

# Optimizing Pattern Matching by Program Transformation

Emilie Balland and Pierre-Etienne Moreau

UHP & LORIA and INRIA & LORIA

**Abstract.** The compilation of pattern matching constructs is crucial to the efficient implementation of functional languages like ML, Caml, or Haskell as well as rewrite rule based languages such as ASF+SDF, ELAN, Maude, or Stratego for example. Until now, the classical approach was to compute an (optimized) automaton before generating, in a straight-forward way, the corresponding implementation code. Optimizations such as tests-sharing are encoded in the construction of the automaton. While efficient, this leads to algorithms which are often complex and difficult to extend and to maintain.

In this paper we present a new compilation and optimization method based on program transformation. The principle is to separate the compilation of pattern matching from the optimization, in order to improve modularity and make extensions simpler. In a first step, the patterns are compiled using a simple, but safe algorithm. Then, optimizations are directly performed on the generated code, using transformation rules. Separating optimization from compilation eases the compilation of extensions, such as new equational theories, or the addition of or-patterns for example. Another contribution of this paper is to define a set of rules which defines the optimization, to show their correction as well as their effectiveness on real programs. The presented approach has been implemented and applied to Tom, a language extension which adds pattern-matching facilities to C and Java.

## 1 Introduction

Pattern matching is an elegant high-level construct which appears in many programming languages. Similarly to method dispatching in object oriented languages, in functional languages like ML [1], or rewrite rule bases languages like ASF+SDF [7,14], ELAN [4], Maude [6] or Tom [12], the notion of pattern matching is essential: it is part of the main execution mechanism.

In this context, it is necessary to have an efficient compilation of this construction. There exists several methods [5,2,9,8] to compile pattern matching. The simplest one is to consider and compile each pattern independently. These kind of algorithms are called *one-to-one* because they operate on only *one* pattern and *one* subject. A more efficient approach consists in considering the system globally and building a discrimination network to efficiently select the pattern that matches. These methods are called *many-to-one*, and they usually consist of three phases: first constructing an automaton, then optimizing it, and finally generating the implementation code. There are two main approaches to construct a matching automaton. The first is based on decision trees [5,9], which correspond

to deterministic automata. By introducing some redundancy, the idea is to ensure that every position of a given term is tested at most once. As a counterpart, the size of the decision tree becomes exponential in the number and the size of patterns. The second approach is based on backtracking automata [2], to avoid duplicating code. As a consequence, the efficiency can no longer be linear in the size of the subject: the compromise between speed and memory space is unavoidable [13].

In this paper, we present a new approach for pattern-matching compilation where the optimization phase is kept separated from the compilation phase. This allows us to design algorithms which are simpler to implement, easier to extend, and that can be formally certified [11,10]. In addition, this work allows to generate efficient implementations.

The presented work takes part of the Tom project, whose goal is to integrate the notion of pattern matching into classical languages such as C and Java. As presented in [12], a Tom program is a program written in a host language and extended by some new instructions like the `%match` construct. Therefore, a program can be seen as a list of Tom constructs interleaved with some sequences of characters. During the compilation process, all Tom constructs are dissolved and replaced by instructions of the host-language, as it is usually done by a pre-processor. The following example shows how a simple symbolic computation (addition) over Peano integers can be defined. This supposes the existence of a data-structure where Peano integers are represented by *zero* and *successor*: the integer 3 is denoted by *suc(suc(suc(zero)))* for example.

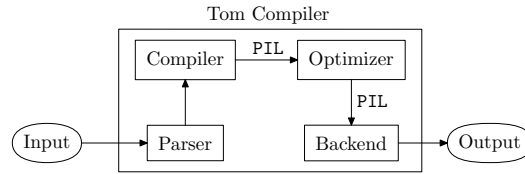
```
public class PeanoExample {
    ...
    Term plus(Term t1, Term t2) {
        %match(t1, t2) {
            x,zero    -> { return x; }
            x,suc(y) -> { return suc(plus(x,y)); }
        }
    }
    void run() {
        System.out.println("plus(1,2) = " + plus(suc(zero),suc(suc(zero))));
    }
}
```

In this example, given two terms  $t_1$  and  $t_2$  (that represent Peano integers), the evaluation of `plus` returns the sum of  $t_1$  and  $t_2$ . This is implemented by pattern matching:  $t_1$  is matched by  $x$ ,  $t_2$  is possibly matched by the two patterns *zero* and *suc(y)*. When *zero* matches  $t_2$ , the result of the addition is  $x$  (with  $x = t_1$ , instantiated by matching). When *suc(y)* matches  $t_2$ , this means that  $t_2$  is rooted by a *suc* symbol: the subterm  $y$  is added to  $x$  and the successor of this number is returned. The definition of `plus` is given in a functional programming style, but the `plus` function can be used in Java to perform computations. This first example illustrates how the `%match` construct can be used in conjunction with the considered native language.

In order to understand the choices we have made when designing the pattern matching algorithm, it is important to consider Tom as a *restricted* compiler (like a pre-processor)

which does not have any information about the host-language. In particular, the data-structure, against which the pattern matching is performed, is not fixed. As a consequence, we cannot assume that a function symbol like *suc* or *zero* is represented by an integer, like it is commonly done in other implementations of pattern matching. In some sense, the data-structure is a parameter of the pattern matching, see [10] for more details.

To allow this flexibility and support several host-languages, a pattern matching problem is compiled into an intermediate language code, called PIL, before being compiled into the selected host-language. The purpose of this work is to add an optimizing phase which performs program transformation on the PIL code. The general architecture of Tom is illustrated in Figure 1.



**Fig. 1.** General architecture of Tom: the compiler generates an intermediate PIL program which is optimized before being pretty-printed by the backend.

Optimization can be seen independently from the compilation step. In our case, optimization is called *source-to-source* because the input and the target languages are the same. In order to manipulate the program, we consider an abstract representation, e.g. the abstract syntactic tree (AST). By representing each instruction by a node, this structure is very closed to the initial code. Using program transformation to do source-to-source optimization has several advantages: the transformations can be easily described using rewrite rules, allowing us to test and activate them separately. The order and combination of different rules can be described using a strategy languages *a la* ELAN [4] or Stratego [15].

*Road-map of the paper.* In Section 2, we present the intermediate language PIL and its semantics. In Section 3, we present a rewrite system which describe an optimizer. We also define a strategy to apply efficiently these rules. In Section 4 we show that the presented rewrite system, and thus optimizations are correct. Finally, we present some experimental results in several revealing examples.

## 2 PIL language

In this section we define the syntax and the semantics of our intermediate language, called PIL.

## 2.1 Preliminary concepts

We assume the reader to be familiar with the basic definitions of first order term given, in particular, in [3]. We briefly recall or introduce the notation for a few concepts that will be used along this paper.

A signature  $\mathcal{F}$  is a set of function symbols, each one associated to a natural number by the arity function ( $\text{ar} : \mathcal{F} \rightarrow \mathbb{N}$ ).  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  is the set of *terms* built from a given finite set  $\mathcal{F}$  of function symbols and a enumerable set  $\mathcal{X}$  of variables. Positions in a term are represented as sequences of integers and denoted by  $\wedge$  or  $\omega$ . The set of positions is noted  $\Omega$ . The empty sequence  $\wedge$  denotes the position associated to the root, and it is called the root (or top) position. The subterm of  $t$  at position  $\omega$  is denoted  $t|_{\omega}$ . The replacement in  $t$  of the subterm  $t|_{\omega}$  by  $t'$  is denoted  $t[t']_{\omega}$ .

The set of variables occurring in a term  $t$  is denoted by  $\text{Var}(t)$ . If  $\text{Var}(t)$  is empty,  $t$  is called a *ground term* and  $\mathcal{T}(\mathcal{F})$  is the set of ground terms.

$\text{Symb}(t)$  is a partial function from  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  to  $\mathcal{F}$ , which associates to each term  $t$  its root symbol  $f \in \mathcal{F}$ . Two ground terms  $t$  and  $u$  of  $\mathcal{T}(\mathcal{F})$  are equal, and we note  $t = u$ , when, for some function symbol  $f$ ,  $\text{Symb}(t) = \text{Symb}(u) = f$ ,  $f \in \mathcal{F}$ ,  $t = f(t_1, \dots, t_n)$ ,  $u = f(u_1, \dots, u_n)$ , and  $\forall i \in [1..n]$ ,  $t_i = u_i$ .

A *substitution*  $\sigma$  is an assignment from  $\mathcal{X}$  to  $\mathcal{T}(\mathcal{F})$ , written, when its domain is finite,  $\sigma = \{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$ . It uniquely extends to an endomorphism  $\sigma'$  of  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ :  $\sigma'(x) = \sigma(x)$  for each variable  $x \in \mathcal{X}$ ,  $\sigma'(f(t_1, \dots, t_n)) = f(\sigma'(t_1), \dots, \sigma'(t_n))$  for each function symbol  $f \in \mathcal{F}$ .

Given a pattern  $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  and a ground term  $t \in \mathcal{T}(\mathcal{F})$ ,  $p$  *matches*  $t$ , written  $p \ll t$ , if and only if there exists a substitution  $\sigma$  such that  $\sigma(p) = t$ .

A conditional rewrite rule is an ordered tuple of terms  $\langle l, r, c \rangle$  noted  $l \rightarrow r$  IF  $c$  such that  $\text{Var}(r) \subseteq \text{Var}(l)$  and  $\text{Var}(c) \subseteq \text{Var}(l)$ . Given a set  $R$  of rewrite rules, called a rewriting system, a term  $t \in \mathcal{T}(\mathcal{F})$  is rewritten into  $t'$  if there exist: a rule  $l \rightarrow r$  IF  $c \in R$ , a position  $\omega$ , and a substitution  $\sigma$  such that  $t|_{\omega} = \sigma(l)$  and  $\sigma(c)$  is true. In that case,  $t' = t[\sigma(r)]_{\omega}$ . When no more rule can be applied, the term obtained is said in *normal form* and noted  $t \downarrow_R$ .

## 2.2 Syntax

The syntax of PIL is given in Figure 2. As mentioned previously, the data-model of this language is a parameter. In practice, this means that a meta-language has to be used to define on which type of data-structure the expressions and instructions operate. For expository reasons, in the rest of this paper, we consider that first order terms define the data-model. Similarly to functional programming languages, given a signature  $\mathcal{F}$  and a set of variables  $\mathcal{X}$ , the considered PIL language can directly handle *terms* and perform operations like checking that a given term  $t$  is rooted by a symbol  $f$  (`is_fsymb(t, f)`), or accessing to n-th child of a this term  $t$  (`subtermf(t, n)`).

The intermediate language PIL has both functional and imperative flavors: the assignment instruction (`let(variable, term, instr)`) defines a scoped unmodifiable variable, whereas the sequence instruction (`instr; instr`) comes from imperative languages. A last particularity of PIL comes from the `hostcode(...)` instruction which is used to abstract

$\begin{aligned} \text{PIL} & ::= \langle instr \rangle \\ \text{symbol} & ::= f \in \mathcal{F} \\ \text{variable} & ::= x \in \mathcal{X} \\ \langle term \rangle & ::= t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\ & \quad   \text{subterm}_f(\langle term \rangle, n) \\ & \quad \quad (f \in \mathcal{F} \wedge n \in \mathbb{N}) \end{aligned}$	$\begin{aligned} \langle expr \rangle & ::= b \in \mathbb{B} \\ & \quad   \text{eq}(\langle term \rangle, \langle term \rangle) \\ & \quad   \text{is\_fsym}(\langle term \rangle, \text{symbol}) \\ \langle instr \rangle & ::= \text{let}(\text{variable}, \langle term \rangle, \langle instr \rangle) \\ & \quad   \text{if}(\langle expr \rangle, \langle instr \rangle, \langle instr \rangle) \\ & \quad   \langle instr \rangle; \langle instr \rangle \\ & \quad   \text{hostcode}(\text{variable}^*) \\ & \quad   \text{nop} \end{aligned}$
--	---

**Fig. 2.** PIL syntax

part of code written in the underlying host-language. This instruction is parametrized by a list of PIL-variables which are used in this part of code. Examples of Tom, and PIL code are given in figure 3.

<p>Tom code:</p> <pre style="margin: 0;">%match(Term t) {   f(a) =&gt; { print(...); }   g(x) =&gt; { print(...x...); }   f(b) =&gt; { print(...); } }</pre>	<p>Generated PIL code:</p> <pre style="margin: 0;">if(is_fsym (t,f), let(t1, subterm_f(t, 1),   if(is_fsym(t1, a), hostcode(), nop)),   nop) ; if(is_fsym (t,g), let(t1, subterm_g(t, 1),   let(x, t1, hostcode(x)))   nop) ; if(is_fsym (t,f), let(t1, subterm_f(t, 1),   if(is_fsym(t1, b), hostcode(), nop)),   nop)</pre>
--	---

**Fig. 3.** Example of PIL code generated by Tom

### 2.3 Semantics

We define PIL semantics as in [10] by a big-step semantics *à la Kahn*. First, we introduce the notion of *environment*, which models the memory of a program during its evaluation. To represent a substitution, we model an environment by a stack of assignments of terms to variables.

**Definition 1.** *An atomic environment  $\epsilon$  is an assignment from  $\mathcal{X}$  to  $\mathcal{T}(\mathcal{F})$ , written  $[x \leftarrow t]$ . The composition of environments is left-associative, and written  $[x_1 \leftarrow t_1][x_2 \leftarrow t_2] \cdots [x_k \leftarrow t_k]$ . Its application is such that:  $\epsilon[x \leftarrow t](y) = t$  if  $y \equiv x$ ,  $\epsilon(y)$  otherwise.*

*We note  $\epsilon|_X$ , the environment  $\epsilon$  whose domain is restricted to  $X$ . We extend the notion of environment to a morphism  $\epsilon$  from PIL to PIL, and we note  $\text{Env}$  the set of all environments.*

The reduction relation of this big-step semantics is expressed on tuples  $\langle \epsilon, \delta, i \rangle$  where  $\epsilon$  is an environment,  $\delta$  is a list of pairs (restricted environment, hostcode), and  $i$  is an

instruction. Thanks to  $\delta$ , we can keep track of the executed hostcode blocks within their environment: the restricted environment associated to each hostcode construct gives the instances of all variables which appear in the block. The reduction relation is the following:

$$\langle \epsilon, \delta, i \rangle \mapsto_{bs} \delta', \text{ with } \epsilon \in \mathcal{Env}, \delta, \delta' \in [\mathcal{Env}, \langle instr \rangle]^*, \text{ and } i \in \langle instr \rangle$$

Before giving the semantics for the instructions, we define a rewrite system, denoted  $R$ , which describes how expressions and terms are evaluated in a PIL program:

$$R = \begin{cases} \text{is\_fsym}(t, f) & \rightarrow \text{Symb}(t) = f & \text{IF } t \in \mathcal{T}(\mathcal{F}) \\ \text{eq}(t_1, t_2) & \rightarrow t_1 = t_2 & \text{IF } t_1, t_2 \in \mathcal{T}(\mathcal{F}) \\ \text{subterm}_f(t, i) & \rightarrow t|_i & \text{IF } \text{Symb}(t) = f \wedge i < \text{arity}(f) \end{cases}$$

**Proposition 1.** *Given  $t \in \langle term \rangle$  (resp.  $t \in \langle expr \rangle$ ), if  $t$  contains no variable,  $t \downarrow_R \in \mathcal{T}(\mathcal{F})$  (resp.  $t \downarrow_R \in \mathbb{B}$ ).*

*Proof.* We reason by induction on  $el \in \langle expr \rangle \cup \langle term \rangle$ :

- when  $e$  is a term from  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ , no rule can be applied. Since it contains no variable, we have  $e \downarrow_R = t \in \mathcal{T}(\mathcal{F})$ ,
- when  $e = \text{subterm}_f(t, i)$ : by induction  $t \downarrow_R \in \mathcal{T}(\mathcal{F})$ , the third rule applies and we obtain  $e \downarrow_R = (t|_i) \downarrow_R \in \mathcal{T}(\mathcal{F})$
- when  $e = b \in \mathbb{B}$ : no rule applies,  $e$  is in normal form and  $e \downarrow_R \in \mathbb{B}$ ,
- when  $e = \text{eq}(t_1, t_2)$ : by induction,  $t_1, t_2 \in \mathcal{T}(\mathcal{F})$ . Therefore, one application of  $R$  gives the normal form  $t_1 = t_2$  which is true or false, and we have  $e \downarrow_R \in \mathbb{B}$ ,
- when  $e = \text{is\_fsym}(t, f)$ : same reasoning.

$$\begin{array}{l} \frac{}{\langle \epsilon, \delta, \mathbf{nop} \rangle \mapsto_{bs} \delta} \quad (nop) \\ \frac{\langle \epsilon[x \leftarrow t], \delta, i \rangle \mapsto_{bs} \delta' \quad \epsilon(u) \rightarrow_{R^*} t}{\langle \epsilon, \delta, \mathbf{let}(x, u, i) \rangle \mapsto_{bs} \delta'} \quad (let) \\ \frac{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta' \quad \epsilon(e) \rightarrow_{R^*} \top}{\langle \epsilon, \delta, \mathbf{if}(e, i_1, i_2) \rangle \mapsto_{bs} \delta'} \quad (iftrue) \\ \frac{\langle \epsilon, \delta, i_2 \rangle \mapsto_{bs} \delta' \quad \epsilon(e) \rightarrow_{R^*} \perp}{\langle \epsilon, \delta, \mathbf{if}(e, i_1, i_2) \rangle \mapsto_{bs} \delta'} \quad (iffalse) \\ \frac{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta' \quad \langle \epsilon, \delta', i_2 \rangle \mapsto_{bs} \delta''}{\langle \epsilon, \delta, i_1 ; i_2 \rangle \mapsto_{bs} \delta''} \quad (seq) \\ \frac{}{\langle \epsilon, \delta, \mathbf{hostcode}(\mathbf{list}) \rangle \mapsto_{bs} \delta :: [\epsilon|_{\mathbf{list}}, \mathbf{hostcode}(\mathbf{list})]} \quad (hostcode) \end{array}$$

**Fig. 4.** Big-step semantics for PIL

## 2.4 Properties

**Definition 2.** A program  $\pi \in \text{PIL}$  is said to be well-formed when it satisfies the following properties:

- each expression  $\text{subterm}_f(t, n)$  is such that  $t$  belongs to  $\langle \text{term} \rangle$ ,  $\text{is\_fsym}(t, f) \equiv \top$  and  $n \in [1..ar(f)]$ ,  
(In practice, we verify that each expression of the form  $\text{subterm}_f(t, n)$  belongs to the **then** part of an instruction  $\text{if}(\text{is\_fsym}(t, f), \dots)$ )
- each variable appearing in a sub-expression is previously initialized by a **let** construct, or in the evaluation environment,
- in the construction  $\text{let}(v, u, i)$ , the instruction block  $i$  must not contain an instruction  $\text{let}(v, u', i')$  (i.e. a variable cannot be redefined).

**Proposition 2.** For all  $\epsilon \in \mathcal{Env}$  and  $\delta \in [\mathcal{Env}, \langle \text{instr} \rangle]^*$ , the reduction of a well-formed instruction  $i \in \langle \text{instr} \rangle$  in the environment  $\epsilon, \delta$  is unique.

*Proof.* The proof can be done by induction on program structures, see [10] for details.

**Definition 3.** Given  $\pi_1$  and  $\pi_2$  two well-formed PIL programs, they are semantically equivalent, noted  $\pi_1 \sim \pi_2$ , iff:

$$\forall \epsilon, \delta, \exists \delta' \text{ s.t. } \langle \epsilon, \delta, \pi_1 \rangle \mapsto_{bs} \delta' \text{ and } \langle \epsilon, \delta, \pi_2 \rangle \mapsto_{bs} \delta'$$

Given a well-formed program  $\pi \in \text{PIL}$ , a position  $\omega$ , an environment  $\epsilon$ , and a hostcode list  $\delta$ , we define  $\Phi$  the function that gives the environment  $\epsilon'$  during the evaluation of  $\pi|_\omega$  in the reduction tree of  $\pi$ :  $\langle \epsilon, \delta, \pi \rangle \mapsto_{bs} \delta'$ . (Note that  $\pi|_\omega$  can be either an expression, a term or an instruction).

**Definition 4.** The function  $\Phi \in \text{PIL} \times \Omega \times \mathcal{Env} \times [\mathcal{Env}, \langle \text{instr} \rangle] \rightarrow \mathcal{Env}$  is defined such that for a given  $\pi, \omega, \epsilon, \delta$  and with  $\epsilon' = \Phi(\pi, \omega, \epsilon, \delta)$ :

- if  $\pi|_\omega = i \in \langle \text{instr} \rangle$ ,  $\epsilon'$  is the environment  $\epsilon$  in which the rule corresponding to  $i$  (*let*, *nop*, *seq*, *iftrue*, *iffalse* or *hostcode*) is applied,
- if  $\pi|_\omega = e \in \langle \text{expr} \rangle$ ,  $e$  appears inevitably in an instruction **if** noted  $i$  and we have  $\epsilon' = \Phi(\pi, \epsilon, \delta, i)$ ,
- if  $\pi|_\omega = t \in \langle \text{term} \rangle$ , we must distinguish two cases. Either  $t$  appears in an expression  $e$ , and we have  $\epsilon' = \Phi(\pi, \epsilon, \delta, e)$ . Either  $t$  appears in an instruction **let**( $-, t, -$ ) noted  $i$ , and we have  $\epsilon' = \Phi(\pi, \epsilon, \delta, i)$ .

Note that  $\Phi$  is defined only for  $\pi|_\omega$  which are reachable at run time.

Let us consider again the program defined in Figure 3. We have:

- $\pi = \text{if}(\text{is\_fsym}(t, f), \text{let}(t_1, \text{subterm}_f(t, 1), \pi'), \text{nop})$  with
- $\pi' = \text{if}(\pi'', \text{hostcode}(), \text{nop})$  and
- $\pi'' = \text{is\_fsym}(t_1, a)$ .

Starting from the environments  $\epsilon = [t \leftarrow f(a)]$  and  $\delta = \{\}$ , we can construct the following derivation tree for  $\pi$ . From this derivation tree, we have:  $\Phi(\pi, \epsilon, \delta, \pi'') = \epsilon[t_1 \leftarrow a]$ . For layout purpose, we note  $\Delta$  the list  $\delta :: [\{\}, \text{hostcode}()]$ , and  $\epsilon' = \epsilon[t_1 \leftarrow a]$ :

$$\frac{\frac{\epsilon(\text{is\_fsym}(t, f)) \rightarrow_{R^*} \top \quad \frac{\frac{\epsilon'(\text{is\_fsym}(t_1, a)) \rightarrow_{R^*} \top \quad \overline{\langle \epsilon', \delta, \text{hostcode}() \rangle \mapsto_{bs} \Delta}}{\langle \epsilon[t_1 \leftarrow a], \delta, \text{if}(\pi'', \text{hostcode}(), \text{nop}) \rangle \mapsto_{bs} \Delta} \text{ (iftrue)}}{\langle \epsilon, \delta, \text{let}(t_1, \text{subterm}_f(t, 1), \pi') \rangle \mapsto_{bs} \Delta} \text{ (let)}}{\langle \epsilon, \delta, \text{if}(\text{is\_fsym}(t, f), \text{let}(t_1, \text{subterm}_f(t, 1), \pi'), \text{nop}) \rangle \mapsto_{bs} \Delta} \text{ (iftrue)}$$

**Definition 5.** Given  $\pi_1$  and  $\pi_2$  two well-formed PIL programs, they are semantically equivalent, noted  $\pi_1 \sim \pi_2$ , when:

$$\forall \epsilon, \delta, \exists \delta' \text{ s.t. } \langle \epsilon, \delta, \pi_1 \rangle \mapsto_{bs} \delta' \text{ and } \langle \epsilon, \delta, \pi_2 \rangle \mapsto_{bs} \delta'$$

**Definition 6.** Given a program  $\pi$ , two expressions  $e_1$  and  $e_2$  are said  $\pi$ -equivalent, and noted  $e_1 \sim_\pi e_2$ , if for all starting environment  $\epsilon, \delta$ ,  $\epsilon_1(e_1) \downarrow_R = \epsilon_2(e_2) \downarrow_R$  where  $\epsilon_1 = \Phi(\pi, \epsilon, \delta, e_1)$  and  $\epsilon_2 = \Phi(\pi, \epsilon, \delta, e_2)$ .

**Definition 7.** Given a program  $\pi$ , two expressions  $e_1$  and  $e_2$  are said  $\pi$ -incompatible, and noted  $e_1 \perp_\pi e_2$ , if for all starting environment  $\epsilon, \delta$ ,  $\epsilon_1(e_1) \downarrow_R \wedge \epsilon_2(e_2) \downarrow_R = \perp$  where  $\epsilon_1 = \Phi(\pi, \epsilon, \delta, e_1)$  and  $\epsilon_2 = \Phi(\pi, \epsilon, \delta, e_2)$ .

We can now define two conditions which are sufficient to determine whether two expression are  $\pi$ -equivalent or  $\pi$ -incompatible. Propositions 3 and 4 are interesting because they can be easily used in practice.

**Proposition 3.** Given a program  $\pi$  and two expressions  $e_1, e_2 \in \langle \text{expr} \rangle$ , we have  $e_1 \sim_\pi e_2$  if:  $\forall \epsilon, \delta$ ,  $\Phi(\pi, \epsilon, \delta, e_1) = \Phi(\pi, \epsilon, \delta, e_2)$  (*cond1*) and  $e_1 = e_2$  (*cond2*).

*Proof.* Given a program  $\pi$  and two expressions  $e_1, e_2$ , for all  $\delta, \epsilon$ , because of *cond1* and *cond2*,  $\Phi(\pi, \epsilon, \delta, e_1)(e_1) = \Phi(\pi, \epsilon, \delta, e_2)(e_2)$  therefore  $\Phi(\pi, \epsilon, \delta, e_1)(e_1) \downarrow_R = \Phi(\pi, \epsilon, \delta, e_2)(e_2) \downarrow_R$ .

**Proposition 4.** Given a program  $\pi$  and two expressions  $e_1, e_2 \in \langle \text{expr} \rangle$ , we have  $e_1 \perp_\pi e_2$  if:  $\forall \epsilon, \delta$ ,  $\Phi(\pi, \epsilon, \delta, e_1) = \Phi(\pi, \epsilon, \delta, e_2)$  (*cond1*) and *incompatible*( $e_1, e_2$ ) (*cond2*), where *incompatible* defined as follows:

$$\text{incompatible}(e_1, e_2) = \text{match } e_1, e_2 \text{ with} \begin{array}{l} | \perp, \top \quad \quad \quad \rightarrow \top \\ | \top, \perp \quad \quad \quad \rightarrow \top \\ | \text{is\_fsym}(t, f_1), \text{is\_fsym}(t, f_2) \rightarrow \top \text{ if } f_1 \neq f_2 \\ | \_, \_ \quad \quad \quad \rightarrow \perp \end{array}$$

*Proof.* Given a program  $\pi$  and two expressions  $e_1, e_2 \in \langle \text{expr} \rangle$ , for all  $\epsilon, \delta$ , we want to prove that *cond1*  $\wedge$  *cond2*  $\rightarrow \epsilon_1(e_1) \downarrow_R \wedge \epsilon_2(e_2) \downarrow_R = \perp$  with  $\epsilon_1 = \Phi(\pi, \epsilon, \delta, e_1)$  and  $\epsilon_2 = \Phi(\pi, \epsilon, \delta, e_2)$ . We will detail every case for which *cond2* is true (because when *cond2* is false, the proposition is trivially verified):

- case 1  $e_1 = \top, e_2 = \perp$ : as  $e_1, e_2$  are yet evaluated and reduced,  $\epsilon_1(e_1)\downarrow_R \wedge \epsilon_2(e_2)\downarrow_R = e_1 \wedge e_2 = \perp$
- case 2  $e_1 = \top, e_2 = \perp$ : similar to case 1
- case 3  $e_1 = \text{is\_fsym}(t, f_1), e_2 = \text{is\_fsym}(t, f_2)$  with  $f_1 \neq f_2$ : because of `cond1`,  $\epsilon_1(t) = \epsilon_2(t) = u$  then  $\epsilon_1(e_1)\downarrow_R = \text{Symb}(u) = f_1$  and  $\epsilon_2(e_2)\downarrow_R = \text{Symb}(u) = f_2$ . Since every ground term has an unique head symbol and  $f_1 \neq f_2$ ,  $\epsilon_1(e_1)\downarrow_R \wedge \epsilon_2(e_2)\downarrow_R = \perp$ .

### 3 Optimizations

An optimization is a transformation which reduces the size of code (*space optimization*) or the time of execution (*time optimization*). In the case of PIL, the presented optimizations reduce the number of assignments (`let`) and tests (`if`) that are executed at run time. When manipulating abstract syntax trees, an optimization can be easily be described by a rewriting system. Its application consists in rewriting an instruction into an equivalent one, using a conditional rewrite rule of the form  $i_1 \rightarrow i_2 \text{ IF } c$ .

**Definition 8.** *An optimization rule  $i_1 \rightarrow i_2 \text{ IF } c$  rewrites a well-formed program  $\pi$  into a program  $\pi'$  if there exists a position  $\omega$  and a substitution  $\sigma$  such that  $\sigma(i_1) = \pi|_\omega$ ,  $\pi' = \pi[\sigma(i_2)]_\omega$  and  $\sigma(c)$  is verified. If  $c = e_1 \sim e_2$  (resp.  $c = e_1 \perp e_2$ ), we say that  $\sigma(c)$  is verified when  $\sigma(e_1) \sim_{\pi|_\omega} \sigma(e_2)$  (resp.  $\sigma(e_1) \perp_{\pi|_\omega} \sigma(e_2)$ ).*

Since the notion of equivalence or incompatibility is dependent on the context of a program. In a rule, the context is given by the left part of the rule.

#### 3.1 Reducing the number of assignments

This kind of optimization is standard, but useful to eliminate useless assignments. In the context of pattern matching, this improves the construction of substitutions, when a variable from the left-hand side is not used in the right-hand side for example.

**Constant propagation.** This first optimization removes the assignment of a variable defined as a constant. Since no side-effect can occur in a PIL program, it is possible to replace all occurrences of the variable by the constant (written  $i[v/t]$ ).

$$\text{ConstProp: } \text{let}(v, t, i) \rightarrow i[v/t] \text{ IF } t \in \mathcal{T}(\mathcal{F}_0)$$

**Dead variable elimination.** Using a simple static analysis, this optimization eliminates useless assignments:

$$\text{DeadVarElim: } \text{let}(v, t, i) \rightarrow i \text{ IF } \text{use}(v, i) = 0$$

**Definition 9.** *use* is a function, defined recursively, which computes how many times the value of a variable  $v$  may be used to evaluate a term, an expression or an instruction  $t$ :

$$\text{use}(v, t) = \text{match } t \text{ with}$$

$v \in \mathcal{X}$	$\rightarrow 1$	$\text{eq}(t_1, t_2)$	$\rightarrow$	$\text{use}(v, t_1) + \text{use}(v, t_2)$
$v' \in \mathcal{X}$	$\rightarrow 0$ if $v \neq v'$	$t \in \mathcal{T}(\mathcal{F})$	$\rightarrow$	$0$
<b>nop</b>	$\rightarrow 0$	$i_1 ; i_2$	$\rightarrow$	$\text{use}(v, i_1) + \text{use}(v, i_2)$
<b>is_fsym</b> ( $t, \_$ )	$\rightarrow \text{use}(v, t)$	<b>let</b> ( $\_, t, i$ )	$\rightarrow$	$\text{use}(v, t) + \text{use}(v, i)$
<b>subterm</b> <sub><math>f</math></sub> ( $t, \_$ )	$\rightarrow \text{use}(v, t)$	<b>hostcode</b> ( <b>list</b> )	$\rightarrow$	$\text{count}(\mathbf{list}, v)$
<b>if</b> ( $e, i_1, i_2$ )	$\rightarrow \text{use}(v, e) + \max(\text{use}(v, i_1), \text{use}(v, i_2))$			

where  $\text{count}(l, v)$  counts  $v$  apparitions in  $l$  list. Note that the value computed by **use** is an upper bound.

**Inlining.** Since no side-effect can occur in a PIL program, an assigned variable cannot be modified. Therefore, when a variable is used only once, we can replace its occurrence by the corresponding term:

$$\text{Inlining: } \mathbf{let}(v, t, i) \rightarrow i[v/t] \text{ IF } \text{use}(v, i) = 1$$

**Fusion.** The following rule merges two successive **let** which assign a same value to two different variables. This kind of optimization rarely applies on human written code, but in the context of pattern matching compilation, rules are compiled independently from others: the variables corresponding to the subject is defined for each rule. By merging the bodies, this allows to recursively perform some optimizations on subterms.

$$\text{LetFusion: } \mathbf{let}(v_1, t, i_1); \mathbf{let}(v_2, t, i_2) \rightarrow \mathbf{let}(v_1, t, i_1; i_2[v_2/v_1])$$

Note that the assignments must be the same for the two variables, to be sure that the values at run time are the same. We also suppose that  $\text{use}(v_1, i_2) = 0$ . Otherwise, it would require to replace  $v_1$  by a fresh variable in  $i_2$ .

### 3.2 Reducing the number of tests

The key technique to optimize pattern matching consists in merging branches, and thus tests that correspond to patterns with identical prefix. Usually, the discrimination between branches is performed by a *switch* instruction. In **Tom**, since the data-structure is not fixed, we cannot assume that a symbol is represented by an integer, and thus, contrary to standard approaches, we have to use an *if* statement instead. This restriction prevents us from selecting a branch in constant time. The two following rules define the fusion and the interleaving of conditional blocks.

**Fusion.** The fusion of two conditional adjacent blocks reduces the number of tests. This fusion is possible only when the two conditions are  $\pi$ -equivalent, This notion of  $\pi$ -equivalence means that the evaluation of the two conditions in a given program are the same (see Definition 6):

$$\text{IfFusion: } \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2) \rightarrow \text{if}(c_1, i_1; i_2, i'_1; i'_2) \text{IF } c_1 \sim c_2$$

To evaluate  $c_1 \sim c_2$  (i.e.  $c_1 \sim_\pi c_2$  with  $\pi$  the redex of the rule), we use Proposition 3. The condition **cond1** (i.e.  $\Phi(\pi, \epsilon, \delta, e_1) = \Phi(\pi, \epsilon, \delta, e_2)$ ) is trivially verified because the semantics of the sequence instruction preserves the environment ( $\forall \delta, \epsilon, \Phi(\pi, \epsilon, \delta, i_1; i_2) = \Phi(\pi, \epsilon, \delta, i_1) = \Phi(\pi, \epsilon, \delta, i_2)$ ) and then  $\forall \delta, \epsilon, \Phi(\pi, \epsilon, \delta, \sigma(c_1)) = \Phi(\pi, \epsilon, \delta, \sigma(c_2))$ . We just have to verify **cond2** (i.e.  $e_1 = e_2$ ) which is easier.

**Interleaving.** As matching code consists of a sequence of conditional blocks, we would like to optimize blocks with  $\pi$ -incompatible conditions (see Definition 7). Some parts of the code cannot be executed at the same time, so swapping statically their order does not change the program behavior.

As we want to keep only one of the conditional block, we determine what instructions must be executed in case of success or failure of the condition and we obtain the two following transformation rules:

$$\begin{aligned} \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_1, i_1; i'_2, i'_1; \text{if}(c_2, i_2, i'_2)) \text{IF } c_1 \perp c_2 \\ \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_2, i'_1; i_2, \text{if}(c_1, i_1, i'_1), i'_2) \text{IF } c_1 \perp c_2 \end{aligned}$$

As for the equivalence in the **IfFusion** rule, to evaluate  $c_1 \perp c_2$ , we just have to verify **cond2** in Proposition 4 (i.e. **incompatible**( $e_1, e_2$ )). A drawback of these two rules is that some code is duplicated ( $i'_2$  in the first rule, and  $i'_1$  in the second one). As we want to maintain linear the size of the code, we consider specialized instances of these rules with respectively  $i'_2$  and  $i'_1$  equal to **nop**, and eliminate the extra **nop** instructions. This gives us the following rules:

$$\begin{aligned} \text{IfInterleaving: } \text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, \text{nop}) &\rightarrow \text{if}(c_1, i_1, i'_1; \text{if}(c_2, i_2, \text{nop})) \text{IF } c_1 \perp c_2 \\ \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, i'_2) &\rightarrow \text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop}); i'_2) \text{IF } c_1 \perp c_2 \end{aligned}$$

These two rules reduce the number of tests at run time because one of the tests is moved into the “else” branch of the other. The second rule can be instantiated and used to swap blocks. When  $i'_1$  and  $i'_2$  are reduced to the instruction **nop**, the second rule can be simplified into:

$$\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop})) \text{IF } c_1 \perp c_2$$

As the two conditions are  $\pi$ -incompatible, we have the following equivalence:

$$\text{if}(c_2, i_2, \text{if}(c_1, i_1, \text{nop})) \equiv \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \text{IF } c_1 \perp c_2$$

After all, we obtain the following rule corresponding to the swapping of two conditional adjacent blocks. This rule does not optimize the number of tests but is useful to join blocks subject to be merged thanks to a smart strategy.

$$\text{IfSwapping: } \text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \text{IF } c_1 \perp c_2$$

<b>DeadVarElim</b>	$\text{let}(v, t, i)$	$\rightarrow i$	<b>IF</b> $\text{use}(v, i) = 0$
<b>ConstProp</b>	$\text{let}(v, t, i)$	$\rightarrow i[v/t]$	<b>IF</b> $t \in \mathcal{T}(\mathcal{F}_0)$
<b>Inlining</b>	$\text{let}(v, t, i)$	$\rightarrow i[v/t]$	<b>IF</b> $\text{use}(v, i) = 1$
<b>LetFusion</b>	$\text{let}(v_1, t, i_1); \text{let}(v_2, t, i_2)$	$\rightarrow \text{let}(v_1, t, i_1; i_2[v_2/v_1])$	
<b>IfFusion</b>	$\text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, i'_2)$	$\rightarrow \text{if}(c_1, i_1; i_2, i'_1; i'_2)$	<b>IF</b> $c_1 \sim c_2$
<b>IfInterleaving</b>	$\text{if}(c_1, i_1, i'_1); \text{if}(c_2, i_2, \text{nop})$	$\rightarrow \text{if}(c_1, i_1, i'_1; \text{if}(c_2, i_2, \text{nop}))$	<b>IF</b> $c_1 \perp c_2$
<b>IfSwapping</b>	$\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop})$	$\rightarrow \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop})$	<b>IF</b> $c_1 \perp c_2$

**Fig. 5.** System of optimization rules: OptSys

### 3.3 Application strategy

Given the rules presented in Section 3.1 and Section 3.2, Figure 5 defines the rewrite systems called OptSys.

Without strategy, this system is clearly not confluent and not terminating. For example, without condition, the **IfSwapping** rule can be applied indefinitely because of the symmetry of incompatibility. The confluence of the system is not necessary as long as the programs obtained are semantically equivalent to the source program but the termination is an essential criterion. Moreover, the strategy should apply the rules to obtain a program, as efficient as possible. Let us consider again the program given in Figure 3, and let us suppose that we interleave the two last patterns. This would result in the following sub-optimal program:

```

if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), nop)), nop) ;
if(is_fsym(t, g), let(t1, subtermg(t, 1), let(x, t1, hostcode(x)))
    if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, b), hostcode(), nop)), nop)

```

**IfSwapping** and **IfFusion** rules can longer be applied to share the  $\text{is\_fsym}(t, f)$  tests. This order of application is not optimal. It is intuitive that the interleaving rule must be applied at the end, when no more optimization or fusion is possible.

The second matter is to ensure the termination. The **IfSwapping** rule is the only rule that does not decrease the size or the number of assignments of a program. To limit its application for interesting cases, we define a condition which ensures that a swapping is performed only if it enables a fusion. This condition can be implemented in two ways, either in using a context, or in defining a total order on conditions. Given a sequence of **if** instructions, the first approach consists in swapping two of them when the right one can be fused with an **if** which earlier in the sequence. This approach is not efficient since we have to find two elements in a list that can react via the **IfFusion** rule. The second approach is more efficient since it consists in considering a total order on conditions and perform the swapping only if two successive **if** have some conditions  $c_1$  and  $c_2$  such that  $c_1 \perp c_2$  and  $c_1 < c_2$ . Similarly to a swap-sort algorithm, this approach ensures the termination of the algorithm. In this way, we obtain a new **IfSwapping** rule:

$$\text{if}(c_1, i_1, \text{nop}); \text{if}(c_2, i_2, \text{nop}) \rightarrow \text{if}(c_2, i_2, \text{nop}); \text{if}(c_1, i_1, \text{nop}) \text{ IF } c_1 \perp c_2 \wedge c_1 < c_2$$

Using basic strategy operators such as  $Innermost(s)$  (which applies  $s$  as many times as possible, starting from the leaves),  $s1 \mid s2$  (which applies  $s1$  or  $s2$  indifferently),  $repeat(s)$  (which applies  $s$  as many times as possible, returning the last unfailing result), and  $r1 ; r2$  (which applies  $s1$ , and then  $s2$  if  $s1$  did not failed), we can easily define a strategy which describes how the rewrite system `OptSys` should be applied to normalize a PIL program:

```
Innermost( repeat(ConstProp | DeadVarElim | Inlining | LetFusion | IfFusion | IfSwapping) ;
           repeat(IfInterleaving))
```

Starting from the program given in Figure 3, we can apply the rule `IfSwapping`, followed by a step of `IfFusion`, and we obtain:

```
if(is_fsym(t, f), let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), nop))
                 ; let(t1, subtermf(t, 1), if(is_fsym(t1, b), hostcode(), nop)), nop) ;
if(is_fsym(t, g), let(t1, subtermg(t, 1), let(x, t1, hostcode(x))), nop)
```

Then, we can apply a step of `Inlining` to remove the second instance of  $t_1$ , a step of `LetFusion`, and a step of `Interleaving` (`is_fsym(t1, a)` and `is_fsym(t1, b)` are  $\pi$ -incompatible). This results in the following program:

```
if(is_fsym(t, f), let(t1, subtermf(t, 1),
                    if(is_fsym(t1, a), hostcode(), if(is_fsym(t1, b), hostcode(), nop))), nop) ;
if(is_fsym(t, g), let(x, subtermg(t, 1), hostcode(x)), nop)
```

Since `is_fsym(t, f)` and `is_fsym(t, g)` are  $\pi$ -incompatible, we can apply a step of `IfInterleaving`, and get the irreducible following program:

```
if(is_fsym(t, f),
   let(t1, subtermf(t, 1), if(is_fsym(t1, a), hostcode(), if(is_fsym(t1, b), hostcode(), nop))),
   if(is_fsym(t, g), let(x, subtermg(t, 1), hostcode(x)), nop)
```

## 4 Properties

When performing optimization by program transformation, it is important to ensure that the generated code has some expected properties. The use of formal methods to describe our optimization algorithm allows us to give proofs. In the section we show that each transformation rule is correct, in the sense that the the optimized program has the same observational behavior as the original. We also show that the optimized code is both more efficient, and smaller than the initial program.

### 4.1 Correction

**Definition 10.** *A transformation rule  $r$  is correct if for all well-formed program  $\pi$ ,  $r$  rewrites  $\pi$  in  $\pi'$  (Definition 8) implies that  $\pi \sim \pi'$  (Definition 5).*

From this definition, we prove that every rule given in Section 3 is correct. For that, two conditions have to be verified:

1.  $\pi'$  is well-formed,
2.  $\forall \epsilon, \delta$ , the derivations of  $\pi$  and  $\pi'$  lead to the same result  $\delta'$ .

The first condition is quite easy to verify. The second one is more interesting: we consider a program  $\pi$ , a rule  $l \rightarrow r$  IF  $c$ , a position  $\omega$ , and a substitution  $\sigma$  such that  $\sigma(l) = \pi|_{\omega}$ . We have  $\pi' = \pi[\sigma(r)]_{\omega}$ . We have to compare the derivations of  $\pi$  and  $\pi'$  in a the context  $\epsilon, \delta$ .

- when  $\omega = \wedge$ , we have to compare the derivation tree of  $\pi = \sigma(l)$  and  $\pi' = \sigma(r)$ ,
- otherwise, we consider the derivation of  $\pi$  (resp.  $\pi'$ ): there is a step which needs in premise the derivation of  $\pi|_{\omega}$  (resp.  $\pi[\sigma(r)]_{\omega}$ ). This is the only difference between the two trees.

In both cases, we have to verify that  $\pi|_{\omega} = \sigma(r)$  and  $\sigma(l)$  have the same derivation in a context which is either equal to  $\epsilon, \delta$  if  $\omega = \wedge$  or given by the including instruction rule in the derivation tree of  $\pi$ (or  $\pi'$ ). To simplify the proof, we consider  $l, r$  and  $c$  instead of  $\sigma(l), \sigma(r)$  and  $\sigma(c)$ .

**Correction of Inlining.** In the Inlining rule, the condition  $\text{use}(v, i) = 1$  is not a required condition for correction. We have to prove that  $l = \mathbf{let}(v, u, i) \sim r = i[v/u]$ . To prove that  $\pi \sim \pi'$ , we have to verify that for a given  $\epsilon, \delta$ ,  $l$  and  $r$  have the same derivation.

The derivation of  $l$  consists in applying the rule (*let*) :

$$\frac{\langle \epsilon[v \leftarrow t], \delta, i \rangle \mapsto_{bs} \delta' \quad \epsilon(u) \rightarrow_{R^*} t}{\langle \epsilon, \delta, \mathbf{let}(v, t, i) \rangle \mapsto_{bs} \delta'} \text{ (let)}$$

We consider that the derivation of  $r$  is a given  $\delta''$  :

$$\langle \epsilon, \delta, i[v/u] \rangle \mapsto_{bs} \delta''$$

We have to prove that  $\delta' = \delta''$ . As these programs are well-formed, in  $i$ , there is no new declaration for variables contained in  $u$  or for  $v$  therefore  $\Phi(i, \epsilon[v \leftarrow t], \delta, v)(v) = \epsilon[v \leftarrow t](v) \rightarrow_{R^*} t$ . Furthermore,  $\Phi(i, \epsilon, \delta, u)(u) = \epsilon(u) \rightarrow_{R^*} t$ . We can conclude that every evaluation of  $v$  or  $u$  in  $i$  are equal to  $t$  thus replacing  $v$  by  $u$  in  $i$  doesn't change the program behavior,  $\delta' = \delta''$ .

**Correction of ConstantProp.** The proof is very similar to the correction of Inlining but as we have the condition  $u \in \mathcal{T}(\mathcal{F}_0)$ , we know that  $\forall \epsilon, \epsilon(u) \rightarrow_{R^*} u$  thus we can directly deduce that every evaluation of  $v$  or  $u$  in  $i$  are equal to  $u$ .

**Correction of DeadVarElim.** In the DeadVarElim rule,  $\text{use}(v, i) = 0$  i.e.  $v$  is never evaluated in  $i$  therefore  $\langle \epsilon[v \leftarrow t], \delta, i \rangle$  and  $\langle \epsilon, \delta, i \rangle$  have the same reduction  $\delta'$ .

**Correction of LetFusion.** In this rule,  $l = \mathbf{let}(v_1, t, i_1); \mathbf{let}(v_2, t, i_2)$  and  $r = \mathbf{let}(v_1, t, i_1; i_2[v_2/v_1])$ . There is no condition  $c$ . Similarly to the correction of **IfSwapping**, we construct the derivation of  $l$ :

$$\frac{\frac{\overline{\langle \epsilon[v_1 \leftarrow t], \delta, i_1 \rangle \mapsto_{bs} \delta'}}{\langle \epsilon, \delta, \mathbf{let}(v_1, t, i_1) \rangle \mapsto_{bs} \delta'} \quad (let) \quad \frac{\overline{\langle \epsilon[v_2 \leftarrow t], \delta', i_2 \rangle \mapsto_{bs} \delta''}}{\langle \epsilon, \delta', \mathbf{let}(v_2, t, i_2) \rangle \mapsto_{bs} \delta''} \quad (let)}{\langle \epsilon, \delta, \mathbf{let}(v_1, t, i_1); \mathbf{let}(v_2, t, i_2) \rangle \mapsto_{bs} \delta''} \quad (seq)}$$

Then, we derive the right part  $r$ :

$$\frac{\frac{\overline{\langle \epsilon[v_1 \leftarrow t], \delta, i_1 \rangle \mapsto_{bs} \delta'}}{\langle \epsilon[v_1 \leftarrow t], \delta, i_1; i_2[v_2/v_1] \rangle \mapsto_{bs} \delta''} \quad (let) \quad \overline{\langle \epsilon[v_1 \leftarrow t], \delta', i_2[v_2/v_1] \rangle \mapsto_{bs} \delta''}}{\langle \epsilon, \delta, \mathbf{let}(v_1, t, i_1; i_2[v_2/v_1]) \rangle \mapsto_{bs} \delta''} \quad (seq)}$$

Since  $\mathbf{let}(v_1, t, i_1); \mathbf{let}(v_2, t, i_2)$  is in a well-formed program  $\pi$ , when evaluating  $i_2$ ,  $v_2$  is equal to  $t$ . Furthermore, we assume that no variable  $v_1$  occurs in  $i_2$ . Therefore, if  $\langle \epsilon, \delta', \mathbf{let}(v_2, t, i_2) \rangle \mapsto_{bs} \delta''$  then we also have  $\langle \epsilon[v_1 \leftarrow t], \delta', i_2[v_2/v_1] \rangle \mapsto_{bs} \delta''$ . As a consequence,  $\delta''$  is the same for the two derivations.

**Correction of IfFusion.** Since  $c_1$  and  $c_2$  are equivalent,  $c_1$  and  $c_2$  are either true or false. Thus, we have to distinguish two cases.

*Case 1:*  $\epsilon(c_1) \rightarrow_{R^*} \top$  and  $\epsilon(c_2) \rightarrow_{R^*} \top$

First, we calculate the derivation of  $l$  :

$$\frac{\frac{\overline{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'}}{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1) \rangle \mapsto_{bs} \delta'} \quad \epsilon(c_1) \rightarrow_{R^*} \top \quad (iftrue) \quad \frac{\overline{\langle \epsilon, \delta', i_2 \rangle \mapsto_{bs} \delta''}}{\langle \epsilon, \delta', \mathbf{if}(c_2, i_2, i'_2) \rangle \mapsto_{bs} \delta''} \quad \epsilon(c_2) \rightarrow_{R^*} \top \quad (iftrue)}{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1); \mathbf{if}(c_2, i_2, i'_2) \rangle \mapsto_{bs} \delta''} \quad (seq)}$$

Then, we give the derivation of  $r$  :

$$\frac{\frac{\overline{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'}}{\langle \epsilon, \delta, i_1; i_2 \rangle \mapsto_{bs} \delta''} \quad \overline{\langle \epsilon, \delta', i_2 \rangle \mapsto_{bs} \delta''}}{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1; i_2, i'_1; i'_2) \rangle \mapsto_{bs} \delta''} \quad (seq) \quad \frac{\epsilon(c_1) \rightarrow_{R^*} \top}{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1; i_2, i'_1; i'_2) \rangle \mapsto_{bs} \delta''} \quad (iftrue)}$$

*Case 2:*  $\epsilon(c_1) \rightarrow_{R^*} \perp$  and  $\epsilon(c_2) \rightarrow_{R^*} \perp$ , the proof is similar.

**Correction of IfSwapping.** In this rule,  $l = \mathbf{if}(c_1, i_1, \mathbf{nop}); \mathbf{if}(c_2, i_2, \mathbf{nop})$  and  $r = \mathbf{if}(c_2, i_2, \mathbf{nop}); \mathbf{if}(c_1, i_1, \mathbf{nop})$ . Since  $c_1$  and  $c_2$  are  $\pi$ -incompatible, three cases have to be studied:

Case 1:  $\epsilon(c_1) \rightarrow_{R^*} \top$  and  $\epsilon(c_2) \rightarrow_{R^*} \perp$

$$\frac{\frac{\overline{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'} \quad \epsilon(c_1) \rightarrow_{R^*} \top}{} (iftrue) \quad \frac{\overline{\langle \epsilon, \delta', \mathbf{nop} \rangle \mapsto_{bs} \delta'} \quad \epsilon(c_2) \rightarrow_{R^*} \perp}{} (nop) \quad \epsilon(c_2) \rightarrow_{R^*} \perp}{} (iffalse)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, \mathbf{nop}) \rangle \mapsto_{bs} \delta'} \quad \overline{\langle \epsilon, \delta', \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta'}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, \mathbf{nop}); \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta'}}{} (seq)$$

We now consider the program  $\mathbf{if}(c_2, i_2, \mathbf{nop}); \mathbf{if}(c_1, i_1, \mathbf{nop})$ . Starting from the same environment  $\epsilon$  and  $\delta$ , we show that the derivation leads to the same state  $\delta'$ , and thus prove that  $\mathbf{if}(c_1, i_1, \mathbf{nop}); \mathbf{if}(c_2, i_2, \mathbf{nop})$  and  $\mathbf{if}(c_2, i_2, \mathbf{nop}); \mathbf{if}(c_1, i_1, \mathbf{nop})$  are equivalent:

$$\frac{\frac{\overline{\langle \epsilon, \delta, \mathbf{nop} \rangle \mapsto_{bs} \delta} \quad \epsilon(c_2) \rightarrow_{R^*} \perp}{} (nop) \quad \epsilon(c_2) \rightarrow_{R^*} \perp}{} (iffalse) \quad \frac{\overline{\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'} \quad \epsilon(c_1) \rightarrow_{R^*} \top}{} (iftrue) \quad \overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, \mathbf{nop}) \rangle \mapsto_{bs} \delta'}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta} \quad \overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, \mathbf{nop}) \rangle \mapsto_{bs} \delta'}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_2, i_2, \mathbf{nop}); \mathbf{if}(c_1, i_1, \mathbf{nop}) \rangle \mapsto_{bs} \delta'}}{} (seq)$$

Since  $\pi$  and  $\pi'$  are well-formed, their derivation in a given context are unique (Proposition 2).  $\langle \epsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'$  is part of these derivation trees, so it is unique, and  $\delta'$  is identical in both derivations.

Case 2:  $\epsilon(c_1) \rightarrow_{R^*} \perp$  and  $\epsilon(c_2) \rightarrow_{R^*} \top$ , the proof is similar.

Case 3:  $\epsilon(c_1) \rightarrow_{R^*} \perp$  and  $\epsilon(c_2) \rightarrow_{R^*} \perp$ , the proof is similar.

**Correction of IfInterleaving.** As for the previous rule, since  $c_1$  and  $c_2$  are  $\pi$ -incompatible, we have to consider three cases. We present the second one which is the most interesting.

Case 2:  $\epsilon(c_1) \rightarrow_{R^*} \perp$  and  $\epsilon(c_2) \rightarrow_{R^*} \top$

$$\frac{\frac{\overline{\langle \epsilon, \delta, i'_1 \rangle \mapsto_{bs} \delta'} \quad \epsilon(c_1) \rightarrow_{R^*} \perp}{} (iffalse) \quad \frac{\overline{\langle \epsilon, \delta', i_2 \rangle \mapsto_{bs} \delta''} \quad \epsilon(c_2) \rightarrow_{R^*} \top}{} (iftrue) \quad \overline{\langle \epsilon, \delta', \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1) \rangle \mapsto_{bs} \delta'} \quad \overline{\langle \epsilon, \delta', \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1); \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''}}{} (seq)$$

$$\frac{\frac{\overline{\langle \epsilon, \delta, i'_1 \rangle \mapsto_{bs} \delta'} \quad \overline{\langle \epsilon, \delta', \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''}}{} (seq) \quad \overline{\langle \epsilon, \delta', i_2 \rangle \mapsto_{bs} \delta''} \quad \epsilon(c_2) \rightarrow_{R^*} \top}{} (iftrue)}{\overline{\langle \epsilon, \delta, i'_1; \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''} \quad \overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1); \mathbf{if}(c_2, i_2, \mathbf{nop}) \rangle \mapsto_{bs} \delta''}}{} (seq)}{\overline{\langle \epsilon, \delta, \mathbf{if}(c_1, i_1, i'_1; \mathbf{if}(c_2, i_2, \mathbf{nop})) \rangle \mapsto_{bs} \delta''} \quad \epsilon(c_1) \rightarrow_{R^*} \perp}{} (iffalse)$$

The proof of Case 1 and Case 3 are similar.

## 4.2 Time and space reduction

In this section we show that the optimized code is both more efficient, and smaller than the initial program. For that, we consider two measures:

- the *size* of a program  $\pi$  is the number of instructions which constitute the program,
- the *efficiency* of a program  $\pi$  is determined by the number of tests and assignments which are performed at run-time.

It is quite easy to verify that each transformation rule does not increase the size of the program: `DeadVarElim`, `ConstProp`, `Inlining`, and `LetFusion` decrease the size of a program, whereas `IfFusion`, `IfInterleaving` and `IfSwapping` maintain the size of the transformed program.

It is also clear that no transformation can reduce the efficiency of a given program:

- each application of `DeadVarElim`, `ConstProp`, and `Inlining` reduces by one the number of assignment that can be performed at run time,
- `IfFusion` reduces by one the number of tests,
- `IfInterleaving` also decreases the number of tests when the first alternative is chosen. Otherwise, there is no optimization,
- `IfSwapping` does not modify the efficiency of a program.

The program transformation presented in Section 3 is an optimization which improves the efficiency of a given program, without increasing its size. Similarly to [8], this result is interesting since it allows to generate efficient pattern matching implementations whose size is linear in the number and size of patterns.

## 5 Experimental Results

The presented algorithm (rules and strategy) has been implemented and integrated into Tom (see Figure 1). We have selected several representative programs and measured the effect of optimization in practice:

	Fibonacci	Eratosthene	Gomoku	Langton	Nspk	Structure
without optimization	21.3s	174.0s	70.0s	15.7s	1.7s	12.3s
with optimization	20.0s	2.8s	30.4s	1.4s	1.2s	11.3s

`Fibonacci` computes several times the 18<sup>th</sup> Fibonacci number. `Eratosthene` computes primes numbers up to 1000, using associative list matching. The improvement comes from the `Inlining` rules which avoids computing a substitution unless the rule applies (i.e. the conditions are verified). `Gomoku` looks for five pawn on a go board, using list matching. This example contains more than 40 patterns and illustrates the interest of test-sharing. `Langton` is a program which computes the 1000<sup>th</sup> iteration of a cellular automaton, using pattern matching to implement the transition function. This example contains more than 100 (ground) patterns. The optimized program is optimal in the sense that a pair

(position,symbol) is never tested more than once. `Nspk` implements the verification of the Needham-Schroeder Public-Key Protocol. `Structure` is a prover for the Calculus of Structures where the inference is performed by pattern matching and rewriting. All these examples are available on the `Tom` web page<sup>1</sup>. Some of these examples have been implemented in OCaml. `Fibonacci`, `Eratosthene`, and `Langton` are respectively executed in 4.0, 0.7 and 2.3 seconds.

These benchmarks show that the proposed approach is effective in practice and allows `Tom` to become competitive with state of the art implementations such as OCaml. We should remind that `Tom` is not dedicated to a unique language. In particular, the fact that data-structure can be user-defined prevents us from using the `switch` instruction. Moreover, all the presented examples have been implemented in Java and executed on a PowerMac 2 GHz.

## 6 Conclusion and future works

In this paper, we have presented a new approach to compile efficiently pattern-matching. This method is based on well-attested program optimization methods. Separating compilation and optimization in order to keep modularity and to facilitate extensions is long-established in compilation community. Furthermore, using program transformation to realize optimization is an elegant way as the correction proof is facilitated contrary to complex algorithms.

As yet, the algorithms proposed to compile pattern-matching [5,9,2,8] were adapted to very much specific forms of matching and assumed properties on target language and term representation. These algorithms hardly conform to evolutions, such as matching modulo AC or non-linear patterns. All these restrictions lead us to give an other approach for `Tom` compiler by compiling simply pattern-matching and optimizing separately. The set of optimization rules presented in this paper is not specific to pattern-matching and can be applied to any program. However, some rules such as `IfSwapping` reply to specific problems related to pattern-matching compilation and formalizing them is an other contribution to this paper. The section 4 shows how simply the correction of the system can be proved and justifies the choice of a method based on program transformations. Moreover, as the `Tom` compiler is bootstrapped, the optimization rules have been implemented in a very natural way using pattern-matching so this work demonstrates how `Tom` tool is well adapted for transforming programs and more generally for compiler construction.

This paper shows that using program transformation rules to optimize pattern-matching is an efficient solution, in respect to algorithms based on automata. The implementation of this work conjugated with the validation tool of `Tom` [10] give us an efficient adaptative certified compiler of pattern-matching.

## References

1. Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming*

---

<sup>1</sup> <http://tom.loria.fr>

- Language Implementation and Logic Programming*, number 528, pages 1–13. Springer Verlag, 1991.
2. Lennart Augustsson. Compiling pattern matching. In *Proceedings of a conference on Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.
  3. Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
  4. Peter Borovanský, Claude Kirchner Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In *WRLA '98: Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, volume 15, Pont-à-Mousson (France), 1998. Electronic Notes in Theoretical Computer Science.
  5. Luca Cardelli. Compiling a functional language. In *LFP'84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, 1984.
  6. Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), 1996. Electronic Notes in Theoretical Computer Science.
  7. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.
  8. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP'01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.
  9. Albert Gräf. Left-to-right tree pattern matching. In *RTA'91: Proceedings of the 4th international conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 323–334. SV, 1991.
  10. Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In Pedro Barahone and Amy Felty, editors, *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–197. ACM, July 2005.
  11. David Lacey, Neil Jones, Eric Van Wyk, and Carl Christian Frederikson. Proving correctness of compiler optimizations by temporal logic. *Higher-Order and Symbolic Computation*, 17(2), 2004.
  12. Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
  13. R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, 1995.
  14. Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.
  15. Eelco Visser, Zine el Abidine Benaïssa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 13–26, 1998.