Optimizing pattern matching compilation by program transformation

Emilie Balland, Pierre-Etienne Moreau

Optimization of Generic Pattern-Matching Compilation

Emilie Balland and Pierre-Etienne Moreau

UHP & LORIA, INRIA & LORIA,
BP 101, 54602 Villers-lès-Nancy Cedex France
{Emilie.Balland,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 approaches 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.

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 ML. 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 is to integrate the notion of pattern matching into classical languages such as C and Java. As presented in [7] and 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.

Due to lack of space, we do not present Tom in details, but the reader should consider it as a powerful language extension which offers syntactic matching, associative matching with neutral element, conditional rewriting, support for built-in data-types, XML transformation facilities, and a modular strategy library *à la* Stratego [11] which allows to define complex recursive normalizing and traversal strategies. Tom has been used to implement various applications from deep inference in the calculus of structures to the generation of web servers using XML for example. One of the biggest application is the Tom compiler itself, written in Tom and Java.

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

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 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 [6] for more details. In practice, this means that a description of the data-structure (a mapping) has to be given to explain Tom how accessing subterms for example.

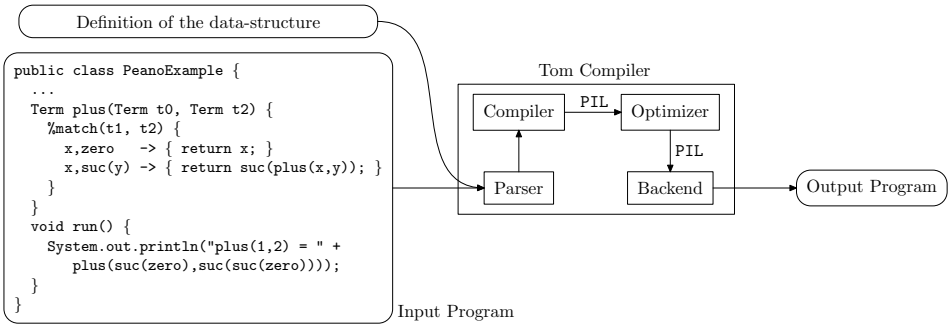

Fig. 1. General architecture of Tom: the compiler generates an intermediate `PIL` program which is optimized before being pretty-printed by the backend.

There exists several methods [2, 1, 5, 4] 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 [2, 5] and the other on backtracking automata [1]. These two approaches emphasize the unavoidable compromise between speed and memory space [9].

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.

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 [6]. 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. Finally, in Section 4, some experimental results are given for several revealing examples.

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 assignments, whereas the sequence instruction $\langle instr\rangle$; $\langle instr\rangle$ comes from imperative languages. A last particularity of `PIL` comes from the `hostcode`(...) 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.

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

Fig. 2. `PIL` syntax

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 ($\texttt{is\_fsym}(t, f)$), or accessing to the n-th child of a term t ($\texttt{subterm}_f(t, n)$). 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 [6] 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 $\mathcal{E}nv$. The reduction relation of the big-step semantics is expressed on tuples $\langle \epsilon, \delta, i\rangle$ 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 which appear in the block. A complete definition of the semantics can be found in [3].

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

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, 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) ;
hostcode(...) ;
```

Fig. 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.

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. Given an environment ϵ and an expression e, `eval` is a function that evaluates e in ϵ to obtain a value (i.e $\top$ or $\bot$). Given an environment ϵ and a host-code list δ, the evaluation of a program $\pi \in$ PIL results in a host-code list: $\langle \epsilon, \delta, \pi \rangle \mapsto_{bs} \delta'$. 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: $\epsilon' = \Phi(\pi, e, \epsilon, \delta)$. More formal definitions can be found in [3]. We can notice that these definitions can be easily extended to terms.

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, $\mathtt{eval}(\epsilon_1, e_1) = \mathtt{eval}(\epsilon_2, e_2)$ where $\epsilon_1 = \Phi(\pi, e_1, \epsilon, \delta)$ and $\epsilon_2 = \Phi(\pi, e_2, \epsilon, \delta)$.*

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, $\mathtt{eval}(\epsilon_1, e_1) \wedge \mathtt{eval}(\epsilon_2, e_2) = \bot$ where $\epsilon_1 = \Phi(\pi, e_1, \epsilon, \delta)$ and $\epsilon_2 = \Phi(\pi, e_2, \epsilon, \delta)$.*

We can now define two conditions which are sufficient to determine whether two expression are π-equivalent or π-incompatible. Propositions 1 and 2 are interesting because the problem is generally undecidable [8], 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 [3].

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

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: $\forall \epsilon, \delta,\ \Phi(\pi, e_1, \epsilon, \delta) = \Phi(\pi, e_2, \epsilon, \delta)$ (cond1) and* `incompatible`(e_1, e_2) *(cond2), where* `incompatible` *defined as follows:*

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

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.

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 $i[v/t]$).

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

Dead variable elimination and Inlining. Using a simple static analysis, these optimizations eliminate useless assignments:

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

where `use`(v, i) is a function that computes how many times the value of a variable v may be used in an instruction i.

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 (see Figure 3), this case often occurs. By merging the bodies, this allows to recursively perform some optimizations on subterms.

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

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.

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):

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

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 $\Phi(\pi, \epsilon, \delta, e_1) = \Phi(\pi, \epsilon, \delta, e_2)$ (cond1) 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 that $e_1 = e_2$ (cond2), which is easier.

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 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:

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

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

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`.

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

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

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. 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.

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

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 two last patterns. This would result in the following sub-optimal program:

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

`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 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 (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 $Innermost(s)$ (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 fail), we can define a strategy which describes how the considered rewrite system should be applied to normalize a `PIL` program:

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

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:

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

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 [3] we show that each transformation rule is correct, in the sense that the the optimized program has the same observational behavior as the original. One advantage of program transformation approach is that if every rule is correct, whatsoever the application strategy, the final program is semantically equivalent.

We also show that the optimized code is both more efficient (in number of tests), and smaller than the initial program. Similarly to [4], this result is interesting since it allows to generate efficient pattern matching implementations whose size is linear in the number and size of patterns.

4 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. 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
Compilation time	0.22 s	0.15 s	2.0 s	0.74 s	0.66 s	0.29 s
Optimisation time	0.26 s	0.19 s	136 s	20.6 s	8.0 s	2.5 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 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).
- `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 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 the `switch` instruction and thus optimizations like those presented in [4].

5 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 [10] which compiles naively ML-style pattern matches and by partial evaluation removes redundant cases instead of constructing directly the decision tree.

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 [6] is another step towards the construction of certified/certifying optimizing compilers.

Acknowledgments

We sincerely thank Claude Kirchner and Antoine Reilles for the useful interactions we had on the topics of this paper.

References

[1] Lennart Augustsson. Compiling pattern matching. In *Proceedings of a conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, 1985.

[2] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217, 1984.

[3] Pierre-Etienne Moreau Emilie Balland. Optimizing pattern matching by program transformation. Technical report, INRIA-LORIA, 2005. http://hal.inria.fr/inria-00001127.

[4] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.

[5] Albert Gräf. Left-to-right tree pattern matching. In *Proceedings of the 4th international conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 323–334. Springer-Verlag, 1991.

[6] 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.

[7] 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.

[8] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In Agostino Cortesi and Gilberto Filé, editors, *SAS*, volume 1694 of *LNCS*, pages 232–247. Springer-Verlag, 1999.

[9] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, 1995.

[10] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 446–464. Springer-Verlag, 1996.

[11] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 13–26, 1998.