

# Formal Validation of Pattern Matching Code

Claude Kirchner  
INRIA & LORIA

Pierre-Etienne Moreau  
INRIA & LORIA

Antoine Reilles  
CNRS & LORIA

Nancy, France  
First.Last@loria.fr

## ABSTRACT

When addressing the formal validation of generated software, two main alternatives consist either to prove the correctness of compilers or to directly validate the generated code. Here, we focus on directly proving the correctness of *compiled* code issued from powerful pattern matching constructions typical of ML like languages or rewrite based languages such as ELAN, Maude or Tom. In this context, our first contribution is to define a general framework for anchoring algebraic pattern-matching capabilities in existing languages like C, Java or ML. Then, using a just enough powerful intermediate language, we formalize the behavior of compiled code and define the correctness of compiled code with respect to pattern-matching behavior. This allows us to prove the equivalence of compiled code correctness with a generic first-order proposition whose proof could be achieved via a proof assistant or an automated theorem prover. We then extend these results to the multi-match situation characteristic of the ML like languages. The whole approach has been implemented on top of the Tom compiler and used to validate the syntactic matching code of the Tom compiler itself.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Correctness proofs, Formal methods, Validation*;  
D.3.4 [Programming Languages]: Processors—*Code generation, Compilers*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

## General Terms

Reliability, rule based languages, verification

## Keywords

Compilation, pattern matching, multi-match, term rewriting, verified code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'05, July 11–13, 2005, Lisbon, Portugal.  
Copyright 2005 ACM 1-59593-090-6/05/0007 ...\$5.00.

## 1. INTRODUCTION

Even if we know, since the beginning of the computer science era, that proving program correctness is profoundly difficult, the quest of software security and dependability due to the general digitalization of most human activities and process control makes this goal both inescapable and extremely important to reach.

When we deal with the previous problem, we should address the whole software conception process that we can reduce, quite schematically, to the following steps: (i) get an informal specification of the software functionalities, (ii) get a formal description of the algorithms assumed to fulfill the informal specification, (iii) get a high-level program implementation of these algorithms, (iv) get a low level program implementation of these programs, (v) get a model of the running hardware.

In this work, we restrict our interest to step (iv) and to high-level languages pattern-matching features. Therefore we address the specific problem of proving the correctness of *compiled* code issued from pattern-matching constructions appearing in high-level programming languages.

*Verifiable —compiler versus compiled— code.* The question of compiler correctness, that is to preserve the properties of the input like its semantics and meta-properties of the underlying algorithm as its termination or the respect of heap invariants, is as old as the first compiler implementations. This is a very challenging goal since it consists in proving that *any* valid input will be correctly compiled. Furthermore, this proof has to be done every time the implementation of the compiler is modified and moreover, the compiler has itself to be compiled.

Much efforts have been done on proving correctness of parts and sometimes even complete compilers either manually [?, ?, ?, ?] or with the help of a proof assistant. But, it is still today mostly out of reach to prove that a program has been correctly compiled. In practice, programmers (and therefore applications) totally rely on the compilers: until one runs the program, we have no idea if the compiler has compiled the program correctly. Even extensively testing the program of course offers no guarantees. So, currently the programmer very often must blindly trust the compiler. But, most of largely used C and Fortran compilers very infrequently generate incorrect code: they are some of the most reliable software tools available. This is due to the large number of developers working to make these compilers correct, as well as the very large number of users who use these compiler, and thus contribute to their debugging. But, when

designing a compiler for a new high-level language, the situation is less comfortable: on one side the number of users and written applications is small, on the other side, the introduction of new high level constructs put a lot on the compiler, and so make it even more complex to write. Since the consequences of an incorrect compiler are disastrous (all compiled programs are potentially faulty), this situation contributes to make users less confident in new languages and compiler implementations.

In this paper, we are concerned by a quite different approach consisting in proving *automatically* the correctness of the compiled code. This “skeptical” approach of the code issued from a compiler allows to deal with two kind of misbehavior: one is due to the classical presence of an unintentional bug of the compiler. The second one concern intended hidden-behavior that could be introduced with malicious intention.

Therefore, assuming a high-level program given as input, we are considering the compiler as a black box escaping our control and we are searching to prove that the generated code is, on its own, correct. This is typical of the seminal work of [?] and more recently of the so called translation validation [?, ?]. A comparable approach presented in [?] is called credible compilation and able to handle pointers in the source program. Note that this is different from the so called proof-carrying code method [?, ?] which is not intended to prove the compiled program to be correct with respect to the source code, but rather on proving certain properties on the output program, such as type safety, memory safety, or the respect of a certain safety invariant.

**Matching power.** Rewriting and pattern-matching are of general use in mathematics and informatics to describe computation as well as deduction. They are central in systems making the notion of rule an explicit and first class object, like expert and business systems (JRule), programming languages based on equational logic (OBJ) or the rewriting calculus (ELAN) or logic (Maude), functional, possibly logic, programming (ML, Haskell, Curry, Teyjus) and model checkers (Murphi). They are also recognized as crucial components of proof assistants (Coq, Isabelle) and theorem provers for expressing computation as well as strategies.

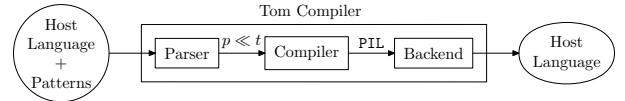
Since pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, it is a first class citizen of functional languages like ML or Haskell and has been considered recently as a useful add-on facility in object programming languages [?]. This is formally backed up by works like [?] and particularly well-suited when describing various transformations of structured entities like, for example, trees/terms, hierarchized objects, and XML documents.

In this context, we are developing the Tom system [?] which provides a generic way to integrate matching power in *existing* programming languages like Java, C or ML. For example, when using Java as the host language, the sum of two integers can be described in Tom as follows:

```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,zero  -> { return x; }
    x,suc(y) -> { return suc(plus(x,y)); }
  }
}
```

In this example, given two terms  $t_1$  and  $t_2$  that represent Peano integers, the evaluation of `plus` computes their sum. This is implemented by pattern-matching:  $t_1$  is matched by the variable  $x$ ,  $t_2$  is possibly matched by one of the two patterns `zero` or `suc(y)`. When `zero` matches  $t_2$ , the result of the addition is  $x$  (where  $x$  has been instantiated into  $t_1$  via 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. This definition of `plus` is given in a functional style, but now the `plus` function can be used elsewhere in a Java program to perform addition.

The general architecture of Tom, depicted as follows,



enlightens that a generic matching problem  $p \ll t$  is compiled into an intermediate language code (PIL) which we would like to compute a substitution  $\sigma$  iff  $\sigma(p) = t$ . As explained in [?], implementing a language (possibly domain-specific) as an extension of an existing *host* language has several advantages. First, we benefit of the existing functionalities and we do not have to re-implement common language constructs. Second, the extensions themselves only need to be transformed to the point where they are expressible in the host language. Third, existing infrastructure can be reused. All these factors result into lower implementation costs and decrease the risk of building an incorrect compiler.

So, in this work we focus on proving the correctness of compiled code issued from pattern-matching constructions, and to the best of our knowledge, this is the first attempt to do so. Other works about pattern-matching compilation address in particular data abstraction e.g. [?], or optimizations for run-time efficiency or code size e.g. [?, ?].

**Roadmap of the paper.** When considering the notion of pattern-matching, we consider a *term* data-structure against which some *patterns* are matched. Since the host language is not fixed and could be typically either C, Java or ML, the data-model is unknown. We therefore introduce in Section ??, the notion of *formal anchor* which formally describes the relationship between the host language data-model and the algebraic notion of term and pattern.

In our language, the host language is also generic, so we have to consider an abstraction which describes the minimal set of functionality the host language should have to express the compilation of pattern-matching. This abstraction is called the *intermediate language* (PIL) and we define its syntax and its big-step semantics in Section ??.

Then Section ?? uses the proposed framework to define the correctness of a single pattern compilation and to show how this correctness can be reduced to the validation of a first-order proposition.

This result is then extended in Section ?? to support Caml or Tom multi-match constructs, and before concluding, Section ?? provides details about the implementation of these concepts in the current version of Tom.

## 2. FORMAL ANCHOR

When considering the problem of proving that the behavior of a program is compatible with its semantics, we have to consider two kinds of entities. On the one side, we consider

algebraic constructions, such as ground terms ( $t \in \mathcal{T}(\mathcal{F})$ ), patterns ( $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ), and matching problems ( $p \ll t$ , with  $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  and  $t \in \mathcal{T}(\mathcal{F})$ ). On the other side, we consider programs, expressed in the PIL intermediate language, which are supposed to solve matching problems. We also consider data which are supposed to represent a term or a pattern. In this section, we define the notions of *representation* and *formal anchor* which define the link between algebraic entities and considered data.

## 2.1 Preliminary concepts

We assume the reader to be familiar with the basic definitions of first order term given, in particular, in [?]. We briefly recall or introduce 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 ( $\text{ar} : \mathcal{F} \rightarrow \mathbb{N}$ ).  $\mathcal{F}_n$  is the subset of function symbols having  $n$  for arity,  $\mathcal{F}_n = \{f \in \mathcal{F} \mid \text{ar}(f) = n\}$ .

$\mathcal{T}(\mathcal{F}, \mathcal{X})$  is the set of *terms* built from a given finite set  $\mathcal{F}$  of function symbols and a denumerable set  $\mathcal{X}$  of variables. A term  $t$  is said to be *linear* if no variable occurs more than once in  $t$ . Positions in a term are represented as sequences of integers and denoted by Greek letters  $\epsilon, \nu$ . The empty sequence  $\epsilon$  denotes the position associated to the root, and it is called the root (or top) position. The subterm of  $t$  at position  $\nu$  is denoted  $t_{|\nu}$ .  $\text{Symb}(t)$  is a partial function from  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  to  $\mathcal{F}$ , which associates to each term  $t$  its root symbol  $f \in \mathcal{F}$ .

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.

Two ground terms  $t$  and  $u$  of  $\mathcal{T}(\mathcal{F})$  are equal, and we note  $t = u$ , when, for some function symbol  $f$ ,  $\text{Symb}(t) = \text{Symb}(u) = f$ ,  $f \in \mathcal{F}_n$ ,  $t = f(t_1, \dots, t_n)$ ,  $u = f(u_1, \dots, u_n)$ , 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  $f \in \mathcal{F}_n$ .

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  $\sigma(p) = t$ . Its negation is written  $p \not\ll t$ .

## 2.2 Object representation

DEFINITION 1. Given a tuple composed of a signature  $\mathcal{F}$ , a set of variables  $\mathcal{X}$ , booleans  $\mathbb{B}$  and integers  $\mathbb{N}$ , given sets  $\Omega_{\mathcal{F}}$ ,  $\Omega_{\mathcal{X}}$ ,  $\Omega_{\mathcal{T}}$ ,  $\Omega_{\mathbb{B}}$ , and  $\Omega_{\mathbb{N}}$ , we consider a family of representation functions  $\ulcorner \cdot \urcorner$  that map:

- function symbols  $f \in \mathcal{F}$  to elements of  $\Omega_{\mathcal{F}}$ , denoted  $\ulcorner f \urcorner$ ,
- variables  $v \in \mathcal{X}$  to elements of  $\Omega_{\mathcal{X}}$ , denoted  $\ulcorner v \urcorner$ ,
- ground terms  $t \in \mathcal{T}(\mathcal{F})$  to elements of  $\Omega_{\mathcal{T}}$ , denoted  $\ulcorner t \urcorner$ ,
- booleans  $b \in \mathbb{B} = \{\top, \perp\}$  to elements of  $\Omega_{\mathbb{B}}$ , denoted  $\ulcorner b \urcorner$ ,
- natural numbers  $n \in \mathbb{N}$  to elements of  $\Omega_{\mathbb{N}}$ , denoted  $\ulcorner n \urcorner$ .

In other words, the representation function  $\ulcorner \cdot \urcorner$  maps algebraic entities (from  $\mathcal{F}$ ,  $\mathcal{X}$ ,  $\mathcal{T}(\mathcal{F})$ ,  $\mathbb{B}$ , and  $\mathbb{N}$ ) to objects manipulable by the intermediate language PIL (elements of  $\Omega_{\mathcal{F}}$ ,  $\Omega_{\mathcal{X}}$ ,  $\Omega_{\mathcal{T}}$ ,  $\Omega_{\mathbb{B}}$ , and  $\Omega_{\mathbb{N}}$ ). We note  $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$  the set containing the representations of terms:  $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner = \{\ulcorner t \urcorner \mid t \in \mathcal{T}(\mathcal{F})\}$ , and we therefore have  $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner \subseteq \Omega_{\mathcal{T}}$ .

EXAMPLE 1. Let us consider  $\mathcal{F} = \{e, s\}$  (with  $\text{ar}(e) = 0$  and  $\text{ar}(s) = 1$ ), and the function  $\ulcorner \cdot \urcorner$  such that  $\ulcorner e \urcorner = 0 \in \Omega_{\mathcal{F}}$ ,  $\ulcorner s \urcorner = 1 \in \Omega_{\mathcal{F}}$ .  $\ulcorner \cdot \urcorner$  maps the symbols  $e$  and  $s$  respectively to “machine integers” 0 and 1 (i.e. the notion of integer in the intermediate language), where we assume an infinite memory. Similarly,  $\ulcorner \cdot \urcorner$  can be extended to map the constant  $e \in \mathcal{T}(\mathcal{F})$  to 0 ( $\ulcorner e \urcorner = 0 \in \Omega_{\mathcal{T}}$ ), and any term of the form  $s(x)$  to the result of the addition of 1 and the representation of  $x$  ( $\ulcorner s(x) \urcorner = 1 + \ulcorner x \urcorner \in \Omega_{\mathcal{T}}$ ).

This representation is a way to map Peano integers to “machine integers”. Another well-known representation is the encoding of algebraic terms into e.g.  $n$ -ary trees.

## 2.3 Object mapping

In Definition ??, the notion of representation mapping has been introduced to establish a correspondence between algebraic objects and their representation in the intermediate language. However, we did not put any constraint on the representation of objects. In particular, the function  $\ulcorner \cdot \urcorner$  does not necessarily preserve structural properties of algebraic objects (all terms could for example be represented by a unique constant).

DEFINITION 2. Given a tuple  $\langle \mathcal{F}, \mathcal{X}, \mathcal{T}(\mathcal{F}), \mathbb{B}, \mathbb{N} \rangle$ , a representation function  $\ulcorner \cdot \urcorner$ , and the mappings  $\text{eq} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{T}} \rightarrow \Omega_{\mathbb{B}}$ ,  $\text{is\_fsym} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{F}} \rightarrow \Omega_{\mathbb{B}}$ , and  $\text{subterm}_f : \Omega_{\mathcal{T}} \times \Omega_{\mathbb{N}} \rightarrow \Omega_{\mathcal{T}}$  ( $f \in \mathcal{F}$ ). A formal anchor is a mapping  $\lceil \cdot \rceil : \mathcal{T}(\mathcal{F}) \rightarrow \ulcorner \mathcal{T}(\mathcal{F}) \urcorner$  such that the structural properties of  $\mathcal{T}(\mathcal{F})$  are preserved, in  $\lceil \mathcal{T}(\mathcal{F}) \rceil$ , by the semantics of  $\text{eq}$ ,  $\text{is\_fsym}$ , and  $\text{subterm}_f$ .

$\forall t, t_1, t_2 \in \mathcal{T}(\mathcal{F}), \forall f \in \mathcal{F}, \forall i \in [1..ar(f)]$  we have:

$$\begin{aligned} \text{eq}(\lceil t_1 \rceil, \lceil t_2 \rceil) &\equiv \lceil t_1 = t_2 \rceil \\ \text{is\_fsym}(\lceil t \rceil, \lceil f \rceil) &\equiv \lceil \text{Symb}(t) = f \rceil \\ \text{subterm}_f(\lceil t \rceil, \lceil i \rceil) &\equiv \lceil t_{|i} \rceil \text{ if } \text{Symb}(t) = f \end{aligned}$$

In the following, we always consider that the representation function is also a formal anchor. Therefore, from now on, the notation  $\ulcorner \cdot \urcorner$  denotes representations that are also formal anchors.

EXAMPLE 2. In C or Java like, the notion of term can be implemented by a record (*sym*:integer, *sub*:array of term), where the first slot (*sym*) denotes the top symbol, and the second slot (*sub*) corresponds to the subterms. It is easy to check that the following definitions of  $\text{eq}$ ,  $\text{is\_fsym}$ , and  $\text{subterm}_f$  (where  $=$  denotes an atomic equality) provide a formal anchor for  $\mathcal{T}(\mathcal{F})$ :

$$\begin{aligned} \text{eq}(t_1, t_2) &\triangleq t_1.\text{sym} = t_2.\text{sym} \wedge \forall i \in [1..ar(t_1.\text{sym})], \\ &\quad \text{eq}(t_1.\text{sub}[i], t_2.\text{sub}[i]) \\ \text{is\_fsym}(t, f) &\triangleq t.\text{sym} = f \\ \text{subterm}_f(t, i) &\triangleq t.\text{sub}[i] \text{ if } t.\text{sym} = f \text{ and } i \in [1..ar(f)] \end{aligned}$$

Defining a correct formal anchor is a key point to allow for the formal verification of the pattern matching code. But since this can be quite technical, we use in practice an external tool which generates for us the mapping for a given signature, as described in Section ??.

### 3. INTERMEDIATE LANGUAGE

We now describe the syntax of PIL, introduce the notion of environment, and give a formal big-step semantics ( $\mapsto_{bs}$ ) to PIL. Informally, this intermediate language is a subset of  $C \cap \text{Java} \cap \text{ML}$  that is expressive enough to describe pattern matching procedures. This language is very close to the host language fragment it will be translated into at the end of the compilation process, and involves only a renaming of the syntactic constructions, so that proving this part of the compilation process should not present difficulties.

#### 3.1 Syntax

Given  $\mathcal{F}$ ,  $\mathcal{X}$ ,  $\mathcal{T}(\mathcal{F})$ ,  $\mathbb{B}$ ,  $\mathbb{N}$ , **eq**, **is\_fsym**, **subterm**, and a formal anchor  $\ulcorner \urcorner$  as defined above, the syntax of the intermediate language PIL is defined as follows:

PIL	::=	$\langle instr \rangle$
<b>symbol</b>	::=	$\ulcorner f \urcorner$ ( $f \in \mathcal{F}$ )
<b>variable</b>	::=	$\ulcorner x \urcorner$ ( $x \in \mathcal{X}$ )
$\langle term \rangle$	::=	$\ulcorner t \urcorner$ ( $t \in \mathcal{T}(\mathcal{F})$ )
		<b>variable</b>
		<b>subterm</b> $_f(\langle term \rangle, \ulcorner n \urcorner)$ ( $f \in \mathcal{F}, n \in \mathbb{N}$ )
$\langle bexpr \rangle$	::=	$\ulcorner b \urcorner$ ( $b \in \mathbb{B}$ )
		<b>eq</b> ( $\langle term \rangle, \langle term \rangle$ )
		<b>is_fsym</b> ( $\langle term \rangle, \mathbf{symbol}$ )
$\langle instr \rangle$	::=	<b>let</b> ( <b>variable</b> , $\langle term \rangle$ , $\langle instr \rangle$ )
		<b>if</b> ( $\langle bexpr \rangle$ , $\langle instr \rangle$ , $\langle instr \rangle$ )
		<b>accept</b>
		<b>refuse</b>

The set of terms  $\langle term \rangle$  is built over the representation of  $\mathcal{T}(\mathcal{F})$ , and the construct **subterm** $_f$  which retrieves the  $i^{\text{th}}$  child of a given term. The set of expressions  $\langle bexpr \rangle$  contains the representation of booleans, as well as two predicates: **eq** which compares two terms, and **is\_fsym** which checks that a given term is rooted by a particular symbol given in argument. The set of instructions  $\langle instr \rangle$  contains only 4 instructions: **let** and **if** correspond respectively to the assignment and the if-then-else test. We consider here that it is forbidden to assign a same variable twice. We use an *if then else* construct instead of the *switch* construct usually used to compile pattern matching because we want the generated matching algorithm to be independent of the mapping and the way terms are effectively represented. The **is\_fsym** expression allows to “query” a term without the need to have function symbols as objects directly manipulated by the host language, providing more abstraction. **accept** and **refuse** are two special instructions aimed to approximate the body part of a function defined by pattern matching. In this work, since we focus on pattern matching, we only need two instructions to put the execution in a given state (**accept** or **refuse**), which denotes whether the pattern matches the subject or not.

Such a program may contain some free variables (variables which are not bound in the program by a **let** construct). They represent the input of the program, in our case the terms the pattern matching algorithm will try to match against. We call such variable *input variable*.

**Assumption A.** In the following, we consider that a program is evaluated in an environment where all its free variables are instantiated by a value, i.e. a term representation.

EXAMPLE 3. Given a signature  $\mathcal{F} = \{a, f\}$ , a set of variables  $\mathcal{X} = \{s, x\}$ , a possible compilation of  $f(x) \ll s$  is:

```

if(is_fsym( $\ulcorner s \urcorner$ ,  $\ulcorner f \urcorner$ ),
  let( $\ulcorner x \urcorner$ , subterm $_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner)$ , accept),
  refuse
)

```

This program is evaluated in an environment which assigns a term representation to the free variable  $\ulcorner s \urcorner$ . This program checks that the root symbol of  $s$  corresponds to the representation of  $f$ . When it is the case, the first subterm of  $s$  is assigned to a variable  $x$ , and the program goes into the **accept** state. Otherwise, it goes into the **refuse** state.

On this example, it is easy to convince ourselves that the program goes into the **accept** state if and only if the pattern effectively matches the subject. Our goal here consists to get a formal proof of this property.

**Notation.** For sake of correctness, mathematical objects ( $\mathbb{B}, \mathbb{N}$ , and  $\mathcal{X}$ ) have to be distinguished from their representation. However, since most of programming languages support the notion of boolean, integer and variable, when there is no ambiguity, we note  $\ulcorner \top \urcorner = \mathbf{true}$ ,  $\ulcorner \perp \urcorner = \mathbf{false}$ ,  $\ulcorner 0 \urcorner = 0$ ,  $\ulcorner 1 \urcorner = 1, \dots, \ulcorner n \urcorner = n$  for  $n \in \mathbb{N}$ , and  $\ulcorner x \urcorner = x$  for  $x \in \mathcal{X}$ .

Among the set of programs of PIL, we consider the subset of programs whose evaluation (under assumption A) always terminates in **accept** or **refuse**, whatever the input is. Those programs are called *well-formed* programs.

DEFINITION 3. A program  $\pi \in \text{PIL}$  is said to be *well-formed* when it satisfies the following properties.

- Each expression **subterm** $_f(t, n)$  is such that  $t$  belongs to  $\langle term \rangle$ , **is\_fsym**( $t, \ulcorner f \urcorner$ )  $\equiv \mathbf{true}$  and  $n \in [1..ar(f)]$ . (In practice, we verify that each expression of the form **subterm** $_f(t, n)$  belongs to the **then** part of an instruction **if**(**is\_fsym**( $t, \ulcorner f \urcorner$ ), ...).)
- Each variable appearing in a sub-expression is previously initialized by a **let** construct, or in the evaluation environment.

We introduce here a simple type system for verifying that a given program is well-formed, in a particular context (modeling the evaluation environment). This context is formed by the variables which have been introduced in the evaluation environment, noted  $\Gamma$ , and a list of couples ( $\langle term \rangle, \mathbf{symbol}$ ), noted  $\Delta$ , representing the fact that in the evaluation environments, the root symbol of a given term is known.

PROPERTY 1. A PIL-program  $\pi$  is said *well-formed* in an evaluation environment if and only if we can build a derivation of  $\Gamma, \Delta \vdash \pi : wf$  in the type system presented in Figure ???.  $\Gamma$  contains the variables initialized by the environment, and  $\Delta$  stores which terms have a particular root symbol.

PROOF. Let  $\pi$  a PIL-program,  $\Gamma, \Delta$  contexts such that there is a derivation  $\Gamma, \Delta \vdash \pi : wf$  in the type system of Figure ???.

So for each variable  $v$  in  $\pi$ , it exists contexts  $\Gamma', \Delta'$  such that  $\Gamma', \Delta' \vdash v : wf$ , and thus  $v \in \Gamma'$ . Since  $v$  can only be introduced in the context  $\Gamma'$  either by early initialisation, or

$\frac{}{\Gamma, \Delta \vdash \ulcorner b \urcorner : wf} (b \in \mathbb{B})$	$\frac{\Gamma, \Delta \vdash \mathbf{t}_1 : wf \quad \Gamma, \Delta \vdash \mathbf{t}_2 : wf}{\Gamma, \Delta \vdash \mathbf{eq}(\mathbf{t}_1, \mathbf{t}_2) : wf}$
$\frac{}{\Gamma, \Delta \vdash \ulcorner t \urcorner : wf} (t \in \mathcal{T}(\mathcal{F}))$	$\frac{\Gamma, \Delta \vdash \mathbf{t} : wf}{\Gamma, \Delta \vdash \mathbf{subterm}_f(\mathbf{t}, i) : wf} \text{ if } (\mathbf{t}, f) \in \Delta \text{ and } i \in [1..ar(f)]$
$\frac{}{\Gamma, \Delta \vdash \mathbf{accept} : wf}$	$\frac{\Gamma, \Delta \vdash \mathbf{t} : wf \quad \Gamma :: v, \Delta \vdash i : wf}{\Gamma, \Delta \vdash \mathbf{let}(v, \mathbf{t}, i) : wf}$
$\frac{}{\Gamma, \Delta \vdash \mathbf{refuse} : wf}$	$\frac{\Gamma, \Delta \vdash \mathbf{is\_fsym}(\mathbf{t}, \ulcorner f \urcorner) : wf \quad \Gamma, \Delta :: (\mathbf{t}, \ulcorner f \urcorner) \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \mathbf{if}(\mathbf{is\_fsym}(\mathbf{t}, \ulcorner f \urcorner), i_1, i_2) : wf}$
$\frac{}{\Gamma, \Delta \vdash \ulcorner x \urcorner : wf} \text{ if } x \in \Gamma$	$\frac{\Gamma, \Delta \vdash e : wf \quad \Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \mathbf{if}(e, i_1, i_2) : wf} \text{ if } e \neq \mathbf{is\_fsym}(\mathbf{t}, \ulcorner f \urcorner)$

Figure 1: Type system for checking validity

by applying the typing rule for **let**, the variable  $v$  has been initialized. Also, for each  $\mathbf{subterm}_f(\mathbf{t}, i)$  construct in  $\pi$ , it exists contexts  $\Gamma', \Delta'$  such that  $\Gamma', \Delta' \vdash \mathbf{subterm}_f(\mathbf{t}, i) : wf$ , and  $(\mathbf{t}, \ulcorner f \urcorner) \in \Delta'$ . Since  $(\mathbf{t}, \ulcorner f \urcorner)$  can only be introduced in the context  $\Delta'$  by applying the typing rule for  $\mathbf{if}(\mathbf{is\_fsym}(\mathbf{t}, \ulcorner f \urcorner), i_1, i_2)$  or by a previous test, the representation  $\mathbf{t}$  has been checked to have root symbol  $f$ .

Let  $\pi$  a well-formed PIL-program in an evaluation environment. If we initialize the contexts  $\Gamma$  and  $\Delta$  with the variables already instantiated in the environment, and with which terms have a particular root symbol, we can build a derivation of  $\Gamma, \Delta \vdash \pi : wf$ , since the typing rules for variables and **subterm** constructs will apply, each variable being either instantiated in the initial environment, or introduced by a **let** construct before its use, and root symbol of terms and arities being checked before the use of a **subterm** construct, either with a test in the program or in the evaluation environment.  $\square$

In practice, when verifying that a given program is well-formed, we initialize the environments with the set of input variables of the program as  $\Gamma$  (corresponding to the subject against which the program matches), and an empty list of couples  $\Delta$ .

Notice that the well-formedness of a PIL-program is linearly decidable, since this property can be decided by the type system in Figure ??.

The program given in Example ?? is well-formed in the environment  $\Gamma = \{s\}$ ,  $\Delta = \emptyset$ , since  $\mathbf{subterm}_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner)$  is protected by the construct  $\mathbf{if}(\mathbf{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \dots)$  with  $1 \in [1..ar(f)]$ ,  $\ulcorner x \urcorner$  is introduced by a **let**, and  $\ulcorner s \urcorner$  is in  $\Gamma$ . On the contrary, the program:  $\mathbf{if}(\mathbf{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \mathbf{if}(\mathbf{eq}(\ulcorner x \urcorner, \mathbf{subterm}_g(\ulcorner s \urcorner, \ulcorner 1 \urcorner)), \mathbf{accept}, \mathbf{refuse}), \mathbf{refuse})$  is not well-formed in the same environment for two reasons:  $\ulcorner x \urcorner$  is not introduced by a **let**, and  $\mathbf{subterm}_g$  is not guarded by an  $\mathbf{if}(\mathbf{is\_fsym}(\ulcorner s \urcorner, \ulcorner g \urcorner), \dots)$ .

### 3.2 Environments

Given a matching problem, its satisfiability is of course of interest. But in most applications it is not enough and we need to compute a witness: i.e. a substitution which assigns values to the variables of the pattern. In this section, 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 concrete terms to variable names. In addition, we also define a function  $\Phi$  which goes back from environments to algebraic substitutions.

DEFINITION 4. An atomic environment  $\epsilon$  is an assignment from  $\ulcorner \mathcal{X} \urcorner$  to  $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ , written  $[x \leftarrow \ulcorner t \urcorner]$ . The composition of environments is left-associative, and written  $[x_1 \leftarrow \ulcorner t_1 \urcorner][x_2 \leftarrow \ulcorner t_2 \urcorner] \dots [x_k \leftarrow \ulcorner t_k \urcorner]$ . Its application is such that:

$$\epsilon[x \leftarrow \ulcorner t \urcorner](y) = \begin{cases} \ulcorner t \urcorner & \text{if } y \equiv x \\ \epsilon(y) & \text{otherwise} \end{cases}$$

We extend the notion of environment to a morphism  $\epsilon'$  from PIL to PIL, and we note  $\mathcal{Env}$  the set of all environments.

DEFINITION 5. Given  $\mathcal{F}$  and  $\mathcal{X}$ , we define the mapping  $\Phi$  from environments to substitutions, by  $\Phi(\epsilon) = \sigma$  where:

$$\sigma = \{x_i \mapsto t_i \mid \epsilon(\ulcorner x_i \urcorner) = \ulcorner t_i \urcorner \text{ with } x_i \in \mathcal{X} \text{ and } t_i \in \mathcal{T}(\mathcal{F})\}$$

Hence, to prove the correctness of the compiled code  $\pi_p$ , we want to ensure that, for a given model of evaluation “eval” and for each term  $t$ , the following diagram commutes:

$$\begin{array}{ccc} p \ll t & \xrightarrow{\text{compile}} & \pi_p(\ulcorner t \urcorner) \\ \text{match} \downarrow & & \downarrow \text{eval} \\ \sigma & \xleftarrow{\text{abstract}} & \epsilon \end{array}$$

We are now going to make the evaluation mechanism explicit.

### 3.3 Big-step semantics

We use a big step semantics *à la Kahn* [?] to express the behavior of the PIL evaluation mechanism. The reduction relation of this big-step semantics is expressed on couples made of an environment and an instruction, denoted  $\langle \epsilon, i \rangle$ . The reduction relation for the big-step semantics is:

$$\langle \epsilon, i \rangle \mapsto_{bs} \langle \epsilon', i' \rangle, \text{ with } i, i' \in \langle instr \rangle, \text{ and } \epsilon, \epsilon' \in \mathcal{Env}$$

and the rules for the big-step semantics are presented in Figure ??. The presented semantics is quite standard, however, the reader should note that conditions are evaluated modulo a formal anchor  $\ulcorner \cdot \urcorner$  and the equivalences given in Section ??. In the line of Example ??, if we evaluate the program in the environment where  $s$  is bound to  $\ulcorner f(a) \urcorner$ , the condition  $[s \leftarrow \ulcorner f(a) \urcorner](\mathbf{is\_fsym}(s, \ulcorner f \urcorner)) \equiv \mathbf{true}$  is equivalent to the condition  $\ulcorner Symb(f(a)) = f \urcorner \equiv \mathbf{true}$ , which in this case is true since the top symbol of  $f(a)$  is  $f$ .





$$\pi_{g(x,b)}(\mathbf{s}) \triangleq$$

```

  if(is_fsym(s, ⌈g⌋),
    let(x1, subtermg(s, 1),
      let(x2, subtermg(s, 2),
        let(x, x1,
          if(is_fsym(x2, ⌈b⌋), accept, refuse))),
        refuse)

```

Given a term  $t$  and an environment  $\epsilon_0 = [\mathbf{s} \leftarrow \lceil t \rceil]$ , let us suppose that  $\langle \epsilon_0, \pi_{g(x,b)}(\mathbf{s}) \rangle \mapsto_{bs} \langle \epsilon', \text{accept} \rangle$ . Figure ?? shows the unique derivation that can be computed by applying the inference rules defined in Section ??.

To make this derivation possible, the following set of constraints has to be satisfied:

$$\mathcal{C}\pi_p(\mathbf{s}) = \begin{cases} \epsilon_0(\text{is\_fsym}(s, \lceil g \rceil)) & \equiv \text{true} & (1) \\ \epsilon_0(\text{subterm}_g(s, 1)) & \equiv \lceil t_{11} \rceil & (2) \\ \epsilon_1(\text{subterm}_g(s, 2)) & \equiv \lceil t_{12} \rceil & (3) \\ \epsilon_2(x_1) & \equiv \lceil t_{11} \rceil & (4) \\ \epsilon_3(\text{is\_fsym}(x_2, \lceil b \rceil)) & \equiv \text{true} & (5) \end{cases}$$

(1) and (5) can be simplified using the equations of the formal anchor, (2), (3), and (4) are tautologies. Thus, to prove the correctness of  $\pi_p$ , we have to prove the equivalence:

$$\begin{aligned} & \forall t \in \mathcal{T}(\mathcal{F}), \\ & \sigma(g(x, b)) = t \wedge \sigma = \{x \mapsto t_{11}\} \\ & \iff \\ & \text{Symb}(t) = g \wedge \text{Symb}(t_{12}) = b \end{aligned}$$

This is proved by first applying the substitution in the first part of the proof obligation, and then using the definitions of terms, symbols and subterms.

For this example, with  $g$  a function symbol of arity 2 and  $b$  a constant symbol, the mapping definition leads to the following axioms:

$$\begin{aligned} & \forall t \in \mathcal{T}(\mathcal{F}), \text{Symb}(t) = g \Leftrightarrow \exists x, y \in \mathcal{T}(\mathcal{F}), t = g(x, y) \\ & \forall t \in \mathcal{T}(\mathcal{F}), \text{Symb}(t) = b \Leftrightarrow (t = b) \\ & \forall x, y \in \mathcal{T}(\mathcal{F}), g(x, y)_{|1} = x \\ & \forall x, y \in \mathcal{T}(\mathcal{F}), g(x, y)_{|2} = y \end{aligned}$$

The first two axioms define the meaning of the  $\text{Symb}$  function. The two remaining axioms define the subterm function over terms rooted by the symbol  $g$ . As we use a first order prover, we need such a definition for each symbol function and for each subterm.

To prove the left to right implication, we simply apply the substitution to the left part, and then apply the first axiom, to obtain  $\text{Symb}(t) = g$ , and the fourth axiom to obtain  $\text{Symb}(t_{12}) = b$ .

To prove the remaining implication ( $\Leftarrow$ ), we apply the first axiom to the first constraint, obtaining  $\exists x, y$  such that  $t = g(x, y)$ . We then apply the third and fourth axiom, to instantiate  $x$  and  $y$  by  $t_{11}$  and  $t_{12}$ . The second constraint with the second axiom gives  $t_{12} = b$ . We can then obtain  $g(t_{11}, b)$ , and extract the substitution.

The form of such propositions is rather simple but a huge number of them could be generated, so this kind of proof should better be done by an automated theorem prover. In our implementation, we are using Zenon [?], a first order tableau based automatic theorem prover. One of the nice capability of Zenon is to generate a formal proof in Coq when a theorem can be proved. In our case, a witness of correctness is generated and associated to the generated code.

Another proof approach will be to use theorem proving modulo [?] using in particular the axioms issued from the mapping definition.

## 5. EXTENSION TO MATCH CONSTRUCTS

Our method can be extended to support *match* constructs à la ML, Caml or Tom. We consider not only single patterns, but also constructs of the form *match*  $s$  with  $(p_1 \rightarrow a_1), \dots, (p_n \rightarrow a_n)$ . The semantics of this construct is the following: if  $p_1$  matches the subject  $s$ , the program goes into the **accept** state, and to keep track of the pattern number, the **accept** state is labeled by  $p_1$ , noted **accept** <sub>$p_1$</sub> . Otherwise, the subproblem *match*  $s$  with  $(p_2 \rightarrow a_2), \dots, (p_n \rightarrow a_n)$  is considered. When no pattern  $p_i$  matches  $s$ , the program goes into the **refuse** state.

This new match construct can be easily compiled using the intermediate language PIL. However, to avoid code duplication and to ease expression in PIL, it is useful to consider the *sequence* construct:  $\langle instr \rangle ; \langle instr \rangle$ .

This sequence construct has the following big-step semantics:

$$\begin{aligned} & \frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle}{\langle \epsilon, i_1 ; i_2 \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle} & (seq_a) \\ & \frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', \text{refuse} \rangle \quad \langle \epsilon', i_2 \rangle \mapsto_{bs} \langle \epsilon'', i \rangle}{\langle \epsilon, i_1 ; i_2 \rangle \mapsto_{bs} \langle \epsilon'', i \rangle} & (seq_b) \end{aligned}$$

It is easy to show that adding the sequence rules does not break the property of uniqueness for the derivation of a well-formed instruction. The notion of correct compilation of a match construct is an extension of the definition of the correct compilation of a pattern. The difference comes from the presence of multiple patterns. Hence, when a pattern is selected to fire a rule (**accept** <sub>$p$</sub>  in our terminology), we should ensure that all previous patterns do not match the subject. In the following, we do not make any assumptions on the form of the code to validate. This ensures that we can consider any optimizations of the matching code, like factorization of common tests.

Let  $\mathcal{P}_m$  be the set of patterns for the match construct, and  $<$  a total ordering relation for patterns in  $\mathcal{P}_m$ . In the case of ML for example, we define  $<$  by the textual ordering:  $p_i < p_j$  if  $p_i$  occurs before  $p_j$  in the match construct.

DEFINITION 10. Given a formal anchor  $\lceil \cdot \rceil$ , a well-formed program  $\pi_m$  is a sound compilation of  $m$  when both:

$$\forall \epsilon \in \mathcal{Env}, \forall t \in \mathcal{T}(\mathcal{F}) :$$

$$\begin{aligned} & \forall p \in \mathcal{P}_m, \exists \epsilon' \in \mathcal{Env}, \langle \epsilon, \pi_m(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle \\ & \Rightarrow \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \text{ s.t. } p' < p, \Phi(\epsilon')(p') \neq t) & (M\text{sound}_{OK}) \end{aligned}$$

$$\begin{aligned} & \exists \epsilon' \in \mathcal{Env}, \langle \epsilon, \pi_m(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \text{refuse} \rangle \Rightarrow \forall p \in \mathcal{P}_m, p \not\ll t & (M\text{sound}_{KO}) \end{aligned}$$

DEFINITION 11. Given a formal anchor  $\lceil \cdot \rceil$ , a well-formed

$$\begin{array}{l}
\text{Let}_3 \triangleq \text{let}(x, x_1, \text{if}(\text{is\_fsym}(x_2, \ulcorner b \urcorner), \text{accept}, \text{refuse})) \\
\text{Let}_2 \triangleq \text{let}(x_2, \text{subterm}_g(s, 2), \text{Let}_3) \\
\text{Let}_1 \triangleq \text{let}(x_1, \text{subterm}_g(s, 1), \text{Let}_2)
\end{array} \left| \begin{array}{l}
\epsilon_0 = [s \leftarrow \ulcorner t \urcorner] \\
\epsilon_1 = \epsilon_0[x_1 \leftarrow \ulcorner t_{11} \urcorner] \\
\epsilon_2 = \epsilon_1[x_2 \leftarrow \ulcorner t_{12} \urcorner] \\
\epsilon_3 = \epsilon_2[x \leftarrow \ulcorner t_{11} \urcorner]
\end{array} \right.$$

$$\frac{\langle \epsilon_3, \text{accept} \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle}{\langle \epsilon_3, \text{if}(\text{is\_fsym}(x_2, \ulcorner b \urcorner), \text{accept}, \text{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle} \boxed{5} = \epsilon_3(\text{is\_fsym}(x_2, \ulcorner b \urcorner)) \equiv \text{true}$$

$$\frac{\langle \epsilon_2, \text{let}(x, x_1, \text{if}(\text{is\_fsym}(x_2, \ulcorner b \urcorner), \text{accept}, \text{refuse})) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle}{\langle \epsilon_1, \text{let}(x_2, \text{subterm}_g(s, 2), \text{Let}_3) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle} \boxed{4} = \epsilon_2(x_1) \equiv \ulcorner t_{11} \urcorner$$

$$\frac{\langle \epsilon_1, \text{let}(x_2, \text{subterm}_g(s, 2), \text{Let}_3) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle}{\langle \epsilon_0, \text{let}(x_1, \text{subterm}_g(s, 1), \text{Let}_2) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle} \boxed{3} = \epsilon_1(\text{subterm}_g(s, 2)) \equiv \ulcorner t_{12} \urcorner$$

$$\frac{\langle \epsilon_0, \text{let}(x_1, \text{subterm}_g(s, 1), \text{Let}_2) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle}{\langle \epsilon_0, \text{if}(\text{is\_fsym}(s, \ulcorner g \urcorner), \text{Let}_1, \text{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle} \boxed{2} = \epsilon_0(\text{subterm}_g(s, 1)) \equiv \ulcorner t_{11} \urcorner$$

$$\frac{\langle \epsilon_0, \text{if}(\text{is\_fsym}(s, \ulcorner g \urcorner), \text{Let}_1, \text{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle}{\langle \epsilon_0, \text{if}(\text{is\_fsym}(s, \ulcorner g \urcorner), \text{Let}_1, \text{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \text{accept} \rangle} \boxed{1} = \epsilon_0(\text{is\_fsym}(s, \ulcorner g \urcorner)) \equiv \text{true}$$

Figure 4: Example of derivation leading to `accept`. We have  $\mathcal{C}\pi_p(s) = \{\boxed{1} \wedge \boxed{2} \wedge \boxed{3} \wedge \boxed{4} \wedge \boxed{5}\}$

program  $\pi_m$  is a complete compilation of  $m$  when both:

$\forall \epsilon \in \mathcal{Env}, \forall t \in \mathcal{T}(\mathcal{F}) :$

$$\begin{array}{l}
\forall p \in \mathcal{P}_m, p \ll t \wedge (\forall p' \in \mathcal{P}_m \text{ s.t. } p' < p, p' \not\ll t) \Rightarrow \\
\exists \epsilon' \in \mathcal{Env}, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle \wedge \Phi(\epsilon')(p) = t \\
\wedge (\forall p' \in \mathcal{P}_m \text{ s.t. } p' < p, \Phi(\epsilon')(p') \neq t) \\
(Mcomplete_{OK})
\end{array}$$

$$\forall p \in \mathcal{P}_m, p \not\ll t \Rightarrow \exists \epsilon' \in \mathcal{Env}, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \text{refuse} \rangle \\
(Mcomplete_{KO})$$

A compilation of a pattern  $p$  into a program  $\pi_p$  is said *correct*, when it is sound and complete.

**PROPERTY 5.** *The derivation of a well-formed instruction  $i$  in  $\langle instr \rangle$  in an environment  $\Gamma, \Delta$ , in the extended language, leads either to `acceptp` or `refuse`, and the reduction is unique.*

**PROOF.** We proceed by induction over the structure of instructions. The proof is similar to the proof of Property ??.

We extend the type system presented Figure ?? with the rule:

$$\frac{\Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash i_1 ; i_2 : wf}$$

Let  $i = i_1 ; i_2$  be a sequence. By induction, the reduction of  $i_1$  is unique and leads either to `acceptp` or `refuse`. In the first case,  $(seq_a)$  is applicable. The reduction of  $i_1 ; i_2$  is equal to the reduction of  $i_1$ , so it is unique. In the second case,  $(seq_b)$  is applicable. Since the reduction of  $i_2$  is unique, the reduction of  $i = i_1 ; i_2$  is unique.  $\square$

**THEOREM 3.** *Given a formal anchor  $\ulcorner \cdot \urcorner$ ,  $m$  a match construct, and  $\pi_m \in \text{PIL}$  a well-formed program, we have:*

$\pi_m$  is a correct compilation of  $m$

$\iff$

$\forall \epsilon \in \mathcal{Env}, \forall t \in \mathcal{T}(\mathcal{F}), \forall p \in \mathcal{P}_m :$

$$\begin{array}{l}
\exists \epsilon' \in \mathcal{Env}, \langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle \\
\iff \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \text{ s.t. } p' < p, \Phi(\epsilon')(p') \neq t)
\end{array}$$

**PROOF.** We want to show, as in Property ??, that  $(Msound_{OK}) \Rightarrow (Mcomplete_{KO})$  and  $(Mcomplete_{OK}) \Rightarrow (Msound_{KO})$ .

In the first case, assume  $(Msound_{OK})$  and  $\forall p \in \mathcal{P}_m, p \not\ll t$ . Since the reduction of  $\langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle$  is unique, we cannot have a reduction of  $\langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \text{accept}_p \rangle$ . This reduction exists, hence we have  $\langle \epsilon, \pi_m(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \text{refuse} \rangle$ . The second case can be proved in a similar way.  $\square$

In order to prove that the compilation  $\pi_m$  of a match constructs  $m$  is correct, we have to consider each statement `acceptp` in the program separately. For each pattern  $p$  in the match construct, we build all derivations in  $\mapsto_{bs}$  leading to `acceptp`, and deduce from it a constraint, formed by a disjunction of conjunctions of single constraints. We can then for each constraint prove the corresponding proof obligation, as expressed in Theorem ??.

Let us note that the number of derivations to build is polynomial in the size of the patterns. The maximum power will be given by the number of patterns in the match construct minus one. However, in practical applications a large number of derivations can be avoided because many patterns are disjoint.

## 6. EARLY EXPERIMENTAL RESULTS

The presented work has been implemented and applied to the intermediate language of Tom [?]. Tom is a language extension which adds pattern matching primitives to C, Java, and Caml. One particularity is to provide support for matching modulo sophisticated theories, like associative operators with neutral element. However, in this work, we only considered the case of the empty theory (i.e. syntactic matching), with possibly non-linear patterns.

Tom is based on the notion of formal anchor presented in Section ??. Thus, it is data structure independent, and customizable for any term implementation. Considering a simple term implementation in C, for example, we can define the following anchor:

```

struct term { int symbol;
              int arity;
              struct term **subterm; };
%typeterm Term {
  implement { struct term* }
  get_subterm(t,n) { t->subterm[n] }
  equal(t1,t2) { term_equal(t1,t2) }
}
%op Term a { is_fsym(t) { t->symbol == A } }
%op Term b { is_fsym(t) { t->symbol == B } }
%op Term f(Term) { is_fsym(t) { t->symbol == F } }

```

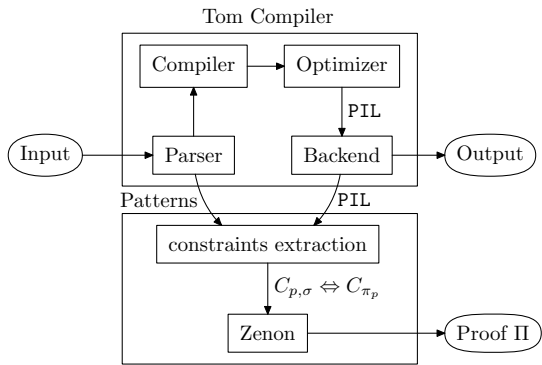


Figure 5: Global architecture of Tom

Given a `%match` construct, as illustrated by Figure ??, the compiler translates patterns into PIL instructions, which use the previously defined formal anchor. In practice, this mapping is supposed correct, in the sense that structural properties of terms should be preserved. To simplify this task, when no particular data-structure is required, a generator of term based implementations, coupled with a generator of formal anchors, can be used [?].

To prove the generated PIL code correct, we recently added to Tom a component (constraints extractor) which generates, for each pattern  $p$ , the constraints  $C_{\pi_p}$  and  $C_{p,\sigma} = (\exists\sigma, \sigma(p) = t)$ , where  $t$  is the input term. In a second step, these two constraints are sent to a prover to show their equivalence. To experiment our approach, we used Zenon, because, in addition to be fully automatic, it can generate a Coq formal proof when it succeeds. This is essential in our “skeptical” approach since it allows the user of the generated program to verify the proof by himself.

The verification tool is integrated into the Tom architecture, but note that no support from the internal compiler is needed:  $C_{p,\sigma}$  are extracted from the AST produced by the parser, and  $C_{\pi_p}$  are extracted from the PIL program produced by the compiler, or any other component such as an optimizer for example. Seeing the compiler as a black-box allows us to perform any kind of optimization unless PIL code is generated. At the moment we handle only the intermediate code of the compiler, ignoring the parser and the code generator. In our case, the backend performs a so straightforward *one-to-one* translation, from PIL instructions to host language instructions, that we trust in its correctness.

The interest of this approach is to allow to verify code corresponding to an optimized many-to-one algorithm, where common tests are factored.

To illustrate the applicability of the present approach, we tested our validator on several small examples, all of which worked with success, in an efficient way. For a more realistic test, to show how the approach scales to biggest problems, we generated proof obligations corresponding to the compilation of Tom itself (written in Tom). 184 patterns were extracted by the parser, after discarding associative patterns. From this set of patterns, 1018 applications of inference rules were needed to compute all derivations which lead to `accept`. This step generated 834 constraints ( $C_{\pi_p}$ ). Most of them were tautologies of the form  $\epsilon(\text{subterm}_g(s, 1)) \equiv s|_1$ . After a first step of simplification, 273 remaining constraints

were simplified using 2533  $\equiv$ -equivalence relation steps. On a PowerMac G5 (2GHz), the compilation of Tom, with the generation of theorems to prove, only increased the compilation time by 20% (going from 70 seconds to 84 seconds). This clearly shows that the approach can scale to large applications. In the current implementation, the translation to Zenon formalism is done fully automatically starting from the Tom program.

## 7. CONCLUSION AND FUTURE WORKS

When using a compiler, we always think it is correct. When writing a compiler, we know it is incorrect. This drives us to present a framework addressing the specific problem of generated pattern matching code validation. The main benefit of such an approach over a traditional compiler is that the compiler output is formally checked after each compilation, thus simplifying testing and development and providing a way to prove the formal validity of the generated code. We have seen that the proposed approach is powerful and flexible enough to validate the compilation of *match* constructs à la Caml or Tom.

We are now attacking the challenging problem of extending this method to support matching with associative (list) operators, like those of the Tom language, and with associative-commutative operators, like in many rewriting based languages like ELAN. Matching modulo theories is much more elaborated than syntactic matching, and so is writing such a pattern matching compiler. Validation of the produced code can then help developing and debugging new optimizations for these matching theories. Furthermore, when matching modulo theories, a new completeness problem has to be solved: the generated matching code has not only to find a substitution if the matching problem has one, but may have to produce *all* possible solutions for the matching problem.

The approach proposed here generates proof obligations of a very strict form. These proof obligations are in general easy to prove, but we should investigate our current conjecture that this class of problems is indeed decidable.

Although we were only interested in this work in the correctness of the generated code against the source problem, some additional properties of the source system could be proved by this method. For example the completeness of definition of a function defined by pattern matching could be proved by showing that there is no possible reduction to `refuse`.

Finally, our ultimate goal is to formally prove the correct compilation of the normalization process induced by a rewrite system. Proving the correct compilation of rewrite system execution will allow us to safely deduce on the program produced by the compilation of a rewriting specification the properties proved for this specification, like termination or confluence. This paper is a first but crucial step in this direction.

**Acknowledgments:** We would like to thank Luigi Liquori for his inside full reading on a preliminary version of this paper, Pierre Weis and Germain Faure for useful comments on this work. Special thanks are due to Damien Doligez for fruitful discussions and his help in connecting our tool to Zenon. We also thank the anonymous referees for valuable comments and suggestions that led to a substantial improvement of the paper.

## 8. REFERENCES

- [1] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
- [2] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
- [3] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction*, pages 416–430. Springer-Verlag, 1992.
- [4] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In Aart Middeldorp, editor, *Proceedings of RTA'2001*, volume 2051 of *LNCS*, Utrecht (The Netherlands), May 2001. Springer-Verlag.
- [5] Damien Doligez. Zenon: an automatic theorem prover for first-order logic. Available as part of the Focal system at <http://focal.inria.fr/download>.
- [6] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.
- [7] 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.
- [8] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–352. ACM Press, 2004.
- [9] Gilles Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.
- [10] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.
- [11] Andreas Krall and Jan Vitek. On extending java. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 321–335. Springer-Verlag, 1997.
- [12] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 283–294. ACM Press, 2002.
- [13] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19*, pages 33–41. AMS, 1967.
- [14] 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*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
- [15] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 144–152. ACM Press, 1973.
- [16] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
- [17] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.
- [18] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp Symb. Comput.*, 8(1-2):111–182, 1995.
- [19] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.
- [20] Martin C. Rinard and Darko Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.
- [21] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2004.
- [22] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–79, April 2005.
- [23] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, 1987.
- [24] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming*, pages 264–274. ACM Press, 2003.