Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006)

Optimizing Pattern Matching Compilation By Program Transformation

Emilie Balland and Pierre-Etienne Moreau

14 pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Optimizing Pattern Matching Compilation By Program Transformation

Emilie Balland[1] **and Pierre-Etienne Moreau**[2]

[1]UHP & LORIA at Nancy, France
Emilie.Balland@loria.fr
[2]INRIA & LORIA at Nancy, France
Pierre-Etienne.Moreau@loria.fr

Abstract: Motivated by the promotion of rewriting techniques and their use in major industrial applications, we have designed Tom: a pattern matching layer on top of conventional programming languages. The main originality is to support pattern matching against native data-structures like objects or records. While crucial to the efficient implementation of functional languages as well as rewrite rule based languages, in our case, this combination of algebraic constructs with arbitrary native data-structures makes the pattern matching algorithm more difficult to compile. In particular, well-known many-to-one automaton-based techniques cannot be used. We present a two-stages approach which first compiles pattern matching constructs in a naive way, and then optimize the resulting code by program transformation using rewriting. As a benefit, the compilation algorithm is simpler, easier to extend, and the resulting pattern matching code is almost as efficient as best known implementations.

Keywords: Pattern matching, Optimization, Program transformation

1 Introduction to Tom

Pattern matching is an elegant high-level construct which appears in many programming languages. Similarly to method dispatching in object oriented languages, it is essential in functional languages like Caml, Haskell, or SML. It is part of the main execution mechanism in rewrite rule based languages like ASF+SDF, ELAN, Maude, or Stratego.

In this paper, we present Tom[1] whose goal, similarly to Prop [Leu96] or Pizza [OW97], is to integrate the notion of pattern matching into classical languages such as C and Java. Following the first ideas presented in [MRV03], illustrated in Figure 1, 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 preprocessor.

In order to understand the choices we have made when designing the pattern matching algorithm, it is important to consider Tom as a generic and *partial* compiler (like a pre-processor) which does not have any information about the host-language. In [BKM06], Tom programs are

[1] `http://tom.loria.fr`

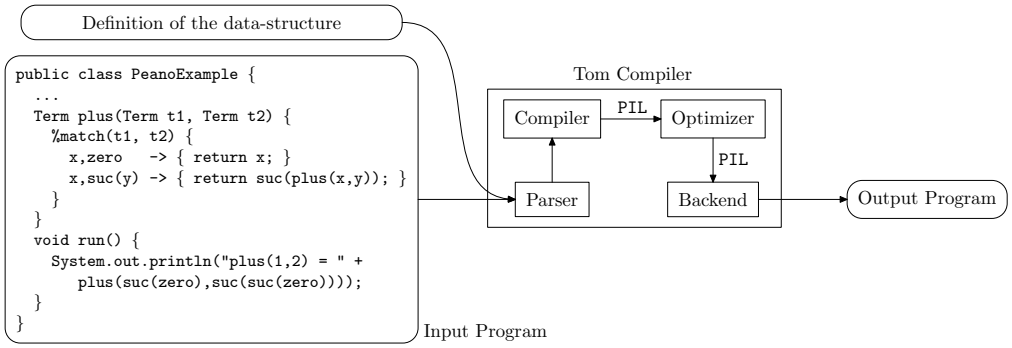

Figure 1: General architecture of Tom: the compiler generates an intermediate PIL program which is optimized before being pretty-printed by the back-end into the host-language.

described as islands anchored in host programs and the link between the two languages is formally defined in a generalized framework. In particular, the data-structure, against which the pattern matching is performed, *is not fixed*. In some sense, the data-structure is a parameter of the pattern matching, see [KMR05] for more details. In practice, this means that a description of the data-structure (a mapping) has to be given to explain Tom how to access subterms for example.

In this paper, we present how the introduced pattern matching construct is compiled, using a program transformation approach. There exists several methods [Car84, Aug85, Grä91, FM01] to efficiently compile pattern matching. The simplest ones, called *one-to-one*, inspect and compile each pattern independently. A more efficient approach consists in considering the system globally and building a discrimination network. These methods are called *many-to-one*, and they usually consist of three phases: constructing an automaton, optimizing it, and finally generating the implementation code. There are two main approaches to construct a matching automaton: one based on decision trees [Car84, Grä91] and the other on backtracking automata [Aug85]. These two approaches emphasize the unavoidable compromise between speed and memory space [SRR95].

In our case, we cannot assume that a function symbol (i.e. a node of a tree) is represented by an integer, like it is commonly done in other implementations of pattern matching. Therefore, the classical `switch/case` instruction can no longer be used to perform the discrimination. Since Tom supports several languages, it is also not possible to use an exception mechanism or a jump statement, like it can be done in ML compilers [Jon87].

The approach chosen in Tom is to keep the optimization phase separated from the *one-to-one* compilation phase. This allows us to design algorithms which are generic, simpler to implement, easier to extend and maintain, and that can be formally certified [KMR05]. In addition, this work allows to generate efficient implementations. In Section 2, we present the compilation algorithm and its intermediate language PIL. In Section 3, we introduce a set of rules which describes the optimizer and a strategy to efficiently apply them. In Section 4 we show that the optimizations are correct and improve the program in execution and size. Finally, in Section 5, some experimental results are given for several revealing examples. This paper assumes some

familiarity with term rewriting notations introduced in [BM05].

2 Compilation

To be data-structure independent and support several host-languages, Tom instructions, like `%match`, are compiled into an intermediate language code, called `PIL`, before being translated into the selected host-language. To compile the `%match` construct, we consider each rule independently.

Contrary to *many-to-one* algorithms which construct decision trees or pattern automata, given a pattern, it is traversed top-down and left-to-right. Nested if-then-else constructs are generated to ensure that constructors of the pattern effectively occur in the subject at a correct position. This technique is inefficient because, for a set of rules, identical tests may be repeatedly performed. The worst-case complexity is thus the product of the number of rules and the size of the subject.

The nested if-then-else are expressed in an intermediate language called `PIL`, whose syntax is given in Figure 2. Note that `PIL` has both functional and imperative flavors: the assignment instruction $\texttt{let}(\textbf{variable}, \langle term\rangle, \langle instr\rangle)$ defines a scoped unmodifiable variable, whereas the sequence instruction $\langle instr\rangle$; $\langle instr\rangle$ comes from imperative languages. A last particularity of `PIL` comes from the $\texttt{hostcode}(\ldots)$ instruction which is used to abstract part of code written in the underlying host-language. This instruction is parameterized by a list of `PIL`-variables which are used in this part of host-code.

$$
\begin{array}{lcl}
\texttt{PIL} & ::= & \langle instr\rangle \\
\textbf{symbol} & ::= & f \in \mathscr{F} \\
\textbf{variable} & ::= & x \in \mathscr{X} \\
\langle term\rangle & ::= & t \in \mathscr{T}(\mathscr{F},\mathscr{X}) \\
 & | & \texttt{subterm}_f(\langle term\rangle, n) \\
 & & (f \in \mathscr{F} \wedge n \in \mathbb{N})
\end{array}
\qquad
\begin{array}{lcl}
\langle expr\rangle & ::= & \texttt{true} \mid \texttt{false} \\
 & | & \texttt{eq}(\langle term\rangle, \langle term\rangle) \\
 & | & \texttt{is\_fsym}(\langle term\rangle, \textbf{symbol}) \\
\langle instr\rangle & ::= & \texttt{let}(\textbf{variable}, \langle term\rangle, \langle instr\rangle) \\
 & | & \texttt{if}(\langle expr\rangle, \langle instr\rangle, \langle instr\rangle) \\
 & | & \langle instr\rangle; \langle instr\rangle \\
 & | & \texttt{hostcode}(\textbf{variable}*) \\
 & | & \texttt{nop}
\end{array}
$$

Figure 2: `PIL` syntax

Similarly to functional programming languages, given a signature $\mathscr{F}$ and a set of variables $\mathscr{X}$, the considered `PIL` language can directly handle *terms*, boolean values (`true`, `false`), and perform operations like checking that a given term t is rooted by a symbol f ($\texttt{is\_fsym}(t,f)$), accessing to the n-th child of a term t ($\texttt{subterm}_f(t,n)$), or comparing two terms ($\texttt{eq}(t_1,t_2)$). The implementation of $\texttt{subterm}_f$, `eq` and `is_fsym` is given by the mapping which describes data-structures. To support the intuition, examples of Tom and `PIL` code are given in Figure 3.

We define `PIL` semantics as in [KMR05] by a big-step semantics *à la* Kahn. To represent a substitution, we model an environment by a stack of assignments of terms to variables. The set of environments is noted $\mathscr{E}nv$. The reduction relation of the big-step semantics is expressed on tuples $\langle \varepsilon, \delta, i\rangle$ where ε is an environment, δ is a list of pairs (environment, host-code), and i is an instruction. Thanks to δ, we can keep track of the executed host-code blocks within their environment: the environment associated to each host-code construct gives the instances of all variables

Tom code:

```
...Java code...
...
%match(Term t) {
    f(a) ⇒ { print(...); }
    g(x)⇒ { print(...x...); }
    f(b) ⇒ { print(...); }
}
...
...Java code...
```

Generated PIL code:

```
hostcode(...) ;
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)))
   nop) ;
if(is_fsym (t,f),let(t1,subtermf(t,1),
   if(is_fsym(t1,b),hostcode(),nop)),
   nop) ;
hostcode(...) ;
```

Figure 3: The left column shows a Tom program which contains three patterns: $f(a), g(x)$, and $f(b)$, where x is a variable. As an example, when the second pattern matches t, this means that t is rooted by the symbol g, and the variable x is instantiated by its immediate subterm. The right column shows the corresponding PIL code generated by Tom. We can notice that this code is not optimal, but will hopefully be optimized by transformation rules afterwards.

which appear in the block. A complete definition of the semantics can be found in [BM05].

$$\langle \varepsilon, \delta, i \rangle \mapsto_{bs} \delta', \text{ with } \varepsilon \in \mathscr{E}nv, \delta, \delta' \in [\mathscr{E}nv, \langle instr \rangle]^*, \text{ and } i \in \langle instr \rangle$$

As PIL programs are predominantly constituted of if-then-else statements, the optimization rules will depend of the evaluation of expressions $e \in \langle expr \rangle$. In the following we introduce the notions of equivalence and incompatibility for expressions, and we consider two functions `eval` and Φ. `eval` is a function which given an environment ε and an expression e evaluates e in ε to obtain a value (i.e `true` for true or `false` for false). Given an environment ε and a host-code list δ, the evaluation of a program $\pi \in$ PIL results in a host-code list: $\langle \varepsilon, \delta, \pi \rangle \mapsto_{bs} \delta'$. During this evaluation, expressions e, subterm of π, are evaluated in environments ε'. We call Φ the function that associates such an environment ε' to a sub-expression e of π: $\varepsilon' = \Phi(\pi, e, \varepsilon, \delta)$. More formal definitions can be found in [BM05].

Definition 1 Given a program π, two expressions e_1 and e_2 are said π-equivalent, and noted $e_1 \sim_\pi e_2$, if for all starting environment ε, δ, $\texttt{eval}(\varepsilon_1, e_1) = \texttt{eval}(\varepsilon_2, e_2)$ where $\varepsilon_1 = \Phi(\pi, e_1, \varepsilon, \delta)$ and $\varepsilon_2 = \Phi(\pi, e_2, \varepsilon, \delta)$.

Definition 2 Given a program π, two expressions e_1 and e_2 are said π-incompatible, and noted $e_1 \perp_\pi e_2$, if for all starting environment ε, δ, $\texttt{eval}(\varepsilon_1, e_1) \neq \texttt{eval}(\varepsilon_4, e_2)$ where $\varepsilon_1 = \Phi(\pi, e_1, \varepsilon, \delta)$ and $\varepsilon_2 = \Phi(\pi, e_2, \varepsilon, \delta)$.

We can now define two conditions which are sufficient to determine whether two expressions are π-equivalent or π-incompatible. Propositions 1 and 2 are interesting because the problem is generally undecidable [RKS99], but here, conditions can be easily used in practice. Indeed cond1 which ensures that the two expressions are evaluated in the same environment is

easy to be checked because of PIL language restrictions and cond2 is a purely syntactic condition. Proofs of these propositions are in [BM05].

Proposition 1 *Given a program π and two expressions $e_1, e_2 \in \langle expr \rangle$ at different positions in, we have $e_1 \sim_\pi e_2$ if: $\forall \varepsilon, \delta$, $\Phi(\pi, e_1, \varepsilon, \delta) = \Phi(\pi, e_2, \varepsilon, \delta)$ (cond1) and $e_1 = e_2$ (cond2).*

The equality $=$ correspond to syntactic equality and the two considered expressions are in a different position in the program so the two environments of evaluation are not trivially equal.

Proposition 2 *Given a program π and two expressions $e_1, e_2 \in \langle expr \rangle$, we have $e_1 \perp_\pi e_2$ if: $\forall \varepsilon, \delta$, $\Phi(\pi, e_1, \varepsilon, \delta) = \Phi(\pi, e_2, \varepsilon, \delta)$ (cond1) and $\texttt{incompatible}(e_1, e_2)$ (cond2), where $\texttt{incompatible}$ is defined as follows:*

$$
\begin{array}{lcll}
\texttt{incompatible}(e_1, e_2) & = & \textit{match } e_1, e_2 \textit{ with} & \\
 & | & \texttt{false}, \texttt{true} & \rightarrow \top \\
 & | & \texttt{true}, \texttt{false} & \rightarrow \top \\
 & | & \texttt{is\_fsym}(t, f_1), \texttt{is\_fsym}(t, f_2) & \rightarrow \top \quad \textit{if} \quad f_1 \neq f_2 \\
 & | & \_, \_ & \rightarrow \bot
\end{array}
$$

3 Optimization

An optimization is a transformation which reduces the size of code (*space optimization*) or the execution time (*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 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 $i_1 \rightarrow i_2$ IF c.

Definition 3 An optimization rule $i_1 \rightarrow i_2$ IF c rewrites a program π into a program π' if there exists a position ω and a substitution σ 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. $\sigma(e_1) \perp_{\pi_{|\omega}} \sigma(e_2)$).

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.

3.1.1 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 $i[v/t]$).

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

3.1.2 Dead variable elimination and Inlining.

Using a simple static analysis, these optimizations eliminate useless assignments:

$$\texttt{DeadVarElim:}\ \texttt{let}(v,t,i) \rightarrow i\ \texttt{IF}\ \texttt{use}(v,i) = 0$$
$$\texttt{Inlining:}\ \texttt{let}(v,t,i) \rightarrow i[v/t]\ \texttt{IF}\ \texttt{use}(v,i) = 1$$

where $\texttt{use}(v,i)$ is a function that computes an upper bound on the number of occurences of a variable v in an instruction i.

3.1.3 Fusion.

The following rule merges two successive `let` which assign the same value to two different variables. This kind of optimization rarely applies on human written code, but in the context of pattern matching compilation (see Figure 3), this case often occurs. By merging the bodies, this allows to recursively perform some optimizations on subterms.

$$\texttt{LetFusion:}\ \texttt{let}(v_1,t_1,i_1);\texttt{let}(v_2,t_2,i_2) \rightarrow \texttt{let}(v_1,t_2,i_1;i_2[v_2/v_1])\ \texttt{IF}\ t_1 \sim t_2$$

Note that the terms t_1 and t_2 must be compatible to ensure that values of v_1 and v_2 are the same at run time. We also suppose that $\texttt{use}(v_1,i_2) = 0$. 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/case` 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.

3.2.1 Fusion.

The fusion of two conditional adjacent blocks reduces the number of tests. This fusion is possible only when the two conditions are π-equivalent. Remind that the notion of π-equivalence means that the evaluation of the two conditions in a given program are the same (see Definition 1):

$$\texttt{IfFusion:}\ \texttt{if}(c_1,i_1,i_1');\texttt{if}(c_2,i_2,i_2') \rightarrow \texttt{if}(c_1,i_1;i_2,i_1';i_2')\ \texttt{IF}\ c_1 \sim c_2$$

To evaluate $c_1 \sim c_2$ (i.e. $c_1 \sim_\pi c_2$ with π the redex of the rule), we use Proposition 1. The condition $\Phi(\pi,\varepsilon,\delta,e_1) = \Phi(\pi,\varepsilon,\delta,e_2)$ (cond1) is trivially verified because the semantics of the sequence instruction preserves the environment ($\forall\delta,\varepsilon,\ \Phi(\pi,\varepsilon,\delta,i_1;i_2) = \Phi(\pi,\varepsilon,\delta,i_1) = \Phi(\pi,\varepsilon,\delta,i_2)$) and then $\forall\delta,\varepsilon,\ \Phi(\pi,\varepsilon,\delta,\sigma(c_1)) = \Phi(\pi,\varepsilon,\delta,\sigma(c_2))$. We just have to verify that $e_1 = e_2$ (cond2), which is easy.

3.2.2 Interleaving.

As matching code consists of a sequence of conditional blocks, we would like to optimize blocks with π-incompatible conditions (see Definition 2). Some parts of the code cannot be both executed in a given environment, 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:

$$\texttt{if}(c_1,i_1,i'_1);\texttt{if}(c_2,i_2,i'_2) \rightarrow \texttt{if}(c_1,i_1;i'_2,i'_1;\texttt{if}(c_2,i_2,i'_2))\ \texttt{IF}\ c_1 \perp c_2$$
$$\texttt{if}(c_1,i_1,i'_1);\texttt{if}(c_2,i_2,i'_2) \rightarrow \texttt{if}(c_2,i'_1;i_2,\texttt{if}(c_1,i_1,i'_1);i'_2)\ \texttt{IF}\ c_1 \perp c_2$$

As for the equivalence in the `IfFusion` rule, to evaluate $c_1 \perp c_2$, we just have to verify that e_1 and e_2 are incompatible (cond2). Note that the two presented rules are not right-linear, therefore 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`.

`IfInterleaving:` $\texttt{if}(c_1,i_1,i'_1);\texttt{if}(c_2,i_2,\texttt{nop}) \rightarrow \texttt{if}(c_1,i_1,i'_1;\texttt{if}(c_2,i_2,\texttt{nop}))\ \texttt{IF}\ c_1 \perp c_2$

$$\texttt{if}(c_1,i_1,\texttt{nop});\texttt{if}(c_2,i_2,i'_2) \rightarrow \texttt{if}(c_2,i_2,\texttt{if}(c_1,i_1,\texttt{nop});i'_2)\ \texttt{IF}\ c_1 \perp c_2$$

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. In practice, we only use the first one labelled by `IfInterleaving`. 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:

$$\texttt{if}(c_1,i_1,\texttt{nop});\texttt{if}(c_2,i_2,\texttt{nop}) \rightarrow \texttt{if}(c2,i_2,\texttt{if}(c_1,i_1,\texttt{nop}))\ \texttt{IF}\ c_1 \perp c_2$$

As c_1 and c_2 are π-incompatible, we have the following equivalence:

$$\texttt{if}(c_2,i_2,\texttt{if}(c_1,i_1,\texttt{nop})) \equiv \texttt{if}(c_2,i_2,\texttt{nop});\texttt{if}(c_1,i_1,\texttt{nop})$$

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 bring closer blocks subject to be merged thanks to the strategy presented in the next section.

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

3.3 Application strategy

From the rules presented in Section 3.1 and 3.2, we define a rewrite system. Without strategy, this system is clearly not confluent and not terminating. For example, 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 last two patterns. This would result in the following sub-optimal program:

$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(t,f),\texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{a}),\texttt{hostcode}(),\texttt{nop})),\texttt{nop})\ ;\\ \texttt{if}(\texttt{is\_fsym}(t,g),\texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{g}(\texttt{t},1),\texttt{let}(\texttt{x},\texttt{t}_1,\texttt{hostcode}(\texttt{x})))\\ \qquad\qquad \texttt{if}(\texttt{is\_fsym}(t,f),\texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{b}),\texttt{hostcode}(),\texttt{nop})),\texttt{nop})\end{array}$$

The `IfSwapping` and `IfFusion` rules can no longer be applied to share the $\texttt{is\_fsym}(t,f)$ tests. This order of application is not optimal. As we want to grant `IfFusion`, the interleaving rule must be applied afterward, when no more optimization is possible.

The second matter is to ensure 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 to 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 noted $<$ (a lexicographic order for example). The second approach is more efficient: similarly to a swap-sort algorithm it ensures the termination of the algorithm. In this way, we obtain a new `IfSwapping` rule:

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

Using basic strategy operators such as $\mathit{Innermost}(s)$ (which applies s as many times as possible, starting from the leaves), $s_1 \mid s_2$ (which applies s_1 or s_2 indifferently), $\mathit{repeat}(s)$ (which applies s as many times as possible, returning the last unfailing result), and $s_1\ ;\ s_2$ (which applies s_1, and then s_2 if s_1 did not fail), we can define a strategy which describes how the considered rewrite system should be applied to normalize a `PIL` program:

$$\begin{array}{l}\mathit{Innermost}(\\ \quad \mathit{repeat}(\texttt{ConstProp} \mid \texttt{DeadVarElim} \mid \texttt{Inlining} \mid \texttt{LetFusion} \mid \texttt{IfFusion} \mid \texttt{IfSwapping})\ ;\\ \quad \mathit{repeat}(\texttt{IfInterleaving})\\ )\end{array}$$

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

$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(t,f),\ \ \texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{a}),\texttt{hostcode}(),\texttt{nop}))\\ \qquad\qquad\qquad ;\ \texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{b}),\texttt{hostcode}(),\texttt{nop})),\texttt{nop})\ ;\\ \texttt{if}(\texttt{is\_fsym}(t,g),\texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{g}(\texttt{t},1),\texttt{let}(\texttt{x},\texttt{t}_1,\texttt{hostcode}(\texttt{x}))),\texttt{nop})\end{array}$$

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` ($\texttt{is\_fsym}(t_1,a)$ and $\texttt{is\_fsym}(t_1,b)$ are π-incompatible). This results in the following program:

$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(t,f),\texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\\ \qquad\qquad\qquad \texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{a}),\texttt{hostcode}(),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{b}),\texttt{hostcode}(),\texttt{nop}))),\texttt{nop})\ ;\\ \texttt{if}(\texttt{is\_fsym}(t,g),\texttt{let}(\texttt{x},\texttt{subterm}_\texttt{g}(\texttt{t},1),\texttt{hostcode}(\texttt{x})),\texttt{nop})\end{array}$$

Since $\texttt{is\_fsym}(t, f)$ and $\texttt{is\_fsym}(t, g)$ are π-incompatible, we can apply a step of `IfInterleaving`, and get the irreducible following program:

$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(t,f),\\ \quad \texttt{let}(\texttt{t}_1,\texttt{subterm}_\texttt{f}(\texttt{t},1),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{a}),\texttt{hostcode}(),\texttt{if}(\texttt{is\_fsym}(\texttt{t}_1,\texttt{b}),\texttt{hostcode}(),\texttt{nop}))),\\ \quad \texttt{if}(\texttt{is\_fsym}(t,g),\texttt{let}(\texttt{x},\texttt{subterm}_\texttt{g}(\texttt{t},1),\texttt{hostcode}(\texttt{x})),\texttt{nop})\end{array}$$

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 this 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 4 Given π_1 and π_2 two well-formed `PIL` programs, they are semantically equivalent, noted $\pi_1 \sim \pi_2$, when:

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

Definition 5 A transformation rule r is correct if for all well-formed program π, r rewrites π in π' (Definition 3) implies that $\pi \sim \pi'$ (Definition 4).

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

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

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

- when ω is the empty position (which corresponds to the root), we have to compare the derivation tree of $\pi = \sigma(l)$ and $\pi' = \sigma(r)$,
- otherwise, we consider the derivation of π (resp. π'): 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 $\pi_{|\omega} = \sigma(l)$ and $\sigma(r)$ have the same derivation in a given context:

- equal to ε, δ when ω is the empty position,
- otherwise, we have to consider the instruction i which immediately contains $\sigma(l)$ (resp. $\sigma(r)$). The context is defined by the context in which i is evaluated in the derivation tree of π (resp. π').

In the following, we give one representative proof of correction: `IfSwapping`[2]. To simplify the proof we consider l, r and c instead of $\sigma(l), \sigma(r)$ and $\sigma(c)$. In this rule, $l = \texttt{if}(c_1, i_1, \texttt{nop}); \texttt{if}(c_2, i_2, \texttt{nop})$ and $r = \texttt{if}(c_2, i_2, \texttt{nop}); \texttt{if}(c_1, i_1, \texttt{nop})$. To prove that $\pi \sim \pi'$, we have to verify that for a given ε, δ, l and r have the same derivation. Since c_1 and c_2 are π-incompatible, three cases have to be studied:

Case 1: $\texttt{eval}(\varepsilon, c_1) = \texttt{true}$ and $\texttt{eval}(\varepsilon, c_2) = \texttt{false}$

$$\frac{\dfrac{\overline{\langle \varepsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'} \quad \texttt{eval}(\varepsilon, c_1) = \texttt{true}}{\langle \varepsilon, \delta, \texttt{if}(c_1, i_1, \texttt{nop}) \rangle \mapsto_{bs} \delta'} \quad \dfrac{\overline{\langle \varepsilon, \delta', \texttt{nop} \rangle \mapsto_{bs} \delta'} \quad \texttt{eval}(\varepsilon, c_2) = \texttt{false}}{\langle \varepsilon, \delta', \texttt{if}(c_2, i_2, \texttt{nop}) \rangle \mapsto_{bs} \delta'}}{\langle \varepsilon, \delta, \texttt{if}(c_1, i_1, \texttt{nop}); \texttt{if}(c_2, i_2, \texttt{nop}) \rangle \mapsto_{bs} \delta'}$$

We now consider the program $\texttt{if}(c_2, i_2, \texttt{nop}); \texttt{if}(c_1, i_1, \texttt{nop})$.

Starting from the same environment ε and δ, we show that the derivation leads to the same state δ', and thus prove that $\texttt{if}(c_1, i_1, \texttt{nop}); \texttt{if}(c_2, i_2, \texttt{nop})$ and $\texttt{if}(c_2, s_2, \texttt{nop}); \texttt{if}(c_1, s_1, \texttt{nop})$ are equivalent:

$$\frac{\dfrac{\overline{\langle \varepsilon, \delta, \texttt{nop} \rangle \mapsto_{bs} \delta} \quad \texttt{eval}(\varepsilon, c_2) = \texttt{false}}{\langle \varepsilon, \delta, \texttt{if}(c_2, i_2, \texttt{nop}) \rangle \mapsto_{bs} \delta} \quad \dfrac{\overline{\langle \varepsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'} \quad \texttt{eval}(\varepsilon, c_1) = \texttt{true}}{\langle \varepsilon, \delta, \texttt{if}(c_1, i_1, \texttt{nop}) \rangle \mapsto_{bs} \delta'}}{\langle \varepsilon, \delta, \texttt{if}(c_2, i_2, \texttt{nop}); \texttt{if}(c_1, i_1, \texttt{nop}) \rangle \mapsto_{bs} \delta'}$$

Since π and π' are well-formed, their derivation in a given context are unique (see [BM05]). $\langle \varepsilon, \delta, i_1 \rangle \mapsto_{bs} \delta'$ is part of these derivation trees, so it is unique, and δ' is identical in both derivations.

Case 2: $\texttt{eval}(\varepsilon, c_1) = \texttt{false}$ and $\texttt{eval}(\varepsilon, c_2) = \texttt{true}$, the proof is similar.

Case 3: $\texttt{eval}(\varepsilon, c_1) = \texttt{false}$ and $\texttt{eval}(\varepsilon, c_2)\texttt{false}$, the proof is similar.

4.2 Time and space reduction

To show that the optimized code is both more efficient, and smaller than the initial program, we consider two measures:

- the *size* of a program π is the number of instructions which constitute the program,

[2] The other proofs can be found in [BM05]

- the *efficiency* of a program π 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 [FM01], 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 Tom compiler is written in Tom and Java. Therefore, the presented algorithm described using rules and strategies, has been implemented in Tom. As illustrated Figure 1, the optimizer is just an extra phase of the compiler, which is now integrated into the main distribution using the strategy given in Section 3.3. In order to illustrate the efficiency of the compiler we have selected several representative programs and measured the effect of optimization in practice:

	`Fibonnacci`	`Eratosthene`	`Langton`	`Gomoku`	`Nspk`	`Structure`
`Tom Java`	21.3 *s*	174.0 *s*	15.7 *s*	70.0 *s*	1.7 *s*	12.3 *s*
`Tom Java Optimized`	20.0 *s*	2.8 *s*	1.4 *s*	30.4 *s*	1.2 *s*	11.3 *s*

- `Fibonacci` computes 500 times the 18^{th} Fibonacci number, using a Peano representation. On this example, the optimizer has a small impact because the time spent in matching is smaller than the time spent in allocating successors and managing the memory.
- `Eratosthene` computes prime 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).
- `Langton` is a program which computes the 1000^{th} iteration of a cellular automaton, using pattern matching to implement the transition function. This example is interesting because it contains more than 100 (ground) patterns. Starting from a simple one-to-one pattern matching

algorithm, the optimizer performs program transformations such that a pair (position,symbol) is never tested more than once. This interesting property, which characterizes deterministic automata based approaches, can unfortunately not be generalized to any program.
- `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.
- `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.

The following table gives some comparisons with other well known implementations.

	Fibonnacci	Eratosthene	Langton
Tom Java Optimized	20.0 *s*	2.8 *s*	1.4 *s*
Tom C Optimized	0.95 *s*	0.36 *s*	0.84 *s*
OCaml	0.44 *s*	0.7 *s*	1.36 *s*
ELAN	0.77 *s*	0.8 *s*	1.26 *s*

All these examples are available on the Tom web page. The measures have been done on a PowerMac 2 GHz, using Java 1.4.2, gcc 4.0, and Ocaml 3.09. They 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 contrary to functional languages prevents us from using exception, `goto`, and `switch` constructs and thus optimizations like those presented in [FM01].

6 Conclusion

In this paper, we have presented a new approach to compile 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 the compiler construction community. Using a program transformation and a formal method approach is an elegant way to describe, implement, and certify the proposed optimizations. This work is closed to Sestoft approach [Ses96] which compiles naively ML-style pattern matches and by partial evaluation removes redundant cases instead of constructing directly the decision tree. Moreover, this two-stage pattern compilation is directly implemented in Tom and shows how Tom language is well-adapted for program analysis-transformation.

We have only be interested in optimizing syntactic matching and thus considered a subset of `PIL` language. As Tom already manages associativity, a future work will consist in developing new transformation rules adapted to this theory, without having to change the rules relative to syntactic one. However, note that the presented rules remain correct when considering an extension of `PIL`.

This paper shows that using program transformation rules to optimize pattern matching is an efficient solution, with respect to algorithms based on automata. The implementation of this

work combined with the formal validation of pattern matching [KMR05] is another step towards the construction of certified/certifying optimizing compilers.

Bibliography

[Aug85] L. Augustsson. Compiling pattern matching. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture*. Pp. 368–381. Springer-Verlag, 1985.

[BKM06] E. Balland, C. Kirchner, P.-E. Moreau. Formal Islands. In Johnson and Vene (eds.), *Proceedings of the 11th international conference on Algebraic Methodology and Software Technology*. LNCS 4019, pp. 51–65. Springer-Verlag, 2006.

[BM05] E. Balland, P.-E. Moreau. Optimizing Pattern Matching by Program Transformation. Technical report, INRIA-LORIA, 2005. http://hal.inria.fr/inria-00000763.

[Car84] L. Cardelli. Compiling a functional language. In *Proceedings of the ACM Symposium on LISP and Functional Programming*. Pp. 208–217. 1984.

[FM01] F. L. Fessant, L. Maranget. Optimizing pattern matching. In *Proceedings of the sixth International Conference on Functional Programming*. Pp. 26–37. ACM Press, 2001. doi:http://doi.acm.org/10.1145/507635.507641

[Grä91] A. Gräf. Left-to-Right Tree Pattern Matching. In *Proceedings of the 4th international conference on Rewriting Techniques and Applications*. LNCS 488, pp. 323–334. Springer-Verlag, 1991.

[Jon87] S. L. P. Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[KMR05] C. Kirchner, P.-E. Moreau, A. Reilles. Formal Validation of Pattern Matching Code. In Barahone and Felty (eds.), *Proceedings of the 7th international conference on Principles and Practice of Declarative Programming*. Pp. 187–197. ACM, July 2005.

[Leu96] A. Leung. C++-based Pattern Matching Language. 1996. citeseer.ist.psu.edu/leung96cbased.html

[MRV03] P.-E. Moreau, C. Ringeissen, M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In Hedin (ed.), *12th Conference on Compiler Construction*. LNCS 2622, pp. 61–76. Springer-Verlag, May 2003.

[OW97] M. Odersky, P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. Pp. 146–159. ACM Press, USA, 1997.

[RKS99] O. Rüthing, J. Knoop, B. Steffen. Detecting Equalities of Variables: Combining Efficiency with Precision. In Cortesi and Filé (eds.), *SAS*. LNCS 1694, pp. 232–247. Springer-Verlag, 1999.

[Ses96] P. Sestoft. ML Pattern Match Compilation and Partial Evaluation. In Danvy et al. (eds.), *Dagstuhl Seminar on Partial Evaluation*. LNCS 1110, pp. 446–464. Springer-Verlag, 1996.

[SRR95] R. C. Sekar, R. Ramesh, I. V. Ramakrishnan. Adaptive Pattern Matching. *SIAM Journal on Computing* 24(6):1207–1234, 1995.