P-TAC: A Parallel Intermediate Language

Computation Structures Group Memo 295
June 1989

Zena M. Ariola
Arvind

To appear in *Preceedings of the Functional Programming Languages and Computer Architectures*, September 1989, London, UK.

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 N00014-84-K-0099 (MIT) and N0039-88-C-0163 (Harvard).

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

$$((Fa),(Fa)) \text{ and } \{x = Fa;\ \mathsf{in}\ (x,x)\}$$

which may arise as a consequence of evaluating $G\ (F\ a)$, where $G\ x = (x,x)$ and "," 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, $R_{P\text{-}TAC}$, 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 "$+\ x\ y$", 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:

$$F_1\ x_1 \ldots\ x_{m_1} = b_1$$
$$\vdots$$
$$F_n\ x_1 \ldots\ x_{m_n} = b_n$$

Legend:

PFi	*means*	Primitive Function with i arguments
UDF	*means*	User Defined Functions
SE	*means*	Simple Expression

Integer	::=	$\underline{1} \mid 2 \mid \cdots \mid \underline{n} \mid \cdots$
Boolean	::=	True | False
Variable	::=	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots$
*PF*1	::=	Nil? | Allocate | Length
*PF*2	::=	+ | − | * | ⋯ | Less | Equal? | Make-tuple | Select | Apply
*PF*3	::=	Cond
UDF	::=	$F^{n_F} \mid G^{n_G} \mid \cdots$
SE	::=	*Variable* | *UDF* | *Integer* | *Boolean* | Nil
Expression	::=	*SE* | *Block* | *PF*1 *SE* | *PF*2 *SE SE* | *PF*3 *SE SE SE*
Block	::=	{[*Statement*;]* in *Expression*}
Statement	::=	*Binding* | *Command*
Command	::=	Store *SE SE SE*
Binding	::=	*Variable* = *Expression*
Definition	::=	$UDF^n \underbrace{Variable \cdots Variable}_{n} = Block$
Program	::=	*Block*

Figure 1. The Grammar of Initial-Terms of P-TAC.

The user-defined functions $F_1 \cdots F_n$ are treated as *combinators*, that is, $FV(b_i) \subseteq \{x_1 \ldots x_{m_i}\}$ for any i. ($FV(b)$ stands for the set of free variables of b). As usual, a program M must be a "closed expression." It is customary in the implementation of functional languages to consider *combinatory normal forms* with respect to $\{F_1 \ldots F_n\}$.

We begin with a discussion of the pure functional subset of P-TAC, which simply amounts to disallowing the user to write Allocate and Store. However, it is important to emphasize that the *implementation* (the operational semantics) of functional P-TAC will make use of these primitives.

2.1 P-TAC as a Minimal Functional Language

P-TAC has higher-order functions, currying and tuples, and a block structure with the usual lexical scoping rules. As is usually the case in functional languages, the variables on the left-hand-side (LHS) of bindings in a block must be pairwise distinct. Bindings may be recursive or mutually recursive, and their order in a block is not significant. The main restriction in P-TAC is that all subexpressions must have a name. Thus, one writes the expression $F\ (a+b)\ (c+d)$ as follows:

$$\begin{aligned}&\{x = a + b;\\ &\ \ y = c + d;\\ &\ \ f = \text{Apply } F\ x;\\ &\ \ \text{in Apply } f\ y\}\end{aligned}$$

The other restriction is that primitive functions, *e.g.*, +, *, Cond, etc., are not curried. This restriction does not cause any loss of expressive power since the curried version of primitive operators can be obtained by giving a user definition, *e.g.*,

$$Plus\ x\ y = \{\text{ in } x + y\}$$

so one can use (Apply *Plus* e) instead of $((+)\ e)$.

In P-TAC, nested function definitions are not permitted. 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].

The only data structure constructor in functional P-TAC is Make-tuple, the non-strict pairing operator. The non-strictness of Make-tuple allows the specification of infinite lists, such as: $\{x = \text{Make-tuple } 1\ x;\ \text{in } x\}$.

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 [3, 16], conditionals behave differently than in other functional languages. For example, the following Id expression

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

is *not* equivalent to

$$\begin{aligned}&\{x = e_1;\\ &\ \ y = e_2;\\ &\ \ \text{in (if } p \text{ then } x \text{ else } y)\}\end{aligned}$$

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, while in the second expression both e_1 and e_2 get evaluated. In fact, (if p then e_1 else e_2) is equivalent to

$$\begin{aligned}&\{\text{Def } F\ x = e_1; && x \notin FV(e_1)\\ &\ \ \text{Def } G\ y = e_2; && y \notin FV(e_2)\\ &\ \ h = (\text{if } p \text{ then } F \text{ else } G);\\ &\ \ \text{in } h\ 0\}\end{aligned}$$

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:

$$\{z = \text{Allocate } 2;\ \text{Store } z\ 1\ x;\ \text{Store } z\ 2\ y;\ \text{in } z\}$$

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 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

$$\{x = \text{Allocate } 2;\ \text{in Apply } G\ x\}$$

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:

$$\begin{aligned} F\ a = \ \ & \{b = \text{Select } 1\ a; \\ & \ c = \text{Select } 2\ a; \\ & \ \text{Store } b\ 1\ 100; \\ & \ \text{Store } c\ 1\ 200; \\ & \ \text{in } 0\} \end{aligned}$$

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: $(F\ N_1) \neq (F\ N_2) \implies N_1 \neq N_2$.

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 xs is a list, then so is (Make-tuple x xs) [1] 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 "difference list" in logic programming. An open list is a list of I-structures where each 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:

[1] This definition of list is not quite correct in Id because Id is a strongly-typed language with Milner-style type deduction scheme. Thus, in Id, a list is an algebraic type with two disjuncts, while Make-tuple has the type $T1 \times T2$. In this paper we have ignored type issues in P-TAC.

- (Allocate 2) is an open list.
- If xs is an open list then so is (cons-open x xs) for any x, where cons-open is defined as follows:

$$\text{cons-open } x\ xs \ = \ \{z = \text{Allocate } 2;\ \text{Store } z\ 2\ xs;\ \text{Store } xs\ 1\ x;\ \text{in } z\}$$

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 $\langle A, \longrightarrow_R\rangle$ where A is a set of terms and $\longrightarrow_R$ is a binary relation on A. The relation $\longrightarrow_R$ 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 *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 *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. We will designate the terms generated by Figure 1 as ***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 $\langle A, \longrightarrow_{R_{P\text{-}TAC}}\ \rangle_D$.

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 ***Locations***. An ***I-structure*** of n elements is represented by n ***Locations***. The ***only*** permissible operations on locations are "I-fetch l", for reading the contents of location l, and "I-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 *Binding* or in the formal list of a *Definition*. Furthermore, locations are ***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 α-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

Legend:

PFi	*means*	Primitive Function with i arguments
UDF	*means*	User Defined Functions
SE	*means*	Simple Expressions
GV	*means*	Ground Values
L	*means*	Locations
FC	*means*	Fast call, see Section 6.2

Integer	::=	$1 \mid 2 \mid \cdots \mid \underline{n} \mid \cdots$
Boolean	::=	True | False
Variable	::=	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots$
L	::=	$l0 \mid l1 \mid \cdots$
PF1	::=	Nil? | Allocate | Length | FC
PF2	::=	+ | − | * | ⋯ | Less | Equal? | Make-tuple | Select | Apply | Make-closure
PF3	::=	Cond
UDF	::=	$F^{n_F} \mid G^{n_G} \mid \cdots$
SE	::=	*Variable* | *UDF* | *GV*
GV	::=	*Integer* | *Boolean* | *Closure* | *I-structure* | Error | T
I-structure	::=	$\langle$I-structure, $\underline{n}$, $\underbrace{L,\ L, \cdots L}_{n}\rangle$ | Nil
Closure	::=	$\langle$Closure, UDF^n, $\underline{m}$, *I-structure*$\rangle$ $(\underline{m} \leq \underline{n})$
Expression	::=	*SE* | *Block* | *PF1 SE* | *PF2 SE SE* | *PF3 SE SE SE* | I-fetch *L*
Block	::=	{[*Statement*;]* in *Expression*}
Statement	::=	*Binding* | *Command* | Store-Error
Command	::=	Store *SE SE SE* | I-store *L SE*
Binding	::=	*Variable* = *Expression*

Figure 2. The Grammar of Terms of P-TAC.

not allow any confusion between the name of a location and its contents. Thus, while a binding like "$x = y$" 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: "{ x = I-fetch l_i; y = I-fetch l_j; in (Equal? x y) }".

Locations are also 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^2 is applied to e, 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. $\{x = 8;\ \text{in}\ \{y = x;\ \text{in}\ x + y\}\}$
2. $\{x = 8;\ y = x;\ \text{in}\ x + y\}$
3. $\{x = 8;\ \text{in}\ x + x\}$
4. $\{\ \text{in}\ 8 + 8\}$

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:

$$\begin{array}{l} \{B_1;\ B_2;\ \cdots \\ \ x = \{BB_1;\ BB_2;\ \cdots \\ \qquad\quad \text{in}\ EE\} \\ \cdots;\ B_n\ ; \\ \text{in}\ E\} \end{array} \xrightarrow[blk_1]{} \begin{array}{l} \{B_1;\ B_2;\ \cdots \\ x = EE'; \\ BB'_1;\ BB'_2;\ \cdots \\ \cdots;\ B_n\ ; \\ \text{in}\ E\} \end{array}$$

$$\begin{array}{l} \{B_1; B_2; \cdots \\ \ \text{in} \\ \qquad \{BB_1;\ BB_2;\ \cdots \\ \qquad\quad \text{in}\ EE\}\ \} \end{array} \xrightarrow[blk_2]{} \begin{array}{l} \{B_1; B_2; \cdots \\ BB'_1;\ BB'_2;\ \cdots \\ \quad \text{in}\ EE'\} \end{array}$$

where BB'_i and EE'_i indicate α-renaming of variables in BB_i and EE'_i, 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.

2. For each binding of the form "$y = x$" in M, where x and y are distinct variables, replace each occurrence of "y" in M by "x" and then erase the binding "$y = x$" from M.

3. 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.

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.

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 locations and bound variables.*

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 α-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 [5]. ■

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

1. Terms $\{\ \text{in}\ x + 1\}$, $\{y = 1;\ \text{in}\ x + y\}$ and $\{z = 2;\ y = 1;\ \text{in}\ x + y\}$ have the same canonical form. However, this canonical form is different from the canonical form of $\{z = \text{Apply}\ F\ 2;\ y = 1;\ \text{in}\ x + y\}$. 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 $\{\text{I-store } l\ v;\ \text{in } 5\}$ and $\{\text{I-store } l\ v;\ x = 5;\ \text{in } x\}$ have the same canonical form which is different from $\{\ \text{in } 5\}$.

3. The following two terms are α-equivalent:

$$\begin{array}{lcl} \{\text{I-store } l_0\ v_0; & \text{and} & \{\text{I-store } l_0'\ v_0; \\ \ \text{I-store } l_1\ v_1; & & \ \text{I-store } l_1'\ v_1; \\ \ \text{in } \langle\text{I-structure}, 2,\ l_0,\ l_1\rangle\} & & \ \text{in } \langle\text{I-structure}, 2,\ l_0',\ l_1'\rangle\} \end{array}$$

4. The following two terms are not α-equivalent:

$$\begin{array}{l} \{\text{I-store } l_0\ \langle\text{I-structure}, 2,\ l_2,\ l_3\rangle; \\ \ \text{I-store } l_1\ \langle\text{I-structure}, 2,\ l_2,\ l_3\rangle; \\ \ \text{in } \langle\text{I-structure}, 2,\ l_0,\ l_1\rangle\} \end{array}$$

and

$$\begin{array}{l} \{\text{I-store } l_0'\ \langle\text{I-structure}, 2,\ l_2',\ l_3'\rangle \\ \ \text{I-store } l_1'\ \langle\text{I-structure}, 2,\ l_4',\ l_5'\rangle; \\ \ \text{in } \langle\text{I-structure}, 2,\ l_0',\ l_1'\rangle\} \end{array}$$

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*:

$$\frac{\text{I-store } l\ 5}{\text{I-fetch } l\ \longrightarrow\ 5}$$

where the command I-store l 5 over the line denotes the precondition. The above rule says that the subterm I-fetch l of M can be rewritten to 5 if the command I-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** *iff 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 "2 + 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 $\langle A, R\rangle$, if M,$N \in A$ and $\sigma \in R$, then M reduces to N in one step ($M \longrightarrow_R N$), iff $\exists$ a context $C[\,]$, and σ-redex ρ , such that, $M \equiv C[\rho]$, $N \equiv C[\rho']$, and $\rho \xrightarrow[\sigma]{} \rho'$. The transitive reflexive closure of $\longrightarrow_R$ is written as $\xrightarrow[R]{}\!\!\!\!\!\rightarrow$.*

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

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

We now present the set of rewrite rules, $R_{P\text{-}TAC}$, for defining the ARS for P-TAC. In the following n and $\underline{n}$ 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**

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

- **Conditional rules**

$$\begin{array}{lcl} \text{Cond True } x\ y & \xrightarrow[cond]{} & x \\ \text{Cond False } x\ y & \xrightarrow[cond]{} & y \end{array}$$

- **Data Constructor rules**

$$\begin{array}{lcl} \text{Make–tuple } x\ y & \xrightarrow[mkt]{} & \{z = \text{Allocate } 2;\ \text{Store } z\ 1\ x; \\ & & \ \text{Store } z\ 2\ y;\ \text{in } z\ \} \end{array}$$

- **I-structure rules**

$$\text{Allocate } \underline{n} \xrightarrow[alc]{} \langle\text{I-structure}, \underline{n}, l_0 \cdots l_{n-1}\rangle$$
$$(l_0 \cdots l_{n-1} \text{ are globally unique new locations})$$

$$\text{Length } \langle\text{I-structure}, \underline{n},\ l_0 \cdots l_{n-1}\rangle \xrightarrow[len]{} \underline{n}$$

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

$$\text{Select } \langle\text{I-structure}, \underline{n},\ l_0 \cdots l_{n-1}\rangle\ \underline{i} \xrightarrow[sel]{} \text{I-fetch } l_i \quad (0 \leq \underline{i} \leq \underline{n}\text{-}1)$$

$$\text{Select } \langle\text{I-structure}, \underline{n},\ l_0 \cdots l_{n-1}\rangle\ \underline{i} \xrightarrow[sel]{} \text{Error} \quad (\underline{i} < 0 \vee \underline{i} > \underline{n}\text{-}1)$$

$$\text{Store } \langle\text{I-structure}, \underline{n},\ l_0 \cdots l_{n-1}\rangle\ \underline{i}\ \ y \xrightarrow[sto]{} \text{I-store } l_i\ y \quad (0 \leq \underline{i} \leq \underline{n}\text{-}1)$$

$$\text{Store } \langle\text{I-structure}, \underline{n},\ l_0 \cdots l_{n-1}\rangle\ \underline{i}\ \ y \xrightarrow[sto]{} \text{Store-Error} \quad (\underline{i} < 0 \vee \underline{i} > \underline{n}\text{-}1)$$

$$\frac{\text{I-store } l\ v}{\text{I-fetch } l\ \xrightarrow[I\text{-}f]{}\ v} \qquad \text{where } v \text{ is a } \textit{ground value}$$

$$\frac{N \equiv \text{I-store } l\ v\ \wedge\ N' \equiv \text{I-store } l\ v'\ \wedge \quad N,\ N' \text{ are distinct subterms of program } M}{M \xrightarrow[blw]{} \top}$$

- **Application rules**

$$\text{Apply (Closure}, F^n, \underline{m}, isv)\ x \underset{ap_1}{\longrightarrow}$$
$$\{chain = \text{Make-tuple}\ x\ isv;$$
$$\text{in Make-closure (Closure}, F^n, \underline{m}, isv)\ chain\ \}$$
$$\text{where}\ \underline{m} > 1$$

$$\frac{F^n\ z_1\ z_2 \cdots z_n = \{B_1;\ B_2; \cdots B_m;\ \text{in}\ z_f\}\ \in D}{\begin{array}{lll}\text{Apply (Closure},\ F^n,\ 1,\ isv)\ x \underset{ap_2}{\longrightarrow} & \{z'_n & = \ x; \\ & z'_{n-1} & = \text{Select}\ isv\ 1; \\ & rest_1 & = \text{Select}\ isv\ 2; \\ & z'_{n-2} & = \text{Select}\ rest_1\ 1; \\ & & \vdots \\ & z'_1 & = \text{Select}\ rest_{n-2}\ 1; \\ & B'_1;\ B'_2; \cdots B'_m; & \\ & \text{in}\ z'_f\} & \end{array}}$$

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**

$$F^n \underset{cl_1}{\longrightarrow} \langle\text{Closure},\ F^n, \underline{n}, \text{Nil}\rangle$$

$$\text{Make-closure}\ \langle\text{Closure}, F^n, \underline{m}, isv\rangle\ isv' \underset{cl_2}{\longrightarrow}$$
$$\langle\text{Closure}, F^n, \underline{m\text{-}1}, isv'\rangle$$

where isv, isv' are *I-structures* values

Discussion:

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

2. According to the *blow-up* rule, $\underset{blw}{\longrightarrow}$, *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 $\top$.

   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 $\{i = \text{Apply}\ F\ a;\ \text{Store}\ x\ i\ v; \cdots\}$, it is not possible, in general, to determine if a program will go to $\top$ without executing it. The confluence of P-TAC will ensure that if any terminating reduction sequence goes to the $\top$, then so will all the terminating ones.

3. Note that the program will go to $\top$ 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 Store only arises indirectly through 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 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:

   $$\{\text{I-store}\ l_0\ 1;$$
   $$\text{I-store}\ l_1\ \langle\text{I-structure}, 2,\ l_0,\ l_1\rangle;$$
   $$\text{in}\ \langle\text{I-structure}, 2,\ l_0,\ l_1\rangle\}$$

   where no further reductions are possible.

6. We have included "errors" in case of I-structures rules because these errors can not 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 $\forall$ terms M, M_1 and M_2:*

$$M \longrightarrow M_1\ \wedge\ M \longrightarrow M_2\ \Longrightarrow\ \exists\ M_3\ \textit{such that}$$

$$M_1 \xrightarrow{0/1} M_3\ \wedge\ M_2 \xrightarrow{0/1} M_3$$

where $\xrightarrow{0/1}$ 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 $R_{P\text{-}TAC}$. 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, $\underset{blw}{\longrightarrow}$, and the I-fetch rule, $\underset{I\text{-}f}{\longrightarrow}$, are the only interfering rules in P-TAC.*

The *blow-up* rule can invalidate the precondition of any rule. The *I-fetch* rule interferes with itself as shown by the example below:

$$\{\text{I-store } l\ 1;\quad \text{I-store } l\ 2;\quad x = \underbrace{\underbrace{\text{I-fetch } l;}_{\rho_1}}_{\rho_2}\quad \text{in } x\}$$

The subterm I-fetch l matches the (LHS) of the *I-fetch* rule. However, the precondition for the rule can be satisfied by either the subterm (I-store l 1) or by the subterm (I-store l 2). Thus, redexes ρ_1 and ρ_2, shown above, do overlap and, consequently, the *I-fetch* rule interferes with itself.

Lemma 4.5 *P-TAC is subcommutative.*

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

$$M \xrightarrow[i]{} M_1 \ \wedge\ M \xrightarrow[j]{} M_2 \implies \exists M_3 \text{ such that}$$

$$M_1 \xrightarrow{0/1} M_3 \ \wedge\ M_2 \xrightarrow{0/1} M_3$$

1. (*Non-interfering rules*) Suppose the i-redex does not interfere with the j-redex. By inspecting the non-interfering rules of $R_{P\text{-}TAC}$, 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 $\xrightarrow[blw]{}$ and j-rule be different from $\xrightarrow[blw]{}$. Since only the ***blow-up*** rule can destroy the precondition for the *blw*-redex, in the following diagram M_2 is still a *blw*-redex.

$$\begin{tikzcd} & M \arrow[dl, "blw"'] \arrow[dr, "j"] & \\ \top & & M_2 \arrow[ll, "blw"] \end{tikzcd}$$

   2.2 Let both i-rule and j-rule be $\xrightarrow[blw]{}$. Then,

$$\begin{tikzcd} M \arrow[d, bend right] \arrow[d, bend left] \\ \top \end{tikzcd}$$

   2.3 Let both i-rule and j-rule be $\xrightarrow[I\text{-}f]{}$. Then M is a *blw*-redex which can not be destroyed by the *I-fetch* reductions. Hence M_1 and M_2 are also *blw*-redexes.

$$\begin{tikzcd} & M \arrow[dl, "I\text{-}f"'] \arrow[dr, "I\text{-}f"] & \\ M_1 \arrow[dr] & & M_2 \arrow[dl] \\ & \top & \end{tikzcd}$$

■

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 v, 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:

$$M_1 \equiv \{\text{ in } 9\} \quad \text{vs} \quad M_2 \equiv \{\text{I-store } l_0\ 4;\ \text{I-store } l_1\ 5;\ \text{ in } 9\}$$

Though these programs are not α-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, $\mathcal{E}(M)$, is the term obtained by erasing all commands of the form* I-store l v, *where v is a ground value, from M.*

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

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

- *if $\mathcal{E}(N)$ is $\top$ then $\top$;*
- *if $\mathcal{E}(N)$ is of the form* { in v} *and*
  - *if v is either integer, boolean or* Error, *then* $\langle$proper-termination, $v\rangle$,
  - *if v is a closure value, then* $\langle$proper-termination, Closure$\rangle$,
  - *if v is an I-structure value, then* $\langle$proper-termination, $\langle$I-structure, $\underline{n}\rangle\rangle$,

  where proper-termination, Closure, *and* I-structure *are reserved constants and $\underline{n}$ denotes the length of the I-structure value;*
- *if $\mathcal{E}(N)$ is of the form $\{B_1;\ B_2; \cdots$ in $v\}$ and*
  - *if v is either integer, boolean or* Error, *then* $\langle$improper-termination, $v\rangle$,
  - *if v is a closure value, then* $\langle$improper-termination, Closure$\rangle$,
  - *if v is an I-structure value, then* $\langle$improper-termination, $\langle$I-structure, $\underline{n}\rangle\rangle$,
  - *else* $\langle$improper-termination, Nothing$\rangle$.

  where improper-termination *and* Nothing *are reserved constants.*

According to the above definition $\mathcal{A}ns(M_1) \equiv \mathcal{A}ns(M_2)$, 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:

$$\begin{array}{llll}
M_3 \equiv \{\text{I-store } l_0\ 4; & & \text{and} & M_4 \equiv \{\text{I-store } l'_0\ 5; \\
\quad \text{I-store } l_1\ 5; & & & \quad \text{I-store } l'_1\ 4; \\
\quad \text{in} & & & \quad \text{in} \\
\langle \text{I-structure}, 2, l_0, l_1 \rangle\} & & & \langle \text{I-structure}, 2, l'_0, l'_1 \rangle\}
\end{array}$$

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 equivalence between programs [11]:

Definition 5.3 *M, $N \in A$ are said to be* **observationally congruent** *if $\forall$ user definable contexts $C[\ ]$*

$$\mathcal{A}ns(C[M]) \equiv \mathcal{A}ns(C[N]).$$

According to this definition programs M_1 and M_2 are *observationally congruent*, while M_3 and M_4 are not. As Meyer points out in [11], we could have limited the notion of observable 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 machine 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 program graphs. Interestingly enough, the compiler optimizations on program graphs can be expressed in terms of an abstract reduction system $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$, where $\mathcal{D}$ is the set of all possible user-defined functions, that is, all the terms generated 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 functions. Thus, a compiler may be defined as follows:

Definition 6.1 *A* **compiler** *is an ARS $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$ which given $D \subseteq \mathcal{D}$ transforms the ARS $\mathcal{P} = \langle A, R \rangle_D$ into another ARS $\mathcal{Q} = \langle A, R \rangle_{D'}$, such that,*

1. *$\forall\ F,\ F\,x_1\,x_2 \cdots x_n = e \in D \implies$ $\exists\ F,\ F\,x_1\,x_2 \cdots x_n = e' \in D'$ and $e \xrightarrow[R_c]{} e'$;*
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's in normal forms with respect to R_c. If R_c is *confluent* then it is guaranteed that all terminating reduction sequences 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 compiler optimizations, is not a serious limitation, because M can be enclosed inside a function body and then optimized.

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 *α-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 definition 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 transformations are correct.

Definition 6.2 *A compiler $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$ which given $D \subseteq \mathcal{D}$ transforms the ARS $\mathcal{P} = \langle A, R_{P\text{-}TAC} \rangle_D$ into another ARS $\mathcal{Q} = \langle A, R_{P\text{-}TAC} \rangle_{D'}$ is* **totally correct** *with respect to $R_{P\text{-}TAC}$ if $\forall\ M \in$ Initial-Terms of A then $\mathcal{A}ns_{\mathcal{P}}(M) \equiv \mathcal{A}ns_{\mathcal{Q}}(M)$.*

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. However, the cases when the original program does not terminate 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 transformed program.*

Definition 6.3 *A compiler $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$ which given $D \subseteq \mathcal{D}$ transforms the ARS $\mathcal{P} = \langle A, R_{P\text{-}TAC} \rangle_D$ into another ARS $\mathcal{Q} = \langle A, R_{P\text{-}TAC} \rangle_{D'}$ is* **partially correct** *with respect to $R_{P\text{-}TAC}$ if $\forall\ M \in$ Initial-Terms of A, if M terminates properly in $\mathcal{P}$ then $\mathcal{A}ns_{\mathcal{P}}(M) \equiv \mathcal{A}ns_{\mathcal{Q}}(M)$.*

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

We begin by examining rules of $R_{P\text{-}TAC}$ that may be applicable in the compilation process. We will call such rules as $R^{-}_{P\text{-}TAC}$. Since a compiler does not allocate storage, it cannot execute Allocate or most other *I-structure* rules. The Apply 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, $R^{-}_{P\text{-}TAC}$ includes the following rules:

- **δ 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_{opt}, to transform a *Program* into an even more efficient version. These optimization rules are usually *not included in* $R_{P\text{-}TAC}$ because it is not clear how the preconditions of some of these rules can be checked efficiently at run time. Let

$$R_{comp} = R^{-}_{P\text{-}TAC} \cup R_{opt}$$

A description of R_{opt} follows:

- **Arity Detection rule**

$$\frac{\begin{array}{ll} f_1 & = \text{Apply } F^n\ x_1; \\ f_2 & = \text{Apply } f_1\ x_2; \\ \vdots & \\ f_{n-1} & = \text{Apply } f_{n-2}\ x_{n-1}; \end{array}}{\text{Apply } f_n\ x_n \xrightarrow[ad]{} \text{FC}_F(x_1, \cdots, x_n)}$$

where it is presumed that there is a combinator FC_F (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**

$$\frac{F^n\ z_1\ z_2 \cdots z_n = \{B_1;\ B_2; \cdots B_m \text{ in } z_f\} \in D}{\text{FC}_F(x_1, \cdots, x_n) \xrightarrow[fc]{} \begin{array}{l} \{z_1' = x_1; \\ \vdots \\ z_n' = x_n; \\ B_1';\ B_2'; \cdots B_m'; \\ \text{in } z_f'\} \end{array}}$$

This rule avoids the overhead of the function call completely by inserting the body of the function in the calling program. Note that the fc-rule must also be included in $R_{P\text{-}TAC}$ 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**

$$\frac{x = a + b;\ \ y = a + b;}{y = a + b; \xrightarrow[cse]{} y = x;}$$

Similar *cse* rules exist for all primitive functions, *PF*1, *PF*2 and *PF*3, except *Allocate*, *FC* and *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**

$$\frac{x = \text{ Make-tuple } a\ b}{\text{Select } x\ 1 \xrightarrow[fe]{} a}$$

$$\frac{x = \text{ Make-tuple } a\ b}{\text{Select } x\ 2 \xrightarrow[fe]{} b}$$

$$\frac{x = \text{Allocate } n}{\text{Length } x \xrightarrow[fe]{} n}$$

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

- **Test Elimination rules**

$$\frac{x = \text{Allocate } \underline{n}}{\text{Nil? } x \xrightarrow[te]{} \text{False}}$$

$$\frac{x = \text{Make-tuple } a\ b}{\text{Nil? } x \xrightarrow[te]{} \text{False}}$$

- **Algebraic Identity rules**

$$\text{And True } x \xrightarrow[alg]{} x$$

$$\text{Or False } x \xrightarrow[alg]{} x$$

$$x + 0 \xrightarrow[alg]{} x$$

$$x * 1 \xrightarrow[alg]{} x$$

$$\vdots$$

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_{comp}

Theorem 6.4 *R_{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_{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_{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 $R_{P\text{-}TAC}$ and R_{comp} that can increase the number of applications and consequently, redexes. These are the Apply ($\xrightarrow[ap_2]{}$) and FC rules, respectively. Application of $\xrightarrow[ap_2]{}$ is automatically excluded from R_{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 (*Defsubst* versus *Def*).

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

Theorem 6.5 *R_{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:

$$\begin{tikzcd}[column sep=small] & \{x = \underbrace{a+b}_{\rho_1};\ y = \underbrace{a+b}_{\rho_2}; \ldots x \ldots y\} \arrow[dl, "\rho_1"'] \arrow[dr, "\rho_2"] & \\ M_1 \equiv \{y = a+b; \cdots y \ldots y\} & & M_2 \equiv \{x = a+b; \ldots x \ldots x\} \end{tikzcd}$$

This sort of interference is benign because $M_1 \equiv_\alpha M_2$.

cse-rule also interferes with δ-rules, and *conditional*-rules, etc. However, all these cases are captured by the following example:

$$\begin{tikzcd}[column sep=small] & \{x = \underbrace{2+3}_{cse_r};\ y = \underbrace{2+3}_{\delta-r}; \cdots x \cdots y\} \arrow[dl, "cse"'] \arrow[dr, "\delta"] & \\ M_1 \equiv \{y = 2+3; \cdots y \cdots y\} & & M_2 \equiv \{x = 2+3;\ \cdots x \cdots 5\} \end{tikzcd}$$

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}

$$x * 0 \xrightarrow[mul_0]{} 0$$

especially given the fact that the rule

$$x * 1 \xrightarrow[mul_1]{} x$$

was included.

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

$$\begin{array}{l} \{y = \text{Cond } b\ 1\ 2; \\ \ x = y * 0; \\ \ b = \text{Equal? } x\ 0\ ; \\ \ \ \text{in } 5\} \end{array}$$

Without the use of rule $\xrightarrow[mul_0]{}$, 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 $\xrightarrow[mul_0]{}$ the answer produced will be

⟨proper-termination, 5⟩

By considering minor variations of the above example, we can show that the $\xrightarrow[mul_0]{}$ rule can also turn

⟨improper-termination, v⟩

into $\top$ or non-termination, and a non-terminating computation into anything. However, an interesting fact is that *rule* $\xrightarrow[mul_0]{}$ *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 $\top$). All the R_{opt_1} rules given below have this property.

- **Test Elimination rule**

$$\frac{x = \text{Allocate } n}{\text{Nil? } x \xrightarrow[te_1]{} \text{False}}$$

- **Algebraic Identity rules**

$$\begin{array}{rcl} \text{And False } x & \xrightarrow[alg_1]{} & \text{False} \\ \text{Or True } x & \xrightarrow[alg_1]{} & \text{True} \\ x * 0 & \xrightarrow[alg_1]{} & 0 \\ x - x & \xrightarrow[alg_1]{} & 0 \\ \text{Equal? } x\ x & \xrightarrow[alg_1]{} & \text{True} \\ & \vdots & \end{array}$$

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

Let us define R_{comp_1} as follows:

$$R_{comp_1} = R_{comp} \cup R_{opt_1}$$

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 λ-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:

$$Y f \xrightarrow[Y]{} f(Y f)$$

we have the following reduction:

$$\underbrace{Equal?\ \underbrace{(Y f)}_{Y-r}\ (Y f)}_{Eq-r} \xrightarrow[Y]{} \underbrace{Equal?\ f(Y f)\ (Y f)}_{desc(Eq-r)}$$

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

$$\begin{array}{lcl} \{x = \underbrace{\text{Apply } Y\ f}_{Y-r}; & \xrightarrow[Y]{} & \{x = \text{Apply } f\ (Y\ f); \\ \ \text{in } \underbrace{Equal?\ x\ x}_{Eq-r}\} & & \ \text{in } \underbrace{Equal?\ x\ x}_{Eq-r}\} \end{array}$$

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

$$R_{comp_2} = R_{comp_1} \cup R_{opt_2}$$

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

- **Fetch Elimination rule**

$$\frac{\text{Store } x\ i\ z}{\text{Select } x\ i \xrightarrow[fe_2]{} z}$$

- **Algebraic Identity rules**

$$\frac{x = n \ + \ \underline{m}}{\mathsf{Less}\ n\ x \ \xrightarrow[alg_2]{} \ \mathsf{True}} \qquad \underline{m} > 0$$

$$\frac{x = n \ + \ \underline{m}}{\mathsf{Less}\ x\ n \ \xrightarrow[alg_2]{} \ \mathsf{False}} \qquad \underline{m} > 0$$

$$\frac{x = n \ + \ \underline{m}}{\mathsf{Greater}\ n\ x \ \xrightarrow[alg_2]{} \ \mathsf{False}} \qquad \underline{m} > 0$$

$$\frac{x = n \ + \ \underline{m}}{\mathsf{Equal?}\ n\ x \ \xrightarrow[alg_2]{} \ \mathsf{False}} \qquad \underline{m} > 0$$

$$\vdots$$

Lemma 6.7 *R_{comp_2} 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:

$$\{y = x + 3;\ x = y + 3;\ b = \mathsf{Less}\ x\ y\ ;\ a = \mathsf{Cond}\ b\ 1\ 2;\ \mathsf{in}\ a\}$$

This program will produce ⟨improper-termination, Nothing⟩ in $R_{P\text{-}TAC}$. The trouble is it can produce two different answers if optimized using R_{comp_2} 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_2}.

The main problem with the rules of R_{comp_2} is that they are potentially overlapping; precondition of any rule can be satisfied in several ways. The only rule in $R_{P\text{-}TAC}$ 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_2}.

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_2} 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 $R_{P\text{-}TAC}$.

Definition 7.1 *A rule $\sigma \in R'$ is said to be a* **derived** *rule in $\langle A, R \rangle$ if $\forall\ M \in A,\ M \xrightarrow[\sigma]{} M_1 \Longrightarrow \exists\ M_2,\ M \twoheadrightarrow_R M_2 \wedge M_1 \twoheadrightarrow_R M_2$.*

Pictorially:

$$\begin{tikzcd} M \arrow[r, "\sigma"'] \arrow[dr, two heads] & M_1 \arrow[d, two heads] \\ & M_2 \end{tikzcd}$$

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

$$\frac{x = \mathsf{Allocate}\ \underline{n}}{\mathsf{Nil?}\ x \xrightarrow[te]{} \mathsf{False}} \quad \text{versus} \quad \frac{x = \mathsf{Allocate}\ n}{\mathsf{Nil?}\ x \xrightarrow[te_1]{} \mathsf{False}}$$

The $\xrightarrow[te]{}$ rule is a derived rule in $R_{P\text{-}TAC}$ while $\xrightarrow[te_1]{}$ is not. This is so because "Allocate $\underline{n}$" can always be reduced in $R_{P\text{-}TAC}$ 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 $R_{P\text{-}TAC}$ 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 $R_{P\text{-}TAC}$ 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 "$x * 1 \longrightarrow x$":

$$M_1 \equiv \{x = y + 1;\ y = x + 1; z = x * 1;\ \ \mathsf{in}\ z + 2\}$$

$$M_2 \equiv \{x = y + 1;\ y = x + 1;\ \ \mathsf{in}\ x + 2\}$$

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

$$Ans(M_1) \equiv Ans(M_2) \equiv \langle \text{improper-termination}, \text{Nothing} \rangle$$

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 "$\frac{x = \mathsf{Make\text{-}tuple}\ a\ b}{\mathsf{Select}\ x\ 1 \longrightarrow a}$" is obtained by substituting a ground value for either "a" or "b", *e.g.*, "$\frac{x = \mathsf{Make\text{-}tuple}\ 2\ b}{\mathsf{Select}\ x\ 1 \longrightarrow 2}$".

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 $x \longrightarrow x$" is "And True False $\longrightarrow$ False"; a grounded instance of the previous Fetch Elimination rule is "$\frac{x = \mathsf{Make\text{-}tuple}\ 2\ 3}{\mathsf{Select}\ x\ 1 \longrightarrow 2}$".

Note that the instance "$\frac{\mathsf{Store}\ x\ i\ 5}{\mathsf{Select}\ x\ i \longrightarrow 5}$" of the Fetch Elimination rule in R_{opt_2}, 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 $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$ is partially correct with respect to $R_{P\text{-}TAC}$ if $\forall\ \sigma \in R_c$ all the grounded instances of σ are derived rules in $R_{P\text{-}TAC}$.*

Proof: By Definition 6.1 a compiler, given R_c, transforms ARS $\mathcal{P} = \langle A, R_{P\text{-}TAC} \rangle_{\mathcal{D}}$ into another ARS $\mathcal{Q} = \langle A, R_{P\text{-}TAC} \rangle_{\mathcal{D}'}$. We want to show that $\forall$ M $\in$ *Initial-Terms* 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 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 $\sigma \in R_c$. 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 $R_{P\text{-}TAC}$ to reduce the clutter. Suppose $M \underset{R}{\twoheadrightarrow} M' \underset{ap_2}{\longrightarrow} M_1$ 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 $M \underset{R}{\twoheadrightarrow} M'$. Suppose $M' \underset{ap_2}{\longrightarrow} M_2$ in $\mathcal{Q}$, where $M_1 \not\equiv_\alpha M_2$. This means that $\exists$ a σ-redex in M_1, say ρ, such that $M_1 \xrightarrow[\sigma]{\rho} M_2$. Pictorially we have

$$\begin{tikzcd}
 & & M_1 \arrow[r] \arrow[dd, "\sigma"] & {} \arrow[r, dashed] & \cdots \\
M \arrow[r, two heads, "R"'] & M' \arrow[ur] \arrow[dr] & & & \\
 & & M_2 \arrow[r] & {} \arrow[r] & \cdots
\end{tikzcd}$$

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

If M_1 terminates properly in $\mathcal{P}$, then all free variables of ρ 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 ρ (see the picture below). We consider two cases.

Case 1

Suppose redexes $\alpha_1 \ldots \alpha_n$ do not interfere with redex ρ, that is, $\alpha_1 \ldots \alpha_n$ do not destroy expression ρ or its precondition. This means that M_1' must contain the expression ρ' where ρ' is a copy of ρ, which is a σ'-redex, where σ' is the grounded instance of σ. Moreover, no rule in R_c can destroy the precondition of a 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 "I-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:

$$\exists M_2',\ \ M_2 \overset{\alpha_1 \ldots \alpha_n}{\twoheadrightarrow} M_2' \text{ in } \mathcal{Q} \text{ and } M_1' \underset{\sigma'}{\longrightarrow} M_2'$$

Since σ' is a grounded instance of σ, σ' is a derived rule in $R_{P\text{-}TAC}$ and hence $\exists$ M_3 such that

$$\begin{tikzcd}
 & & M_1 \arrow[r, "\alpha_1"] \arrow[dd, "\sigma"] & \cdots \arrow[r, "\alpha_n"] & M_1' \arrow[dd, "\sigma'"] \arrow[dr, two heads] & \\
M \arrow[r, two heads, "R_{P\text{-}TAC}"'] & M' \arrow[ur] \arrow[dr] & & & & M_3 \\
 & & M_2 \arrow[r, "\alpha_1"] & \cdots \arrow[r, "\alpha_n"] & M_2' \arrow[ur, two heads] &
\end{tikzcd}$$

Case 2

Suppose redexes $\alpha_1 \ldots \alpha_{j-1}$ do not interfere with ρ and α_j does. (As an example of this situation consider $M_1 \equiv \{x =$ Make-tuple 1 y; $y =$ Select x 1; in $y\}$ and $M_2 \equiv \{x =$ Make-tuple 1 y; $y = 1$; in $y\}$. M_2 is the optimized version of M_1 obtained by applying the fe-rule.) Furthermore, let α_j be the redex needed to ground, say variable "y", of rule σ. Thus if, $M_1 \overset{\alpha_1 \ldots \alpha_{j-1}}{\twoheadrightarrow} M_1'$ and $M_2 \overset{\alpha_1 \ldots \alpha_{j-1}}{\twoheadrightarrow} M_2'$ then M_1' still contains a copy of ρ, which is a σ''-redex, where σ'' is an instance of σ in which variable "y" is not grounded. However, an attempt to ground "y" by applying redex α_j, either the precondition of ρ or ρ itself gets destroyed due to interference. Clearly due to proper-termination in $\mathcal{P}$ it must be the case that both the precondition of ρ and ρ itself are not strict on y. Notice that, if σ'' were grounded then there would exists reduction sequences $\beta_1 \ldots \beta_n$, $\gamma_1 \ldots \gamma_m$, in $\mathcal{P}$ and $\mathcal{Q}$ respectively such that

$$M_1' \overset{\beta_1 \ldots \beta_n}{\twoheadrightarrow} M_3 \ \ \wedge \ \ M_2' \overset{\gamma_1 \ldots \gamma_m}{\twoheadrightarrow} M_3.$$

Due to non-strictness and by the fact that variable y gets grounded it must be the case that the above reduction sequences are indeed applicable to M_1' and M_2' respectively. ∎

Corollary 7.5 *Compilers $\langle \mathcal{D}, \longrightarrow_{R_{comp}} \rangle$, $\langle \mathcal{D}, \longrightarrow_{R_{comp_1}} \rangle$ and $\langle \mathcal{D}, \longrightarrow_{R_{comp_2}} \rangle$ are partially correct with respect to $R_{P\text{-}TAC}$.*

Theorem 7.6 *A compiler $\langle \mathcal{D}, \longrightarrow_{R_c} \rangle$ is totally correct with respect to $R_{P\text{-}TAC}$ if*

1. $\forall\ \sigma \in R_c$ all the grounded instances of σ are derived rules in $R_{P\text{-}TAC}$; and

2. $\forall\ \sigma \in R_c$ the instances of σ that can introduce a ground value on the RHS are derived rules in $R_{P\text{-}TAC}$.

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 $M_1 \xrightarrow[\sigma]{\rho} M_2$, if M_2 terminates properly in $\mathcal{Q}$ then all free variables of σ 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 σ-reduction does introduce a new ground value in $\mathcal{Q}$ and σ is not a derived rule, which contradicts the second condition. What we have shown so far is that a P-TAC program M terminates properly in $\mathcal{P}$ iff M terminates properly in $\mathcal{Q}$. We can then conclude that a variables gets grounded in $\mathcal{P}$ iff it gets grounded in $\mathcal{Q}$, that is, no information can be deleted or created. From this we also derive that both improper-termination, $\top$ 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 $\mathcal{Q}$. Note that M can not produce $\top$ in $\mathcal{Q}$ because this means that a new ground value is produced in $\mathcal{Q}$, and this violates what proved previously. Analogously M can not produce $\bot$ in $\mathcal{Q}$. ∎

Corollary 7.7 *The compiler $\langle \mathcal{D}, \longrightarrow_{R_{comp}} \rangle$ is totally correct with respect to $R_{P\text{-}TAC}$.*

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 ⊤ or improper-termination into a properly terminating program. *Dead code elimination* can also interfere with deadlock detection and optimizations in R_{comp_2}.

It appears that even though *ARS*'s 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 N00014-84-K-0099 (MIT) and N0039-88-C-0163 (Harvard).

References

[1] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5), October 1988.

[2] Arvind, S. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proceedings of the Fourth International Symposium on Biological and Artificial Intelligence Systems, E. Clementi and S. Chin (eds), Trento, Italy*, pages 255–286. ESCOM Science Publishers, Leiden, The Netherlands, S ptember 1988.

[3] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands, Springer-Verlag LNCS 259*, June 1987. (To appear in IEEE Transactions on Computers).

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, Springer-Verlag LNCS 279*, pages 336–369, September/October 1987.

[5] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[6] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Towards an Intermediate Language Based on Graph Rewriting. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands, Springer-Verlag LNCS 259*, pages 159–175, June 1987.

[7] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands, Springer-Verlag LNCS 259*, pages 141–158, June 1987.

[8] T. Hardin. Résultat de Confluence pour les Règles Fortes da la Logique Combinatoire Categoique et Liens avec les Lambda-calculs. Ph.D. thesis, Université Paris VII, October 1987.

[9] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of Functional Programming Languages and Computer Architecture Conference, Nancy, France, Springer-Verlag LNCS 201*, September 1985.

[10] J. Klop. Term Rewriting Systems. Course Notes, Summer course organized by Corrado Boehm, Ustica,Italy, September 1985.

[11] A. R. Meyer and S. S. Cosmadakis. Semantical Paradigms: Notes for an Invited Lecture. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1988.

[12] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[13] R. S. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, July 1986.

[14] S. L. Peyton Jones. FLIC — a Functional Language Intermediate Code. *ACM SIGPLAN Notices*, 23(8):30–48, August 1988.

[15] S. K. Skedzielewski and M. L. Welcome. Data Flow Graph Optimization in IF1. In *Proceedings of Functional Programming Languages and Computer Architecture Conference, Nancy, France, Springer-Verlag LNCS 201*, pages 17–34, September 1985.

[16] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[17] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report LCS TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988.