The New Rewriting Engine of Dedukti

Gabriel Hondet, Frédéric Blanqui

The New Rewriting Engine of Dedukti

Gabriel Hondet
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria
Laboratoire Spécification et Vérification, 94235, Cachan, France

Frédéric Blanqui
Université Paris-Saclay, ENS Paris-Saclay, CNRS, Inria
Laboratoire Spécification et Vérification, 94235, Cachan, France

Abstract

Dedukti is a type-checker for the λΠ-calculus modulo rewriting, an extension of Edinburgh's logical framework LF where functions and type symbols can be defined by rewrite rules. It therefore contains an engine for rewriting LF terms and types according to the rewrite rules given by the user. A key component of this engine is the matching algorithm to find which rules can be fired. In this paper, we describe the class of rewrite rules supported by Dedukti and the new implementation of the matching algorithm. Dedukti supports non-linear rewrite rules on terms with binders using higher-order pattern-matching as in Combinatory Reduction Systems (CRS). The new matching algorithm extends the technique of decision trees introduced by Luc Maranget in the OCaml compiler to this more general context.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Operational semantics

Keywords and phrases rewriting, higher-order pattern-matching, decision trees

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.12

Category System Description

Related Version `https://hal.inria.fr/hal-02317471`

Supplement Material `https://github.com/deducteam/lambdapi.git`

Acknowledgements The authors thank Bruno Barras and Rodolphe Lepigre for their help in developing the new rewriting engine of Dedukti.

1 Introduction

Dedukti is primarily a type-checker for the so-called λΠ-calculus modulo rewriting, $\lambda\Pi/R$, an extension of Edinburgh's logical framework LF [9] where function and type symbols can be defined by rewrite rules. This means that Dedukti takes as input type declarations and rewrite rules, and check that expressions are well typed modulo these rewrite rules and the β-reduction of λ-calculus.

The λΠ-calculus is the simplest type system on top of the pure untyped λ-calculus combining both the usual simple types of (functional) programming (*e.g.* the type $\mathbb{N} \to \mathbb{N}$ of functions from natural numbers to natural numbers) with value-dependent types (*e.g.* the type $\Pi n : \mathbb{N}, V(n)$ of vectors of some given dimension). In fact, a simple type $A \to B$ is just a particular case of dependent type $\Pi x : A, B$ where x does not occur in B. Syntactically, this means that types are not defined prior to terms as usual, but that terms and types are mutually defined.

Moreover, in $\lambda\Pi/R$, a term of type A is also seen as a term of type B if A and B are equivalent not only modulo β-reduction but also modulo some user-defined rewrite rules R. Therefore, to check that a term t is of type A, one has to be able to check when two

expressions are equivalent modulo β-reduction and rewrite rules. This is why there is a rewriting engine in Dedukti.

Thanks to the Curry-Howard correspondence between λ-terms and proofs on the one hand, and (dependent) types and formulas on the other hand, Dedukti can be used as a proof checker. Hence, in recent years, many satellite tools have been developed in order to translate to Dedukti proofs generated by automated or interactive theorem provers: Krajono for Matita, Coqine for Coq, Holide of OpenTheory (HOL Light, HOL4), Focalide for Focalize, Isabelle, Zenon, iProverModulo, ArchSAT, etc. [1].

By unplugging its type verification engine to only retain its rewriting engine, Dedukti can also be used as a programming language. Thanks to its rewriting capabilities, Dedukti can be used to apply transformation rules on terms and formulas with binders [18, 4].

A rewrite rule is nothing but an oriented equation [2]. Rewriting consists in applying some set of rewrite rules R (and β-reduction) as long as possible so as to get a term in (weak head) normal form. At every step it is therefore necessary to check whether a term matches some left-hand side of a rule of R. It is therefore important to have an efficient algorithm to know whether a rule is applicable and select one:

▶ **Example 1.** Consider the following rules in the new Dedukti syntax (pattern variables are prefixed by `$` to avoid name clashes with other symbols):

```
rule f (c (c $x)) a  $x
with f       $x   b  $x
```

To select the correct rule to rewrite a term, the naive algorithm matches the term against each rule left-hand side from the top rule to the bottom one. Let us apply the algorithm on the matching of the term `t = f (c (c e)) b`.

The first argument of `t` is matched against the first argument of the first left-hand side `c (c $x)`. As `c (c e)` matches `c (c $x)`, it succeeds. However, when we pass to the second argument, `b` does not match the pattern `a`. So the second rule is tried. Pattern `$x` filters successfully `c (c e)`, and `b` matches `b`, so it succeeds.

Yet, matching `c (c e)` against `c (c $x)` can be avoided. Indeed, if we start by matching the second argument of `f`, then the first rule is rejected in one comparison. The only remaining work is to match `c (c e)` against `$x`.

In [14], Maranget introduces a domain-specific language of so-called decision trees for describing matching algorithms, and a procedure for compiling some set of rewrite rules into this language. But his language and compilation procedure handle rewrite systems whose left-hand sides are linear constructor patterns only. In Dedukti, as we are going to see it soon, we use a more general class of patterns containing defined symbols and λ-abstractions. They can also be non-linear and contain variable-occurrence conditions as in Klop's Combinartory Reduction Systems (CRS) [11].

In this paper, we describe an extension of Maranget's work to this more general setting, and present some benchmark.

Outline of the paper In Section 2, we start by giving examples of the kind of rewrite rules that can be handled by Dedukti, before giving a more formal definition. In Section 3, we present our extension of Maranget's decision trees, their syntax and semantics, and how to compile a set of rewrite rules into this language. In Section 4, we compare this new implementation with previous ones and other tools implementing rewriting. Finally, in Section 5, we discuss some related work and conclude.

2 Rewriting in Dedukti

We will start by providing the reader with various examples of rewrite rules accepted by Dedukti before giving a formal definition. To this end, we will use the new Dedukti syntax (see `https://github.com/Deducteam/lambdapi`). In this new syntax, one can use Unicode characters, some function symbols can be written in infix positions and, in rewrite rules, pattern variables need to be prefixed by `$` to avoid name clashes with function symbols. Note however that, for the sake of simplicity, we may omit some declarations.

Dedukti can of course handle the "Hello world!" example of first-order rewriting, the addition on unary natural numbers, as follows:

```
symbol : TYPE      symbol 0:       symbol s:

symbol +:

rule      0  + $m  $m
with (s $n) + $m  s ($n + $m)
```

More interestingly is the fact that, in constrast to functional programming languages like OCaml, Haskell or Coq, rule left-hand sides can overlap each other. Consequently, in Dedukti, addition on unary numbers can be more interestingly defined as follows:

```
rule       0 + $m       $m
with (s $n) + $m       s ($n + $m)
with      $m + 0       $m
with      $m + (s $n)  s ($m + $n)
```

With the first definition, one has $0 + t$ equivalent to t modulo rewriting, written $0 + t \simeq t$, for all terms t (of type $\mathbb{N}$), but not $t + 0 \simeq t$. Hence, the interest of the second definition.

It is also possible to match on defined symbols and not just on constructors like in usual functional programming languages. Hence, for instance, one can add the following associativity rule on addition:

```
rule ($x + $y) + $z  $x + ($y + $z)
```

Moreover, one can use non-linear patterns, that is, require the equality of some subterms to fire a rule like in:

```
rule $x + (- $x)  0
```

Therefore, Dedukti can handle any first-order rewriting system [2]. But it can also handle higher-order rewriting in the style of Combinatory Reduction Systems (CRS) [11].

The simplest example of higher-order rewriting is given by the `map` function on lists, which applies an argument function to every element of a list:

```
symbol map: (  )  List  List

rule map $f (cons $x $l)  cons ($f $x) (map $f $l)
```

Unlike first-order rewriting, function symbols can be partially applied, including in patterns. Hence, in Dedukti, one can write the following:

```
symbol id:
rule id $x  $x
rule plus 0  id with plus (s $n) $m  s (plus $n $m)
rule map id $l  $l
```

It is also possible to match λ-abstractions as follows:

```
symbol cos:
symbol sin:
symbol *: ( ) ( ) ( )
symbol diff: ( ) ( )

rule diff(x,sin($v[x]))  diff(x,$v[x]) * cos
```

Following the definition of CRS, in a rule left-hand side, a higher-order pattern variable can only be applied to distinct bound variables (this condition could be slightly relaxed though [13]). A similar condition appears in λProlog [15]. It ensures the decidability of matching.

It can also be used to check variable-occurrence conditions. The differential of a constant function can thus be simply defined as follows in Dedukti:

```
rule diff(x,$v)  x,0
```

While in the rule for `sin`, we had `$v`$[x]$, meaning that the term matching `$v`$[x]$ may depend on `x`, here we have `$v` applied to no bound variables, meaning that the term matching `$v` cannot depend on x.

2.1 Terms, Patterns, Rewrite Rules and Matching, Formally

We now define more formally terms, patterns, rewrite rules and rewriting. Following [3], the terms of the $\lambda\Pi$-calculus are inductively defined as follows:

$$t, u ::= \texttt{TYPE} \mid \texttt{KIND} \mid x \mid f \mid tu \mid \lambda x : t,\ u \mid \Pi x : t,\ u$$

where x is a term variable, f is a function symbol, tu is the application of the function t to the term u, $\lambda x : t,\ u$ is the function mapping x of type t to u, which type is the dependent product $\Pi x : t,\ u$. The simple type $t \to u$ is syntactic sugar for $\Pi x : t,\ u$ where x is any fresh term variable not occurring in u.

In $\lambda x : t,\ u$ and $\Pi x : t,\ u$, the occurrences of x in u are bound, and terms equivalent modulo renaming of their bound variables are identified, as usual. In Dedukti, this is implemented by using the Bindlib library [12].

A (possibly empty) ordered sequence of terms $t_1, \ldots, t_n$ is written $\vec{t}$ for short.

Patterns are inductively defined as follows:

$$p ::= \$x\,[\vec{y}] \mid f\vec{p} \mid \lambda y,\ p$$

where $\$x$ is a pattern variable and $\vec{y}$ is a sequence of distinct bound variables.

A rewrite rule is a pair of terms, written $\ell \to r$, such that ℓ is a pattern of the form $f\vec{p}$ and every pattern variable occurring in r also occurs in ℓ.

In the following, we will assume given a set of user-defined rewrite rules R.

Matching a term t against a pattern p whose bound variables are in the set V, written $p \preceq_V t$ is inductively defined as follows:

$$\begin{array}{rll}
\$x\,[\vec{y}] \preceq_V t & \text{iff } \mathrm{FV}(t) \cap V \subseteq \{\vec{y}\} & \text{(MatchFv)} \\
\mathtt{f}\,p_1\ \ldots\ p_n \preceq_V \mathtt{f}\,t_1\ \ldots\ t_n & \text{iff } (p_1\ \ldots\ p_n) \preceq_V (t_1\ \ldots\ t_n) & \text{(MatchSymb)} \\
\lambda y, p \preceq_V \lambda y : A, t & \text{iff } p \preceq_{V \uplus \{y\}} t & \text{(MatchAbst)} \\
(p_1\ \ldots\ p_n) \preceq_V (t_1\ \ldots\ t_n) & \text{iff } \forall i, p_i \preceq_V t_i \wedge \forall j, p_i = p_j \Rightarrow t_i = t_j & \text{(MatchTuple)}
\end{array}$$

and we say that the term t *matches* the pattern p or that the pattern p *filters* the term t.

The indexing set V of variables is used to record which binders have been traversed, which is necessary to perform variable-occurrence tests.

The condition in the (MatchTuple) rule translates non-linearity conditions: if a variable occurs twice in a pattern, then the matching values must be equal.

3 Implementing Matching With Decision Trees

The rewriting engine described in this paper is based on the work of Maranget [14]. Maranget introduces a domain-specific language for matching and an algorithm to transform a (ordered) list of first-order linear constructor patterns into a program in this language. In this section, we explain how we extend Maranget's language and compilation procedure to our more general setting with non-linear higher-order patterns, partially applied function symbols, and no order on patterns.

We start by defining the language of decision trees D and switch case lists L:

$$\begin{array}{rcl} D, E & ::= & \texttt{Leaf}(r) \mid \texttt{Fail} \mid \texttt{Swap}_i(D) \mid \texttt{Store}(D) \mid \texttt{Switch}(L) \\ & \mid & \texttt{BinNl}(D, \{i, j\}, E) \mid \texttt{BinCl}(D, (n, X), E) \\ L & ::= & (s, D)::L \mid (\lambda, D)::L_\lambda \mid T \\ L_\lambda & ::= & (s, D)::L_\lambda \mid T \\ T & ::= & (*, D)::\texttt{nil} \mid \texttt{nil} \end{array}$$

where r is a rule right-hand side, i, j and n are integers, X is a finite set of variables. For case lists, s is a function symbol annotated with the number of arguments it is applied to, :: is the cons operator on lists and `nil` is the empty list.

An element of a switch case list is a pair mapping:

- a function symbol s to a tree for matching its arguments,
- a λ to a tree for matching the body of an abstraction,
- a default case $*$ to a tree for matching the other arguments.

Note that a list L_λ has no element (λ, D) and, in a list L, there is at most one element of the form (λ, D). Finally, in both cases, there is at most one element of the form $(*, D)$ and, if so, it is the last one (default case).

Semantics Decision trees are evaluated along with a stack of terms $\vec{v}$ to filter and an array $\vec{s}$ used by the decision tree to store elements. Informally, the semantics of each tree constructor is as follows:

`Leaf`(r) matching succeeds and yields right-hand side r.

`Fail` matching fails.

`Swap`$_i(D)$ moves the ith element of $\vec{v}$ to the top of $\vec{v}$ and carries on with D.

`Store`(D) stores the top of the stack into $\vec{s}$ and continues with D.

`Switch`(L) branches on a tree in L depending on the term on top of $\vec{v}$.

`BinNl`$(D, \{i, j\}, E)$ checks whether s_i and s_j are equal and continues with D if this is the case, and E otherwise.

`BinCl`$(D, (n, X), E)$ checks whether $\mathrm{FV}(s_n) \subseteq X$ and continues with D if this is the case, and with E otherwise.

▶ **Example 2.** The matching algorithm described in Example 1 can be represented by the following decision tree:

$$\begin{array}{l} \texttt{Swap}_2(\texttt{Switch}([(\texttt{a}_0, \texttt{Switch}([(\texttt{c}_0, \texttt{Switch}([(\texttt{c}_0, \texttt{Leaf}(\$x))]))]); \\ \qquad\qquad (\texttt{b}_0, \texttt{Leaf}(\$x))])) \end{array}$$

which can be graphically represented as follows: where leaves are the right-hand sides of the

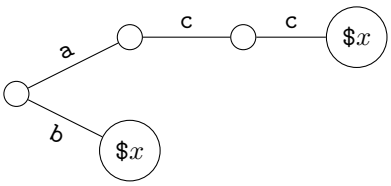

Figure 1 Graphical representation of the decision tree of Example 2

rewrite rules and a path from the root to a leaf is a successful matching. The tree of Figure 1 can be used to rewrite any term of the form $\texttt{f}\ \vec{t}$. The sequence of operations to filter the term $\texttt{f}\ (\texttt{f}\ \texttt{a})\ \texttt{b}$ can be read from the tree. The initial vector $\vec{v}$ is $\vec{v} = (\texttt{f}\ \texttt{a}, \texttt{b})$ and the array $\vec{s}$ won't be necessary here.

1. The $\texttt{Swap}_2$ transforms $\vec{v}$ into $(\texttt{b}, \texttt{f}\ \texttt{a})$, so the next operations will be carried out on $\texttt{b}$.
2. The `Switch` node with the case list $[(\texttt{a}_0, D), (\texttt{b}_0, E)]$ allows to branch on D or E depending on the term on top of $\vec{v}$, that is, $\texttt{b}$. Since $\texttt{b}$ is applied to no argument, it matches $\texttt{b}_0$ and filtering continues on E. The stack is now $\vec{v} = (\texttt{f}\ \texttt{a})$.
3. Node E is in fact a `Leaf` and so the matching succeeds.

Note that the top symbol `f` is not matched. Top symbols are analysed prior to filtering as they are needed to get the appropriate decision tree to filter the arguments.

The formal semantics is given in Figure 2. An evaluation is written as a judgement $\vec{v}, \vec{s}, V \vdash D \rightsquigarrow r$ which can be read: "stack $\vec{v}$, store $\vec{s}$ and abstracted variables V yield the term r when matched against tree D". We overload the comma notation, using it for the cons $(\texttt{s}, \vec{v})$ and the concatenation $(\vec{v}, \vec{w})$. The $|$ is used as the alternative.

Matching succeeds with the Match rule. Terms are memorised on the stack $\vec{s}$ using the Store rule. Matching on a symbol is performed with the SwitchSymb rule. If the stack has a term `f` applied on top and the switch-case list L contains an element $(\texttt{f}, D)$, then the symbol `f` can be removed, and matching continues using sub-tree D. The rule SwitchDefault allows to match on any symbol or abstraction, provided that the switch-case list L has a default case (and that we can apply neither rule SwitchSymb nor SwitchAbst). The binary constraint rules guide the matching depending on failure or success of the constraints. The last three rules allow to search for a symbol in a switch-case list. A judgement $s \vdash L \rightsquigarrow p$ reads "looking for symbol s in list L yields pair p". Cont skips a cell of the list, Default returns unconditionally the default cell of the list (which is the last by construction) and Found returns the cell that matches the symbol looked for.

3.1 Matrix Representation of Rewrite Systems

In order to compile a set of rewrite rules into this language, it is convenient to represent rewrite systems as tuples containing a matrix and three vectors. The matrix contains the patterns and can have lines of different lengths because function symbols can be partially applied. The vectors contain the right-hand side of the rewriting system and the constraints. Hence, a rewrite system for a function symbol f, that is, a set of rewrite rules $f\vec{p}^1 \to r^1, \ldots, f\vec{p}^m \to r^m$

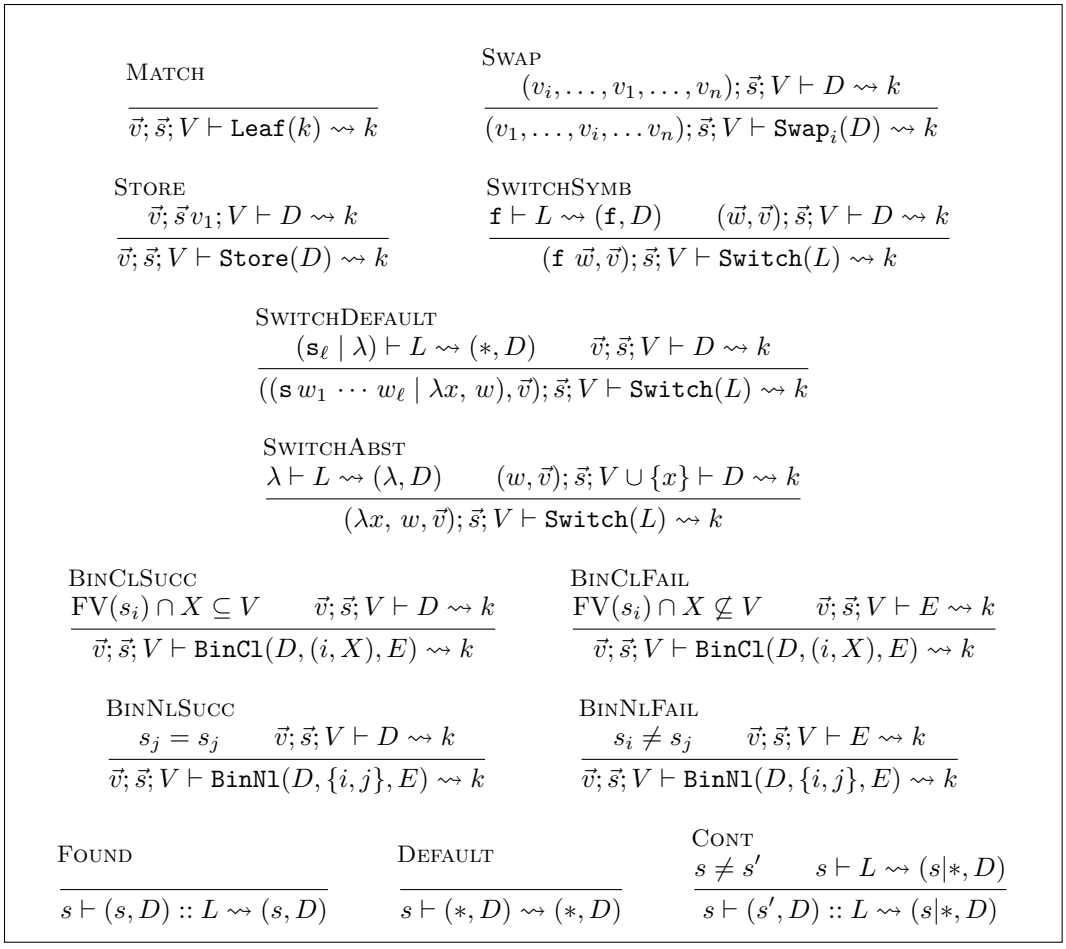

Figure 2 Evaluation of decision trees

is represented by:

$$\left(\begin{bmatrix} p_1^1 & \cdots & p_{n_1}^1 \\ p_1^2 & \cdots & p_{n_2}^2 \\ & \vdots & \\ p_1^m & \cdots & p_{n_m}^m \end{bmatrix}, \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_m \end{bmatrix}, \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_m \end{bmatrix}, \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix}\right)$$

where C_i encodes the variable-occurrence constraints in $\vec{p^i}$ and N_i encodes non-linearity constraints in $\vec{p^i}$.

A variable-occurrence constraint given by a pattern variable $\$x\,[\vec{y}]$ is encoded as a pair $(a, \vec{y})$ where a is the position of the variable in the main term.

Non-linearity constraints between two terms at positions a and b are encoded by the unordered pair $\{a, b\}$.

In the above matrix, we can then replace a pattern of the form $\$x\,[\vec{y}]$ or $\$x$ by $\_$.

For the sake of completeness, we recall the definition of positions:

▶ **Definition 3** (Positions in a term). *The set of* positions *of a term t is the set of words over the alphabet of positive integers inductively defined as follows:*

- $\mathcal{P}os(x) \triangleq \{\epsilon\}$
- $\mathcal{P}os(\mathtt{f}\ t_1\ \cdots\ t_n) \triangleq \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ia \mid a \in \mathcal{P}os(t_i)\}$
- $\mathcal{P}os(\lambda x,\, t) \triangleq \{\epsilon\} \cup \{1a \mid a \in \mathcal{P}os(t)\}$

The position ϵ is called the root position *of the term t and the symbol at this position is called the* root symbol *of t.*

For $a \in \mathcal{P}os(t)$, the subterm of t at position a*, denoted by $t|_a$, is defined by induction on the length of a, $t|_\epsilon \triangleq t$ and $\mathtt{f}\ t_1\ \cdots\ t_n|_{ia} \triangleq t_i|_a$*

The notion of position is extended to sequences of terms by taking $\vec{t}\,\big|_{ia} \triangleq t_i|_a$.

▶ **Example 4.** The rewrite system

```
rule f a   (λx,λy,$g[x]) ↪ 0
with f $x  $x            ↪ 1
with f a   b             ↪ 2
```

is represented by the following matrix:

$$\left(\begin{bmatrix} \mathtt{a} & \lambda x,\, \lambda y,\, \_ \\ \_ & \_ \\ \mathtt{a} & \mathtt{b} \end{bmatrix}, \begin{bmatrix} \emptyset \\ \{\{1,2\}\} \\ \emptyset \end{bmatrix}, \begin{bmatrix} \{(211, (x))\} \\ \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}\right).$$

The variable-occurence constraint of the first rule is encoded by $(211, (x))$ since only the variable x is authorised in $\$g\,[x]$. The non-linearity constraint `f &x &x` is translated by $\{1, 2\}$, hence the constraints set $\{\{1, 2\}\}$.

3.2 Compiling Rewrite Systems to Decision Trees

We will describe the compilation process as a non-deterministic recursively defined relation $\triangleright$ between matrices and decision trees.

To this end, we use the transformations on matrices defined in Table 1.

- $\mathsf{Spec}\left(\mathtt{f}, a, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right)$ keeps rows whose first pattern filters the application of function `f` a arguments:

Pattern p_1^j	Rows of $\mathsf{Spec}(\mathtt{f}, a, P \to A)$	Rows of $\mathsf{Spec}_\lambda(P \to A)$	Rows of $\mathsf{Def}(P \to A)$
$\mathtt{f}\ q_1\ \cdots\ q_a$	$q_1\ \cdots\ q_a\ p_2^j\ \cdots p_n^j$	No row	No row
$\mathtt{f}\ q_1\ \cdots\ q_b$	No row if $a \neq b$	No row	No row
$\mathtt{f}\ q_1\ \cdots\ q_b$	No row	No row	No row
$\lambda x,\ q$	No row	$q\ p_2^j\ \cdots p_n^j$	No row
$\_$	$\overbrace{\_\ \cdots\ \_}^{\times a}$	$\_\ p_2^j\ \cdots\ p_n^j$	$p_2^j\ \cdots\ p_n^j$

Table 1 Decomposition operators

▶ **Example 5.** Let $P = \begin{bmatrix} \mathtt{r}\ \$x & \mathtt{q} \\ \mathtt{r} & \mathtt{f}\ \$x \\ \$x & \mathtt{r} \\ \lambda x,\ \$x & \lambda x,\ \mathtt{r} \end{bmatrix}$. Then,

$$\mathsf{Spec}\left(\mathtt{r}, 1, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right) = \left(\begin{bmatrix} \$x & \mathtt{q} \\ \_ & \mathtt{r} \end{bmatrix}, \vec{N}, \vec{C}, \begin{bmatrix} r_1 \\ r_3 \end{bmatrix}\right)$$

- $\mathsf{Spec}_\lambda\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right)$ keeps rows whose first pattern filters a λ-abstraction:
  ▶ **Example 6.** Let P be the same as in Example 5.

$$\mathsf{Spec}_\lambda\left(P, \vec{N}, \vec{C}, \vec{r}\right) = \left(\begin{bmatrix} \_ & \mathtt{r} \\ \$x\,[x] & \lambda x,\ \mathtt{r} \end{bmatrix}, \vec{N}, \vec{C}, \begin{bmatrix} r_3 \\ r_4 \end{bmatrix}\right)$$

- $\mathsf{Def}\left(P, \vec{N}, \vec{C}, \vec{r}\right)$ keeps rows whose first pattern is a pattern variable:
  ▶ **Example 7.** Let P be the same as in Example 5.

$$\mathsf{Def}\left(P, \vec{N}, \vec{C}, \vec{r}\right) = \left(\left[\mathtt{r}\right], \vec{N}, \vec{C}, \left[r_3\right]\right)$$

To sum up, given a pattern matrix P, a simplification function removes rows of P that are not compatible with some assumption on the form of the first pattern.

The same idea is used for constraints. Note that we will abuse set notations and write $k \in N$ or $N\backslash\{k\}$ even if N is not a set of elements of the type of k. In that case $k \in N$ is false and $N\backslash\{k\}$ is N.

- $\mathsf{csucc}(k, (P, \vec{N}, \vec{C}, \vec{r}))$ keeps all the rows and simplify the constraint sets

$$\mathsf{csucc}(k, (\vec{p}, N, C, r)) \triangleq (\vec{p}, N\backslash\{k\}, C\backslash\{k\}, r)$$

- $\mathsf{cfail}(k, (P, \vec{N}, \vec{C}, \vec{r}))$ keeps rows that don't have k in their constraint sets

$$\mathsf{cfail}(k, (\vec{p}, N, C, r)) \triangleq \begin{cases} \text{No row} & \text{if } k \in N \text{ or } k \in C \\ (\vec{p}, N, C, r) & \text{if } k \notin N \text{ and } k \notin C \end{cases}$$

A compilation process consists in reducing the matrix step by step, compiling the sub-matrices and aggregating the sub-trees obtained using the node that corresponds to the computed sub-matrices (e.g. a `Switch` if the Spec, Def and Spec_λ sub-matrices have been computed).

To say that the matrix $\left(P, \vec{N}, \vec{C}, \vec{r}\right)$ compiles to the decision tree D, we write

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright D$$

where:

- $\vec{\rho}$ are the positions in the term that will be matched against during evaluation.
- $\mathcal{E}$ is a map from positions to integers such that $\mathcal{E}(\rho)$ is the index in $\vec{s}$ of the subterm at position ρ used during the evaluation of decision trees. The empty map is denoted $\emptyset$.
- n is the size of the store, which is incremented each time an element is added.

We now describe the compilation process implemented in Dedukti:

▶ **Definition 8** (Compilation). **1.** *If the matrix P has no row ($m = 0$), then matching always fails, since there is no rule to match,*

$$\vec{\rho}, \left(\emptyset, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E} \rhd \texttt{Fail} \tag{2}$$

2. *If there is a row k in P composed of unconstrained variables, matching succeeds and yields right-hand side r*

$$\left(\vec{\rho}, \begin{pmatrix} p_1^1 & \cdots & p_{n_1}^1 & N_1 & C_1 & \to & r_1 \\ & & & \vdots & & & \\ \_ & \cdots & \_ & \emptyset & \emptyset & \to & r_k \\ & & & \vdots & & & \\ p_1^m & \cdots & p_{n_m}^m & N_m & C_m & \to & r_m \end{pmatrix}, n, \mathcal{E}\right) \rhd \texttt{Leaf}(r_k) \tag{3}$$

3. *Otherwise, there is at least one row with either a symbol or a constraint or an abstraction. We can choose to either specialise on a column or solve a constraint.*

a. *Consider a specialisation on the first column, assuming it contains at least a symbol or an abstraction.*
If ρ_1 is constrained in some N_i or C_i, then define $n' = n + 1$ and $\mathcal{E}' = \mathcal{E} \cup \{\rho_1 \mapsto n\}$. Otherwise, let $n' = n$ and $\mathcal{E}' = \mathcal{E}$.
Let Σ be the set of root symbols of the terms of the first column and k the number of arguments f is applied to. Then for each $f \in \Sigma$, we compile

$$\left((\rho_1|_1 \ \cdots \ \rho_1|_k \ \ \rho_2 \ \cdots \ \rho_n), \mathsf{Spec}\left(f, k, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_{f_k}$$

Let L be the switch case list defined as (we use the bracket notation for list comprehension as the order is not important here)

$$L \triangleq [(f, D_{f_k}) | f \in \Sigma]$$

If there is an abstraction in the column, the Spec_λ sub-matrix is computed and compiled to D_λ, and an abstraction case is added to the mapping

$$\left((\rho_1|_1 \ \ \rho_2 \ \cdots \ \rho_n), \mathsf{Spec}_\lambda\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_\lambda$$
$$L \triangleq [(s, D_{f_k}) \mid f \in \Sigma] :: (\lambda, D_\lambda) :: \texttt{nil}$$

Similarly, if the column contains a variable, the Def *sub-matrix is computed and compiled to D_*, and the mapping is completed with a default case, (the abstraction case may or may not be present)*

$$\left((\rho_2 \ \cdots \ \rho_n), \mathsf{Def}\left(\left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n', \mathcal{E}'\right) \rhd D_*$$
$$L \triangleq [(f_k, D_{f_k}) | s \in \Sigma] :: (\lambda, D_\lambda) :: (*, D_*) :: \texttt{nil}$$

Now that the switch case list L is complete (all the symbols, the abstractions and the pattern variables are handled) and the sub-trees are defined and related to their pattern matrix, we can create the top node $\texttt{Switch}(L)$.
Furthermore, if ρ_1 is constrained, the term must be saved during evaluation. In that case, we add a `Store` *node,*

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright \texttt{Store}(\texttt{Switch}(L)).$$

Otherwise,

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright \texttt{Switch}(L).$$

b. *If a term has been stored and is subject to a closedness constraint, then this constraint can be checked.*
That is, for any position μ such that $\mathcal{E}(\mu)$ is defined and there is a constraint set C_i and a variable set V such that $(\mu, V) \in C_i$, we compute the sub-matrices csucc *and* cfail *and we compile them to D_s and D_f,*

$$\left(\vec{\rho}, \mathsf{csucc}\left((\mu, V), \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \triangleright D_s$$

$$\left(\vec{\rho}, \mathsf{cfail}\left((\mu, V), \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \triangleright D_f$$

with $(\mu, X) \in F^j$ for some row number j and we finally define

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright \texttt{BinCl}(D_s, (\mathcal{E}(\mu), X), D_f)$$

c. *A non linearity constraints can be enforced when the two terms involved in the constraint have been stored, that is, when there is a couple $\{\mu, \nu\}$ $(\mu \neq \nu)$ such that $\mathcal{E}(\mu)$ and $\mathcal{E}(\nu)$ are defined and there is a row j such that $\{\mu, \nu\} \in N_j$. If it is the case, then compute* csucc, cfail *and compile them,*

$$\left(\vec{\rho}, \mathsf{csucc}\left(\{\mu, \nu\}, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \triangleright D_s$$

$$\left(\vec{\rho}, \mathsf{cfail}\left(\{\mu, \nu\}, \left(P, \vec{N}, \vec{C}, \vec{r}\right)\right), n, \mathcal{E}\right) \triangleright D_f$$

and define

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright \texttt{BinNl}(D_s, \{\mathcal{E}(i), \mathcal{E}(j)\}, D_f)$$

4. *If column i contain either a symbol, an abstraction or a constraint, and each pattern vector of P is at least of length i, then compile $\left(\vec{\mu}, \left(P', \vec{N}, \vec{F}, \vec{r}\right), n, \mathcal{E}\right) \triangleright D'$ where $\vec{\mu} = (\rho_i\, \rho_1\, \ldots\, \rho_n)$ and P' is P with column i moved to the front; to build*

$$\left(\vec{\rho}, \left(P, \vec{N}, \vec{C}, \vec{r}\right), n, \mathcal{E}\right) \triangleright \texttt{Swap}_i(D') \tag{4}$$

▶ **Example 9** (Example 1, 2 continued)**.** We consider again the rewriting system used in Example 1. We start by computing the matrices:

$$(P, \emptyset, \emptyset, \vec{r}) \triangleq \left(\begin{bmatrix} \mathsf{c}\ (\mathsf{c}\ \_) & \mathsf{a} \\ \_ & \mathsf{b} \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \emptyset \\ \emptyset \end{bmatrix}, \begin{bmatrix} \$x \\ \$x \end{bmatrix}\right).$$

1. We saw that it is better to start examining the second argument, so we start by swapping columns of the matrix, $P' = \begin{bmatrix} \mathtt{a} & \mathtt{c}\ (\mathtt{c}\ \_) \\ \mathtt{b} & \_ \end{bmatrix}$, define D such that $(2\,1), (P', \emptyset, \emptyset, \vec{r}), 0, \emptyset \triangleright D$. and we thus have

   $$((1\,2), (P, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \triangleright \mathtt{Swap}_2(D).$$

2. To continue and compute D, we can match on the symbols of the first column of P' with a `Switch` node. For this, we compute
   - $P_{\mathtt{a}} = \mathsf{Spec}(\mathtt{a}, 0, P') = [\mathtt{c}\ (\mathtt{c}\ \_)]$, and
   - $P_{\mathtt{b}} = \mathsf{Spec}(\mathtt{b}, 0, P') = [\_]$.

   Then we compute $D_{\mathtt{a}}$ and $D_{\mathtt{b}}$ such that $(2, (P_{\mathtt{a}}, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \triangleright D_{\mathtt{a}}$ and $(2, (P_{\mathtt{b}}, \emptyset, \emptyset, \vec{r}), 0, \emptyset) \triangleright D_{\mathtt{b}}$. The switch case list $L \triangleq [(\mathtt{a}_0, D_{\mathtt{a}}), (\mathtt{b}_0, D_{\mathtt{b}})]$ can be defined and so the compilation step produces

   $$((2\,1), (P', \emptyset, \emptyset, \vec{r}), 0, \emptyset) \triangleright \mathtt{Switch}(L).$$

3. Since $P_{\mathtt{b}}$ contains only unconstrained variables, we are in the case item 2 and so we have $(1, (P_{\mathtt{b}}, \emptyset, \emptyset, \$x), 0, \emptyset) \triangleright \mathtt{Leaf}(\$x)$.
4. A specialisation on $P_{\mathtt{a}}$ with respect to `c` can be performed, let $Q_{\mathtt{a}} \triangleq \mathsf{Spec}(\mathtt{c}, 0, P_{\mathtt{a}}) = [\mathtt{c}\ \_]$ and define E such that $(1, (Q_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \triangleright E$. The compilation step produces

   $$(1, (P_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \triangleright \mathtt{Switch}([(\mathtt{c}, E)]).$$

5. Similarly, we can specialise $Q_{\mathtt{a}}$ on `c` yielding the matrix $[\_]$ which compiles to `Leaf`. We thus have,

   $$(1, (Q_{\mathtt{a}}, \emptyset, \emptyset, \$x), 0, \emptyset) \triangleright \mathtt{Switch}([(\mathtt{c}, \mathtt{Leaf}(\$x))]).$$

The soundness and completeness proofs for this compilation process can be found in [10].

We have seen that at each compilation step, several options are possible. The stack can be swapped with `Swap` to orient the filtering. If a constraint can be solved, either is is solved with a `BinNl` or `BinCl` node, or a `Switch` can be performed. These possibilities make the compilation process undeterministic. Therefore, a given matrix can be compiled to several decision trees. Maranget compares different heuristics based on the shape of patterns as well as some more complex ones. In Dedukti, since verifying constraints can involve non trivial operations (non-linearity and variable occurrence tests), constraint checking is postponed as much as possible. Regarding `Swap`, we process in priority columns that have many constructors and few constraints.

4 Results

This section compares the performance of the new rewriting engine with previous implementations of Dedukti, and other tools as well.

4.1 Hand-written examples

We consider 3 different implementations of Dedukti:

- Dedukti2.6 is the latest official release of Dedukti available on `opam`. Its matching algorithm also implements decision trees but non-linearity and variable-occurrence constraints are not integrated in decision trees. Its implementation, primarily due to Ronan Saillard [17], is available on `https://github.com/Deducteam/dedukti`.

Table 2 Time needed to solve Sudoku and SAT formulae in seconds.

	Sudoku			DPLL-SAT	
	easy	med	hard	`2_ex`	`ok_50x80`
Dedukti2.6	0.7	7.7	8 min 43	2	10
Lambdapi1.0	1.2	13	16 min 2	1.6	10
Dedukti3.0	0.5	5.2	5 min 15	0.2	2

- Lambdapi1.0 is an alternative implementation of Dedukti due to Rodolphe Lepigre [12]. It implements a naive algorithm for matching. It is available on `https://github.com/rlepigre/lambdapi/tree/fix_ho`.
- Dedukti3.0 is our new implementation of Dedukti. It adds to Lambdapi1.0 the decision trees described in this paper. It is available on `https://github.com/Deducteam/lambdapi`.

The git repository `https://github.com/deducteam/libraries` contains several hand-written Dedukti examples, including a Sudoku solver with 3 examples labelled easy, medium and hard respectively, and a DPLL-based SAT solver to decide the satisfiability of propositional logic formulae in conjunctive normal form with two example files:

- `2_ex` contains a function that when given a integer n, produces n literals named v_n and the formula $p(0) = v_0 \wedge \bigwedge_{k=1}^{n}(p(k) = p(k-1) \wedge (v_{k-1} \neq v_k))$
- `ok_50x80` contains a formula with 50 literals and 80 clauses of the form $\neg x \vee \neg y \vee \neg z$.

Because of the nature of the problems, they require a substantial amount of rewriting steps to be solved.

Table 2 shows the performance of each tool on these examples. Using decision trees increases significantly performance on Sudoku since Lambdapi1.0 is twice as slow as Dedukti2.6 which is slower than Dedukti3.0. SAT problems confirm that Dedukti3 is more efficient than Lambdapi1.0 and Dedukti2.

More benchmarks are described in [10].

4.2 Rewriting Engine Competition (REC)

The Rewriting Engine Competition[1] (REC), first organized in 2009, aims to compare rewriting engines. F. Duràn and H. Garavel revived the competition in 2018, and another study has been done in 2019 [5]. There are 14 rewriting engines tested, among which Haskell's GHC and OCaml.

REC problems are written in a specific REC syntax which is then translated into one of the 14 target languages with Awk scripts. To use REC benchmarks with Dedukti, a translation from Haskell benchmarks to Dedukti has been implemented[2].

Our rewriting engine[3] has been compared on the problems that do not use conditional rewriting with OCaml and Haskell. For each language, we have measured both the interpretation time with `ocaml` and `runghc`, and the compiling and running time with `ocamlopt` and `ghc`. The results are in Table 3.

[1] `http://rec.gforge.inria.fr`
[2] file `tools/rec_to_lp/rec_hs_to_lp.awk` available from revision e8388b73 (published on May 12, 2020)
[3] with revision a0009fdaa (published on Jan. 10, 2020)

We can divide our observations on classes of problems. There are 43 problems, among which 22 are solved in less that one second by at least two other solvers than Dedukti (the first group of the table). On these problems, our rewriting engine is in average 4 times faster than the median of the other rewriting engines. The second group contains problems on which no other tool than Dedukti needs more than ten seconds. On this group, Dedukti is in average 10 times slower than the median of the other tools. However, Dedukti performs better than interpreted OCaml on `add8` and better than compiled OCaml on`benchtree10`. On the last group, Dedukti is in average 60 times slower than other engines. Interestingly `ocamlopt` has more memory overflows than Dedukti (seven against four).

5 Conclusion & Related Work

This article describes the implementation of the new rewriting engine of Dedukti. It extends Maranget's techniques of decision trees used in the OCaml compiler [14] to the class of non-linear higher-order patterns used in Combinatory Reduction Systems (CRS) [11]. We define the language of decision trees and how to compile a set of rewrite rules into a decision tree. We finally present some benchmarks showing good performances.

A similar algorithm had been implemented in Dedukti2.6 by Ronan Saillard [17]. However, Saillard's rewriting engine used decision trees for first-order linear matching and handled non-linearity and variable-occurrence constraints afterwards in a naive way. In the new implementation, these constraints are fully integrated in decision trees.

Other rewriting engine uses decision trees as well like CRSX, which is a rewrite engine for an extension of Combinatory Reduction Systems [16], and Maude under certain conditions [8], but Maude considers first-order terms only.

Other pattern-matching algorithm are also possible, in particular using backtracking automata instead of trees, which allow to have smaller data structures. The interested reader can look at Prolog implementations or Egison (see [6] and, more particularly on the question of pattern matching, [7]).

Further useful extensions would be interesting: conditional rewrite rules (the REC database contains many files with conditional rewrite rules) and matching modulo associativity and commutativity (AC). Conditional rewriting could be implemented without too much difficulty since it would consist in extending the constraints mechanism which is modular. A prototype implementation of matching modulo AC has already been developed for Dedukti2.6 by Gaspard Férey[4] but performance is not very good yet. This new implementation could provide a better basis to implement matching modulo AC.

References

1. A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the λΠ-calculus modulo theory, 2019. Draft.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
3. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.
4. R. Cauderlier. Tactics and Certificates in Meta Dedukti. In *Proceedings of the 9th International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 10895, 2018.

[4] revision 5990bc6c (published on Feb. 21, 2021)

Table 3 Performance on REC benchmark in seconds. N/A is for out of memory. T/O is for timeout (30 minutes). The last line indicates that on the `langton7` problem, Dedukti ran out of memory, the command `runghc langton7.hs` took 533.2 seconds to finish, `ocaml langton7.ml` took 101.7 seconds, `ghc langton7.hs && ./langton7` took 66 seconds and `ocamlopt langton7.ml && ./a.out` took 39.6 seconds.

	Dedukti	runghc	ocaml	ghc	ocamlopt
revelt	0.026	0.517	0.065	0.271	0.168
check1	0.030	0.417	0.065	0.270	0.170
calls	0.017	0.416	0.065	0.271	0.168
check2	0.033	0.466	0.065	0.271	0.168
garbagecollection	0.033	0.416	0.115	0.271	0.170
fibonacci05	0.033	0.416	0.065	0.271	0.168
soundnessofparallelengines	0.033	0.467	0.065	0.271	0.168
factorial5	0.033	0.416	0.066	0.271	0.168
empty	0.033	0.416	0.065	0.271	0.170
revnat100	0.064	0.516	0.115	0.276	0.170
factorial6	0.065	0.467	0.066	0.271	0.169
tautologyhard	0.065	0.716	0.165	0.271	0.221
fibonacci18	0.115	0.466	0.065	0.275	0.170
fibonacci21	0.166	0.566	0.068	0.178	0.174
benchexpr10	0.165	0.667	0.266	0.275	0.221
benchsym10	0.165	0.667	0.266	0.275	0.221
natlist	0.165	0.868	0.266	0.275	0.220
fibonacci19	0.168	0.517	0.065	0.174	0.170
factorial7	0.215	0.467	0.065	0.275	0.170
fibonacci20	0.265	0.567	0.065	0.283	0.174
permutations6	0.366	0.868	0.115	0.299	0.182
factorial8	1.467	0.817	0.115	0.299	0.183
revnat1000	3.300	3.578	0.266	0.482	0.281
benchtree10	3.476	0.667	126.027	0.275	11.546
factorial9	N/A	2.974	N/A	0.532	0.282
permutations7	6.376	3.276	0.416	0.531	0.331
add8	10.899	N/A	12.605	0.232	N/A
benchsym20	14.335	6.581	1.067	0.734	0.381
benchexpr20	14.787	6.786	1.417	0.932	0.983
mul16	T/O	11.154	6.640	0.735	N/A
add32	T/O	19.714	8.640	0.331	N/A
benchtree20	N/A	21.333	T/O	3.301	T/O
mul32	T/O	27.562	10.444	1.638	N/A
benchsym22	54.491	23.534	2.727	1.435	0.833
benchexpr22	52.995	24.492	3.979	2.092	2.337
add16	79.472	22.697	7.140	0.303	N/A
mul8	167.500	4.479	3.771	0.331	N/A
omul32	T/O	49.863	25.251	1.885	N/A
benchtree22	N/A	101.176	T/O	10.047	T/O
revnat10000	400	244.459	6.987	19.241	4.092
omul8	797.406	5.430	3.971	0.381	N/A
langton6	N/A	377.208	75.709	38.437	24.463
langton7	N/A	533.197	101.695	66.093	39.640

5 Francisco Durán and Hubert Garavel. The rewrite engines competitions: A rectrospective. In *TACAS*, 2019.

6 Satoshi Egi. Egison: Non-linear pattern-matching against non-free data types. *ArXiv*, abs/1506.04498, 2015.

7 Satoshi Egi and Yuichi Nishiwaki. Non-linear pattern matching with backtracking for non-free data types. *ArXiv*, abs/1808.10603, 2018.

8 S. Eker. Fast matching in combinations of regular equational theories. *Electronic Notes in Theoretical Computer Science*, 4:90 – 109, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications. URL: `http://www.sciencedirect.com/science/article/pii/S1571066104000350`, `doi:https://doi.org/10.1016/S1571-0661(04)00035-0`.

9 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

10 G. Hondet. Efficient rewriting using decision trees. Master's thesis, ENAC and Université Paul Sabatier, 2019.

11 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

12 R. Lepigre and C. Raffalli. Abstract Representation of Binders in OCaml using the Bindlib Library. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Electronic Proceedings in Theoretical Computer Science 274, 2018.

13 T. Libal and D. Miller. Functions-as-constructors Higher-order Unification. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 52, 2016.

14 L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the ACM SIGPLAN Workshop on ML*, 2008.

15 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

16 Kristoffer H. Rose. CRSX - Combinatory Reduction Systems with Extensions. In Manfred Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2011/3130`, `doi:10.4230/LIPIcs.RTA.2011.81`.

17 R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.

18 F. Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Electronic Proceedings in Theoretical Computer Science 274, 2018.