P-TAC: A Parallel Intermediate Language

Zena Ariola
Aiken Computational Laboratory
Harvard University
Cambridge MA, 02138

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge MA, 02139

Abstract

P-TAC is an intermediate-level language designed to capture the sharing of computation. It is a more suitable internal language for functional language compilers than the λ-calculus or combinators, especially for compiler optimizations. Using P-TAC, a proof for the confluence of Id, a higher-order functional language augmented with I-structures, is given. Using the notion of observational congruence the correctness of some compiler optimizations is shown.

1 Introduction

We present P-TAC (for Parallel Three Address Code), a simple and powerful declarative language, to study the confluence of the higher-level language Id and the correctness and confluence of the optimizations used in the Id-to-dataflow-graph compiler [16]. Id is a modern, higher-order, strongly-typed declarative language with I-structures [1, 12], arrays whose elements get "refined" during the course of computation [4]. I-structures have the flavor of variables in logic languages because it is possible to create an I-structure without giving a definition for each of its elements. (In a purely functional language a variable is given exactly one binding or definition at creation time.) All Id programs produce unique results, but a formal proof of the confluence of Id had eluded us until now.

A formalism that can express unambiguously the operational semantics of Id or compiler optimizations must be able to capture the sharing of computations. For example, the formalism must distinguish between the following two programs:

\[(Fa), (Fa) \] and \[ \{x = Fa; \in (x, x) \} \]

which may arise as a consequence of evaluating \( G (Fa) \), where \( G x = (x, x) \) and "\( \in \)" is the pairing operation. (The expressions that appear before the keyword "\( \in \)" are bindings, while the expression that follows the keyword "\( \in \)" is the main expression.) P-TAC captures such distinctions. In order to model accurately the implementation of a functional language, the sharing of computation is important. Moreover, it becomes a necessity when the language is extended with I-structures. So, one way to give an operational semantics for Id is to give a translation of Id into P-TAC, and a well-defined operational semantics for P-TAC. The advantage of this approach is that P-TAC, unlike other formalisms such as the λ-calculus or the combinatory calculus, leaves no room for choice in the sharing of subexpressions.

There are many intermediate languages, such as IF1[15], FLIC[14] and Term Graph Rewriting Systems[6, 7], that have been used to model functional language implementations. P-TAC is related to Traub's Functional Quads [17], which is a formalization of the dataflow-inspired model for the sharing of computation presented in [4, 13]. Traub provided the "three address" syntax for dataflow graphs and proved the confluence of functional Id using an Abstract Reduction System (ARS). However, we believe that P-TAC models data structures and locations in a novel way.

We begin the paper by giving the syntax of P-TAC (Section 2) and introduce the reader to the concept of I-structures. In Section 3, we give the operational semantics of P-TAC in terms of a set of rewrite rules, \( R_{P-TAC} \), and in Section 4 we show that P-TAC is confluent. A notion of observable behavior and program equivalence is introduced in Section 5. In Section 6, the optimizations used in the Id compiler are described in terms of some additional rewrite rules; in particular, some interesting optimizations which only approximate the behavior of the original program are discussed (Section 6.4). Furthermore, the confluence and strong normalization properties of some of these optimizations are proven. In Section 7 the correctness of the optimizations is proven. We end the paper with some possible directions for future work.

2 Syntax of P-TAC

The syntax of P-TAC is described by the grammar of Figure 1, whose start symbols are Program and Definition. For better readability, we will take some liberties with the concrete syntax. For example, instead of writing "\( + y \)" we will use the familiar infix notation and write "\( x + y \)." A program \( M \) in P-TAC is evaluated with respect to \( D \), a set of user-defined functions \( F_1 \ldots F_n \) which are generated by the syntactic category Definition. As an example of \( D \), consider the following set of function definitions which may be mutually recursive:

\[
F_1 x_1 \ldots x_m = b_1 \\
\vdots \\
F_n x_1 \ldots x_n = b_n
\]

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0-89791-328-0/89/0009/0230 $1.50 230
The user-defined functions of P-TAC, which simply amounts to disallowing the user to write Allocate and Store. However, it is important to emphasize that the primitive functions, e.g., +, *, Cond, etc., are not curried. This restriction does not cause any loss of expressive power because it is always possible to eliminate all nested function definitions by “lambda lifting,” i.e., by passing all free variables of a function as parameters to the function [9].

Even though we believe that the translation of Id into P-TAC is quite straightforward, particular care has to be devoted to conditional expressions because, in Id and dataflow languages, conditionals behave differently in functional languages. For example, the following Id expression

\[(\text{if } p \text{ then } e_1 \text{ else } e_2)\]

is not equivalent to

\[(x = e_1; \ y = e_2; \ \text{in } (\text{if } p \text{ then } x \text{ else } y))\]

If we restrict our attention to functional Id (Id without I-structures), the above two expressions do indeed denote the same value; the semantic distinction shows up only when Id with I-structures is considered. Yet even in functional Id, the two expressions behave differently. In the first expression either \(e_1\) or \(e_2\) gets evaluated, whereas in the second expression both \(e_1\) and \(e_2\) get evaluated. In fact, \(\text{if } p \text{ then } e_1 \text{ else } e_2\) is equivalent to

\[
\text{case } p \text{ of } 0 \Rightarrow e_1; \ 1 \Rightarrow e_2
\]

where “0” represents a dummy argument. Embedding the terms \(e_1\) and \(e_2\) inside function definitions prevents their evaluation because a function body gets evaluated only when the function is applied to the number of arguments specified in its definition. It is straightforward to translate the above Id expression into P-TAC because the if-expression becomes the P-TAC Cond and definitions for \(F\) and \(G\) can be lambda-lifted to the top level.

2.2 I-structures

Implementation of any functional data structure constructor requires allocation of storage and binding an expression to each location of the allocated storage. To compute the \(i^{th}\) element of a data structure, the expression associated to the \(i^{th}\) location is evaluated. Thus, an implementation of “Make-tuple \(x\ y\)” can be written as follows:

\[
\text{Def } F \ x = e_1; \ 2; \ \text{in } \text{Allocate } 2; \ \text{Store } x 1 x; \ \text{Store } z 2 y; \ \text{in } a
\]

It is not legal for the user to write the above term in functional P-TAC, because it contains Allocate and Store. It is a legal term in full P-TAC.

Normally, these details of data structure operations are not included explicitly in the operational semantics of functional languages. Moreover, P-TAC departs from other functional languages by giving users direct access to Allocate and Store primitives. Data structures defined using these primitives are called I-structures. P-TAC places the so-called “single-assignment” restriction that no more than one store operation can be performed in any location. In general, this

Legend:

- FPi means Primitive Function with \(i\) arguments
- UDF means User Defined Function
- SE means Simple Expression
- Expression ::= Simple Expression
- SE ::= Variable | User Defined Functions | Integer | Boolean | Nil
- Block ::= Block * Statement
- Statement ::= Binding | Command
- Command ::= Store SE SE SE
- Binding ::= Variable = Expression
- Definition ::= UDF Variable . . . Variable = Block
- Program ::= Block

Figure 1. The Grammar of Initial-Terms of P-TAC.
restriction cannot be checked at compile time. In spite of this drawback, P-TAC programs have a clear “declarative meaning” in a manner similar to logic programs.

We go back to the example given in the introduction to show how to define the operational semantics of I-structures correctly. Suppose we evaluate

\[ x = \text{Allocate 2}; \text{in} \, \text{Apply} \, G \, x \]

where the function \( G \) is defined as

\[ G \, x = \{ \text{in Make-tuple} \, x \, x \} \]

P-TAC dictates that the answer is

\[ x = \text{Allocate 2}; \text{in Make-tuple} \, x \, x \]

(call it \( N_1 \)) and not the Id expression (Allocate 2, Allocate 2) which may be written in P-TAC as follows:

\[ N_2 \equiv \{ x = \text{Allocate 2}; y = \text{Allocate 2}; \text{in Make-tuple} \, x \, y \} \]

This distinction is important because \( N_1 \) and \( N_2 \) are not equivalent. Since “Allocate 2” allocates storage, we can say intuitively that \( N_1 \) returns a pair containing two references to the same I-structure, while \( N_2 \) represents a pair of two different I-structures. The following program can distinguish between \( N_1 \) and \( N_2 \):

\[ F \, a = \{ b = \text{Select} \, 1 \, a; \]
\[ c = \text{Select} \, 2 \, a; \]
\[ \text{Store} \, b \, 1 \, 100; \]
\[ \text{Store} \, c \, 1 \, 200; \]
\[ \text{in} \, 0 \} \]

Note that \( (F \, N_1) \neq (F \, N_2) \). Intuitively, the above program stores the value 100 in the first I-structure (named “\( b^* \)) and the value 200 in the second I-structure (named “\( c^* \)) of the pair “\( a^* \). The program \( (F \, N_1) \) violates the single-assignment restriction, because both \( b \) and \( c \) refer to the same I-structure. As will be clarified in the successive sections, \( (F \, N_1) \) will enter a “contradictory” state, while \( (F \, N_2) \) will terminate successfully. Thus, we have:

\[ (F \, N_1) \neq (F \, N_2) \rightarrow N_1 \neq N_2 \]

We illustrate the advantage of augmenting a functional language with I-structures by considering the problem of “flattening” a list-of-lists. A list in P-TAC can be defined inductively as follows:

- Nil is a list.
- If \( zs \) is a list, then so is \( \text{Make-tuple} \, x \, zs \) for any \( x \).

Flattening a list of lists requires the repeated appending of two lists. In order to append two lists, in functional P-TAC one ends up making a copy of the first list and replacing the Nil by the second list. Using I-structures, however, one can define an “open” list [2] along the lines of a “differential” list in logic programming. An open list is a list of I-structures where each open I-structure has two components and is referred to as a pair. The first slot of the first pair and the second slot of the last pair are always empty; an empty open list can be represented by a pair with two empty slots. More precisely:

\[ (\text{Allocate} \, 2) \text{ is an open list.} \]

- If \( zs \) is an open list then so is \( \text{cons-open} \, x \, zs \) for any \( x \), where \( \text{cons-open} \) is defined as follows:

\[ \text{cons-open} \, x \, zs = \{ x = \text{Allocate} \, 2; \text{Store} \, x \, 2 \, x; \]
\[ \text{Store} \, x9 \, 1 \, x; \text{in} \, x \} \]

The head of the list is contained in the second slot of the first pair. Appending two open lists only requires storing the head of the second list in the second component of the pair at the end of the first list! The use of open lists can avoid unnecessary copying, as well as expose more parallelism in the flattening operation. A fuller discussion of the advantages of I-structures is beyond the scope of this paper; the interested reader is referred to [4].

3 Operational Semantics for P-TAC

The operational semantics is given in terms of an Abstract Reduction System [10], which is a structure \((A, \rightarrow_R)\) where \( A \) is a set of terms and \( \rightarrow_R \) is a binary relation on \( A \). The relation \( \rightarrow_R \) is derived from the set of rewrite rules, \( R \), on terms of \( A \). For our purposes, \( A \) contains all closed P-TAC terms, i.e., terms without free variables, generated by the grammar of Figure 1, with \( \text{Program} \) as start symbol, plus all terms that can appear during a program evaluation. Therefore, we need to extend the grammar of Figure 1 with new syntactic categories, such as \( \text{Locations, I-structures and Closures} \), in order to name all the different objects that can appear at run time. The meaning of locations and closures is informally explained in the next section. The grammar that generates all elements of \( A \) is described in Figure 2. Will designate the terms generated by Figure 1 as \( \text{Initial-Terms of P-TAC} \). The ground values of \( A \) cannot be reduced any further and correspond to the values of P-TAC, which are integers, booleans, closures and I-structures.

Though the user-defined functions are constants, their operational meaning, in contrast to the operational meaning of the primitive functions, is given by the user via the set \( D \). Thus, the abstract reduction system is actually parameterized by \( D \) and therefore, when \( D \) is not clear from the context, we will write \((A, \rightarrow_{R_{\text{P-TAC}}}) \, D\).

3.1 Locations

The most novel aspect of P-TAC is the way it models data structure operations using a special class of identifiers called \( \text{Locations} \). An I-structure of \( n \) elements is represented by \( n \) \( \text{Locations} \). The only permissible operations on locations are “l-fetch \( l \)”, for reading the contents of location \( l \), and “l-store \( l \, v \)”, for storing \( v \), some ground value, in location \( l \). Confluence of P-TAC crucially depends upon the “write-once” restriction, that is, only one store operation is permitted on a location.

Unlike variables, which are names for expressible values, locations are merely names of memory locations and cannot appear in a left-hand side of a \( \text{Binding} \) or in the formal list of a \( \text{Definition} \). Furthermore, locations are \( \text{globally unique} \), that is, the scope of a location identifier is the entire program. Therefore, unlike a variable with a binding in a block, a location cannot be \( \alpha \)-renamed locally within a block. P-TAC also does not permit two different location identifiers to refer to the same location. The syntax and the associated rewrite rules (to be described in the next section) do
not allow any confusion between the name of a location and its contents. Thus, while a binding like ‘z = y’ has the meaning that the variables x and y are names for the same value, the corresponding binding for locations ‘l_i = l_j’ does not make sense because two location identifiers can never be the same. No equality-test on locations is permitted, but equality on location contents can be expressed by writing: ‘( z = 1; fetch i; y = 1; fetch j; in (Equal? z y) )’.

Locations are used to implement Closures (partial applications of functions) as described next. We denote the number of arguments specified in the definition of a user-defined function F by n_F and often write F^n_F for clarity. In almost all implementations of functional languages, substitutions inside the body of F are not performed until all the arguments for F have been specified. Thus, when F^n is applied to ξ, instead of performing a substitution, a data structure known as a Closure is built. A closure contains an I-structure to remember the arguments specified so far. It also contains the number of arguments still missing (the m).

### 3.2 Canonical Representation of Terms

Consider the following terms:

1. {x = 8; in (y = x; in z + y)}
2. {x = 8; y = z; in x + y}
3. {x = 8; in z + x}
4. {in 8 + 8}

Although these terms are syntactically distinct from each other, they all behave the same operationally. For example, the Id compiler would represent all these terms using the same dataflow graph, i.e., 8 + 8. Therefore, in our reduction system terms that have the same graph, up to isomorphism, are equivalent. We define then a canonicalization procedure in order to select a representative of each equivalence class, i.e., the canonical form of a term. Reductions will be performed only on canonical terms.

**Definition 3.1** The canonical form of a term M is obtained as follows:

1. Flatten all blocks according to the following two rules:
   - Terms { in z + l}, {y = l; in 2 + y} and {Z = 2; y = 1; in z f y} have the same canonical form. However, this canonical form is different from the canonical form of {z = Apply F 2; y = 1; in z + y}. This example shows that a binding whose right-hand-side (RHS) is not grounded is meaningful in determining the operational structure of a term.
   - Notice only ground values can be substituted freely. This restriction on substitution of terms for variables is what allows P-TAC to model sharing of computation precisely.

2. Each P-TAC term has a unique canonical form up to a-renaming.

3. Each P-TAC term has a unique canonical form up to a-renaming.

**Definition 3.2** Two closed terms M and N in canonical form are said to be α-equivalent if each can be transformed into the other by a consistent renaming of variables in M and N, respectively, to avoid names clashes with the names in the surrounding scope. Note that we do not need to rename locations because they are globally unique.

4. For each binding of the form “y = z” in M, where x and y are distinct variables, replace each occurrence of “y” in M by “z” and then erase the binding “y = z” from M.

5. For each binding of the form “y = v” in M, where v is a ground value, replace each occurrence of “y” in M by “v” and then erase the binding “y = v” from M.

**Lemma 3.3** Each P-TAC term has a unique canonical form up to α-renaming.

**Proof:** Since there are only a finite number of blocks and occurrences of a variable in a term M, the canonicalization procedure is strongly normalizing. It is easy to see that the block flattening rules are locally confluent up to β-renaming. Since variable substitution rules within a flattened block are also locally confluent, the uniqueness of the canonical form of M up to α-renaming follows trivially from Newman’s Lemma [6].

Some examples will clarify canonicalization of terms and α-equivalence.

1. Terms { in x + 1}, {y = 1; in x + y} and {z = 2; y = 1; in z + y} have the same canonical form. However, this canonical form is different from the canonical form of {z = Apply F 2; y = 1; in x + y}. This example shows that a binding whose right-hand-side (RHS) is not grounded is meaningful in determining the operational structure of a term.
2. Terms \{l-store l v; in 5\} and \{l-store l v; x = 5; in x\} have the same canonical form which is different from \{in 5\}.

3. The following two terms are α-equivalent:
\{l-store l0 v0; l-store l1 v1; in (l-structure, 2, l0, l1)\}
\{l-store l0' v0; l-store l1' v1; in (l-structure, 2, l0', l1')\}

4. The following two terms are not α-equivalent:
\{l-store l0 (l-structure, 2, l0, l1); l-store l1 (l-structure, 2, l0, l1); in (l-structure, 2, l0, l1)\}

These terms could arise during the execution of the terms \(N_1\) and \(N_2\) introduced in Section 2.2. This further emphasizes the point that terms \(N_1\) and \(N_2\) are different.

3.3 Reduction Notions

Intuitively, we define the evaluation of a program \(M\) as consisting of repeatedly rewriting its subterms until no further rewriting is possible. However, as we shall see shortly, some of our rewrite rules have a precondition. The following is an example of a rule with a precondition:

\[
\begin{align*}
\text{l-store} & \ 5 \\
\rightarrow & \\
\text{l-fetch} & \ 5
\end{align*}
\]

where the command l-store \(l\ 5\) over the line denotes the precondition. The above rule says that the subterm l-fetch \(l\) of \(M\) can be rewritten to 5 if the command l-store \(l\ 5\) occurs in the context, i.e., \(M\).

Note from the above example that the precondition does not merely allow or disallow the rewriting, but also affects the outcome of the rewriting. Therefore we define a redex in our ARS as follows:

**Definition 3.4** Given a program \(M\), let \(N\) be a subterm of \(M\). \(N\) is said to be a redex if it matches the left hand side of a rule and the precondition of the rule is satisfied. If the precondition of the rule can be satisfied by more than one statement then \(N\) represents a distinct redex corresponding to each such statement.

Following Klop [10], we will sometimes qualify a redex with the name of the specific rule it matches; thus, we will say, for example, that \(a + 2\) is a δ-redex, for it matches the LHS of a δ-rule. We remind the reader that a context \(C[\ ]\) is a term with a hole in it, such that, when a suitable term is plugged in the hole, \(C[\ ]\) becomes a proper term [5]. A closed context is a context with no free variables.

**Definition 3.5** Given an ARS \((A, R)\), if \(M, N \in A\) and \(\sigma \in R\), then \(M\) reduces to \(N\) in one step \((M \rightarrow R N)\), iff \(\exists\) a context \(C[\ ]\), and a redex \(\rho\), such that, \(M \equiv C[\rho]\), \(N \equiv C[\rho']\), and \(\rho \rightarrow \rho'\). The transitive reflexive closure of \(-\rightarrow R\) is written as \(\rightarrow R\).

We will sometimes explicitly mention the rule being applied in the reduction by writing \(M \rightarrow \rho N\). We will also make use of a different notation, \(M \xrightarrow{\rho} N\), where \(\rho\) is the subterm (of term \(M\)) being reduced.

3.4 The Rewrite Rules of P-TAC \((R_{P-TAC})\)

We now present the set of rewrite rules, \(R_{P-TAC}\), for defining the ARS for P-TAC. In the following \(n\) and \(\alpha\) represent a variable and a numeral, respectively. Remember that the rules are only to be applied to terms in canonical form. To avoid a distracting discussion of type errors, we assume that a primitive function is only applied to arguments of appropriate types.

- **δ rules**

  \[
  m + n \xrightarrow{\delta} m + n
  \]

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

- **Conditional rules**

  \[
  \begin{align*}
  \text{Cond True } x y & \xrightarrow{\text{cond}} x \\
  \text{Cond False } x y & \xrightarrow{\text{cond}} y
  \end{align*}
  \]

- **Data Constructor rules**

  \[
  \begin{align*}
  \text{Make-tuple } x y & \xrightarrow{\text{mkt}} \{x = \text{Allocate} 2; \text{Store } z 1 x; \text{Store } z 2 y; \text{in } x\}
  \end{align*}
  \]

- **I-structure rules**

  \[
  \begin{align*}
  \text{Allocate } n & \xrightarrow{\text{alc}} (l-structure, n, l_0 \ldots l_{n-1}) \\
  \text{Length } (l-structure, n, l_0 \ldots l_{n-1}) & \xrightarrow{\text{len}} n
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Nil? Nil} & \xrightarrow{\text{nil}} \text{True} \\
  \text{Nil? isv} & \xrightarrow{\text{nil}} \text{False} \ (\text{if isv is a ground value other than Nil})
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Select } (l-structure, \alpha, l_0 \ldots l_{n-1}) i & \xrightarrow{\text{sel}} l-fetch l_i \ (0 \leq i \leq n-1) \\
  \text{Select } (l-structure, \alpha, l_0 \ldots l_{n-1}) i & \xrightarrow{\text{sel}} \text{Error} \ (i < 0 \lor i > n-1)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Store } (l-structure, \alpha, l_0 \ldots l_{n-1}) i y & \xrightarrow{\text{sto}} \text{Store-Error} \ (0 \leq i \leq n-1) \\
  \text{Store } (l-structure, \alpha, l_0 \ldots l_{n-1}) i y & \xrightarrow{\text{sto}} \text{Store-Error} \ (i < 0 \lor i > n-1)
  \end{align*}
  \]

\[
\begin{align*}
\text{l-store} & \ l v \\
\text{l-fetch} & \ l \rightarrow v
\end{align*}
\]

where \(v\) is a ground value

\[
\begin{align*}
N \equiv \text{l-store} \ l \ v \land N' \equiv \text{l-store} \ l \ v' \land \\
N, N' \text{ are distinct subterms of program } M
\end{align*}
\]
• Application rules

\[
\text{Apply (Closure, } F^n, m, \text{ isu}) \ x \rightarrow \ \text{up} \\
\{ \text{chain } = \text{Make-tuple } x \ \text{isu;} \\
\text{in Make-closure (Closure, } F^n, m, \text{ isu) chain } \} \\
\text{where } m > 1
\]

\[
F^n z_1 z_2 \cdots z_n = \{ B_1; B_2; \cdots B_m; \ \text{in } z \} \in D
\]

\[
\text{Apply (Closure, } F^n, 1, \text{ isu}) \ x \rightarrow \ \text{up} \\
\{ \text{chain } = \text{Select isu 1;} \\
\text{rest } = \text{Select isu 2;} \\
\text{in } z \}
\]

All local variables in the block on the right hand side are considered to be new. Note that we will have one instance of the last rule for every user-defined function.

• Closure rules

\[
F^n \rightarrow \text{(Closure, } F^n, m, \text{ Nil)}
\]

\[
\text{Make-closure (Closure, } F^n, m, \text{ isu}) \ isu' \rightarrow \ \text{up} \\
\text{(Closure, } F^n, m-1, \text{ isu'})
\]

where isu, isu' are I-structures values

Discussion:

1. By insisting that \( v \) be a ground value in the \( I-\)fetch rule, \( \text{I-f} \), we capture the intuition that only ground \( I-\)values can be stored into locations. Also notice that \( I-\text{store} \ l \ v \) is only a precondition for the \( I-\text{fetch} \) rule, and thus, it remains unaffected by the application of the \( I-\text{fetch} \) rule.

2. According to the blow-up rule, \( \text{b-f} \), the whole program \( M \) is a redex. That is, if an attempt is made to store something in an \( I-\)structure location that already contains a value, the whole program \( M \) goes into a "contradictory state" or \( T \).

The blow-up rule seems rather drastic. However, localizing the effect of multiple stores would require keeping track of what computations have depended upon which fetches. Since a location can be computed at run time as in \( \{ i = \text{Apply } F \ a; \ \text{Store } x \ v; \cdots \} \), it is not possible, in general, to determine if a program will go to \( T \) without executing it. The confluence of P-TAC will ensure that if any terminating reduction sequence goes to the \( T \), then so will all the terminating ones.

3. Note that the program will go to \( T \) even if the second value to be stored in a location is the same as the first one. We could relax this condition by requiring that the values being stored in a location be "unifiable" with the one already in the location. This poses some conceptual problems when the values under consideration are closures, and some efficiency problems when the values are I-structures.

4. An important fact to be noted is that the blow-up rule will never be applicable in the functional subset of P-TAC. If the use of \( \text{Store} \) only arises indirectly through \( \text{Make-tuple} \), then it is clear that each location has exactly one store operation associated with it. This can be seen by examining the rewrite rule for \( \text{Make-tuple} \).

5. The I-structure rules allow the specification of infinite lists, such as: \( \{ x = \text{Make-tuple } 1 \ x \; ; \; \text{in } x \} \). This expression gets reduced to:

\[
\{ I-\text{store} l_0 \ 1; \\
I-\text{store} l_1 \ (I-\text{structure}, 2, l_0, l_1); \\
in (I-\text{structure}, 2, l_0, l_1) \}
\]

where no further reductions are possible.

6. We have included "errors" in case of I-structures rules because these errors cannot be avoided by static checking. However, we have not provided any rule for error propagation.

4 Confluence of P-TAC

We prove the confluence of P-TAC by showing that P-TAC is subcommutative, a property that is defined as follows:

Definition 4.1 An ARS is said to be subcommutative, if for terms \( M, M_1 \), and \( M_2 \):

\[
M \rightarrow M_1 \wedge M \rightarrow M_2 \Rightarrow \exists M_3 \text{ such that }
\]

\[
M_1 \xrightarrow{a/n} M_3 \wedge M_2 \xrightarrow{a/n} M_3
\]

where \( a/n \) means in zero or one step.

Lemma 4.2 A reduction relation that is subcommutative is confluent. [5]

In order to show that P-TAC is subcommutative, we need to examine the interaction between rules of RPA-TAC. In a Term Rewriting System (TRS), the interaction between rules is often characterized by the notion of "overlapping patterns". The pattern of a rule is the abstract syntax tree associated to the LHS of the rule, where each variable is replaced by a hole. Thus, the constant symbols on the LHS of a rule determine the pattern. Two rules are said to be ambiguous if their patterns overlap. In our ARS two rules may not overlap, but one of the them may still affect the other by destroying its preconditions. The following notion of interfering rules subsumes the notion of overlapping.

Definition 4.3 i-rule interferes with j-rule iff:
1. the application of i-rule can invalidate the precondition of j-rule; or
2. the pattern of i-rule overlaps with the pattern of j-rule.

Fact 4.4 The blow-up rule, \( \text{b-f} \), and the \( I-\text{fetch} \) rule, \( \text{I-f} \), are the only interfering rules in P-TAC.

The blow-up rule can invalidate the precondition of any rule. The \( I-\text{fetch} \) rule interferes with itself as shown by the
The subterm \( I\text{-fetch } l \) matches the (LHS) of the \( I\text{-fetch } \) rule. However, the precondition for the rule can be satisfied by either the subterm (I-store \( l \)) or by the subterm (I-store \( l \)). Thus, redexes \( p_1 \) and \( p_2 \), shown above, do overlap and, consequently, the \( I\text{-fetch } \) rule interferes with itself.

**Lemma 4.5** P-TAC is subcommutative.

**Proof:** We want to show that \( \forall \) (canonical) \( M, M_1, M_2 \in \mathcal{A} \), and \( \forall \) i-rule, j-rule \( \in \mathcal{R}_P \text{-TAC} \), the following holds:

\[
M \rightarrow M_1 \land M \rightarrow M_2 \Rightarrow \exists M_3 \text{ such that } M_1 \stackrel{\text{I}}{\rightarrow} M_3 \land M_2 \stackrel{\text{J}}{\rightarrow} M_3
\]

1. (Non-interfering rules) Suppose the i-redex does not interfere with the j-redex. By inspecting the non-interfering rules of \( \mathcal{R}_P \text{-TAC} \), and the structure of canonical terms, we can see that the i-redex and j-redex have to be disjoint. By the definition of interference, the \( i\)-reduction (analogously, the \( j\)-reduction) cannot invalidate the precondition of the j-redex (i-redex). Therefore, the i and j reductions commute.

2. (Interfering rules)

2.1 Let i-rule be \( \rightarrow \) and j-rule be different from \( \rightarrow \). Since only the blow-up rule can destroy the precondition for the blow-redex, in the following diagram \( M_2 \) is still a blow-redex.

\[
\begin{array}{c}
M \\
\stackrel{\text{blw}}{\rightarrow} \\
T \\
\stackrel{\text{blw}}{\rightarrow} \\
M_2
\end{array}
\]

2.2 Let both i-rule and j-rule be \( \rightarrow \). Then,

\[
\begin{array}{c}
M \\
\left( \begin{array}{c}
T \\
\end{array} \right)
\end{array}
\]

2.3 Let both i-rule and j-rule be \( \rightarrow \). Then \( M \) is a blow-redex which cannot be destroyed by the \( I\text{-fetch } \) reductions. Hence \( M_1 \) and \( M_2 \) are also blow-redexes.

\[
\begin{array}{c}
M \\
\left( \begin{array}{c}
M_1 \\
M_2
\end{array} \right)
\end{array}
\]

5 Program Equivalence and Observable results

Notice that, unlike functional TRS's, the normal form of a P-TAC program will usually contain a number of commands, such as I-store \( l \), because commands are never erased according to our rewrite rules. Such store commands reflect the state of storage at the end of an execution. Consider the following two programs, \( M_1 \) and \( M_2 \), which are in normal form:

\[
M_1 \equiv \{ \text{in } 9 \} \quad \text{vs} \quad M_2 \equiv \{ \text{I-store } l \text{ 4}, \text{I-store } l \text{ 5, in } 9 \}
\]

Though these programs are not \( \alpha \)-equivalent, they can be said to produce the same "answer", i.e., 9. That is, if the user does not care about the contents of the store at the termination of his program, he may observe "9" at his terminal in both cases. Exactly what should be observable about a value is a tricky question if the value is of a higher-type, i.e., an I-structure or a closure. For example, to a user the internal representation of a function (or of a partial application) is of no interest; he can only apply it to another term. As is to be expected the question of program equivalence, i.e., whether two programs can be substituted freely for each other, is related to the notion of "answer" we choose. We introduce the notion of an erasure to factor out the internal store of a term.

**Definition 5.1** Given a term \( M \), where \( M \) is in normal form, the L-Erasure of \( M \), \( E(M) \), is the term obtained by erasing all commands of the form I-store \( l \), where \( v \) is a ground value, from \( M \).

We introduce the notion of proper-termination and improper-termination to deal with the fact that \( E(M) \) may contain unresolved I-fetches or circular bindings.

**Definition 5.2** Given an ARS \( P = (\mathcal{A}, \mathcal{R}_P \text{-TAC})_P \) and \( M \in \mathcal{A}\text{-Initial-Terms of } A \), the answer produced by \( M \) with respect to \( P \), \( Ansp(M) \), is undefined if \( M \) does not have a normal form. Otherwise, \( M \) reduces to a normal form \( N \), and \( Ansp(M) \) is

- if \( E(N) = T \) then \( T \),
- if \( E(N) \) is of the form \{ in \( v \) \} and
  - if \( v \) is either integer, boolean or Error, then (proper-termination, \( v \)),
  - if \( v \) is a closure value, then (proper-termination, Closure),
  - if \( v \) is an I-structure value, then (proper-termination, (I-structure, \( n \))),
where proper-termination, Closure, and I-structure are reserved constants and \( n \) denotes the length of the I-structure value;
- if \( E(N) \) is of the form \{ \( B_1 ; B_2 ; \ldots \) in \( v \) \} and
  - if \( v \) is either integer, boolean or Error, then (improper-termination, \( v \)),
  - if \( v \) is a closure value, then (improper-termination, Closure),
  - if \( v \) is an I-structure value, then (improper-termination, (I-structure, \( n \))),
  - else (improper-termination, Nothing).
where improper-termination and Nothing are reserved constants.
According to the above definition, \( \text{Ans}(M_1) = \text{Ans}(M_2) \), for
\( M_1 \) and \( M_2 \) defined at the beginning of this section. The
same is true for the following two programs \( M_3 \) and \( M_4 \):

\[
\begin{align*}
M_3 & \equiv \{\text{l-store} l_0 \ 4; \ \text{l-store} l_1 \ 5; \ \text{l-store} l_2 \ 4; \ \text{l-store} l_3 \ 5; \ \text{in} \{\text{l-structure}, 2, l_0, l_1\}\} \\
M_4 & \equiv \{\text{l-store} l_0' \ 5; \ \text{l-store} l_1' \ 4; \ \text{l-store} l_2' \ 5; \ \text{l-store} l_3' \ 4; \ \text{in} \{\text{l-structure}, 2, l_0', l_1'\}\}
\end{align*}
\]

However, \( M_3 \) and \( M_4 \) are not substitutable for each other,
because, for example, the answers produced by “Select \( M_3 \) 1”
and “Select \( M_4 \) 1” are not the same. This example clearly
shows that in the presence of higher-order types, context
plays an important role. Thus, we define the following equiv-
laence between programs [11]:

Definition 5.3: \( M, N \in A \) are said to be observationally
congruent if for all user definable contexts \( C \):

\[
\text{Ans}(C[M]) = \text{Ans}(C[N]).
\]

According to this definition programs \( M_1 \) and \( M_2 \) are ob-
ervationally congruent, while \( M_3 \) and \( M_4 \) are not. As Meyer
points out in [11], we could have limited the notion of ob-
serveable values to, say, the integer “3” without having any
impact on the equivalence of programs!

6. Operational Semantics of an Optimizing Compiler for
P-TAC

The Id compiler [16] has several phases. It first translates
Id into high-level dataflow graphs (“program graphs”) and
then repeatedly applies many optimizations to these graphs.
It then generates the “machine graphs”, which is the ma-
chine language of the Tagged-Token Dataflow Architecture
(TTDA). It also does optimizations on the machine graphs,
but these are specific to TTDA. P-TAC is very close to pro-
gram graphs. Interestingly enough, the compiler optimiza-
tions on program graphs can be expressed in terms of an
abstract reduction system \( (D, \rightarrow_{\text{comp}}) \), where \( D \) is the set of
all possible user-defined functions, that is, all the terms gen-
erated by the grammar of Figure 1 with Definition as start
symbol.

6.1 The Definition of a Correct Compiler

The rules stated in Section 3.4 are applied only to the main
expression of a program; the definitions of functions are not
involved. The domain of action of a compiler, on the other
hand, is the set of user definitions; the rules are applied to
subterms on the right-hand side of the user-defined func-
tions. Thus, a compiler may be defined as follows:

Definition 6.1: A compiler is an ARS \( (D, \rightarrow_{R_c}) \) which
given \( D \subseteq D \) transforms the ARS \( P=(A, R_{\text{comp}})D \) into another
ARS \( Q=(A, R_{\text{TAC}})D' \) such that,

1. \( \forall F, F \ x_1 \ x_2 \ldots x_n = e \in D, \implies \exists F', F \ x_1 \ x_2 \ldots x_n = e' \in D' \text{ and e} \rightarrow_{R_c} e' \)
2. \( R_c \) is strongly normalizing.

Discussion:

1. There are several useful optimizations that, if included
in \( R_c \), will destroy its strongly normalizing property.
   In such cases we need to restrict, in some manner,
   the number of times such an optimization rule can be
   applied. We will discuss one such optimization later.

2. Usually a compiler will only produce \( e' \) in normal
   forms with respect to \( R_c \). If \( R_c \) is confluent
   then it is guaranteed that all terminating reduction se-
   quences in \( R_c \) lead to the same “efficient” program, and we can
   concentrate on finding an efficient reduction strategy
   for generating the optimized program.

3. The exclusion of \( M \), the main expression, from com-
   piler optimizations, is not a serious limitation, because
   \( M \) can be enclosed inside a function body and then opti-
mized.

The foremost question regarding any optimization is whether
it preserves the meaning of the original program. Usually,
questions about meanings take us into the denotational
semantics of programs because the optimized program is
not going to be syntactically equivalent or \( \alpha \)-equivalent
to the source program. We will sidestep the denotational
semantics of P-TAC programs, by formulating the correctness
question in terms of observational congruence. The defi-
nition given below basically states that if no program can
“observe” a difference (produce different answers) between
the use of the old and new definitions then the transforma-
tions are correct.

Definition 6.2 A compiler \( (D, \rightarrow_{R_c}) \) which given \( D \subseteq D \)
transforms the ARS \( P=(A, R_{\text{TAC}})D \) into another ARS
\( Q=(A, R_{\text{TAC}})D' \) is totally correct with respect to \( R_{\text{TAC}} \)
if \( \forall M \in \text{Initial-Terms of } A \text{ then Ans}(M) = \text{Ans}(M) \).

In some sense the correctness criterion chosen is too strict.
Generally all we care about is that the transformed program
produces exactly the same answers as the original. How-
ever, the cases when the original program does not termi-
nate or terminates improperly, it may be all right for the
transformed program to take some “liberties” and produce
a result. In the denotational jargon we would say that
the original program should be an approximation of the trans-
formed program.

Definition 6.3 A compiler \( (D, \rightarrow_{R_c}) \) which given \( D \subseteq D \)
transforms the ARS \( P=(A, R_{\text{TAC}})D \) into another ARS
\( Q=(A, R_{\text{TAC}})D' \) is partially correct with respect to \( R_{\text{TAC}} \)
if \( \forall M \in \text{Initial-Terms of } A \text{, if } M \text{ terminates properly in } P \text{ then } \text{Ans}(M) = \text{Ans}(M) \).

6.2 Compiler Rules that Preserve Total Correctness \( (R_{\text{comp}}) \)

We begin by examining rules of \( R_{\text{TAC}} \) that may be applica-
tible in the compilation process. We will call such rules as
\( R_{\text{TAC}} \). Since a compiler does not allocate storage, it can-
not execute Allocate or most other I-structure rules. The Ap-
ply operator cannot be executed indiscriminately at compile
time either. Besides the difficulty of allocating storage for
closures, the execution of Apply has the potential to get the
compiler into an infinite recursion. This is not acceptable
because a compiler must terminate, regardless of whether
the program is correct or incorrect. Thus, in addition to the
canonicalization procedure given in Definition 3.1, \( R_{\text{TAC}} \)
includes the following rules:

- \( \delta \) rules
- Conditional rules
- Nil? rules.

As usual we will consider reductions only on the canonical
representation of terms.
A compiler may apply some additional rewrite rules, \( R_{\text{opt}} \), to transform a Program into an even more efficient version. These optimization rules are usually not included in \( R_{\text{P-TAC}} \) because it is not clear how the preconditions of some of these rules can be checked efficiently at run time. Let

\[
R_{\text{comp}} = R_{\text{P-TAC}} \cup R_{\text{opt}}
\]

A description of \( R_{\text{opt}} \) follows:

- **Arity Detection rule**
  
  \[
  \begin{align*}
  f_1 &= \text{Apply } F^m x_1; \\
  f_2 &= \text{Apply } f_1 x_2; \\
  \vdots \\
  f_{n-1} &= \text{Apply } f_{n-2} x_{n-1}; \\
  \text{Apply } f_n x_n &\rightarrow_a F_C(x_1, \ldots, x_n)
  \end{align*}
  \]
  
  where it is presumed that there is a combinator \( F_C \) (acronym for Fast Call) for each user-defined function. This rule detects if all the arguments for a user-defined function are available. If so, the function is invoked directly, saving the overhead of creating the intermediate closures.

- **Inline Substitution rule**

  \[
  \begin{align*}
  F^m x_1 x_2 \cdots x_m &= (B_1; B_2; \cdots B_m \text{ in } f) \\
  F_C(x_1, \ldots, x_n) &\rightarrow_{f_C} \{z_i = x_1; \\
  \vdots \\
  z_n = x_n; \\
  B_1; B_2; \cdots B_m \text{ in } z_j\}
  \end{align*}
  \]

  This rule avoids the overhead of the function call completely by inserting the body of the function in the calling program. Note that the \( f_C \) rule must also be included in \( R_{\text{P-TAC}} \) because, as we shall see later, not all FC's can be eliminated at compile time. FC's can also be included in the Initial-Terms of P-TAC.

- **Common Subexpression Elimination rule**

  \[
  a + b; \quad y = a + b; \\
  a + b; \quad y = x; \\
  \]

  Similar \( \text{case} \) rules exist for all primitive functions, \( PF_1, PF_2 \) and \( PF_3 \), except \( \text{Allocate}, \ FC \) and \( \text{Apply} \). As the name suggests this rule avoids recomputation of the same subexpression. Function calls cannot be eliminated because of the possibility of side-effects via \( I \)-structures.

- **Fetch Elimination rules**

  \[
  \begin{align*}
  x &= \text{Make-tuple } a \ b \\
  \text{Select } x 1 &\rightarrow_a f_e \\
  x &= \text{Make-tuple } a \ b \\
  \text{Select } x 2 &\rightarrow_b f_e \\
  x &= \text{Allocate } a \\
  \text{Length } x &\rightarrow_f n
  \end{align*}
  \]

  The above rules eliminate a run time fetch from a data structure.

- **Test Elimination rules**

  \[
  \begin{align*}
  \text{Allocate } n &\rightarrow_{te} \text{False} \\
  \text{Make-tuple } a \ b &\rightarrow_{te} \text{False}
  \end{align*}
  \]

- **Algebraic Identity rules**

  \[
  \begin{align*}
  \text{And True } x &\rightarrow_a x \\
  \text{Or False } x &\rightarrow_a x \\
  x + 0 &\rightarrow_a x \\
  x * 1 &\rightarrow_a x \\
  \vdots
  \end{align*}
  \]

  Any algebraic rule can be included as long it does not have a precondition and it does not produce a ground value on the RHS. We will explain the reasons for these restrictions in Section 6.4.

### 6.3 Confluence and Normalization of \( R_{\text{comp}} \)

Theorem 6.4 \( R_{\text{comp}} \) without the Inline Substitution rule is strongly normalizing.

**Proof:** The proof is straightforward from the following two observations:

1. Application of any rule in \( R_{\text{comp}} \) destroys a redex and does not duplicate any existing redexes.
2. Let \( n \) be the number of occurrences of primitive functions in a term \( M \). It is clear that the application of any rule of \( R_{\text{comp}} \) decreases this number by one. Consequently, the maximal number of redexes that can be reduced in a term is at most \( n \).

There are only two rules in \( R_{\text{P-TAC}} \) and \( R_{\text{comp}} \) that can increase the number of applications and consequently, redexes. These are the \( \text{Apply} \) (\( \rightarrow_{ap} \)) and \( \text{FC} \) rules, respectively. Application of \( \rightarrow_{ap} \) is automatically excluded from \( R_{\text{comp}} \) because storage allocation is excluded, and we deliberately exclude the FC rule to guarantee strong normalization. However, inline substitution is a very important optimization in the Id compiler, therefore, the Id compiler passes the burden of guaranteeing termination to the user in the following way. In Id a user declares every function definition to be “substitutable” or “not substitutable” by using a keyword (\( \text{Defsubat versus Def} \)).

Notice that because of the elimination of interfering rules, \( R_{\text{P-TAC}} \) is confluent. What we want to show is that by extending \( R_{\text{P-TAC}} \) with \( R_{\text{opt}} \), which has new interfering rules, the subcommutative property, and hence, the confluence property, still holds.

Theorem 6.5 \( R_{\text{comp}} \) is confluent.

**Proof:** The proof is similar to the one given in Section 4 so, we only sketch the basic steps.
Notice that the cse-rule interferes with itself as shown below:

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

This sort of interference is benign because \( M_1 \equiv M_2 \), and the cse-rule also interferes with \( \delta \)-rules, and conditional-rules, etc. However, all these cases are captured by the following example:

\[
\{ x = 2 + 3; \ y = 2 + 3; \ldots x \ldots y \} \\
\]

It is straightforward to see that the above diagram can be closed in one step.

### 6.4 Optimizations that Preserve only Partial Correctness

#### 6.4.1 Confluent Rules (\( R_{opt_1} \))

The reader may be wondering why the following rule was not included in \( R_{comp} \):

\[
x * 0 \longrightarrow 0 \\
\]

especially given the fact that the rule

\[
x * 1 \longrightarrow x \\
\]

was included.

The reason is that this rule can change the termination behavior of the program as illustrated by the following example:

\[
\{ y = \text{Cond} \ b \ 1 \ 2; \\
\ x = y * 0; \\
\ b = \text{Equal?} \ x \ 0 ; \\
\ \text{in 5} \} \\
\]

Without the use of rule \( \longrightarrow \mu_{0} \), the above program will produce the answer (improper-termination,5) because of a circular dependency between \( x \), \( y \) and \( b \). However, if we apply the rule \( \longrightarrow \) the answer produced will be

\[
(\text{proper-termination,5}) \\
\]

By considering minor variations of the above example, we can show that the \( \longrightarrow \) rule can also turn

\[
(\text{improper-termination,v}) \\
\]

into \( T \) or non-termination, and a non-terminating computation into anything. However, an interesting fact is that rule \( \longrightarrow \) cannot change the answer of a computation that produces an answer of the type (proper-termination,\( v \)). Furthermore, it cannot change \( v \), the value part of the answer, even when the unoptimized computation terminates with (improper-termination,\( v \)) (of course, as noted above, improper termination can turn into \( T \)). All the \( R_{opt_1} \) rules given below have this property.

- **Test Elimination rule**

\[
x = \text{Allocate} \ n \\
\text{Nil?} \ x \longrightarrow \text{False} \\
\]

- **Algebraic Identity rules**

\[
\text{And False} \ x \longrightarrow \text{False} \\
\text{Or True} \ x \longrightarrow \text{True} \\
x * 0 \longrightarrow 0 \\
x - x \longrightarrow 0 \\
\text{Equal?} \ x \ x \longrightarrow \text{True} \\
\]

Any algebraic rule that does not have a precondition can be included here.

Let us define \( R_{comp_1} \) as follows:

\[
R_{comp_1} = R_{comp} \cup \text{R_{opt_1}} \\
\]

**Lemma 6.6** \( R_{comp_1} \) is strongly normalizing and confluent.

**Proof:** Omitted. Similar to proofs given earlier.

Discussion: The confluence is quite surprising because \( R_{comp_1} \) contains the algebraic Equal?-rule, which is a non-left-linear rule. Since adding this rule to \( \lambda \)-calculus destroys confluence [10], we want to clarify why it does not cause any harm in P-TAC. Let's first recall the reduction rule for the fix-point operator \( Y \):

\[
Yf \longrightarrow f(Yf) \\
\]

we have the following reduction:

\[
\text{Equal?} \ (Yf) \ (Yf) \ Y \longrightarrow \text{Equal?} \ f(Yf) \ (Yf) \\
\text{Y-r} \ \text{desc(Eq-r)} \\
\]

Note that the descendent of the Eq-redex is not a redex any more, while in P-TAC we get

\[
\{ x = \text{Apply} \ Y \ f ; \ \longrightarrow \{ x = \text{Apply f} \ (Y \ f) ; \ \\
\ \text{in Equal?} \ x \ x \ \\
\ \text{in Equal?} \ x \ x \} \\
\text{Eq-r} \ \text{Eq-r} \\
\]

The main point to grasp is that by naming "\( Y \ f \)" the descendent of "Equal? \( x \ x \)" remains a redex, and the duplication of the computation "\( Y \ f \)" is avoided.

#### 6.4.2 Non Confluent Rules (\( R_{opt_2} \))

We now discuss some optimizations that are not confluent.

Let

\[
R_{comp_2} = R_{comp_1} \cup \text{R_{opt_2}} \\
\]

where rules in \( R_{opt_2} \) are defined as follows:

- **Fetch Elimination rule**

\[
\text{Store} \ x \ i \ z \\
\text{Select} \ x \ i \longrightarrow z \\
\]

239
Algebraic Identity rules

\[
x = n + m - \quad & \text{Less } n \ x \ \text{True} \quad & m > 0 \\
x = n + m - \quad & \text{Less } n \ x \ \text{False} \quad & m > 0 \\
x = n + m - \quad & \text{Greater } n \ x \ \text{False} \quad & m > 0 \\
x = n + m - \quad & \text{Equal? } n \ x \ \text{False} \quad & m > 0
\]

Lemma 6.7 \( R_{comp} \) is strongly normalizing.
Proof: Trivial.

We illustrate, by an example, that these transformations can create havoc in the presence of deadlocks. Consider the following program:

\[
\{ y = x + 3; \ x = y + 3; \ a = \text{Less } b \ 1; \ n \}
\]

This program will produce (improper-termination, Nothing) in \( R_{P-TAC} \). The trouble is it can produce two different answers if optimized using \( R_{comp} \) because “Less \( x \) \( y \)” represents two different overlapping redexes – one corresponding to the precondition “\( y = x + 3 \)” and the other corresponding to the precondition “\( x = y + 3 \)”.

Therefore, the above program can produce either 1 or 2 as an answer after having applied \( R_{comp} \).

The main problem with the rules of \( R_{comp} \) is that they are potentially overlapping; precondition of any rule can be satisfied in several ways. The only rule in \( R_{P-TAC} \) that had this characteristic was the I-fetch rule. However, by disallowing multiple writes in a location via the blow-up rule we were able to preserve the confluence. We can include a slightly modified version of the blow-up rule to deal with the modified version of the fetch elimination rule in \( R_{comp} \).

However, there is no easy way of dealing with the “circular dependency” problem short of doing the dataflow analysis of the programs. If we could detect all such programs then we can declare them illegal and conveniently use all the \( R_{comp} \) rules. Though algorithms for dataflow analysis are well developed, we have not yet examined them from the correctness point of view, that is, we don’t know if they detect all and nothing but deadlocked cycles in a program.

7 The Correctness of \( R_{comp} \)

The correctness of \( R_{comp} \) would be trivial to decide if all the rules in \( R_{opt} \) were “derived rules” in \( R_{P-TAC} \).

Definition 7.1 A rule \( \sigma \in R' \) is said to be a derived rule in \( ( A, R ) \) if \( \forall M \in A, M \rightarrow_A M_1 \rightarrow_A M_2 \land M_1 \rightarrow_R M_2 \).

Pictorially.

\[
\begin{array}{c}
M_0 \\
\sigma \\
M_1 \\
\downarrow \\
M_2
\end{array}
\]

Consider the following two versions of the Test elimination rule, neither of which is in \( R_{P-TAC} \):

\[
\begin{array}{c}
x = \text{Allocate } n \quad \text{Nil? } x \rightarrow \text{False} \quad \text{versus} \quad x = \text{Allocate } n \\
\text{Nil? } x \rightarrow \text{False}
\end{array}
\]

The \( \rightarrow \) rule is a derived rule in \( R_{P-TAC} \) while \( \rightarrow \) is not. This is so because “Allocate \( n \)” can always be reduced in \( R_{P-TAC} \) to get an I-structure value, and then the Nil? test can be applied to this I-structure value to get the answer False. On the other hand no rule in \( R_{P-TAC} \) matches “Allocate \( n \)” unless \( n \) is grounded, and consequently the Nil? test may not be applicable either. However, in case \( n \) is grounded, the result according to \( R_{P-TAC} \) would also be False. Thus, the difference between the two rules shows up only when we consider the application of the rules to open terms or terms where the RHS of the binding corresponding to \( n \) does not get grounded either because of non-termination or improper-termination.

The only rules in \( R_{opt} \) that are derived rules are the Test elimination rules.

To better understand the behavior of non-derived rules, consider the following example where \( M_2 \) is obtained from \( M_1 \) by applying the algebraic rule “\( x + 1 \rightarrow z \)”:

\[
M_1 \equiv \{ z = y + 1; y = z + 3; x = z + 1; \text{ in } z + 2 \}
\]

\[
M_2 \equiv \{ z = y + 1; y = z + 1; \text{ in } z + 2 \}
\]

It so happens that both \( M_1 \) and \( M_2 \) are in normal form and are not \( \alpha \)-equivalent. However, they do produce the same observable answers, i.e.,

\[
\text{Ans}(M_1) \equiv \text{Ans}(M_2) \equiv \text{(improper-termination, Nothing)}
\]

The correctness of the compiler crucially depends upon the fact that the effect of a non-derived rule cannot be “observed” by any program.

Definition 7.2 An instance of a rule is a rule obtained by substituting a ground value for a free variable in the rule.

For example, an instance of “\( \varepsilon = \text{Make-tuple } a \ b \)” is obtained by substituting a ground value for either “\( a \)” or “\( b \)” e.g., “\( \varepsilon = \text{Make-tuple } 2 \ 5 \)”.

Definition 7.3 A grounded instance of a rule is a rule obtained by substituting ground values for all the free variables in the rule.

For example, a grounded instance of “And True \( x \rightarrow x \)” is “And True False \( \rightarrow \) False”; a grounded instance of the previous Fetch Elimination rule is “\( \varepsilon = \text{Make-tuple } 2 \ 3 \)”.

Note that the instance “\( \text{Store } x \ i \ 6 \)” of the Fetch Elimination rule in \( R_{opt} \), even though it introduces a ground value on the RHS, is not a grounded instance. Furthermore, it is not a derived rule, while any grounded instance is a derived rule.
Theorem 7.4 A compiler \((D, \rightarrow_R)\) is partially correct with respect to \(\text{P-TAC}\) if \(\forall \sigma \in \mathcal{R}_c\) all the grounded instances of \(\sigma\) are derived rules in \(\text{P-TAC}\).

Proof: By Definition 6.1 a compiler, given \(\mathcal{R}_c\), transforms \(\text{ARS} \mathcal{P} = (A, \text{P-TAC})_D\) into another \(\text{ARS} \mathcal{Q} = (A, \text{P-TAC})_D'\).

We want to show that \(\forall M \in \text{Initial-Terms} \text{ of } A\), if \(M\) terminates properly then \(M'\) produces the same answer in both \(\mathcal{P}\) and \(\mathcal{Q}\). We prove the partial correctness of \(\mathcal{R}_c\) by induction on the number of reduction steps applied to \(\mathcal{P}\).

We give the proof only for the base case, that is, suppose \(\mathcal{Q}\) is obtained from \(\mathcal{P}\) in one step by applying rule \(\sigma\) of \(\mathcal{R}_c\). Thus, the only difference between \(\mathcal{P}\) and \(\mathcal{Q}\) is that \(\mathcal{Q}\) contains an "optimized" version of a function, say, \(F_i\). In the following we will write \(R\) instead of \(\text{P-TAC}\) to reduce the clutter. Suppose \(M \rightarrow M' \rightarrow M_1 \in \mathcal{P}\), where \(M'\) contains the first invocation of \(F_i\). Since \(F_i\) is first applied in \(M'\), it is possible to mimic in \(\mathcal{Q}\) the reduction sequence \(M \rightarrow M'\). Suppose \(M' \rightarrow M_2 \in \mathcal{Q}\), where \(M_1 \neq M_2\).

This means that \(\exists \sigma\)-redex in \(M_1\), say \(\rho\), such that \(M_1 \rightarrow_\sigma M_2\). Pictorially we have

\[
\begin{array}{c}
M \\
r
M' \\
\downarrow_\sigma
\end{array} 
\rightarrow
\begin{array}{c}
M_1 \\
\downarrow_\sigma
\end{array} 
\rightarrow
to
\begin{array}{c}
M_2
\end{array}
\]

We will show that if \(M_1\) terminates properly in \(\mathcal{P}\) then it's possible to close the above diagram in \(\text{P-TAC}\).

If \(M_1\) terminates properly in \(\mathcal{P}\), then all free variables of \(\rho\) get grounded; consider then the reduction sequence, \(\alpha_1 \ldots \alpha_n\), where redexes \(\alpha_1 \ldots \alpha_n\) are the redexes needed to ground all free variables of \(\rho\) (see the picture below). We consider two cases.

Case 1
Suppose redexes \(\alpha_1 \ldots \alpha_n\) do not interfere with redex \(\rho\), that is, \(\alpha_1 \ldots \alpha_n\) do not destroy expression \(\rho\) or its precondition. This means that \(M_1\) must contain the expression \(\rho'\) where \(\rho'\) is a copy of \(\rho\), which is a \(\sigma'\)-redex, where \(\sigma'\) is the grounded instance of \(\sigma\). Moreover, no rule in \(\mathcal{R}_c\) can destroy the precondition of a \(\text{P-TAC}\) redex for the following reasons:

1) the grounded instances of any rule that deletes a command can not be derived rules;
2) any rule that modifies a "Store" command has to rewrite that command into the correspondent "1-store" command.

We conclude that it is possible to apply the reduction sequence \(\alpha_1 \ldots \alpha_n\), to \(M_2\) in \(\mathcal{Q}\). We thus obtain:

\[\exists M'_2, \ M_2 \rightarrow_{\alpha_1 \ldots \alpha_n} M'_2 \text{ in } \mathcal{Q} \text{ and } M_1 \rightarrow_{\sigma'} M'_2\]

Since \(\sigma'\) is a grounded instance of \(\sigma\), \(\sigma'\) is a derived rule in \(\text{P-TAC}\) and hence \(\exists M'_3\) such that

\[
\begin{array}{c}
M \\
r_{\text{P-TAC}}
\end{array} 
\rightarrow
to
\begin{array}{c}
M'_1 \\
\downarrow_{\sigma_1 \ldots \sigma_n}
\end{array} \rightarrow
to
\begin{array}{c}
M'_3
\end{array}
\]

Proof: From the previous theorem the first condition guarantees partial correctness. What we want to show next is that \(\forall M \in A\) if \(M\) terminates properly in \(\mathcal{Q}\) then \(M\) terminates properly in \(\mathcal{P}\). The proof is similar to the previous one, therefore we will only sketch the basic steps. Suppose \(M_1 \rightarrow_\rho M_2\), if \(M_2\) terminates properly in \(\mathcal{Q}\) then \(\forall\) free variables of \(\sigma\) get grounded in \(\mathcal{Q}\). Consider the reduction sequence \(\alpha_1 \ldots \alpha_n\) that ground those variables. In both cases of non-interfering and interfering redexes it must be the case that \(\alpha_1 \ldots \alpha_n\) are applicable in \(\mathcal{P}\). If not it means that \(\sigma\)-reduction does introduce a new ground value in \(\mathcal{Q}\) and \(\sigma\) is not a derived rule, which contradicts the second condition.

What we have shown so far is that a \(\text{P-TAC}\) program \(M\) terminates properly in \(\mathcal{P}\) if \(M\) terminates properly in \(\mathcal{Q}\). We can then conclude that a variables gets grounded in \(\mathcal{P}\) if it gets grounded in \(\mathcal{Q}\), that is, no information can be deleted or created. From this we also derive that both improper-termination, \(T\) and \(\bot\) are preserved, since the same argument applies to all cases we will only analyze the case of improper-termination. We want to show that if \(M\) terminates improperly in \(\mathcal{P}\) then \(M\) terminates improperly in \(\mathcal{Q}\).

Corollary 7.7 The compiler \((D, \rightarrow_{\text{comp}})\) is totally correct with respect to \(\text{P-TAC}\).

8 Conclusions

P-TAC has already proven very useful in understanding and classifying optimizations used in the Id compiler. The only
optimization that is in use in the Id compiler but has not been discussed here is dead code elimination, which essentially deletes bindings corresponding to those variables that are not needed to produce the final answer. This optimization also requires dataflow analysis and can turn a program producing T or improper-termination into a properly terminating program. Dead code elimination can also interfere with deadlock detection and optimizations in R_{combine}.

It appears that even though ARSs are very useful to describe correctness of many optimizations they are inadequate for describing optimizations that require dataflow analysis. Hence, it may be worth extending the work in the direction that facilitates bringing in graph-theoretic results. We also think that exploring connections with the work on strictness analysis may be profitable.

Acknowledgments

Part of the motivation for this work came from observing Ken Traub’s success in modeling sharing of computations in his Functional Quads. He also provided many detailed and useful comments on several drafts of this paper. We got the idea of modeling compiler optimizations as an ARS from a talk P.L. Curien gave at M.I.T in November 1987, where he described the work of Thérèse Hardin. Unfortunately we have not been able to read the thesis [8] because it is in French!

We are grateful to Vinod Kathail for innumerable discussions about Term Rewriting Systems and the λ-calculus, and the specific issues discussed in this paper. A discussion with Albert Meyer was very helpful in clarifying the question regarding observational congruence and compiler correctness.

Many thanks to Shail Gupta, Jamey Hicks, Gary Lindstrom (Utah), R.S. Nikhil, Surya Mantha (Utah), Jon Ricke and Jonathan Young for reading the current draft of the paper and for providing insightful comments.

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 N 00014-84-K-0099 (MIT) and N 0039-88-C-0163 (Harvard).

References


