \overrightarrow{t_1} = +(X_1, D); \\
\quad \overrightarrow{t_p} = -(t_1, U); \\
\quad \overrightarrow{t_n'} = \text{FLoop}_n (U, D, B, \overrightarrow{t_n}, t_p) \\
\quad \ln \overrightarrow{t_n'} \}
\]

\[
\text{FLoop}_n (U, D, B, \overrightarrow{X_n}, \text{False}) \rightarrow \overrightarrow{X_n}
\]
Tuple rule

\[ X = \text{Make\_tuple\_n} \left( X_n \right) \]
\[ \text{Detuple}_n \left( X \right) \rightarrow X_n \]

List rules

\[ \text{Cons} \left( X, Y \right) \rightarrow \{ \ t = \text{Open\_cons} () ; \]
\[ \text{Cons\_store}\_1 \left( t, X \right) ; \]
\[ \text{Cons\_store}\_2 \left( t, Y \right) \]
\[ \text{In} \ t \} \]

\[ \text{Cons\_store}\_1 \left( X, Y \right) \]
\[ \text{Cons}_1 \left( X \right) \rightarrow Y \]

\[ \text{Cons\_store}\_2 \left( X, Y \right) \]
\[ \text{Cons}_2 \left( X \right) \rightarrow Y \]

\[ \text{Cons\_store}\_1 \left( X, Y \right) \]
\[ \text{Cons\_store}_1 \left( X, Y' \right) \rightarrow \mathcal{T}_s \]

\[ \text{Cons\_store}\_2 \left( X, Y \right) \]
\[ \text{Cons\_store}_2 \left( X, Y' \right) \rightarrow \mathcal{T}_s \]

Array rules

\[ X = \text{I\_nD\_array} \left( X_b \right) \]
\[ \text{nD\_Bounds} \left( X \right) \rightarrow X_b \]

\[ \text{P\_nD\_store} \left( X, Y, Z \right) \]
\[ \text{P\_nD\_select} \left( X, Y \right) \rightarrow Z \]

\[ \text{P\_nD\_store} \left( X, Y, Z \right) \]
\[ \text{P\_nD\_store} \left( X, Y, Z' \right) \rightarrow \mathcal{T}_s \]
Multivariable rule

\[ \overrightarrow{X_n} = \overrightarrow{Y_n} \quad \rightarrow \quad (X_1 = Y_1; \ldots; X_n = Y_n) \]

Substitution rules

\[ \begin{align*}
X &= Y \\
\overrightarrow{X} &\rightarrow \overrightarrow{Y} \\
X &= C \\
\overrightarrow{X} &\rightarrow \overrightarrow{C}
\end{align*} \]

Block Flattening rule

\[ \begin{align*}
\{ \overrightarrow{X_n} = \{ n \overrightarrow{S_{S_1}}; \overrightarrow{S_{S_2}}; \ldots \} \\
\overrightarrow{\ln Y_n} \} &\rightarrow \{ \overrightarrow{X_n} = \overrightarrow{Y_n}; \\
\overrightarrow{S_{S_1}}; \overrightarrow{S_{S_2}}; \ldots \} \\
\overrightarrow{S_{S_1}}; \ldots; \overrightarrow{S_{S_n}} \\
\overrightarrow{\ln Z_m} \} \\
\overrightarrow{\ln Z_m} \}
\end{align*} \]

Propagation of \( T \)

\[ \begin{align*}
\{ \overrightarrow{m X} = T; \overrightarrow{S_{S_1}}; \overrightarrow{S_{S_2}}; \ldots; \overrightarrow{S_{S_n}} \ln \overrightarrow{Z_m} \} &\rightarrow \quad T \\
\{ \overrightarrow{m T}; \overrightarrow{S_{S_1}}; \overrightarrow{S_{S_2}}; \ldots; \overrightarrow{S_{S_n}} \ln \overrightarrow{Z_m} \} &\rightarrow \quad T
\end{align*} \]
4 Translating Desugared $Id^-$ into Kid

4.1 Simplification of Expressions

We give the translation in terms of the following functions:

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

The exact syntax of Case_expression is given in Section 4.3.

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

Throughout, the emphasis is on clarity of $Id^-$ to Kid translation rather than its efficiency. We will use the same conventions introduced in Section 3.2, with the addition of meta-variable $P_i$ that ranges over Patterns. As before $X_n$ indicates multiple meta-variables. The lower case variables, such as $t_i, x_i$, that appear in the translated expression represent new Kid variables. The translation procedure given below does not require lexical scope analysis for variables. However, after the translation is complete a procedure to make all bound variables unique has to be applied before any rewriting can be done.

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

\text{TE}[X] = X

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

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

\text{TE}[C] = C

\text{TE}[(\text{UOP } E)] = \{ t_1 = \text{TE}[E];
\text{t} = \text{TO}[\text{UOP}](t_1)
\text{ln } t\}\n
\text{TE}[(E_1 \text{BOP } E_2)] = \{ t_1 = \text{TE}[E_1];
\text{t}_2 = \text{TE}[E_2];
\text{t} = \text{TO}[\text{BOP}](t_1, t_2)
\text{ln } t\}\n
\text{TE}[(E_1 E_2)] = \{ t_1 = \text{TE}[E_1];
\text{t}_2 = \text{TE}[E_2];
\text{t} = \text{Apply}(t_1, t_2)
\text{ln } t\}\n
\text{TE}[E_1[E_2]] = \{ t_1 = \text{TE}[E_1];
\text{t}_2 = \text{TE}[E_2];
\text{t} = \text{Ap}_2(\text{Select}, t_1, t_2)
\text{ln } t\}\n
where Select is a standard function definition in Kid (see Section 4.2).

\text{TE}[E.\text{cons}_1] = \{ t_1 = \text{TE}[E];
\text{t} = \text{List}._\text{case}(t_1, \text{Error}, \text{Cons}_1(t_2))
\text{ln } t\}\n
Similarly for \text{E.\text{cons}_2}.  

14
\[\text{TE}[(E_1, \ldots, E_n)] = \{ t_1 = \text{TE}[E_1]; \]
\[
\quad \vdots \quad
\]
\[
\quad t_n = \text{TE}[E_n]; \quad t = \text{Make\_tuple}_n(\langle t_n \rangle) \quad \text{ln } t\} \]
\[\text{TE}[(\text{if } E_1 \text{ then } E_2 \text{ else } E_3)] = \{ t_1 = \text{TE}[E_1]; \]
\[
\quad t = \text{Bool\_case}(t_1, \text{TE}[E_2], \text{TE}[E_3]) \quad \text{ln } t\} \]
\[\text{TE}[(\text{case } E \text{ of } P_1 = E_1)\]
\[
\quad \vdots \quad
\]
\[
\quad P_n = E_n \}) \quad t = \text{PM}[(\text{case } t_1 \text{ of } P_1 = \text{TE}[E_1]) \]
\[
\quad \vdots \quad
\]
\[
\quad P_n = \text{TE}[E_n]) \}) \quad \text{ln } t\} \]

Where PM is described in the Section 4.3.

\[\text{TE}[(\text{while } E \text{ do } S_1; \quad S_n; \quad \text{finally } E_f)] = \text{TSLE}[(\text{while } \text{TE}[E] \text{ do } S_1; \quad S_n; \quad \text{finally } \text{TE}[E_f])] \]

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

\[\text{TSLE}[(\text{while } E \text{ do } p = \lambda_n(\overrightarrow{X_n}) \cdot (E);\]
\[
\quad \vdots \quad
\]
\[
\quad \text{next}(X_1) = E_1; \quad b = \lambda_n(\overrightarrow{X_n}) \cdot (\{ \text{next}(X_1) = E_1; \quad Y_1 = E_{y_1}; \quad \vdots \quad \text{finally } E_f \}) \quad \text{ln next}(X_1), \ldots, \text{next}(X_n) \})\]
\[
\quad \text{tp} = \text{Ap}_n(p, \overrightarrow{X_n}); \quad \overrightarrow{t_n} = \text{WLoop}_n(p, b, \overrightarrow{X_n}, t_p); \quad t_f = \text{E}_f [\overrightarrow{t_n} / X_n]_B \quad \text{ln } t_f \} \]

15
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.

\[
\text{TE}\{ \text{For } X \leftarrow E_i \text{ to } E_b \text{ step } E_s \text{ do } \} = \text{TSLE}\{ \text{For } X \leftarrow \text{TE}[E_i] \text{ to } \text{TE}[E_b] \text{ step } \text{TE}[E_s] \text{ do } \}
\]

\[
\text{TSLE}\{ \text{For } X_1 \leftarrow E_i \text{ to } E_b \text{ step } E_s \} = \{ b = \lambda_{n,n-1}(X_n). \\
\text{\{ (next}(X_2) = E_2; \\
\quad \vdots \\
\quad \text{next}(X_n) = E_n; \\
\quad Y_1 = E_{v_1}; \\
\quad \vdots \\
\quad \text{Finally } E_f \} \\
\}
\]

\[
\text{TE}\{ \text{Fun } \overrightarrow{X_n} = E \} = \lambda_n(\overrightarrow{X_n}).(\text{TE}[E])
\]

\[
\text{TE}\{ \text{Fun } \overrightarrow{X_n} = E \} = \lambda_n(\overrightarrow{X_n}).(\text{TE}[E])
\]
\[ \text{TE} \{ S_1; \ldots; S_n; \ln E \} = \{ \text{TS}[S_1]; \\
\quad \vdots \\
\quad \text{TS}[S_n]; \\
\quad t = \text{TE}[E]; \\
\quad \ln t \} \]

**TS: Id\textsuperscript{-} Statement \rightarrow list (Kid Statement)**

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

\[ \text{TS}[P = E] = (t = \text{TE}[E]; \text{TB}[P = t]) \]

\[ \text{TS}[E_1[E_2] = E_3] = (t_1 = \text{TE}[E_1]; \\
\quad t_2 = \text{TE}[E_2]; \\
\quad t_3 = \text{TE}[E_3]; \\
\quad t = \text{Ap}_3(\text{Store}, t_1, t_2, t_3)) \]

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

\[ \text{TS}[E_1.\text{cons}_1 = E_2] = (t_1 = \text{TE}[E_1]; \\
\quad t_2 = \text{TE}[E_2]; \\
\quad t = \text{List_case}(t_1, \{ \text{Store}_\text{error} \ln (), \{ \text{Cons}\_\text{store}_1(t_1, t_2) \ln () \} \})) \]

Similarly for cons\_2.

**TB: Id\textsuperscript{-} Bindings \rightarrow list (Kid Binding)**

\[ \text{TB}[(P_1, \ldots, P_n) = X] = (t_n = \text{Detuple}_n(X); \\
\quad \text{TB}[P_1 = t_1]; \\
\quad \vdots \\
\quad \text{TB}[P_n = t_n]) \]

\[ \text{TB}[(P_1 : P_2) = X] = (t_1, t_2 = \text{List_case}_2(X, (\text{Error}, \text{Error}), (\text{Cons}\_1(X), \text{Cons}\_2(X))); \\
\quad \text{TB}[P_1 = t_1]; \\
\quad \text{TB}[P_2 = t_2]) \]

\[ \text{TB}[Y = X] = (Y = X) \]
\[ T_{B}[Next \ Y = X] = (next(Y) = X) \]

**TO: Id\- Operator \rightarrow Kid Operator**

\[ T_{O}[UOP] = \text{the corresponding Kid PF1} \]
\[ T_{O}[BOP] = \text{the corresponding Kid PP2} \]

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

### 4.2 Definition of Standard Functions

In our translation from Id\- to Kid we have introduced two new functions Select and Store. These are not primitive operators in Kid, therefore we give their definitions. Similarly we give a definition for Make\_1D\_array. For the sake of brevity we give these definitions in Id\-. These definitions and Make\_1D\_array can be written as follows

\[
\text{Make\_1D\_array = } \{ \text{Fun b f = } \{ (1,u) = b ; \\
\quad a = 1 \_array b ; \\
\quad i = 1 ; \\
\quad d = \{ \text{While i } \leq u \ do \\
\quad \quad a[i] = f \ i ; \\
\quad \quad \text{Next i } = i + 1 \\
\quad \quad \text{Finally } () \} \\
\quad \text{In a } \} \}
\]

Like lists we are treating I-arrays and functional arrays as the same type. However, functional arrays have a different degree of polymorphism than \_1D\_array. We are ignoring these subtle issues in the above translation.

\[
\text{Select = } \{ \text{Fun x i = } \{ (1,u) = 1D\_Bounds x ; \\
\quad \text{In} \\
\quad \quad \text{If } (i > u \ Or \ i < 1) \ \text{then} \\
\quad \quad \quad \text{Error} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \text{P\_1D\_select x i } \} \}
\]

\[
\text{Store = } \{ \text{Fun x i y = } \{ (1,u) = 1D\_Bounds x ; \\
\quad \text{In} \\
\quad \quad \text{If } (i > u \ Or \ i < 1) \ \text{then} \\
\quad \quad \quad \{ \text{Store\_error In () } \} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \{ \text{P\_1D\_store x i y In () } \} \} \}
\]
The above $\text{Id}^-$ definitions can be translated into Kid by adding the following rules:

\begin{align*}
\text{TE}[\text{P.nD.select} \ x \ i] &= \text{P.nD.select} \ (x, \ i) \\
\text{TS}[\text{P.nD.store} \ x \ i \ y] &= \text{P.nD.store} \ (x, \ i, \ y) \\
\text{TE}[\text{Error}] &= \text{Error} \\
\text{TE}[\text{Store.error}] &= \text{Store.error}
\end{align*}

4.3 Pattern Matching: Elimination of Complex Patterns

Pattern matching in Id follows a different philosophy than other functional languages such as Miranda and Haskell. In both these languages clauses in patterns are examined from top to bottom and patterns in a clause are examined from left to right. Within the limitations of the above rule, pattern matching does not force the evaluation of a pattern whose type is irrefutable (tuples are examples of irrefutable patterns) [7]. Miranda in addition allows repeated occurrences of a variable in a clause.

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

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

\begin{verbatim}
{ Case (x, y, z) of
  | (1, 0, w) = 1
  | (0, w, 1) = 2
  | (w, 1, 0) = 3 }
\end{verbatim}

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

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

Now we informally describe a simplified version of the pattern matching algorithm used by the Id compiler [5]. For $\text{Id}^-$ we take into consideration lists and tuples only. In
the following we describe various cases that arise during pattern matching.

The signature of the pattern matching function is

\[ \text{PM} : \text{Case-expression} \mapsto \text{Kid Expression} \]

where Case-expression is defined as follows

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

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

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

**Variable rule**

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

\[ \text{PM} \{ \text{Case} \ (X_{\overrightarrow{i,-1}}, \ X_i, \ X_{\overrightarrow{i+1,n}}) \ of \]
\[ \quad \mid \ (P_{\overrightarrow{1,1-i}}, \ Y_1, \ P_{\overrightarrow{1,i-1}}) = E_1 \]
\[ \quad \mid \ \vdots \]
\[ \quad \mid \ (P_{\overrightarrow{m,1-i}}, \ Y_m, \ P_{\overrightarrow{m,i-1}}) = E_m \} = \]

\{ t = X_i; \]
\[ t_f = \text{PM} \{ \text{Case} \ (X_{\overrightarrow{i,-1}}, \ X_{\overrightarrow{i+1,n}}) \ of \]
\[ \quad \mid \ (P_{\overrightarrow{1,1-i}}, \ P_{\overrightarrow{1,i-1}}) = E_1[t/Y_1]_B \]
\[ \quad \mid \ \vdots \]
\[ \quad \mid \ (P_{\overrightarrow{m,1-i}}, \ P_{\overrightarrow{m,i-1}}) = E_m[t/Y_m]_B \} \]

\[ \ln t_f \}

Note that meta-variables \( Y_1 \cdots Y_m \) may be distinct.
Irrefutable pattern rule

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

\[
\begin{align*}
\mathbf{PR}[ \{\text{Case } & (X_{i,i-1}, X_i, X_{i+1,n}) \text{ of } \\
& (P_{1,i-1}, Y_1, P_{1,i+1,n}) = E_1 \\
& \vdots \\
& (P_{k,i-1}, Y_k, P_{k,i+1,n}) = E_k \\
& (P_{l,i-1}, (Yl_1, Yl_2), P_{l,i+1,n}) = E_l \\
& \vdots \\
& (P_{m,i-1}, (Ym_1, Ym_2), P_{m,i+1,n}) = E_m \} ] = \\
\{t_1, t_2 = \text{Detuple}_2(X_i); \\
t = X_i; \\
t_f = \mathbf{PR}[ \{\text{Case } & (X_{i,i-1}, t_1, t_2, X_{i+1,n}) \text{ of } \\
& (P_{1,i-1}, t_{1,1}, t_{1,2}, P_{1,i+1,n}) = E_1[t/Y_1]_B \\
& \vdots \\
& (P_{k,i-1}, t_{k,1}, t_{k,2}, P_{k,i+1,n}) = E_k[t/Y_k]_B \\
& (P_{l,i-1}, Yl_1, Yl_2, P_{l,i+1,n}) = E_l \\
& \vdots \\
& (P_{m,i-1}, Ym_1, Ym_2, P_{m,i+1,n}) = E_m \} ] \\
\ln t_f \}
\end{align*}
\]
Refutable pattern rule

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

\[
\begin{align*}
\text{PM}[ \{ \text{Case } (X_{1,i-1}, \ X_i, \ X_{i+1,n}) \rightarrow \text{of } & \ 
\quad (P_{1,i-1}, \ \text Nil, \ P_{1,i+1,n}) = E_1 \\
\quad \quad \quad \quad \vdots \\
\quad (P_{k,i-1}, \ \text Nil, \ P_{k,i+1,n}) = E_k \\
\quad (P_{l,i-1}, \ Yl_1 : Yl_2, \ P_{l,i+1,n}) = E_l \\
\quad \quad \quad \quad \vdots \\
\quad (P_{m,i-1}, \ Ym_1 : Ym_2, \ P_{m,i+1,n}) = E_m \} ] = \\
\text{List case } (X_i, \\
\text{PM}[ \{ \text{Case } (X_{1,i-1}, \ X_{i+1,n}) \rightarrow \text{of } & \ 
\quad (P_{1,i-1}, \ P_{1,i+1,n}) = E_1 \\
\quad \quad \quad \quad \vdots \\
\quad (P_{k,i-1}, \ P_{k,i+1,n}) = E_k \} ], \\
\{ \ t_h, t_i = \text{Cons}_1 (X_i), \ \text{Cons}_2 (X_i) ; \ \\
\ t_f = \text{PM}[ \{ \text{Case } (t_h, \ t_i, \ X_{1,i-1}, \ X_{i+1,n}) \rightarrow \text{of } & \ 
\quad (Yl_1, \ Yl_2, \ P_{l,i-1}, \ P_{l,i+1,n}) = E_l \\
\quad \quad \quad \quad \vdots \\
\quad (Ym_1, \ Ym_2, \ P_{m,i-1}, \ P_{m,i+1,n}) = E_m \} ] \\
\ \text{ln } t_f \} )
\end{align*}
\]

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

\[
\begin{align*}
\text{PM}[ \{ \text{Case } X \rightarrow \text{of } & \ 
\quad Y_1 : Y_2 = E \} ] = \text{PM}[ \{ \text{Case } X \rightarrow \text{of } \\
\quad Y_1 : Y_2 = E \} ] \\
\text{PM}[ \{ \text{Case } X \rightarrow \text{of } & \ 
\quad \text Nil = E \} ] = \text{PM}[ \{ \text{Case } X \rightarrow \text{of } \\
\quad \text Nil = E \} ] \\
\end{align*}
\]
Mixed variable-refutable patterns

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

\[
\text{PM}\{\text{Case } (X_1, X_2, X_3 ) \text{ of }
\begin{align*}
& (\text{Nil} , Y_{1,1} : Y_{1,2} , Y_1 ) = E_1 \\
& (Y_{2,1} : Y_{2,2} , Y_2 , \text{Nil} ) = E_2 \\
& (Y_3 , \text{Nil} , Y_{3,1} : Y_{3,2} ) = E_3 
\end{align*}
\} =
\]

\[
\{ \quad t_3 = X_3; \quad t_f = \text{PM}\{\text{Case } (X_1, X_2, X_3 ) \text{ of }
\begin{align*}
& (\text{Nil} , Y_{1,1} : Y_{1,2} , \text{Nil} ) = E_1[t_3/Y_1]_b \\
& (\text{Nil} , Y_{1,1} : Y_{1,2} , t_{1,1} : t_{1,2} ) = E_1[t_3/Y_1]_b \\
& (Y_{2,1} : Y_{2,2} , \text{Nil} , \text{Nil} ) = E_2[t_2/Y_2]_b \\
& (Y_{2,1} : Y_{2,2} , t_{2,1} : t_{2,2} , \text{Nil} ) = E_2[t_2/Y_2]_b \\
& (\text{Nil} , \text{Nil} , Y_{3,1} : Y_{3,2} ) = E_3[t_1/Y_3]_b \\
& (t_{3,1} : t_{3,2} , \text{Nil} , Y_{3,1} : Y_{3,2} ) = E_3[t_1/Y_3]_b 
\end{align*}
\} \quad \ln t_f \}
\]

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

Empty variable rule

\[
\text{PM}\{\text{Case } () \text{ of } = \text{ Compiler Error "overlapping patterns"
\begin{align*}
& ( ) = E_1 \\
& \vdots \\
& ( ) = E_m 
\end{align*}
\}
\]

\[
\text{PM}\{\text{Case } () \text{ of } = E \\
\begin{align*}
& ( ) = E 
\end{align*}
\}
5 Optimizations of Kid Programs

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

It is believed that all optimizations to be presented in this section preserve at least partial correctness. So far this has been proven for only a small subset of them [2].

Kid Rewrite rules

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

Inline Substitution

\[
F = \lambda_{n,m} (\overline{Z_n}) \cdot (E) \\
Ap_{n,m} (F, \overline{X_n} \rightarrow (RB [E]) [\overline{X_n} / \overline{Z_n}]
\]

Partial Evaluation

\[
F = \lambda_{n,m} (\overline{Z_n}) \cdot (E) \\
Apply (F, X) \rightarrow \{ f = \lambda_{n-1,m} (\overline{z_{n-1}}) \cdot ((RB [E]) [\overline{z_{n-1}} / \overline{Z_{2,n}}, X/Z_1]) \ln f \}
\]

A similar rule applies for \( \lambda_{n,m} \).

Fetch Elimination

\[
X = Cons (X_1, X_2) \\
Cons.1 (X) \rightarrow X_1
\]
\[
\begin{align*}
X &= \text{Cons} (X_1, X_2) \\
\text{Cons}_2 \ (X) &\rightarrow X_2
\end{align*}
\]

**Algebraic Identities**

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

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

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

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

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

: 

**Common Subexpression Elimination**

\[ \overline{Y_m} = \overline{PFN_m(X_n)} \]
\[ \overline{PFN_m(X_n)} \quad \rightarrow \quad Y_m \]

Primitive functions \text{TnD}_{\text{array}}, \text{Open}_{\text{cons}}, \text{Apply} \text{ and } \text{Ap}_{n,m} \text{ are excluded from this optimization because they (may) cause side-effects.}

**Lift Free Expressions**

\[ \overline{\lambda_{n,m}(Z_n)} \cdot (\{ n Y = E; \text{ln} \overline{X_m} \}) \]
\[ \text{m t}_1 = E; \]
\[ t = \overline{\lambda_{n,m}(Z_n)} \cdot (\{ n Y = t_1; \text{ln} \overline{X_m} \}) \]
\[ \text{ln} t \}

Where \( FE(e, e') \) return true if the expression \( e \) is free in \( e' \). This optimization allows us to deal with loop invariants, that is, expressions that do not depend on the nextified variables. A similar rule applies for \( \lambda_{n,m} \). (See the restrictions in the common subexpression elimination rule).

**Hoisting Code out of a Conditional**

\[ \overline{\lambda_{n}(\text{Bool}_{\text{case}}_n(X, \{ n Y = E; \text{ln} \overline{X_n} \}, \{ n Y' = E; \text{ln} \overline{X'_n} \})))} \]
\[ \overline{\text{Bool}_{\text{case}}_n(X, \{ n Y = E; \text{ln} \overline{X_n} \}, \{ n Y' = E; \text{ln} \overline{X'_n} \})} \]
\[ \{ n \text{ t}_1 = E; \]
\[ \text{t}_n = \overline{\text{Bool}_{\text{case}}_n(X, \{ n Y = t_1; \text{ln} \overline{X_n} \}, \{ n Y' = t_1; \text{ln} \overline{X'_n} \})} \]
\[ \text{ln} \text{ t}_n \} \]

26
Eliminating Circulating Variables

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

\[
\begin{align*}
P &= \lambda_n \langle X_n \rangle \cdot (E) \\
B &= \lambda_{n,a} \langle X'_n \rangle \cdot \{ \text{\textit{S in } } Z_{n-1}, X'_n \} \\
\text{WLoop}_n (P, B, \overrightarrow{Y_n}, Y_p) &\rightarrow \\
&\{ p = \lambda_{n-1} \langle z_{n-1} \rangle \cdot (\text{RB}[E] [\overrightarrow{z_{n-1}} / \overrightarrow{X_{n-1}}, Y_n/X_n]); \\
b = \lambda_{n-1,a-1} \langle z_{n-1} \rangle \cdot (\text{RB}[\{ \text{\textit{S in } } Z_{n-1} \}] [\overrightarrow{z_{n-1}} / \overrightarrow{X_{n-1}}, Y_n/X'_n]); \\
&\text{ln } t_{n-1} = \text{WLoop}_{n-1} (p, b, \overrightarrow{Y_{n-1}}, Y_p) \\
&\text{ln } t_{n-1}, Y_n \}
\end{align*}
\]

A similar optimization applies to for-loops.

Eliminating Circulating Constants

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

\[
\begin{align*}
\{ \text{While } (p \times y) \text{ do } \\
\text{Next } x = t; \\
\text{Next } y = f \times y; \\
\text{Finally } y \}
\end{align*}
\]

This may be transformed as follows:

\[
\begin{align*}
\text{If } (p \times y) \text{ then } \\
\{ y_1 = f \times y; \\
\text{ln } \\
\{ \text{While } (p \times t \times y_1) \text{ do } \\
\text{Next } y_1 = f \times t \times y_1; \\
\text{Finally } y_1 \} \}
\end{align*}
\]

\[
\text{else}
\]

\[
y
\]

27
Notice that it is only after the first iteration that the value of "x" is \( t \). Thus, to avoid the circulation of the nextified variable "x", the loop has to be peeled once. This rule can be expressed as follows. Please note that we could have also written \( Z_n \) instead of \( t_n \) on the right-hand-side.

\[
\begin{align*}
P & = \lambda_{n, \overline{X_n}} (\overline{X_n}), (E) \\
B & = \lambda_{n, \overline{X_n}} (\overline{X_n}^t), \left( \{ n, S \in \overline{Z_n} \} \right) \& \ FE(Z_n, \rho)
\end{align*}
\]

\[
W\text{Loop}_n \ (P, B, \overline{Y_n}, Y_p) \rightarrow \\
\text{Bool\_case}_n \ (Y_p, \overline{\text{X}_n})
\]

\[
\begin{align*}
\{ n \; p \} & = \lambda_{n-1} (\overline{x_{n-1}}), \left( \overline{\text{RB}\left[ E \right]} \left[ \overline{x_{n-1}} / \overline{X_{n-1}, \; t_n / X_n} \right] \right) \\
b & = \lambda_{n-1, n} (\overline{x_{n-1}^t}), \left( \overline{\text{RB}\left[ \{ n-1, \; S \in \overline{Z_n-1} \} \right]} \left[ \overline{x_{n-1}^t} / \overline{X_{n-1}^t, \; t_n / X_n} \right] \right) \\
t_n & = \text{Ap}_n (B, \overline{Y_n}) \\
t_p & = \text{Ap}_{n-1} (B, t_{n-1}) \\
t_{n-1} & = W\text{Loop}_{n-1} (p, b, t_{n-1}, t_p) \\
\text{ln} \; t_{n-1}, t_n, \overline{Y_n})
\end{align*}
\]

**Peeling the Loop once**

\[
F\text{Loop}_n \ (U, D, B, \overline{X_n}, X) \rightarrow \text{Bool\_case}_n \ (X, \{ n \; t_2, n \} = \text{Ap}_{n-1} (B, \overline{X_n}) \\
t_1 = + (X_1, D) \\
t_p = < (t_1, U) \\
t_n' = F\text{Loop}_n (U, D, B, t_n, t_p) \\
\text{ln} \; t_n')}
\]

28
Loop Body Unrolling K times

\[ \text{FLoop}_n (U, D, B, \overrightarrow{X_n}, X_p) \rightarrow \{ n \ b = \lambda_{n,n-1}(\overrightarrow{x_n}) \cdot (\{ n-1 \ \overrightarrow{t_{1,n}} = \text{A}_p_{n,n-1} (B, \overrightarrow{x_n}); \overrightarrow{t_{1,n}} = + (X_1, D)); \overrightarrow{t_{2,n}} = \text{A}_p_{n,n-1} (B, t_{1,n}); \overrightarrow{t_{2,n}} = + (t_{1,n}, D)); \ldots \overrightarrow{t_{2,n}} = \text{A}_p_{n,n-1} (B, t_{k-1,n}); \overrightarrow{t_{2,n}} = + (t_{k-1,n}, D)); \ln \overrightarrow{t_n} = \text{FLoop}_n (U, D, b, \overrightarrow{X_n}, X_p); \ln \overrightarrow{t_n} \} \]

Suppose \( r = \text{remainder}((U - X_1)/D,k) \), and \( r \) is not zero. We can still apply the above transformation by first peeling the loop \( r \) times.

6 P-TAC: Parallel Three Address Code

6.1 P-TAC

The syntax of P-TAC is given in Figure 4. In P-TAC, I-structure Storage is modelled in greater detail which requires the notion of Labels. All composite objects, that is, data structures and closures are stored in I-structure store and assigned unique labels, which are treated as constants that can be substituted freely.

6.2 Rewrite rules of P-TAC

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

\[ \delta \text{ rules} \]

\[ + (m, n) \overset{\delta}{\rightarrow} + (m, n) \]

\[ \vdots \]
\[
\begin{align*}
\text{UDF} & \in \text{User Defined Function} \\
\text{GV} & \in \text{Ground Value} \\
\text{Integer} & \ ::= \ 1 \ | \ 2 \ | \ \ldots \ | \ n \ | \ \ldots \\
\text{Boolean} & \ ::= \ \text{True} \ | \ \text{False} \\
\text{Variable} & \ ::= \ x \ | \ y \ | \ z \ | \ \ldots \ | \ a \ | \ b \ | \ \ldots \ | \ f \ | \ \ldots \ | \ x_1 \ | \ x_2 \ | \ \ldots \\
\text{MV}_m & \ ::= \ \text{Variable}_1, \ldots, \text{Variable}_m \\
\text{Label} & \ ::= \ L \ | \ L_1 \ | \ \ldots \ | \ L' \ | \ \ldots \\
\text{PF1} & \ ::= \ \text{Negate} \ | \ \text{Not} \ | \ \text{Allocate} \\
\text{PF2} & \ ::= \ + \ | \ - \ | \ \ast \ | \ \ldots \ | \ \text{Less} \ | \ \text{Equal?} \ | \ \text{P\_select} \\
\text{PF3} & \ ::= \ \text{Ack\_store} \\
\text{UDF} & \ ::= \ F \ | \ G \ | \ \ldots \\
\text{SE} & \ ::= \ \text{Variable} \ | \ \text{UDF} \ | \ \text{GV} \\
\text{SE}_m & \ ::= \ \underbrace{\text{SE}, \ldots, \text{SE}}_{m} \\
\text{GV} & \ ::= \ \text{Integer} \ | \ \text{Boolean} \ | \ () \ | \ \text{Label} \ | \ \text{Error} \ | \ \top \\
\text{E}_1 & \ ::= \ \text{SE} \ | \ \text{PF1} (\text{SE}) \ | \ \text{PF2} (\text{SE}_2) \ | \ \text{PF3} (\text{SE}_3) \ | \ \text{Ap}_n (\text{SE}_{n+1}) \\
& \quad | \ \text{Dispatch}_n (\text{SE}, E, \ldots, E) \ | \ \text{WLoop}_1 (\text{SE}_4) \ | \ \text{FLoop}_1 (\text{SE}_5) \ | \ \text{Block} \\
\text{E}_m & \ ::= \ \text{SE}_m \ | \ \text{Ap}_{n,m} (\text{SE}_{n+1}) \ | \ \text{Dispatch}_{n,m} (\text{SE}, E_{m}, \ldots, E_{m}) \\
& \quad | \ \text{WLoop}_m (\text{SE}_{m+3}) \ | \ \text{FLoop}_m (\text{SE}_{m+4}) \ | \ \text{Block}_m \\
\text{Block}_m & \ ::= \ \{n [\text{Statement};]^* \ \text{In} \ \text{SE}_m\} \\
\text{Statement} & \ ::= \ \text{Binding} \ | \ \text{Command} \ | \ \text{Store\_Error} \\
\text{Command} & \ ::= \ \text{P\_store} (\text{SE}_3) \ | \ \top_s \\
\text{Binding} & \ ::= \ \text{MV}_m = \text{E}_m \\
\end{align*}
\]

Figure 4: Grammar of P-TAC

30
Conditional rule

\[ \text{Dispatch}_{n,m} (i, \overrightarrow{E_{i-1}}, E_i, \overrightarrow{E_{i+1,n}}) \rightarrow E_i \]

I-structure rules

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

where \( L \) is a brand new label.

\[ \begin{align*}
\frac{\text{P\_store } (L, i, V)}{\text{P\_select } (L, i) \rightarrow V} \\
\frac{\text{P\_store } (L, i, V) \quad \text{P\_store } (L, i, V')}{\text{P\_store } (L, i, V') \rightarrow T_s}
\end{align*} \]

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

Application rule

\[ F = \lambda_{n,m} (\overrightarrow{Z_n}) . (E) \]

\[ \text{Apl}_{n,m} (F, \overrightarrow{X_n}) \rightarrow \text{RB}[E]) [\overrightarrow{X_n} / \overrightarrow{Z_n}] \]

A similar rule applies for \( \lambda_{n,m} \).

Loop rules

\[ \text{WLoop}_{n} (P, B, \overrightarrow{X_n}, \text{True}) \rightarrow \{ n \overrightarrow{t_n} = \text{Apl}_{n,n} (B, \overrightarrow{X_n}); \overrightarrow{t_p} = \text{Apl}_{n} (P, \overrightarrow{t_n}); \overrightarrow{t'_n} = \text{WLoop}_{n} (P, B, \overrightarrow{t_n}, \overrightarrow{t_p}); \text{ln } \overrightarrow{t'_n} \} \]

\[ \text{WLoop}_{n} (P, B, \overrightarrow{X_n}, \text{False}) \rightarrow \overrightarrow{X_n} \]
\[\text{FLoop}_n (U, D, B, \overrightarrow{X}_n, \text{True}) \rightarrow \{ n \ t_{2,n} = \text{A}_p_{n,n-1}(B, \overrightarrow{X}_n); \]
\[ t_1 = + (X_1, D); \]
\[ t_p = < (t_1, U); \]
\[ t'_n = \text{FLoop}_n (U, D, B, \overrightarrow{t_n}, t_p) \]
\[ \ln t'_n \} \]

\[\text{FLoop}_n (U, D, B, \overrightarrow{X}_n, \text{False}) \rightarrow \overrightarrow{X}_n \]

**Multivariable rule**

\[\overrightarrow{X}_n = \overrightarrow{Y}_n \rightarrow (X_1 = Y_1; \cdots X_n = Y_n)\]

**Substitution rules**

\[
\begin{align*}
X &= Y \\
\overrightarrow{X} &\rightarrow \overrightarrow{Y} \\
X &= V \\
\overrightarrow{X} &\rightarrow \overrightarrow{V}
\end{align*}
\]

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

**Block Flattening rule**

\[
\begin{align*}
\{ m \ \overrightarrow{X}_n &= \{ n \ SS_1; SS_2; \cdots \}
\end{align*}
\]

\[
\begin{align*}
\ln Y_n \} &\rightarrow \{ m \ \overrightarrow{X}_n = \overrightarrow{Y}_n; \\
S_1; \cdots S_n \\
\ln \overrightarrow{Z}_m \} \rightarrow \{ m \ SS_1; SS_2; \cdots \\
S_1; \cdots S_n \\
\ln \overrightarrow{Z}_m \}
\end{align*}
\]

**Propagation of \( \top \)**

\[
\begin{align*}
\{ m \ \overrightarrow{X} = \top; S_1; \cdots S_n \ln \overrightarrow{Z}_m \} &\rightarrow \top \\
\{ m \overrightarrow{\top_s}; S_1; \cdots S_n \ln \overrightarrow{Z}_m \} &\rightarrow \top
\end{align*}
\]

32
7 Translation of Kid into P-TAC

Prior to translating Kid to P-TAC, λ-lifting is performed. A Kid program after λ-lifting only contains closed λ-expressions. The translator, given a Kid program, produces the corresponding P-TAC program and a set, "D", of definitions. The set D is initialized with constants that are introduced by the translator.

7.1 Simple Kid Expressions

\( \text{TE} [X] = X \)

\( \text{TE} [C] = C \)

where \( C \) ranges over Integers and Booleans.

\( \text{TE} [\text{Negate} \ (X)] = \text{Negate} \ (X) \)

The same holds for Not.

\( \text{TE} [+ \ (X, Y)] = + \ (X, Y) \)

\( \text{TE} [\text{WLoop}_m \ (P, B, X_m, X_b)] = \text{WLoop}_m \ (P, B, X_m, X_b) \)

The same holds for FLoop.

\( \text{TE} [\text{Bool_case}_m \ (X, E_1, E_2)] = \{ m \ t_m = \text{BoolToInt} \ (X); \)
\( \quad t_m = \text{Dispatch}_{2,m} \ (t, E_1, E_2); \)
\( \quad \text{ln} \ t_m \} \)

\text{BoolToInt} is a coercion function which converts True to 0 and False to 1.

\( \text{TE} [\{ m \ X_1 = E_1; \cdots \ X_n = E_n; \ S_1; \cdots \ S_m \ \text{ln} \ Y_m \}] = \{ m \ X_1 = \text{TE} [E_1]; \)
\( \cdots \)
\( \quad X_n = \text{TE} [E_n]; \)
\( \quad \text{TE}[S_1]; \)
\( \cdots \)
\( \quad \text{TE}[S_m]; \)
\( \quad \text{ln} \ Y_m \} \)
7.2 Data Structure Representations

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

Tuples

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{TE}[\text{Make\_tuple}_n (X_n)] = \{ t = \text{Allocate} (n + 1) ; \\
\quad \text{P\_store} (t, \text{Type, "n\_tuple"}) ; \\
\quad \text{P\_store} (t, 1, X_1) ; \\
\quad \vdots \\
\quad \text{P\_store} (t, n, X_n) ; \\
\quad \text{ln} \ t \}\n
\text{TE}[\text{Detuple}_m (X)] = \{ m \ t_1 = \text{P\_select} (X, 1) ; \\
\quad \vdots \\
\quad t_m = \text{P\_select} (X, m) ; \\
\quad \text{ln} \ t_m \} \]
1D-Arrays

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

\[
\begin{array}{c|c|c|c|c|}
\text{Type} & \text{Bounds} & \text{ } & \text{ } & \text{ } \\
\hline
0 & 1 & 2 & u-1 & 2 \\
\end{array}
\]

The constant definition \texttt{HeaderSize = 2} should be included in the set "D".

\[
\text{TE}[\text{nD\_Bounds} (X)] = \text{P\_select} (X, \text{Bounds})
\]

\[
\text{TE}[\text{Larray} (X)] = \{ l = \text{P\_select} (X, \text{Lower}); \\
u = \text{P\_select} (X, \text{Upper}); \\
s = -(u, l); \\
size = + (s, 3); \\
t = \text{Allocate} (size); \\
\text{P\_store} (t, \text{Type, "Array"}); \\
\text{P\_store} (t, \text{Bounds, } X); \\
\text{ln } t \}
\]

\[
\text{TE}[\text{P\_1D\_select} (X_1, X_2)] = \{ t_b = \text{P\_select} (X_1, \text{Bounds}); \\
l = \text{P\_select} (t_b, \text{Lower}); \\
t_1 = -(X_2, l); \\
t_2 = +(t_1, \text{HeaderSize}); \\
t = \text{P\_select} (X_1, t_2) \\
\text{ln } t \}
\]

\[
\text{TE}[\text{P\_1D\_store} (X_1, X_2, X_3)] = \{ t_b = \text{P\_select} (X_1, \text{Bounds}); \\
l = \text{P\_select} (t_b, \text{Lower}); \\
t_1 = -(X_2, l); \\
t_2 = +(t_1, \text{HeaderSize}); \\
t = \text{P\_store} (X_1, t_2, X_3) \\
\text{ln } t \}
\]

A representation that will be more efficient for computing the slot address may want to store \(l\) and \(u\) values redundantly in two additional fields.
2D-Arrays

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

\[
\begin{align*}
\text{First\_dim} & = 1 & \text{Second\_dim} & = 2 \\
\text{First\_index} & = 1 & \text{Second\_index} & = 2 \\
\end{align*}
\]

\[
\text{TE}[\text{1\_2D\_array}(X)] = \{ \begin{align*}
d_1 &= \text{P\_select}(X, \text{First\_dim}) \\
d_2 &= \text{P\_select}(X, \text{Second\_dim}) \\
l_1 &= \text{P\_select}(d_1, \text{Lower}) \\
u_1 &= \text{P\_select}(d_1, \text{Upper}) \\
l_2 &= \text{P\_select}(d_2, \text{Lower}) \\
u_2 &= \text{P\_select}(d_2, \text{Upper}) \\
s_1 &= -(u_1, l_1) \\
s_2 &= -(u_2, l_2) \\
s_1' &= +(s_1, 1) \\
s_2' &= +(s_2, 1) \\
s &= *(s_1', s_2') \\
\text{size} &= +(s, \text{Headersize}) \\
t &= \text{Allocate} (\text{size}) \\
\text{P\_store}(t, \text{Type}, \text{"Array"}) \\
\text{P\_store}(t, \text{Bounds}, X) \\
\text{ln } t \}
\]

Translation for \text{1\_nD\_array} can be given in a similar fashion.
\[ \text{TE}[\text{P.2D.select} (X_1, X_2)] = \{ \begin{array}{l}
  b = \text{P.select} (X_1, \text{Bounds}) ; \\
  d_1 = \text{P.select} (b, \text{First.dim}) ; \\
  d_2 = \text{P.select} (b, \text{Second.dim}) ; \\
  l_1 = \text{P.select} (d_1, \text{Lower}) ; \\
  u_1 = \text{P.select} (d_1, \text{Upper}) ; \\
  l_2 = \text{P.select} (d_2, \text{Lower}) ; \\
  u_2 = \text{P.select} (d_2, \text{Upper}) ; \\
  r = - (u_2, l_2) ; \\
  r' = + (r, 1) ; \\
  i = \text{P.select} (X_2, \text{First.index}) ; \\
  j = \text{P.select} (X_2, \text{Second.index}) ; \\
  i' = - (i, l_1) ; \\
  j' = - (j, l_2) ; \\
  o = * (i', r') ; \\
  o' = + (o, 1) ; \\
  ad = + (o, j') ; \\
  ad' = + (ad, \text{Headersize}) ; \\
  t = \text{P.select} (X_1, ad') \\
\end{array} \}
\]

A similar rule applies for \text{P.2D.store}, and higher dimensional arrays.

**Lists**

The list data type can be represented as follows:

Cons :

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type</td>
<td>Tag</td>
<td>Hd</td>
<td>Tl</td>
</tr>
</tbody>
</table>

Nil :

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type</td>
<td>Tag</td>
</tr>
</tbody>
</table>

It is often possible to store niladic constructors using much less space by combining them with pointers. We won’t discuss such machine dependent representations in this paper.
The following constant definitions should be included in the set “D”. 

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil_size</td>
<td>2</td>
</tr>
<tr>
<td>Cons_size</td>
<td>4</td>
</tr>
<tr>
<td>Nil_Tag</td>
<td>0</td>
</tr>
<tr>
<td>Cons_Tag</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{TE}[\text{Nil}] = \{ t = \text{Allocate}(\text{Nil}.\text{size}); \\
\text{P}_{\text{store}}(t, \text{Type}, "\text{List}"); \\
\text{P}_{\text{store}}(t, \text{Tag}, \text{Nil}.\text{Tag}); \\
\text{In } t \}\n\]

\[
\text{TE}[\text{Open}.\text{cons}()] = \{ t = \text{Allocate}(\text{Cons}.\text{size}); \\
\text{P}_{\text{store}}(t, \text{Type}, "\text{List}"); \\
\text{P}_{\text{store}}(t, \text{Tag}, \text{Cons}.\text{Tag}); \\
\text{In } t \}\n\]

\[
\text{TE}[\text{Cons}.\text{_1}(X)] = \text{P}_{\text{select}}(X, \text{Hd})
\]

\[
\text{TE}[\text{Cons}.\text{_2}(X)] = \text{P}_{\text{select}}(X, \text{Tl})
\]

\[
\text{TE}[\text{Cons}(X_1, X_2)] = \{ t = \text{Allocate}(\text{Cons}.\text{size}); \\
\text{P}_{\text{store}}(t, \text{Type}, "\text{List}"); \\
\text{P}_{\text{store}}(t, \text{Tag}, \text{Cons}.\text{Tag}); \\
\text{P}_{\text{store}}(t, \text{Hd}, X_1); \\
\text{P}_{\text{store}}(t, \text{Tl}, X_2); \\
\text{In } t \}\n\]

\[
\text{TS}[\text{Cons}.\text{store}_1(X_1, X_2)] = \text{P}_{\text{store}}(X_1, \text{Hd}, X_2); \\
\]

\[
\text{TS}[\text{Cons}.\text{store}_2(X_1, X_2)] = \text{P}_{\text{store}}(X_1, \text{Tl}, X_2); \\
\]

\[
\text{TE}[\text{List}.\text{case}_m(X, E_1, E_2)] = \{ m \xrightarrow{} t_m = \text{P}_{\text{select}}(X, \text{Tag}); \\
\xrightarrow{} t_m = \text{Dispatch}_{2,m}(t, E_1, E_2); \\
\text{In } t_m \}\n\]

### 7.3 Function Calls and Closures

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

\[
\begin{array}{|c|c|c|c|}
\hline
0 & \text{Function name} & \text{Fast call name} & \text{Arity} \\
\hline
1 & \text{Arg} & \text{Rest} & \text{Chain} \\
\hline
\end{array}
\]

The constant definition \( \text{Closure\_size} = 5 \) should be included in the set "D".

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

\[
\text{Make\_closure} = \lambda (cl, X) . \ ( \{ f = \text{P\_select}(cl, \text{Funcname}); \\
\text{frc} = \text{P\_select}(cl, \text{Fastcallname}); \\
n = \text{P\_select}(cl, \text{Arity}); \\
ch = \text{P\_select}(cl, \text{Chain}); \\
cl' = \text{Allocate}(\text{Closure\_size}); \\
\text{P\_store}(cl', \text{Type}, \text{"Closure"}); \\
\text{P\_store}(cl', \text{Functionname}, f); \\
\text{P\_store}(cl', \text{Fastcallname}, f_{rc}); \\
\text{P\_store}(cl', \text{Arity}, n'); \\
\text{P\_store}(cl', \text{Chain}, ch'); \\
n' = -(n, 1); \\
ch' = \text{Ap}2(\text{Arg\_chain}, X, ch); \\
\text{ln cl'} \} )
\]

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

\[
\text{Arg\_chain} = \lambda (X, Xs) . \ ( \{zs' = \text{Allocate}(2); \\
\text{P\_store}(zs', \text{Arg}, X); \\
\text{P\_store}(zs', \text{Rest}, Xs); \\
\text{ln zs'} \} )
\]

The argument chain can be destructed using the following function:
\[ \text{Args}_n = \Delta(X) \cdot \{ n \cdot t_1 = \text{P.select} (X, \text{Chain}); \]
\[ a_n = \text{P.select} (t_1, \text{Arg}); \]
\[ t_2 = \text{P.select} (t_1, \text{Rest}); \]
\[ a_{n-1} = \text{P.select} (t_2, \text{Arg}); \]
\[ t_3 = \text{P.select} (t_2, \text{Rest}); \]
\[ \vdots \]
\[ a_1 = \text{P.select} (t_{n-1}, \text{Arg}); \]
\[ \ln \overline{a_n} \} \}\]

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

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

\[ \text{TE}[\text{Apply} (F, X)] = \{ n = \text{P.select} (F, \text{Arity}); \]
\[ \text{fire}_b = \text{Equal} (n, 1); \]
\[ \text{fire}_c = \text{BooleanInt} (\text{fire}_b); \]
\[ \text{res} = \text{Dispatch}_{2,2} (\text{fire}_c, \]
\[ \{ \text{fun} = \text{P.select} (F, \text{Functionname}); \]
\[ \text{as} = \text{P.select} (F, \text{Chain}); \]
\[ \text{as'} = \text{Ap}_2 (\text{Arg.chain}, X, \text{as}); \]
\[ \text{res'} = \text{Ap} (\text{fun}, \text{as'}); \]
\[ \ln \text{res'}, \]
\[ \text{Ap}_2 (\text{Make.closure}, F, X) \}
\[ \ln \text{res} \}

\[ \text{TE}[\text{Ap}_{n,m} (F, X)] = \{ m \cdot f' = \text{P.select} (F, \text{Fastcallname}); \]
\[ t_m = \text{Ap}_{n,m} (f', X); \]
\[ \ln t_m \} \]

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

\[ \text{TE}[\lambda n (X_n) \cdot (E)] = \{ cl = \text{Allocate} (\text{Closure.size}); \]
\[ \text{P.store} (cl, \text{Type}, \text{"Closure"}); \]
\[ \text{P.store} (cl, \text{Functionname}, \text{\textquoteleft}T\textquoteright_c); \]
\[ \text{P.store} (cl, \text{Fastcallname}, \text{\textquoteleft}T_{fc}\textquoteright); \]
\[ \text{P.store} (cl, \text{Arity}, n); \]
\[ \text{P.store} (cl, \text{Chain}, \text{\textquoteleft}End\textquoteright); \]
\[ \ln cl \} \]

The following two function definitions are included in the set D.
\[ T_c = \lambda_1(Xs) \cdot \{ \overline{X_n} = \text{Ap}_{1,n}(\text{Args}_n, Xs); \}
\]
\[ t = \text{TE}[E]; \]
\[ \ln t \}
\]
\[ T_{fc} = \lambda_n(X_n) \cdot \text{TE}[E] \]

'T\_c indicates the name T\_c and not the value associated to T\_c. Note that TE[E] can be computed once and shared between the curried aabd the fastcall version of the function. The same translation rule applies for \( \lambda_n \).

8 Signals

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

\[ S[\lambda_{n,m}(\overline{X_n}) \cdot \{ m \ Y_i = SE_1 \cdots (\{_{m+1} Y_i = SE_1 } \}
\]
\[ \cdots \]
\[ Y_n = SE_n \]
\[ Y_{n+1} = NSE_1 \]
\[ \cdots \]
\[ Y_{n+m} = NSE_m \]
\[ C_1 \]
\[ \cdots \]
\[ C_k \]
\[ \ln \overline{R_m} \} ]
\]
\[ S_{m+k} = \text{SE}[NSE_m] \]
\[ S_{m+k+1} = \text{SE}[C_k]; \]
\[ \cdots \]
\[ S' = \text{SC}(\text{Deadvariables}, \overline{R_m}, S') \]

Where SE\_i stands for an expression involving strict operators, whilst NSE\_i stands for either an applicative or a loop expression. Deadvariables are the parameters that are not being used in the body of the function.
\[
\text{SE}[\text{WLoop}_n (P, B, \overrightarrow{X}_n, Y)] = \text{WLoop}'_n (P, B, \overrightarrow{Y}, S_p, Y)
\]

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

\[
\text{SE}[\text{Ap}_{n,m} (F, \overrightarrow{X}_n)] = \text{Ap}_{n,m+1} (F, \overrightarrow{X}_n)
\]

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

Where \text{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:

\[
\text{WLoop}'_n (P, B, \overrightarrow{X}_n, S, \text{True}) \rightarrow \{n+1 \ t_n, S_b = \text{Ap}_{n,n+1} (B, \overrightarrow{X}_n); \ t_p, S_p = \text{Ap}_{n,2} (P, \overrightarrow{t_n}); \ S' = \text{Sync}_3 (S, S_b, S_p); \ t'_n, S'_l = \text{WLoop}'_n (P, B, \overrightarrow{t_n}, S', t_p) \}
\]

\[
\text{WLoop}'_n (P, B, \overrightarrow{X}_n, S, \text{False}) \rightarrow \overrightarrow{X}_n, S
\]

\[
\text{Ack\_store} (L, i, V) \rightarrow \{t = \text{Signal}; \ P\_store (L, i, V); \ \text{In } t\}
\]

\[
\text{Sync}_n (\overrightarrow{V}_n) \rightarrow ()
\]

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

**Acknowledgments**

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

**References**


