Efficient rewriting using decision trees

Gabriel Hondet

UT3 Master of computer science
Data and knowledge representation
ENAC Engineer diploma
SITA, artificial intelligence

Master's thesis

prepared at

ENS Paris-Saclay, Laboratoire Spécification et Vérification, Deducteam

from 2019–02–11 to 2019–07–31

by

Gabriel Hondet

Efficient rewriting using decision trees

Defended on August 29, 2019

LSV supervision	Frédéric Blanqui	Inria, Saclay
	Bruno Barras	Inria, Saclay
Jury	Nicolas Barnier	ENAC, Toulouse
	Jérôme Mengin	UT3, Toulouse

Contents

List of Figures

List of Tables

Abstract

Proving programs becomes compulsory as software is increasingly complex. Our work stands in the context of proof interoperability, the global objective is to ease the development of formal proofs. Our work is focused on the development of Dedukti, a proof checker implementing the $\lambda\Pi$ calculus modulo rewriting. This internship is centred on the rewriting engine of the successor of Dedukti, namely Dedukti3. This memoir starts with a presentation of the theories involved, mainly λ calculus, rewriting and $\lambda\Pi$ calculus. The algorithm used in the new rewriting engine is then formalised and a correctness proof is provided. This algorithm is based on the pattern matching algorithm by Maranget and used in OCaml. It is extended to rewriting rules, λ terms and non linear patterns. Some interesting implementation details are evinced and then we compare the efficiency of the new engine to a naive matching algorithm and to the rewriting engine of Dedukti. The results show that our implementation handles large rewrite systems better that the naive algorithm, and is always better than Dedukti's. We finally present some miscellaneous work concerning Logipedia, an encyclopedia of formal proofs powered by Dedukti and concerning the translation from Isabelle –a proof assistant from the University of Cambridge and TU Münich– to Dedukti.

♣ ♣ ♣

La preuve de programmes devient nécessaire avec la complexification des logiciels. Le cadre général est l'interoperabilité de preuves dans le but de faciliter le développement des preuves formelles. Le travail est centré sur le développement de l'outil Dedukti, un vérificateur de preuve implémentant le $\lambda\Pi$ calcul modulo réécriture. Ce stage concerne le moteur de réécriture du successeur de Dedukti, Dedukti3. Le rapport commence par apporter les éléments théoriques impliqués, c'est-à-dire en particulier le λ le $\lambda\Pi$ calcul et la réécriture. L'algorithme implémenté est ensuite formalisé et une preuve de correction est présentée. L'algorithme est une extension du pattern matching utilisé en OCaml et développé par Maranget. En particulier, notre algorithme est étendu à la réécriture, aux λ termes et aux motifs non linéaires. Quelques détails d'implémentation jugés pertinents sont exposés. L'implémentation est ensuite comparée a un algorithme naïf et a l'implémentation de Dedukti. Les résultats montrent que notre implémentation est plus efficace que l'algorithme naïf, surtout en présence de systèmes de réécriture importants en taille ; et est toujours plus efficace que le moteur de Dedukti. Le mémoire finit par la présentation de travail auxiliaire, le développement de Logipedia, une encyclopédie des preuves alimentée par Dedukti, et la traduction de Isabelle (un assistant de preuve développé à l'université de Cambridge et TU Münich) en Dedukti.

Acknowledgements

I am grateful to all French people for their financial contribution to a bit less than twenty years of studies. More precisely, I would like to thank the ENAC for some of their courses which allowed me to dive into the beautiful world of theoretical computing ; and Université Paul Sabatier for accepting two of us for an interesting douple diploma.

I am also grateful to Frédéric Blanqui and the LSV, who accepted me as a master student, and even as a PhD student. Concerning this work, I must thank C. Picard and C. Allignol for offering their help with the then new world of formal proofs. I also thank G. Scherer who dedicated a Sunday afternoon to describe Inria teams, introduce me further to formal proofs and answer my questions.

0. Introduction

We describe in this chapter the environment in which the internship took place. We discuss the administrative structures, their role and the general project to which this work contributes. A rough planning of the internship is laid out to conclude.

0.1. General introduction

To ease the life of people, we tend to automate all we can. As a consequence, software is an increasingly crucial component in our lives. It has become crucial up to a point where our lives can depend on it ; see for instance a pace-maker or the auto pilot of an aircraft. In addition, software is generally getting more and more complex ; and thus more and mode error prone. In this context, proving properties on software and their behaviour seems mandatory in order to ensure the liability of what is known as the "digital transition".

I looked toward the field of formal proof mainly for two reasons. The first is that I wanted to work on the algebraic side of computer science, after having studied machine learning at ENAC and performed an internship in that area. It was as well the opportunity to work in fundamental research. Second, formal proof is deeply entangled with mathematics.

In this context, I have performed my internship in the Deducteam who works on formal proofs and verification. This team is an Inria project located at the ENS Paris-Saclay. The work is focused on the development of the flagship product of the team, the proof checker—and now proof assistant—Dedukti.

The first chapter serves two purposes. It handles the state of the art as well as introducing the theoretical content needed for the rest. The second chapter exposes the formal aspect of the algorithm used. We then continue with some implementation details and some metrics, in particular, we compare our algorithm with two other version of Dedukti. The last chapter is dedicated to other work performed in the team not directly related to the topic of the internship.

Reading guide People not familiar with λ calculus should definitely read the first chapter completely. People having some experience with it might still need to read section 1.2.

The second and third chapters expose the work performed, in a theoretical way as well as a concrete one. The non-programmer reader can skip section 3.2 to focus on the results.

The last chapter is not essential but details some other work performed.

0.2. Further information and context

0.2.1. Inria

Inria is a French national institute dedicated to research on computer science and applied mathematics. It has been founded in 1967 with its headquarters in Rocquencourt. There are 8 research centres in France : Saclay, Paris, Bordeaux, Lille, Grenoble, Nancy, Rennes and Sophia-Antipolis.

Inria is organised into "project-teams". A team can go from two permanent researchers to a dozen. Teams are categorised by fields which are

- proofs and verification ;
- applied mathematics, computation and simulation ;
- networks, systems and services, distributed computing ;
- perception, cognition and interaction ;
- digital health, biology and earth.

The Deducteam is in the category proofs and verification.

0.2.2. Deducteam

The Deducteam is located in the facilities of the ENS Paris-Saclay, in the *Laboratoire de Spécification et Vérification* (LSV). It is led by Gilles Dowek and is composed of four permanent researchers, one post doc and four PhD students. The team was founded two years ago.

The target of the team is to develop proof interoperability. Indeed many proof systems have emerged, such as Coq developed at Inria, Agda, PVS, Lean, Isabelle &c. and proofs developed in a system cannot be used in another one. Consequently, a non negligible amount of tedious work is duplicated, namely rewriting proofs to be able to use a formal proof into one's favourite proof system.

To palliate the duplication, one might want to write automatic translators from one system to the other. This approach works and has already been explored, with translators from HOL light to Coq. However, given n proof systems, this gives rise to n^2 translators, and writing a translator is a non trivial task.

Deducteam's proposal is to provide a formalism into which any proof system can be translated, and then translate from this general formalism to any other proof system. This reduces obviously the number of translators from n^2 to n. This formalism is the $\lambda\Pi$ calculus modulo rewriting described in section 1.4 implemented in Dedukti.

The subject of the internship is the improvement of the rewriting engine of Dedukti3, more precisely using a more elaborate algorithm based on trees instead of the naive one currently implemented.

0.2.3. Logipedia

From this idea of gathering all existing proofs into one formalism emerged the idea of an encyclopedia of proofs, where all existing formal proofs would be available to download in any system. The website currently exists at the address `https ://logipedia.science`, but is in a very early version.

For this purpose, Logipedia has to be capable of

- translating proofs from foreign system into its own encoding ;
- translating the encoded proofs into other foreign systems ;
- serving these proofs via a user-friendly interface, which is currently a web front-end.

Logipedia is the main use case for Dedukti, where it is used for typechecking the translated proofs.

0.2.4. Planning

The work has been carried out as follows. The first step has been to develop first order matching on algebraic patterns. It thus did not include section 2.2.1 nor section 2.2.2. Once the first order rewriting engine written out, an optimisation phase took place. The objective was to avoid complexity explosion during the compilation of trees and have an evaluation code as fast as possible. For instance, the complexity of an operation in the compilation has been reduced from n^2 to $n \log n$, resulting in the reduction from fifteen minutes to less than a minute to build trees on a specially designed example. Regarding the evaluation code, the objective was to be at least as fast as the naive algorithm, except for some really trivial defined cases.

Once this optimisation step done, we moved on to non linearity and higher order. Again, the implementation of these two features done, a second optimisation pass was carried out.

The two aforementioned tasks took approximately 4 months. I was then supposed to work on matching modulo associativity and commutativity. However, some other tasks have been assigned to me as well, as described in chapter 4.

Additionally, I attended the International School on Rewriting[1] during the first week of July. I followed the basic track and passed the final exam, with a grade of 64/100.

In fine, I could only read some material on matching modulo AC, and had afterward to focus on the formalism of the algorithm for the memoir.

1. `https://isr2019.inria.fr`

1. Theoretical background

We will here give an overview of the theoretical content involved in this work. We first expose the basics of λ calculus, which is essential for the rest of the document, as well as the next section on rewriting. The two next sections are less important as they only give more context to the work, describing roughly the theories that use the tools we work on. The chapter is concluded by an introduction of the main subject of the memoir and a short paragraph replacing the contribution of this work in a more general context.

1.1. λ calculus

The λ calculus, defined by Alonzo Church in the 30s, is a computation model where everything is a function. It is the basis of functional languages such as Lisp, OCaml, Haskell &c.

1.1.1. Syntax of terms

Let $\mathcal{V}$ be a countable infinite set of variables $x, y, ...$ We begin by defining the set of terms of the λ calculus, whose syntax is described in Figure 1.1.

Definition 1.1 (Terms[Pie02])**.** The set of terms $\mathcal{T}$ is the smallest set such that

1. $x \in \mathcal{T}$ if $x \in \mathcal{V}$
2. if $t \in \mathcal{T}$ and $x \in \mathcal{V}$, then $\lambda x.\, t \in \mathcal{T}$,
3. if $t \in \mathcal{T}$ and $u \in \mathcal{T}$, then $t\ u \in \mathcal{T}$.

An abstraction $\lambda x.\, t$ can be seen as a mathematical function taking as argument x and whose result is t, which may or may not depend on x. An application $t_1\ t_2$ is equivalent to, in mathematical notation, $t(u)$ (where t is seen as a function). A variable is *bound* by a binder, for example, in $\lambda s.\, \lambda z.\, sz$, variables s and z are bound. A variable that is not bound is said *free*. A formal definition of a free variable is given in Definition 1.2.

$t ::=$	**Terms**
x	variable, $x \in \mathcal{V}$
$\lambda x.\, t$	abstraction
$t_1\ t_2$	application

Figure 1.1. : Syntax of λ-terms

Definition 1.2 (Free variable[Pie02])**.** The set of free variables of a term t written $\mathrm{FV}(t)$ is defined as

$$\begin{aligned}\mathrm{FV}(x) &\triangleq \{x\}\\ \mathrm{FV}(\lambda x.\, t) &\triangleq \mathrm{FV}(t)\backslash\{x\}\\ \mathrm{FV}(t\ t') &\triangleq \mathrm{FV}(t) \cup \mathrm{FV}(t')\end{aligned}$$

1.1.2. Substitution

A substitution is the replacement of a free variable in a term by another term. Substitutions are usually noted in postfix notation, that is, if σ is a substitution, its application to term M is written $M\sigma$.

Definition 1.3 (Substitution[Pie02])**.**

$$\begin{aligned}x\{x := s\} &\triangleq s\\ y\{x := s\} &\triangleq x && \text{if } y \neq x\\ (\lambda y.\, t)\{x := s\} &\triangleq \lambda y.\, t\{x := s\} && \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(s)\\ (t\ t')\{x := s\} &\triangleq t\{x := s\}\ t'\{x := s\}\end{aligned}$$

1.1.3. β reduction

The β reduction is the computation step of the λ calculus. Mathematically, it can be seen as the application of a function to a variable that yield a new value, as $x \mapsto 2 \times x$ applied to 2 yields 4 can be seen as $(\lambda x.\, 2 \times x)\ 2$ which β reduces to 2×2.

Definition 1.4 (β reduction)**.** The arrow $\to$ stands for "evaluates to",

$$(\lambda x.\, t_1)\ t_2 \to t_1\{x := t_2\}$$

A term that can be reduced by a β step is called a *β-redex*.

For a more complete introduction to λ calculus, see [Pie02].

1.2. Rewriting

1.2.1. Abstract reduction system

Definition 1.5 (Abstract reduction system[BN99])**.** An *abstract reduction system* is a pair $(A, \to)$ where A is a set and $\to \subset A \times A$ is a binary relation on A. For $(a, b) \in A \times A$, we write $a \to b$ rather than $(a, b) \in \to$ and we say that $a \to b$ is a *rewrite step*.

Definition 1.6 ([BN99])**.** We will use the following definitions throughout the report,

$\overset{*}{\to} \triangleq \bigcup_{i \geq 0} \to$	reflexive and transitive closure
$\leftarrow \triangleq \{(y, x) \mid x \to y\}$	symmetric
$\leftrightarrow \triangleq \to \cup \leftarrow$	symmetric closure
$\overset{*}{\leftrightarrow} \triangleq \bigcup_{i \geq 0} \leftrightarrow$	transitive reflexive symmetric closure

Definition 1.7 (Convertibility)**.** Let $(A, \to)$ be an ARS and $a, b \in A \times A$. We say that a and b are *convertible* whenever $a \overset{*}{\leftrightarrow} b$.

Definition 1.8 ([BN99])**.** A reduction $\to$ is said

confluent whenever if $u \overset{*}{\leftarrow} t \overset{*}{\to} v$ then there is a w such that $u \overset{*}{\to} w \overset{*}{\leftarrow} v$

terminating iff there is no infinite rewrite sequence $a_0 \to a_1 \cdots$

normalising iff each element has a normal form

Notation 1.9. If a term x has a unique normal form, we write it $\downarrow x$.

Theorem 1.10 ([BN99])**.** *If $\to$ is confluent and normalising, then $x \overset{*}{\leftrightarrow} y \iff \downarrow x = \downarrow y$.*

Proposition 1.11. *The problem*

Instance : *ARS $(A, \to)$, $(a, b) \in A \times A$*
Question : *are a and b convertible ?*

Is decidable whenever $(A, \to)$ is confluent and normalising.

Proof. Thanks to Theorem 1.10, it is enough to compute normal forms of a and b to answer the question. □

1.2.2. Term rewrite system

Definition 1.12 (Signature[BN99])**.** A *signature* Σ is a set of function symbols where each $\mathtt{f} \in \Sigma$ is associated with a non-negative integer n, the arity of $\mathtt{f}$. The elements of arity zero are also called *constants*.

Definition 1.13 ([BN99])**.** Given a signature Σ and a set of variables V such that $V \cap \Sigma = \emptyset$. The set $\mathcal{T}(\Sigma, V)$ is the set of all Σ-terms over V defined inductively as

- $X \subset \mathcal{T}(\Sigma, V)$ (i.e. every variable is a term) ;
- for all $n \geq 0$, all $\mathtt{f} \in \Sigma$ of arity n, and all $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, V)$, we have $\mathtt{f}(t_1, t_2, \ldots, t_n) \in \mathcal{T}(\Sigma, V)$.

Definition 1.14 (Height)**.** The *height* h of a term is inductively defined as

$$h(\lambda x.\, t) = 1 + h(t) \qquad h(\mathtt{f}\, t_1 \ \ldots \ t_n) = 1 + \max(h(t_1), \ldots, h(t_n))$$
$$h(x) = 0 \text{ otherwise}$$

Definition 1.15 (Position[BN99])**.** Let Σ be a signature and V a set of variables.

1. The set of positions of the term t is the set of strings defined over the alphabet of positive integers inductively defined as follows :
   - If $t \in V$, then $\mathcal{P}\text{os} \triangleq \{\epsilon\}$.

- If $t = \mathtt{f}\ t_1\ \cdots\ t_n$, then

$$\mathcal{P}\mathrm{os}(t) \triangleq \{\epsilon\} \bigcup_{i=1}^{n} \{ip | p \in \mathcal{P}\mathrm{os}(t_i)\}.$$

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

2. The *size* $|t|$ of a term t is the cardinality of $\mathcal{P}\mathrm{os}(t)$.

3. For $p \in \mathcal{P}\mathrm{os}(t)$, the *subterm of t at position p*, denoted by $t|_p$, is defined by induction on the length of p :

$$t|_\epsilon \triangleq t$$
$$\mathtt{f}\ t_1\ \cdots\ t_n|_{iq} \triangleq t_i|_q$$

4. The type of positions will be noted $\mathcal{P}\mathrm{os}$.

Definition 1.16 (Rewrite rule)**.** A *rewrite rule* is an equation $\ell \approx r$ that satisfies the following two conditions

- the left-hand side ℓ is not a variable,
- the variables which occur in the right-hand side r occur also in ℓ.

Rewrite rule $\ell \approx r$ will be written $\ell \to r$. A *term rewrite system* or TRS for short is a set of rewrite rules. A *redex* (*red*ucible *ex*pression) is an instance of the left-hand side of a rewrite rule.

Notation 1.17. We will generally denote $\to$ the rewrite relation induced by a TRS R instead of $\to_R$.

Remark 1.18 (Higher order rewriting)**.** This section describe rewriting on terms in $\mathcal{T}(\Sigma, V)$ for some signature Σ and a set of variables V, which is called *first order rewriting*.

Rewriting with terms of the λ calculus is called *higher order rewriting*. The rewriting is performed as for first order, but modulo β and η reductions[1].

In the context of higher order rewriting, the set of terms $\mathcal{T}(\Sigma, V)$ contains abstractions $\lambda x.\, t \in \mathcal{T}(\Sigma, V)$ with $x \in V$ and $t \in \mathcal{T}(\Sigma, V)$.

Remark 1.19. The β reduction of beta reduction can be seen as a rewrite rule $\to_\beta$

$$(\lambda x.\, t)\ u \to t\{x := u\}$$

One can then define, given a TRS R a relation $\to_{R,\beta} = \to_R \cup \to_\beta$. We will write $\to$ the rewrite relation encompassing β reduction when working on λ terms. The convertibility regarding $\to_{R,\beta}$ will be written $\equiv$.

1. η equivalence states that $\lambda x.\, (t\ x)$ is equivalent to t.

1.3. Type theory and proof assistant

Type theory[Rus08] is an alternative to set theory proposed by Bertrand Russel in the early 20$^{\text{th}}$ century. The main idea is to classify objects within *types* to ensure each object has a meaning. Church then reformulated type theory using λ calculus, creating the *simple type theory*[Chu40].

In the 60s, the "propositions as types" principle appeared[How69]. Also called *Curry-Howard correspondence*, it states that mathematical propositions can be seen as types and that proofs of propositions can be seen as terms of those types. A mathematical proof can be seen as a program, and thus, *proof* checking becomes *type* checking.

Many proof checkers stemmed from this principle, each one using one logical system, such as Coq [dt18], Matita [ARCT11], Agda [BDN09]&c. There are also *logical frameworks* that allow to embed several logical systems into encodings so that proposition validity becomes type inhabitation[HHP87]. Some current logical frameworks are Isabelle [PN94] or Twelf [Sch09].

Given that the purpose of Dedukti is proof interoperability, the logical framework approach seems suitable, and indeed, Dedukti is a logical framework.

1.4. Dependent types and $\lambda\Pi$ calculus modulo

Dedukti is based on the $\lambda\Pi$ calculus modulo rewriting theory[CD07].

Remark 1.20. We introduce here this calculus for the interested reader, but since rewriting operates on objects (see Figure 1.2) of the language, a basic understanding of λ calculus is enough.

The $\lambda\Pi$ calculus is an extension of the λ calculus with the dependent product $\Pi x : A.\, B$ where B is a type that depends on value x which is of type A. Through the Curry-Howard isomorphism, it can be seen as first order logic (with $\Pi x : A.\, P$ written $\forall x : A.\, P$). The $\lambda\Pi$ calculus can encode several logics, but it can't deal efficiently with computations.

To solve this issue, we extend $\lambda\Pi$ calculus with rewrite rules. The syntax is then the syntax of the λ calculus, with dependent product and with rewrite rules of the form $M \to N$. The syntax of $\lambda\Pi$ calculus modulo is given in Figure 1.2.

1.5. Matching

We will here give the basics on term matching. We begin by defining a *substitution* on terms (not only in the λ calculus).

Definition 1.21 (Substitution[Mid19])**.** Let Σ be a signature and V a countably infinite set of variables. A $\mathcal{T}(\Sigma, V)$-*substitution*–or simply *substitution*– is a function $\sigma : V \to \mathcal{T}(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many $x \in V$. The (finite) set of variables that σ does not map to themselves are called the *domain* of σ, $\text{dom}(\sigma) \triangleq \{x \in V | \sigma(x) \neq x\}$. If $\text{dom}(\sigma) = \{x_1, \dots, x_n$, then we may write

$$\sigma = \{x_1 := \sigma(x_1), \dots, x_n := \sigma(x_n)\}$$

We write $t\sigma$ the result of applying substitution σ to term $t \in \mathcal{T}(\Sigma, V)$,

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \in V \\ \mathtt{f}\ t_1\sigma\ \cdots t_n\sigma & \text{if } t = \mathtt{f}\ t_1\ \cdots\ t_n \end{cases}$$

and we say that $t\sigma$ is an *instance* of t.

$K ::=$		**Kinds**
	$\texttt{Type}$	Top kind
	$\Pi x : A.\, K$	Dependent product
$A, B ::=$		**Types**
	$\Pi x : A.\, B$	Dependent product
	$A\ M$	Type application
$M, N ::=$		**Objects**
	x	Variable, $x \in \mathcal{V}$
	$\lambda x : A.\, M$	Abstraction
	$M\ N$	Application
$\Gamma ::=$		**Contexts**
	$\emptyset$	Empty context
	$\Gamma, x : A$	
$R ::=$		**Rewrite rules**
	$\Gamma, M \to N$	Rewrite rule

Figure 1.2.: Syntax of $\lambda\Pi$ calculus modulo rewriting, where K are kinds, A, B are types, M, N are objects, Γ are contexts and R are rewrite rules

$$\frac{\{\mathtt{f}\ s_1 \dots s_n \coloneqq \mathtt{f}\ t_1\ \dots\ t_n\} \uplus S}{\{s_1 \coloneqq t_1, \dots, s_n \coloneqq t_n\} \cup S} \qquad \frac{\{\mathtt{f}\ s_1\ \cdots\ s_n \coloneqq \mathtt{g}\ t_1\ \cdots\ t_m\} \uplus S}{\bot} \text{if } f \neq g$$

$$\frac{\{\mathtt{f}\ s_1\ \cdots\ s_n \coloneqq x\} \uplus S}{\bot} \qquad \frac{\{x \coloneqq t\} \uplus S}{\bot} \text{if } x \coloneqq t' \in S \text{ with } t \neq t'$$

Figure 1.3. : Matching procedure

Definition 1.22 (Matching[Mid19])**.** We say that a term s matches a term t if t is an instance of s, that is, if there is a substitution σ such that $t = s\sigma$.

Theorem 1.23. *The* matching problem *is decidable*

Instance *: terms* s, t

Question *: is there a substitution* σ *such that* $s\sigma = t$ *?*

Proof. A simple procedure is described and analysed in [Mid19]. The procedure is the set of rules given in Figure 1.3 where $\uplus$ is the disjoint union starting on set $S = \{s \coloneqq t\}$. □

Pattern matching has been defined by D. Miller in [Mil90] and has been used extensively as it has been shown decidable in 90.

Notation 1.24. We denote lists or vectors by bold italic letters $\boldsymbol{x}$.

- A curly braced vector $\{\boldsymbol{x}\}$ is considered as a set.
- We note $\boldsymbol{x}@\boldsymbol{y}$ the concatenation of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.
- The concatenation operator is overloaded on values, that is, we will write $x@\boldsymbol{x}$ for the concatenation of the vector with one element x with the vector $\boldsymbol{x}$.
- We extend the subterm definition on vectors,

  $$(x_1\ \cdots x_n)|_{ip} \triangleq x_i|_p$$

  given that $p \in \mathcal{P}\text{os}(x_i)$.

Definition 1.25 (Pattern[Mil90])**.** A term t is a *(higher order) pattern* if every free occurrence of a variable F is in a subterm $(F\,\boldsymbol{u})$ of t such that $\boldsymbol{u}$ is a (possibly empty) list of distinct bound variables.

Definition 1.26 (Unification[BN99])**.** A *unification problem* is a finite set of equations $S = \{s_1 =^? t_1, \dots, s_n =^? t_n\}$. A *unifier* is a substitution σ such that $s_i\sigma = t_i\sigma$ for all i.

Theorem 1.27 (Higher order unification with patterns)**.** *The following problem is decidable*

Instance *: Higher order patterns p, q,*
Question *: are there two substitutions σ, τ such that $p\sigma =_{\beta,\eta} q\tau$?*

Proof. See [Mil90] for an algorithm. □

Remark 1.28. The higher order pattern matching problem can be seen as a particular case of the higher order unification problem, and is thus decidable as well.

1.6. Introduction to rule filtering

Computing can be seen as normalising terms. For instance, considering arithmetic expressions, a computation of the form $2\times2+2=6$ can be seen as the normalisation of the term $2\times2+2$ to a form that can't be reduced further : 6. In that case, the normalisation can be decomposed into $2\times2+2=4+2=6$: there are two elementary reduction steps. In our context, an elementary reduction step is either a β reduction, or the application of a rewrite rule.

In Dedukti, a symbol may have several rewriting rules attached, as can be seen in Figure 1.4 showing some Dedukti code. Therefore, the efficiency of rewriting depends on the ability to select the correct rule to apply given a stack of terms. For instance, considering our arithmetic expression, we knew that we had to apply first the product as it has a higher priority. Algorithmically, we had to choose between the $+$ and the $\times$. More formally, when we want to apply a rewrite rule, the term to be rewritten is analysed and compared to the left-hand side of the rewrite rules that have the same head symbol. If there is a left-hand side such that the term is an instance of it, we say that the term *matches* the left-hand side, or equivalently, the left-hand side *filters* the term.

instance def.1.16 p.20

```
symbol d : (F ⇒ F) ⇒ (F ⇒ F)
rule d (F + G) → (d F) + (d G)
 and d (F · G) → ((d F) · G) + (F · (d G))
 and d (F ∘ G) → ((d F) ∘ G) · (d G)
```

Figure 1.4. : Rewrite rules for derivation of functions

$p ::=$	**Patterns**
$(p_1 \vert p_2)$	Disjunctive pattern
$\mathtt{c}(p_1, \dots, p_n)$	Constructor pattern
$\_$	Wildcard

Figure 1.5. : Pattern language of OCaml

Example 1.29 (Selection efficiency)**.** Consider the following string rewrite system on musical scales with $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \sharp, \flat, \mathtt{m}\}$ with $\sharp$ and $\flat$ in postfix notation,

$$\mathtt{m\ C\ D\ E\ F\ G\ A\ B} \to \mathtt{C\ D\ E}\flat\ \mathtt{F\ G\ A\ B}$$

The function `m` transforms a major scale into a minor one.

Consider using the first rule on, say, the G major scale $\mathtt{GABCDEF}\sharp$. A naïve algorithm performs only one comparison to see that the rule does not apply since $\mathtt{G} \neq \mathtt{C}$.

Consider using the same rule on the C minor scale $\mathtt{CDE}\flat\mathtt{FGAB}$. This time, a naïve algorithm comparing symbols sequentially from left to right will perform three comparisons : $\mathtt{C} = \mathtt{C}, \mathtt{D} = \mathtt{D}, \mathtt{E} \neq \mathtt{E}\flat$.

An algorithm comparing directly the third note of the scale would have performed only one comparison $\mathtt{E} \neq \mathtt{E}\flat$.

1.7. Contribution

1.7.1. Extension of [Mar08]

The main contribution of this master's thesis is the extension of an algorithm for first order pattern matching using decision trees to higher order matching. The original algorithm developed in [Mar08] is used for pattern matching in the OCaml [LDF+13] language. It handles the pattern language defined in 1.5. Our language handles λ terms and allow to enforce convertibility constraints between values. Furthermore, the constraints are used in a general fashion, possibly allowing to implement conditional rewriting easily.

Additionally, we provide a mechanism to build the substitution, this mechanism not being mentioned in [Mar08].

Other pattern matching algorithm are 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 [Egi15] and more particularly, on the question of pattern matching, see [EN18].

1.7.2. Related work

The Ph.D of Ronan Saillard [Sai15] already establishes the use of Maranget's decision trees for rewriting ; along with a proof of soundness and completeness. However, decision trees are not used for higher order terms nor non linearity conditions. Instead, higher order and non linearity constraints are solved naïvely as the last step before yielding the result.

Similarly the higher order rewriting engine CRSX [Ros11] uses decision trees ("similarly to the way normal functional language") for pattern matching.

The Maude rewriting engine as described in [Eke96], performs first order rewriting modulo associativity and commutativity using backtracking automata in the general case ; and trees if the right hand sides contain only one variable.

1.7.3. Implementations

In the wake of Ronan Saillard's work, Dedukti2 already implements decision trees à la Maranget, however, the convertibility and closedness checks are performed naïvely.

Dedukti3, the new implementation of Dedukti, published in September 2018, uses a naïve rule filtering strategy analysing symbols from left to right using the rules 1.3 page 22.

Our algorithm replaces the naïve rule filterting algorithm of the current implementation of Dedukti3.

2. A formal rule filtering algorithm

This section is devoted to the formalisation of the algorithm developed. It describes the different steps, the properties of the algorithm in order to be able to prove that it is correct regarding our specifications. The algorithm compiles the language composed of the objects of the λΠ calculus modulo and patterns to decision trees. We finish by the proof of correctness of our algorithm.

2.1. Introduction

This section extends the formalism introduced by Maranget in [Mar08]. As [Mar08] uses pattern matching for OCaml, neither higher order pattern matching nor non left-linear rules are considered. In rewriting, higher order is crucial, and non linearity is used from time to time, for instance to encode the intern operation of a group, as in 2.1, where it enforces the equality of two terms. It can be used to encode the equality of bound variables as well.

$$\begin{array}{cc} X^{-1} \cdot X \to \mathtt{e} & X \cdot X^{-1} \to \mathtt{e} \\ X^{-1} \cdot (X \cdot Y) \to Y & X \cdot (X^{-1} \cdot Y) \to Y \end{array} \tag{2.1}$$

In this set of equations, we specify that to apply the first rule $X^{-1} \cdot X \to \mathtt{e}$, we need the two X to be convertible. convertible

Higher order rewriting gives birth to *closedness* constraints. Consider for instance the partial derivation rule $\frac{\partial F(y)}{\partial x} = 0$. This rule assumes that $y \neq x$ to be applied. This assumption will be verified by testing whether the variable x occurs in the term $F(y)$. In our language, it will be written $\partial_1\ \lambda xy.\, F[y] \to 0$; which means that the term matched by the variable F might contain at most y among its free variables. free variable

def.1.7 p.19

These two constraints boil down to a binary choice : whether the two terms are convertible or whether the term contains a given free variable. def.1.2 p.17

2.2. Source language

The values of the language are the terms that will be filtered by patterns. They are therefore the objects of the λΠ calculus modulo in 1.2.

Given a signature Σ and a set of variables V, the set of values is written $\mathcal{W}(\Sigma, V)$ and the syntax is defined in 2.1.

Remark 2.1. A symbol of Σ can have several arities. For instance, one might define, with `id` the identity function,

$$\begin{array}{c} \mathtt{plus}\ (\mathtt{s}\ N)\ M \to \mathtt{s}\ (\mathtt{plus}\ N\ M) \\ \mathtt{plus}\ \mathtt{zero} \to \mathtt{id} \end{array}$$

in which `plus` has arity one in the first rule and arity two in the second.

Patterns are used to filter values. A pattern represents a set of values, possibly reduced to a singleton. To match several terms, patterns need variables, noted as capital letters X coming

$v \in \mathcal{W}(\Sigma, V) ::=$		**Values**
	$\mathtt{f}\ v_1\ ...\ v_a$	function symbol, $a \geq 0$, $\mathtt{f} \in \Sigma$
	$\lambda x.\, v$	abstraction
	$x \in V$	bound variable

Figure 2.1. : Syntax of values $\mathcal{W}$

$p \in \mathcal{P}(\Sigma, V) ::=$		**Patterns**
	$\mathtt{f}\ p_1\ ...\ p_a$	function pattern, $a \geq 0$, $\mathtt{f} \in \Sigma$
	$\lambda x.\, p$	abstraction
	$X\left[x_1, ..., x_{a(X)}\right]$	pattern variable with $a(X) \geq 0$ distinct bound variables, $x_i \in V$

Figure 2.2. : Syntax of patterns $\mathcal{P}$

from a countably infinite set V_p disjoint from V. A pattern variable X has a fixed arity given by a function $a : V_p \to \mathbb{N}$. The set of patterns will be denoted $\mathcal{P}$ and the syntax of patterns is described in Figure 2.2. One can verify that the language of patterns follows the definition of Miller's patterns 1.25

Notation 2.2.

- A pattern variable with no bound variable and not subject to non linearity constraints may be written as the usual wildcard _.
- A pattern variable with no bound variable but a name is written X. There is a linearity constraint whenever two pattern variables have the same identifier in the left hand side of a rule.
- We will only write $\mathcal{W}$ and $\mathcal{P}$ instead of $\mathcal{W}(\Sigma, V)$, $\mathcal{P}(\Sigma, V)$ whenever specifying the signature and the variable set isn't relevant.
- In the following, we will consider Σ to be a signature and $\mathcal{V}$ a countably infinite set of variables.

Definition 2.3 (Rewrite rule in Dedukti)**.** Let Σ be a signature and V a set of variables. A rewrite rule in Dedukti $\ell \to r$ is a rewrite rule of a TRS with ℓ of the form $\mathtt{f}\ \boldsymbol{p}$ where $f \in \Sigma$ and $\boldsymbol{p} \in \mathcal{P}(\Sigma, V)^*$.

Vectors are grouped into matrices P. A matching problem is represented by a clause matrix $P \to A$.

$$P \to A = \begin{pmatrix} p_1^1 & \cdots & p_n^1 & \to & a^1 \\ p_1^2 & \cdots & p_n^2 & \to & a^2 \\ & & & \vdots & \\ p_1^m & \cdots & p_n^m & \to & a^m \end{pmatrix}$$

2.2.1. Semantics of Dedukti matching

Definition 2.4. Two pattern variables $X[v_1, \dots, v_a]$ and $Y[w_1, \dots, w_b]$ are similar, noted $X[v_1, \dots, v_a] \sim Y[w_1, \dots, w_b]$ iff X is syntactically equal to Y.

The *instance* relation denoted $\prec_X$ is defined inductively between a pattern and a value v, indexed by a set of variables X as

(MatchPatt)	$Z \prec_X v$	
(MatchSymb)	$\mathtt{f}\, p_1 \dots p_a \prec_X \mathtt{f}\, v_1 \dots v_a$	iff $(p_1 \cdots p_a) \prec_X (v_1 \cdots v_a)$
(MatchAbst)	$\lambda x.\, p \prec_X \lambda x.\, v$	iff $p \prec_{X \cup \{x\}} v$
(MatchFv)	$Y[\boldsymbol{x}] \prec_X v$	iff $\mathrm{FV}(v) \cap X \subseteq \{\boldsymbol{x}\}$
(MatchTuple)	$(p_1 \ \dots \ p_a) \prec_X (v_1 \ \dots \ v_a)$	iff $\forall i, (\text{if } \exists j \neq i, p_i \sim p_j \text{ then } v_i \equiv v_j) \wedge (p_i \prec_X v_i)$

The condition in the MatchTuple rule translates non linearity checking : if a variable occurs twice in the pattern, then the matching values must be convertible.

The indexing set of variables is used to record which binder have been traversed to ignore variables that are free in the first (and biggest) term being filtered.

Definition 2.5 (DEDUKTI matching)**.** Let P be a pattern matrix of width n and height m. Let $\boldsymbol{v}$ be a value vector of size n. Let j be a row index.

Row j of P filters $\boldsymbol{v}$ when vector $\boldsymbol{v}$ is an instance of $\boldsymbol{p}^j$. Let $P \to A$ be a clause matrix. If row j of P filters $\boldsymbol{v}$, we write

$$\mathrm{Match}[\boldsymbol{v}, P \to A] \triangleq a^j$$

Separating constraints

Non linearity and closedness constraints are encoded using two vectors.

$$P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\} = \begin{pmatrix} p_1^1 & \cdots & p_n^1 & \to & a^1 & \{N^1, F^1\} \\ p_1^2 & \cdots & p_n^2 & \to & a^2 & \{N^2, F^2\} \\ & & & \vdots & & \\ p_1^m & \cdots & p_n^m & \to & a^m & \{N^m, F^m\} \end{pmatrix}$$

With F^i encoding closedness constraints and N^i encoding non linearity constraints.

A closedness constraint consists in allowing bound variables to appear at some position of the root term, therefore, an element of an F^i will be of type $\mathcal{F}v \triangleq \mathcal{P}\mathrm{os} \times \mathcal{V}^*$, and thus, $F^i : 2^{\mathcal{F}v}$. $\mathcal{P}\mathrm{os}$

A convertibility constraint requires two subterms of the root term to be convertible. An element of N^i is thus a pair of positions. As the order is not important, these pairs are sets of two elements. The type of a convertibility constraint is $\mathcal{N}l \triangleq 2^{\mathcal{P}\mathrm{os}}$. def.1.15 p.19

We now define precisely F (resp. N) using a function $\mathsf{extract}_{\mathsf{fv}}$ (resp. $\mathsf{extract}_{\equiv}$) which extracts the closedness (resp. non linearity) constraints from a vector of patterns.

To avoid redundancy and lighten the process, we introduce *nameless patterns* which are patterns without constraints.

Definition 2.6. Let $\boldsymbol{p}$ be a vector of patterns. We define $\mathsf{extract}_{\equiv} : \mathcal{P}^* \to 2^{\mathcal{N}l}$ relatively to $\boldsymbol{p}$ as

$$\mathsf{extract}_{\equiv}(\boldsymbol{p}) \triangleq \left\{ \{o_1, o_2\} \middle| o_1 \neq o_2 \wedge \boldsymbol{p}|_{o_1} \sim \boldsymbol{p}|_{o_2} \right\} \tag{2.3}$$

Definition 2.7. Let $\boldsymbol{p}$ be a vector of patterns. We define $\mathsf{extract}_{\mathsf{fv}} : \mathcal{P}^* \to 2^{\mathcal{F}v}$ inductively as

$$
\begin{aligned}
\mathsf{extract}_{\mathsf{fv}}(\boldsymbol{p}) &\triangleq \bigcup_i \mathsf{extract'}_{\mathsf{fv}}(p_i, \emptyset, i) \\
\mathsf{extract'}_{\mathsf{fv}}(\lambda x.\, t, X, o) &\triangleq \mathsf{extract'}_{\mathsf{fv}}(t, \{x\} \cup X, o1) \\
\mathsf{extract'}_{\mathsf{fv}}(\mathtt{f}\ t_1\ \cdots\ t_n, X, o) &\triangleq \bigcup_{i=1}^{n} \mathsf{extract'}_{\mathsf{fv}}(t_i, X, oi) \\
\mathsf{extract'}_{\mathsf{fv}}(F[\boldsymbol{x}], X, o) &\triangleq \begin{cases} \{(o, X)\} & \text{if } \{\boldsymbol{x}\} \subsetneq X \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{extract'}_{\mathsf{fv}}(\mathtt{f}, X, o) &\triangleq \emptyset
\end{aligned}
\tag{2.4}
$$

Remark 2.8. In Equation 2.4, we have $\subsetneq X$ (instead of a simple $\subseteq X$) because no test is required (because trivially true) if $\{\boldsymbol{x}\} = X$.

Definition 2.9 (Nameless pattern)**.** Let $p \in \mathcal{P}$. The *nameless pattern* extracted from p is defined as

$$
\begin{aligned}
\sharp(\lambda x.\, p) &\triangleq \lambda x.\, \sharp(p) \\
\sharp(\mathtt{f}\ q_1\ \cdots\ q_n) &\triangleq \mathtt{f}\ \sharp(q_1)\ \cdots\ \sharp(q_n) \\
\sharp(Z[\boldsymbol{x}]) &\triangleq \_
\end{aligned}
$$

We extend the definition to vectors,

$$
\sharp(p_1\ \cdots\ p_a) = (\sharp(p_1)\ \cdots\ \sharp(p_a))
$$

and to matrices,

$$
\sharp \begin{bmatrix} \boldsymbol{p}^1 \\ \vdots \\ \boldsymbol{p}^m \end{bmatrix} = \begin{bmatrix} \sharp(\boldsymbol{p}^1) \\ \vdots \\ \sharp(\boldsymbol{p}^m) \end{bmatrix}.
$$

We now define a new filtering relation that takes advantage of this representation. Intuitively, a vector of patterns filters values when the symbols match individually (same notion as previous matching) and when the constraints encoded by the pattern are satisfied by the values. Since constraints are recorded into sets, the elements of these sets will be considered as "to be satisfied" constraints, that is, once a constraint is satisfied, it is removed from the set. The satisfaction of all constraints is thus equivalent to the emptyness of the constraint set. The following function encodes this filtering relation.

Notation 2.10.

$$
{}^2\bigcup_{i=a}^{b}(e_i, e'_i) \triangleq \left(\bigcup_{i=1}^{a} e_i, \bigcup_{i=1}^{a} e'_i\right)
$$

Definition 2.11. Let X be a set of non linear constraints, Y a set of closedness constraints and $o \in \mathcal{P}\mathrm{os}$. We define the operator $\mathsf{filter} : 2^{\mathcal{N}l} \to 2^{\mathcal{F}v} \to \mathcal{P}^* \to \mathcal{W}^* \to 2^{\mathcal{N}l} \times 2^{\mathcal{F}v}$ as

$$
\mathsf{filter}(N, F, \boldsymbol{p}, \boldsymbol{v}) \triangleq \mathsf{filter'}(N, F, \boldsymbol{p}, \boldsymbol{v}, \epsilon, \emptyset)
$$

where $\mathsf{filter'} : 2^{\mathcal{N}l} \to 2^{\mathcal{F}v} \to \mathcal{P}^* \to \mathcal{W}^* \to \mathcal{P}\mathrm{os} \to \mathcal{V} \to 2^{\mathcal{N}l} \times 2^{\mathcal{F}v}$ is defined as

$$
\begin{aligned}
\mathsf{filter'}(N, F, \lambda x.\, p, \lambda x.\, v, o, X) &\triangleq \mathsf{filter'}(N, F, p, v, o1, X \cup \{x\}) \\
\mathsf{filter'}(N, F, \mathtt{f}\ \boldsymbol{p}, \mathtt{f}\ \boldsymbol{v}, o, X) &\triangleq \mathsf{filter'}(N, F, \boldsymbol{p}, \boldsymbol{v}, o, X) \\
\mathsf{filter'}(N, F, \_, v, o, X) &\triangleq (N, \mathsf{simpl}_{\mathsf{fv}}(F, v, o, X)) \\
\mathsf{filter'}(N, F, \boldsymbol{p}, \boldsymbol{v}, o, X) &\triangleq {}^2\bigcup_{i=1}^{a} \mathsf{filter'}(\mathsf{simpl}_{\equiv}(N, \boldsymbol{v}, o), F, p_i, v_i, oi, X)
\end{aligned}
$$

and where $\mathsf{simpl}_{\equiv} : 2^{\mathcal{N}l} \to \mathcal{W}^* \to \mathcal{P}\mathrm{os} \to 2^{\mathcal{N}l}$ is defined as

$$\mathsf{simpl}_{\equiv}(\{\{oi, oo'\}\} \uplus N, \boldsymbol{v}, o) \triangleq N \quad \text{if } \boldsymbol{v}|_i \equiv \boldsymbol{v}|_{o'} \text{ with } i \in \mathbb{N}, o' \in \mathcal{P}\mathrm{os}(\boldsymbol{v})$$
$$\mathsf{simpl}_{\equiv}(N, \boldsymbol{v}, o) \triangleq N \qquad \text{if } \nexists i, oi \in \{o' | \exists o'', \{o', o''\} \in N\}$$

and $\mathsf{simpl}_{\mathsf{fv}} : 2^{\mathcal{F}v} \to \mathcal{W}^* \to \mathcal{P}\mathrm{os} \to 2^{\mathcal{V}} \to 2^{\mathcal{F}v}$ is defined as

$$\mathsf{simpl}_{\mathsf{fv}}(F \uplus (o, Y), v, o, X) \triangleq \begin{cases} F & \text{if } \mathrm{FV}(v) \cap X \subseteq Y \\ F \cup (o, Y) & \text{otherwise} \end{cases}$$
$$\mathsf{simpl}_{\mathsf{fv}}(\emptyset, v, o, X) \triangleq \emptyset$$

Each simpl function reduces the sets of constraints if the constraint holds on some terms of v. The set of convertibility constraints is reduced when the function is called on a vector because we need several variables.

We define another matching,

Definition 2.12 (Nameless matching)**.** Let $P \to A$ be a clause matrix of width n and of height m, $\boldsymbol{v}$ a value vector of size n and j an index.

Row j of $\sharp\, P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}$ filters $\boldsymbol{v}$ when $\mathsf{filter}(N^j, F^j, \sharp(\boldsymbol{p}^j), \boldsymbol{v}) = (\emptyset, \emptyset)$. In that case, we write

$$\mathrm{Match}'[\boldsymbol{v}, \sharp\, P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}].$$

Example 2.13. The rewrite system

$$\mathtt{f}\,\mathtt{a}\,(\lambda x.\,\lambda y.\,G\,[x]) \to 0$$
$$\mathtt{f}\,X\,X \to 1$$
$$\mathtt{f}\,\mathtt{a}\,\mathtt{b} \to 2$$

is transformed into

$$\begin{bmatrix} \mathtt{a} & \lambda x.\,\lambda y.\,\_ \\ \_ & \_ \\ \mathtt{a} & \mathtt{b} \end{bmatrix} \to \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \left\{ \begin{bmatrix} \emptyset, \{(11, (x))\} \\ \{\{0, 1\}\}, \emptyset \\ \emptyset, \emptyset \end{bmatrix} \right\}$$

.

Lemma 2.14. *Let $\boldsymbol{p} \in \mathcal{P}^*$ of size n and $\boldsymbol{v} \in \mathcal{W}^*$ of size n as well. Compute $N \triangleq \mathsf{extract}_{\equiv}(\boldsymbol{p})$ and $F \triangleq \mathsf{extract}_{\mathsf{fv}}(\boldsymbol{p})$. We have*

$$\boldsymbol{p} \prec \boldsymbol{v} \iff \mathsf{filter}(N, F, \sharp\, \boldsymbol{p}, v) = (\emptyset, \emptyset).$$

Proof. By induction on n, definition of filter, N, F and $\prec$. □

The following proposition ensures that one can use any of the two matching definitions equivalently.

Proposition 2.15. *Let $P \to A$ be a clause matrix of width n and $\boldsymbol{v} \in \mathcal{W}^*$ a vector of size n. Define*

$$\boldsymbol{N} \triangleq (\mathsf{extract}_{\equiv}(\boldsymbol{p}))_{\boldsymbol{p} \in P}; \quad \boldsymbol{F} \triangleq (\mathsf{extract}_{\mathsf{fv}}(\boldsymbol{p}))_{\boldsymbol{p} \in P},$$

we have the equality

$$\mathrm{Match}[\boldsymbol{v}, P \to A] = \mathrm{Match}'[\boldsymbol{v}, \sharp\, P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}]$$

Proof. By Lemma 2.14. □

2.2.2. Matrix reduction

Clause matrices are reduced through the compilation process by several decomposition operations. A decomposition operator simplifies a matrix by filtering each of its rows under some assumption. For each row, if the assumption holds, it is kept (and modified). Otherwise the row is discarded.

Handling constraints

Constraints propagation The sets N and F previously described allow to declare a constraint at compile time. We introduce the actual constraints as propositions.

- $\mathfrak{N} : \mathbb{N} \to \mathbb{N} \to \mathcal{W}^* \to \mathbb{P}$ defined as $\mathfrak{N}(i, j, \boldsymbol{s}) = s_i \equiv s_j$;
- $\mathfrak{F} : \mathbb{N} \to 2^{\mathcal{V}} \to 2^{\mathcal{V}} \to \mathcal{W}^* \to \mathbb{P}$ defined as $\mathfrak{F}(i, V, X, \boldsymbol{s}) = \mathrm{FV}(s_i) \cap X \subseteq V$

Constraint $\mathfrak{N}$ is used for non linearity and constraint $\mathfrak{F}$ is used for closedness.

The additional argument of type $2^{\mathcal{V}}$ for closedness constraints is used to record the binders traversed and their bound variables, that is, it is the set X in the definition of filter and the X of MatchFv.

Simplification of constraint sets Let $\mathfrak{P} : \mathcal{W}^* \to \mathbb{P}$ be a constraint on a value vector. In practice, constraint $\mathfrak{P}$ can be either $\mathfrak{N}$ or $\mathfrak{F}$ partially applied. We suppose for conciseness that the constraint sets contain directly constraints of type $\mathcal{W}^* \to \mathbb{P}$. The constraint set is noted K^j, and will be in practice either a N set or a F set. We will use the function $\mathsf{filter}(K, \boldsymbol{p}, \boldsymbol{v})$ to represent the filter function working on one set at a time.

Define two operators $\overline{\mathcal{C}} : \mathcal{P}^* \to \mathcal{P}^*$ assuming a satisfaction of a constraint and $\underline{\mathcal{C}} : \mathcal{P}^* \to \mathcal{P}^*$ assuming the failure of satisfaction, (where $(K_1 \ \cdots \ K_n)\backslash X = (K_1\backslash X \ \cdots \ K_n\backslash X)$)

$$\overline{\mathcal{C}}(\mathfrak{P}, \boldsymbol{p}^j \to a^j \{C^j\}) = \begin{cases} \boldsymbol{p}^j \to a^j \{C^j\backslash\mathfrak{P}\} & \mathfrak{P} \in C^j \\ \boldsymbol{p}^j \to a^j \{C^j\} & \mathfrak{P} \notin C^j \end{cases}$$

$$\underline{\mathcal{C}}(\mathfrak{P}, \boldsymbol{p}^j \to a^j \{C^j\}) = \begin{cases} \text{No row} & \mathfrak{P} \in C^j \\ \boldsymbol{p}^j \to a^j \{C^j\} & \mathfrak{P} \notin C^j \end{cases}$$

Lemma 2.16. *Let $\boldsymbol{K}$ be a vector of sets of constraints (one per row),*

$$(\mathfrak{P} \in K^k \implies \mathfrak{P}(\boldsymbol{v})) \wedge \mathrm{Match}'[\boldsymbol{v}, P \to A\{\boldsymbol{K}\}] = k$$
$$\Updownarrow$$
$$\mathrm{Match}'[\boldsymbol{v}, \overline{\mathcal{C}}(\mathfrak{P}, P \to A\{\boldsymbol{K}\})] = k$$

Proof. Let $P \to A\{\boldsymbol{K}\}$ be a matching problem.

$\Downarrow$ Suppose there is a row $r = \ell \to k\{K_k\}$ in $P \to A\{\boldsymbol{K}\}$ such that $\mathsf{filter}(K_k, \ell, \boldsymbol{v}) = (\emptyset, \emptyset)$.

- If $\mathfrak{P} \notin K_k$, then $\overline{\mathcal{C}}(r) = r$ and thus $\mathrm{Match}'[\boldsymbol{v}, \overline{\mathcal{C}}(\mathfrak{P}, P \to A\{\boldsymbol{K}\})] = k$.
- If $\mathfrak{P} \in K_k$, $\overline{\mathcal{C}}(\mathfrak{P}, P \to A\{\boldsymbol{K}\}) = P \to A\{\boldsymbol{K}\}\backslash\{\mathfrak{P}\}$, by definition of filter, we have $\mathsf{filter}(K_k\backslash\{\mathfrak{P}\}, \ell, \boldsymbol{v}) = (\emptyset, \emptyset)$ and thus we have $\mathrm{Match}'[\boldsymbol{v}, \overline{\mathcal{C}}(\mathfrak{P}, P \to A\{\boldsymbol{K}\})] = k$

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

Table 2.1. : Decomposition operators

$\Uparrow$ Assume $\mathrm{Match}'[\boldsymbol{v}, \overline{\mathcal{C}}(\mathfrak{P}, P \to A\,\{K\})] = k$ and let $\ell \to k\,\{K_k\}$ the kth row. The kth row of $P \to A\,\{K\}$ is either

- $\ell \to k\,\{K_k\}$, we have $\mathsf{filter}(K, \ell, \boldsymbol{v})$ by hypothesis, and since $\mathfrak{P} \notin K_k$, the condition $\mathfrak{P} \in K_k \implies \mathfrak{P}(\boldsymbol{v})$ is true.
- $\ell \to k\,\{K_k\} \cup \{\mathfrak{P}\}$, by definition of $\overline{\mathcal{C}}$, $\mathfrak{P}(\boldsymbol{v})$ holds, we have $\mathsf{filter}(K_k, \ell \to k, \boldsymbol{v}$ (when applied to non linearity or closedness constraints), the matching still holds.

□

Lemma 2.17. *Let $\boldsymbol{K}$ be a set of constraints,*

$$(\mathfrak{P} \in K^k \wedge \neg\mathfrak{P}(\boldsymbol{v})) \wedge \mathrm{Match}'[\boldsymbol{v}, P \to A\,\{\boldsymbol{K}\}\}] = k$$
$$\Updownarrow$$
$$\mathrm{Match}'[\boldsymbol{v}, \underline{\mathcal{C}}(\mathfrak{P}, P \to A\,\{\boldsymbol{K}\})] = k$$

Proof. by definition of $\underline{\mathcal{C}}$. □

Operators $\underline{\mathcal{C}}$ and $\overline{\mathcal{C}}$ can now be declined for non linear constraints and free variable : $\underline{^{\equiv}\mathcal{C}}$, $\overline{^{\equiv}\mathcal{C}}$ and $\underline{^{\mathrm{FV}}\mathcal{C}}$, $\overline{^{\mathrm{FV}}\mathcal{C}}$.

Specialisation and abstraction

As we won't work with constraints in this section, we will use Match and $\prec$ rather than Match$'$ and filter. We will only use wildcards as well.

Specialisation assumes the first pattern filters some symbol. The specialisation $\mathcal{S}(\mathtt{f}, a, P \to A)$ is carried out with $\mathtt{f}$ with arity a. The definition is given in Table 2.1 page 33.

Lemma 2.18. *Let $P \to A$ be a clause matrix. Let $\mathtt{f}$ be a rewrite symbol and a an integer. Then the following equivalence holds,*

$$\mathrm{Match}[(\mathtt{f}\ w_1\ \cdots\ w_a)\, v_2 \cdots v_n, P \to A] = k$$
$$\Updownarrow$$
$$\mathrm{Match}[w_1 \cdots w_a v_2 \cdots v_n, \mathcal{S}(\mathtt{f}, a, P \to A)] = k$$

Proof. Let $P \to A$ be a clause matrix.

$\Downarrow$ Let $\boldsymbol{v} = \mathtt{f}\, w_1 \cdots w_a\, v_2 \cdots v_n$ be a value vector, and let $\boldsymbol{p} \to k$ be a row of $P \to A$ such that $\boldsymbol{p} \prec \boldsymbol{v}$. Looking at 2.2, we have either

- $\boldsymbol{p} = \mathtt{f}\, q_1 \cdots q_a\, p_2 \cdots p_n$ with $\forall i, q_i \prec w_i$;
- or $\boldsymbol{p} = X\, p_2 \cdots p_n$.

In both cases, the image of $\boldsymbol{p}$ by $\mathcal{S}(\mathtt{f}, \cdot)$ filters $w_1 \cdots w_a\, v_2 \cdots v_n$.

$\Uparrow$ Let $\boldsymbol{v}$ be a vector of values and $\boldsymbol{p}$ a row of $\mathcal{S}(\mathtt{f}, a, P \to A)$ such that $\boldsymbol{p} \prec \boldsymbol{v}$. By definition of $\mathcal{S}$, we have $\boldsymbol{p}$ and $\boldsymbol{v}$ of the form $\boldsymbol{p} = (q_1 \cdots q_a\ p_2 \cdots p_n)$ and $\boldsymbol{v} = (w_1 \cdots w_a\ v_2 \cdots v_n)$.

Then the row $\boldsymbol{p}'$ of $P \to A$ such that $\boldsymbol{p} = \mathcal{S}(\mathtt{f}, a, \boldsymbol{p}')$ is of the form either $\boldsymbol{p}' = \_\, p_2 \cdots p_n$ or $\boldsymbol{p}' = (\mathtt{f}\, q_1 \cdots q_a)\, p_2 \cdots p_n$.

- In the first case, by Rule MatchSymb, we have that $\boldsymbol{p}' \prec (\mathtt{f}\, w_1\ ...\ w_a)\, \boldsymbol{v}$.
- In the second case, we have the same result by Rule MatchPatt and MatchTuple.

□

A specialisation can be similarly defined for free variables considering that a free variable is a symbol of arity zero.

Similarly, one can specialise on an abstraction, which will be the operation $\mathcal{S}_\lambda(P \to A)$ described in Table 2.1 page 33.

Lemma 2.19.

$$\begin{gathered}\mathrm{Match}[(\lambda x.\, v)\, v_2 \cdots v_n, P \to A] = k \\ \Updownarrow \\ \mathrm{Match}[v\, v_2 \cdots v_n, \mathcal{S}_\lambda(P \to A)] = k\end{gathered}$$

Proof. By definition of $\mathcal{S}_\lambda$. □

Default

The default case is used when a value doesn't match any pattern of the form $\mathtt{f} \cdots$ and there is at least one row having a variable as first pattern. The default case $\mathcal{D}(P \to A)$ is described in Table 2.1 page 33.

Lemma 2.20. *Let $P \to A$ be a clause matrix with no abstraction in the first column. We have the following equivalence,*

$$\begin{gathered}\mathrm{Match}[\lambda x.\, w\, v_2 \cdots v_n, P \to A] = k \\ \Updownarrow \\ \mathrm{Match}[v_2 \cdots v_n, \mathcal{D}(P \to A)] = k\end{gathered}$$

Proof. By definition of $\mathcal{D}$. □

Lemma 2.21. *Let $P \to A\,\{C\}$ be a pattern matching problem. Suppose there are no symbol $\mathtt{f}$ applied in the first column. Then the following equivalence holds,*

$$\begin{gathered}\mathrm{Match}[\mathtt{f}\, w_1 \cdots w_a\ v_2 \cdots v_n, P \to A] \\ \Updownarrow \\ \mathrm{Match}[v_2 \cdots v_n, \mathcal{D}(P \to A)]\end{gathered}$$

Proof. By definition of $\mathcal{D}$. □

$D, D' ::=$		**Decision trees**
	$\mathsf{Leaf}(k)$	success (k is an action)
	Fail	failure
	$\mathsf{Switch}(L)$	multi-way test
	$\mathsf{BinFv}(D, (n, X), D')$	Binary switch on closedness constraint, $n \in \mathbb{N}, X \in 2^{\mathcal{V}}$
	$\mathsf{BinNl}(D, \{i, j\}, D')$	Binary switch on convertibility constraint, $(i, j) \in \mathbb{N}^2$
	$\mathsf{Swap}_i(D)$	stack swap ($i \in \mathbb{N}$)
	$\mathsf{Store}(D)$	save term on top of the stack
$L ::=$		**Switch case list**
	$(s, D) :: L$	tree symbol
	nil	empty list

Figure 2.3. : Syntax of decision trees

$s ::=$		**Tree symbols**
	c	tree constructor
	$*$	default case
$c ::=$		**Tree constructor**
	$\mathtt{f}_n$	rewrite symbol $\mathtt{f} \in \Sigma$ applied to n arguments
	λ	abstraction

Figure 2.4. : Syntax of tree constructors

2.3. Target language

Now that we have defined the source language, we describe the target language of the compilation process, that is, the language used by the rewriting engine. The target language is thus the language of decision trees whose syntax is given in Figure 2.3.

The terminal cases are Leaf and Fail. Structures Swap and Store provide control instructions to evaluate terms. A switch case list L is a non empty association list from tree symbols to decision trees defined in 2.3. The tree symbols are defined in Figure 2.4.

A switch case list is composed of all the tree symbols that can be matched, completed with an optional abstraction case if an abstraction should be accepted and an optional default case if there was a variable in the patterns. Evaluation rules to use such a list are given in Figure 2.5. We write $a|b$ in the rules to say we consider a or b respectively. By construction, a switch case list shall always be of the form

$$(\mathtt{a}_n, D_a) :: (\mathtt{b}_m, D_b) :: \cdots :: (\lambda, D_\lambda) :: (*, D_d) :: \mathsf{nil}$$

where $\mathtt{a}, \mathtt{b}, ... \in \Sigma$ (of respective arities n and m), D_a is a tree associated to $\mathtt{a}$. The elements (λ, D_λ) and $(*, D_d)$ may or may not be present. nil is the empty list.

$$\textsc{Found}\ \frac{}{c \vdash (c, D) :: L \rightsquigarrow (c, D)} \qquad \textsc{Default}\ \frac{}{c \vdash (*, D) \rightsquigarrow (*, D)} \qquad \textsc{Cont}\ \frac{c \neq c' \qquad c \vdash L \rightsquigarrow (c|*, D)}{c \vdash (c', D) :: L \rightsquigarrow (c|*, D)}$$

Figure 2.5. : Evaluation rules for association lists

Definition 2.22. The *domain* of an association list L is defined as

$$\operatorname{dom} L = \{s | \exists e, (s, e) \in L\}.$$

A semantics of evaluation with decision trees is given in Figure 2.6. Decision trees are evaluated with an input stack of terms. In addition, an array of terms is maintained during the evaluation. This array is filled thanks to Store nodes. Terms are stored to be used by constraints. Since the closedness constraints require to remember the removed abstractions (see Rule MatchFv), a set of free variables is also completed each time there is an abstraction.

An evaluation judgment $\boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k$ means that with stack $\boldsymbol{v}$, array of stored terms $\boldsymbol{s}$, variables from binders V; tree D yields action k.

2.4. Compilation scheme

The compilation is described as a relation between matrices $P \to A$ and their compiled form (that is, a tree) D.

$$P \to A \rhd D \text{ if } ((1\ 2\ \cdots\ m), \sharp\, P \to A\,\{\mathsf{extract}_{=}(P), \mathsf{extract}_{\mathsf{fv}}(P)\}, 0, \emptyset) \rhd D \tag{2.5}$$

Where the relation $\rhd$ is defined from elements of the form $(\boldsymbol{o}, P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E})$ to trees, with $\boldsymbol{o} \in \mathcal{P}\text{os}$ a vector of positions, $P \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}$ a clause matrix, n a storage counter and $\mathcal{E}$ an environment.

- The vector of positions represents at compile-time the values used during evaluation.
- The environment $\mathcal{E}$ maps a position to its index storage, $\mathcal{E} : \mathcal{P}\text{os} \to \mathbb{N}$. We note $\emptyset$ for the empty mapping.
- The storage counter is needed to create constraints on terms that are saved during evaluation. It allows to go from positions in the initial matrix to slots in an array of saved terms, and therefore transforms data of sets $N : 2^{\mathcal{N}l}, F : 2^{\mathcal{F}v}$ into actual to-be-checked constraints.

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

$$\boldsymbol{o}, \emptyset \to A\,\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E} \rhd \mathsf{Fail} \tag{2.6}$$

2. If there is a row k in P composed of unconstrained variables, matching succeeds and yields action k

$$\left(\boldsymbol{o}, \begin{pmatrix} p_1^1 & \cdots & p_n^1 & \to & a_1 & \{N^1, F^1\} \\ & & & \vdots & & \\ \_ & \cdots & \_ & \to & a_k & \{\emptyset, \emptyset\} \\ & & & \vdots & & \\ p_1^m & \cdots & p_n^m & \to & a^m & \{N^m, F^m\} \end{pmatrix}, n, \mathcal{E}\right) \rhd \mathsf{Leaf}(a_k) \tag{2.7}$$

$$\textsc{Match}\ \frac{}{\boldsymbol{v}, \boldsymbol{s}, V \vdash \mathsf{Leaf}(k) \rightsquigarrow k} \qquad \textsc{Swap}\ \frac{(v_i \cdots v_1 \cdots v_n), \boldsymbol{s}, V \vdash D \rightsquigarrow k}{(v_1 \cdots v_i \cdots v_n), \boldsymbol{s}, V \vdash \mathsf{Swap}_i(D) \rightsquigarrow k}$$

$$\textsc{Store}\ \frac{(v_1\ \cdots\ v_n), \boldsymbol{s}@v_1, V \vdash D \rightsquigarrow k}{(v_1\ \cdots\ v_n), (s_1\ \cdots\ s_m), V \vdash \mathsf{Store}(D) \rightsquigarrow k}$$

$$\textsc{SwitchSymb}\ \frac{\mathtt{f} \vdash L \rightsquigarrow (\mathtt{f}, D) \qquad (w_1\ \cdots\ w_a\ v_2\ \cdots\ v_n), \boldsymbol{s}, V \vdash D \rightsquigarrow k}{(\mathtt{f}\ w_1 \cdots w_a\ v_2 \cdots v_n), \boldsymbol{s}, V \vdash \mathsf{Switch}(L) \rightsquigarrow k}$$

$$\textsc{SwitchDefault}\ \frac{\mathtt{s}_a | \lambda \vdash L \rightsquigarrow (*, D) \qquad (v_2\ \cdots v_n), \boldsymbol{s}, V \vdash D \rightsquigarrow k}{(\mathtt{s}\ w_1 \cdots w_a | \lambda x.\, w\ v_2\ \cdots\ v_n), \boldsymbol{s}, V \vdash \mathsf{Switch}(L) \rightsquigarrow k}$$

$$\textsc{SwitchAbst}\ \frac{\lambda \vdash L \rightsquigarrow (\lambda, D) \qquad (w\{x\}\ v_2\ \cdots\ v_n), \boldsymbol{s}, V \cup \{x\} \vdash D \rightsquigarrow k}{(\lambda x.\, w\ v_2\ \cdots\ v_n), \boldsymbol{s}, V \vdash \mathsf{Switch}(L) \rightsquigarrow k}$$

$$\textsc{BinFvSucc}\ \frac{\boldsymbol{s}, V \vdash \mathrm{FV}(s_i) \cap V \subseteq X \qquad \boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k}{\boldsymbol{v}, \boldsymbol{s}, V \vdash \mathsf{BinFv}(D, (i, X), D_f) \rightsquigarrow k}$$

$$\textsc{BinFvFail}\ \frac{\boldsymbol{s}, V \vdash \mathrm{FV}(s_i) \cap V \not\subseteq X \qquad \boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k}{\boldsymbol{v}, \boldsymbol{s}, V \vdash \mathsf{BinFv}(D_s, (i, X), D) \rightsquigarrow k}$$

$$\textsc{BinNlSucc}\ \frac{(s_1\ \cdots\ s_i\ \cdots\ s_j\ \cdots\ s_m) \vdash s_i \equiv s_j \qquad \boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k}{\boldsymbol{v}, (s_1\ \cdots\ s_i\ \cdots s_j\ \cdots\ s_m), V \vdash \mathsf{BinNl}(D, \{i, j\}, D_f) \rightsquigarrow k}$$

$$\textsc{BinNlFail}\ \frac{(s_1\ \cdots\ s_i\ \cdots\ s_j\ \cdots\ s_m) \vdash s_i \not\equiv s_j \qquad \boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k}{\boldsymbol{v}, (s_1\ \cdots\ s_i\ \cdots s_j\ \cdots\ s_m), V \vdash \mathsf{BinNl}(D_s, \{i, j\}, D) \rightsquigarrow k}$$

Figure 2.6. : Evaluation of decision trees

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.

   Let $\mathfrak{S}$ be the predicate being true iff the first column contains a constrained variable and $n' = n + 1$ if $\mathfrak{S}$ holds and $n' = n$ otherwise. If $\mathfrak{S}$ holds, we also have to save the position to be able to pick back the term in the list $\boldsymbol{s}$. For this, we update the environment,

$$\mathcal{E}' = \begin{cases} \mathcal{E} \cup \{ \boldsymbol{o}|_1 \mapsto n \} & \mathfrak{S} \\ \mathcal{E} & \text{otherwise} \end{cases}$$

   Let Σ be the set of tree constructors defined from the symbols in the column. Then for each $s \in \Sigma$, a tree is built, with a the arity of constructor s,

$$\left((o_1|_1 \ \cdots \ o_1|_a \ o_2 \ \cdots \ o_n), \mathcal{S}(s, a, P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n', \mathcal{E}' \right) \rhd D_s$$

   Decision trees D_s are grouped into an association list L (we use the bracket notation for list comprehension as the order is not important here)

$$L \triangleq [(s, D_s) | s \in \Sigma]$$

   If there is an abstraction in the column, an abstraction case is added to the mapping

$$\begin{gathered} \left((o_1|_1 \ o_2 \ \cdots \ o_n), \mathcal{S}_\lambda(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n', \mathcal{E}' \right) \rhd D_\lambda \\ L \triangleq [(s, D_s) | s \in \Sigma] :: (\lambda, D_\lambda) \end{gathered} \tag{2.8}$$

   If the column contains a variable, the mapping is completed with a default case, (the abstraction case may or may not be present)

$$\begin{gathered} ((o_2 \ \cdots \ o_n), \mathcal{D}(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n') \rhd D_d \\ L \triangleq [(s, D_s) | s \in \Sigma] :: (\lambda, D_\lambda) :: (*, D_d) \end{gathered} \tag{2.9}$$

   A switch can then be created. Additionally, if $\mathfrak{S}$ holds, the term must be stored during evaluation to be able to enforce the constraint later. We finally define

$$(\boldsymbol{o}, P \to A\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E}) \rhd \begin{cases} \mathsf{Store}(\mathsf{Switch}(L)) & \text{if } \mathfrak{S} \\ \mathsf{Switch}(L) & \text{otherwise} \end{cases} \tag{2.10}$$

   b) If a term has been stored and is subject to a closedness constraint, then this constraint can be checked.

   That is, for any index in $\{i | i \in \operatorname{dom} \mathcal{E} \cap \{\boldsymbol{F}\}\}$, build two trees D_s used in case of satisfaction of the constraint, and D_f used otherwise,

$$(\boldsymbol{o}, \overline{\mathrm{FV}\mathcal{C}}(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n, \mathcal{E}) \rhd D_s; \quad (\boldsymbol{o}, \underline{\mathrm{FV}\mathcal{C}}(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n, \mathcal{E}) \rhd D_f$$

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

$$(\boldsymbol{o}, P \to A\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E}) \rhd \mathsf{BinFv}(D_s, (\mathcal{E}(i), X), D_f)$$

c) A non linearity constraints can be enforced when the two terms involved have been stored, that is, when there is a couple $\{i, j\}$ $(i \neq j)$ such that $(i, j) \in \operatorname{dom}\mathcal{E} \times \operatorname{dom}\mathcal{E}$ and $\{i, j\} \in \{\boldsymbol{N}\}$.

If it is the case, then compute

$$(\boldsymbol{o}, \overline{\overline{\equiv \mathcal{C}}}(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n, \mathcal{E}) \rhd D_s; \quad (\boldsymbol{o}, \underline{\equiv \mathcal{C}}(P \to A\{\boldsymbol{N}, \boldsymbol{F}\}), n, \mathcal{E}) \rhd D_f \tag{2.11}$$

and define

$$(\boldsymbol{o}, P \to A\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E}) \rhd \mathsf{BinNI}(D_s, \{\mathcal{E}(i), \mathcal{E}(j)\}, D_f) \tag{2.12}$$

4. If column i contain either a switch or a constraint, and $(\boldsymbol{o}', P' \to A\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E}) \rhd D'$ where $\boldsymbol{o}' = (o_i\, o_1\, ...\, o_n)$ and P' is P with column i moved to the front ; then we have

$$(\boldsymbol{o}, P \to A\{\boldsymbol{N}, \boldsymbol{F}\}, n, \mathcal{E}) \rhd \mathsf{Swap}_i(D') \tag{2.13}$$

Thus, one can decide to swap the columns to orient the compilation.

Example 2.23. Consider the following rewrite system, with $\Sigma = \{\mathtt{g}, \mathtt{b}, \mathtt{f}\}$

$$\mathtt{f}\ (\mathtt{g}\ X)\ \_ \to \mathtt{f}\ X\ X \qquad \mathtt{f}\ \mathtt{b}\ (\mathtt{g}\ X) \to \mathtt{f}\ \mathtt{b}\ X$$
$$\mathtt{f}\ X\ \mathtt{b} \to \mathtt{b}$$

This gives the clause matrix

$$P \to A = \begin{bmatrix} \mathtt{g}\ X & \_ \\ \mathtt{b} & \mathtt{g}\ X \\ X & \mathtt{b} \end{bmatrix} \to \begin{bmatrix} \mathtt{f}\ X\ X \\ \mathtt{f}\ \mathtt{b}\ X \\ \mathtt{b} \end{bmatrix}$$

The matrix is not empty and there is no row with only pattern variables. We are thus in case 3 or 4.

The first column of the matrix contains two symbols, that is, $\Sigma_1 = \{\mathtt{g}, \mathtt{b}\}$. A swap is not needed. We can proceed to case 3.

The first column contains a pattern variable, namely the X, however, it is not constrained, so no storage is necessary.

We first compute the specialised matrices as well as the default matrix,

$$\mathcal{S}(\mathtt{g}, 1, P) = \begin{bmatrix} X & \_ \\ \_ & \mathtt{b} \end{bmatrix} = M_1 \qquad \mathcal{S}(\mathtt{b}, 0, P) = \begin{bmatrix} \mathtt{g}\ X \\ \mathtt{b} \end{bmatrix} = M_2$$
$$\mathcal{D}(P) = \begin{bmatrix} \mathtt{b} \end{bmatrix} = M_3$$

then the trees stemming from these matrices can be built,

- $(o_1|_1\ o_2 \cdots o_n, M_1, 0, \emptyset) \rhd D_{\mathtt{g}}$
- $(o_2 \cdots o_n, M_2, 0, \emptyset) \rhd D_{\mathtt{b}}$
- $(o_2 \cdots o_n, M_3, 0, \emptyset) \rhd D_d$

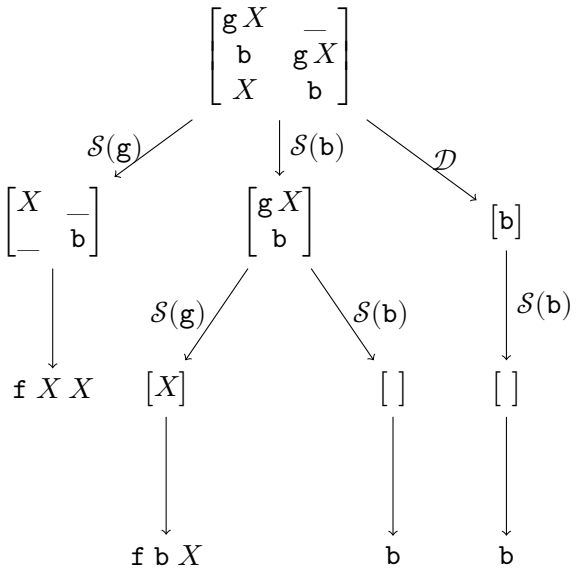

Figure 2.7. : Decision tree built for rewriting system of Example 2.23

Those three trees are the three subtrees of the root of Figure 2.7.
We can then build the switch case list

$$L_1 = (\mathtt{g}_1, D_{\mathtt{g}}) :: (\mathtt{b}_0, D_{\mathtt{b}}) :: (*, D_D) :: \mathsf{nil}$$

And the top node is finally

$$\mathsf{Switch}(L)$$

with

$$P \to A \rhd \mathsf{Switch}(L)$$

The complete tree is drawn Figure 2.7. The nodes of the tree are the left hand side matrices, the labels on the edges are the operations performed on matrices to reach children nodes.

2.5. Soundness & Completeness

Theorem 2.24 (Soundness)**.** *Let $P \to A$ be a clause matrix. If for some value vector $\boldsymbol{v}$, storage vector $\boldsymbol{s}$, set of variables V and decision tree D such that $P \to A \rhd D$ we have $\boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k$ then we have* $\mathrm{Match}[\boldsymbol{v}, P \to A] = k$.

Proof. By induction on $\vdash$,

- $D = \mathsf{Leaf}(k)$, $P \to A\,\{C\}$ has line k with only unconstrained pattern variables, and, by definition, $\mathrm{Match}[\boldsymbol{v}, P \to A] = k$;
- $D = \mathsf{Switch}(L)$, $\boldsymbol{v} = u@\boldsymbol{u}$, we proceed on case distinction on u of arity a,
  - Suppose $u \in \mathrm{dom}\, L$. By construction of L, there is a couple $(u, B) \in L$ with $(\mathcal{S}(u, a, P \to A)) \rhd B$. By induction hypothesis, there is a $\boldsymbol{w}$ such that $k = \mathrm{Match}[\boldsymbol{w}, \mathcal{S}(u, a, P \to A)]$. By Lemma 2.18, we have $\mathrm{Match}[\boldsymbol{v}, P \to A] = k$
  - Suppose $u = \lambda x.\, t$. For the same reason as above, $(\lambda, B) \in L$. We have $\mathcal{S}_\lambda(P \to A) \rhd B$. By induction hypothesis, there is a $\boldsymbol{w}$ such that $k = \mathrm{Match}(\boldsymbol{w}, \mathcal{S}_\lambda(P \to A))$. By Lemma 2.19, $k = \mathrm{Match}(v, P \to A)$.

– Suppose that $u \notin \operatorname{dom} L$. Consequently, there is $(*, D) \in L$, otherwise, the matching would fail. Since $P \to A \rhd D$, $\mathcal{D}(P \to A) \rhd D$. By induction hypothesis, there is a $\boldsymbol{w}$ such that $k = \text{Match}[\boldsymbol{w}, \mathcal{D}(P \to A)]$, and by Lemma 2.21, $k = \text{Match}[\boldsymbol{v}, P \to A]$.

- $D = \mathsf{BinNl}(D_s, \{i, j\}, D_f)$,
  – Suppose $s_i \equiv s_j$. Since $P \to A \rhd D$, we have $\overline{\equiv\mathcal{C}}(P \to A) \rhd D_s$. By induction hypothesis, $k = \text{Match}[\boldsymbol{v}, \overline{\equiv\mathcal{C}}(P \to A)]$, and by Lemma 2.16, $k = \text{Match}[\boldsymbol{v}, P \to A]$.
  – Suppose $s_i \not\equiv s_j$. The case is similar to the previous one using tree $D_f = \underline{\equiv\mathcal{C}}(P \to A)$ and Lemma 2.17.

- $D = \mathsf{BinFv}(D_s, (i, X), D_f)$,
  – Suppose $\text{FV}(s_i) \cap V \subseteq X$. The case is similar to the convertibility condition using $\overline{\text{FV}\mathcal{C}}(P \to A) \rhd D_s$.
  – Suppose $\text{FV}(s_i) \cap V \not\subseteq X$. Similar to convertibility condition with $\underline{\text{FV}\mathcal{C}}(P \to A) \rhd D_f$.

□

Theorem 2.25 (Completeness)**.** *Let $P \to A$ be a clause matrix. If for some value vector $\boldsymbol{v}$, we have* $\text{Match}[\boldsymbol{v}, P \to A] = k$*, then there is a decision tree $P \to A \rhd D$ and a vector of items $\boldsymbol{s}$ and a set of variables V such that we have $\boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k$.*

Proof. We proceed inductively on the definition of $\rhd$,

- $D = \mathsf{Leaf}(m)$, we have necessarily $m = k$ as row m is composed only of unconstrained vars. By rule (**Match**), we have immediately, with $\mathbf{0}$ the null vector

$$\mathbf{0}, \mathbf{0}, \emptyset \vdash D \rightsquigarrow k$$

- $D = \mathsf{Switch}(L)$, we proceed by case distinction on v_1 and L,
  – There is a tree B such that $(v_1, B) \in L$. Let $Q = \mathcal{S}(v_1, a, P \to A\,\{C\})$. We have $Q \rhd B$. By Lemma 2.18, we obtain a vector $\boldsymbol{w}$ such that $\text{Match}[\boldsymbol{w}, Q] = k$ and thus, by induction hypothesis, there is a $\boldsymbol{u}$ and a V such that $\boldsymbol{w}, \boldsymbol{u}, V \vdash B \rightsquigarrow k$. Applying Rule (**SwitchSymb**), we obtain $\boldsymbol{v}$ and $\boldsymbol{s}$ such that

$$\boldsymbol{v}, \boldsymbol{s}, V \vdash D \rightsquigarrow k$$

  – v_1 is not in L. We have the default case in L, otherwise, the matching would fail and that is not the case. Let $Q = \mathcal{D}(P \to A\,\{C\})$ and $Q \rhd B$ and $(*, B) \in L$. By Lemma 2.21, there is a vector $\boldsymbol{w}$ such that $\text{Match}[\boldsymbol{w}, Q] = k$. By induction hypothesis, we have $\boldsymbol{w}, \boldsymbol{s}, V \vdash B \rightsquigarrow k$, and we can apply Rule (**SwitchDefault**).
  – v_1 is an abstraction. If L does not contain an abstraction case, we fall into the previous case. Otherwise, the proof is similar to the first item.

- $D = \mathsf{BinNl}(D_s, (i, j), D_f)$,
  – if $(i, j) \in C_k$ and $s_i \equiv s_j$ or $\{i, j\} \notin C_k$, by Lemma 2.16, we have

$$\text{Match}[\boldsymbol{v}, \overline{\equiv\mathcal{C}}((i, j), P \to A\,\{C\})] = k$$

  As by construction we have $(\overline{\equiv\mathcal{C}}((i, j), P \to A\,\{C\})) \rhd D_s$, by induction hypothesis we have $\boldsymbol{v}, \boldsymbol{s} \vdash D_s \rightsquigarrow k$. We can thus apply inference rule (**BinNlSucc**) to obtain $\boldsymbol{v}, \boldsymbol{s} \vdash D \rightsquigarrow k$.

– if otherwise we have $\{i, j\} \in C_k$ and $s_i \not\equiv s_j$, by Lemma 2.17 we have

$$\mathrm{Match}[\boldsymbol{v}, \underline{\overset{\equiv}{\mathcal{C}}}(\{i, j\}, P \to A\,\{C\})] = k$$

By similar arguments, with $D_f = \underline{\overset{\equiv}{\mathcal{C}}}(\{i, j\}, P \to A\,\{C\})$, and applying inference rule (**BinNlFail**), we obtain the desired result.

- $D = \mathsf{BinFv}(D_s, (i, X), D_f)$,
  – if (i, X) in F^k and $\mathrm{FV}(s_i) \cap V \subseteq X$ or $(i, X) \notin F^k$, by Lemma 2.16 we have

$$\mathrm{Match}[\boldsymbol{v}, P \to A\left\{\overline{\mathcal{C}}(C)\right\}]$$

□

3. Implementation and results

While the previous chapter formalised the algorithm, we are here interested in the practical aspect. We show where, when and why, in Dedukti3, the algorithm is used ; then we evince some interesting aspects. The last section concerns the results, in which we mainly compare the performances of the rewriting engines of Dedukti2, Dedukti3 and Dedukti3 with decision trees.

3.1. Normalisation algorithm

We here show how the trees are used in the context of reduction. We begin by defining the reduced terms, and then explain how to reach those kind of terms.

Definition 3.1 (Head structure)**.** Let _ be a constant. The *head structure* of a term is defined inductively as

$$\begin{aligned} \mathsf{hs}(\lambda x : A.\, t) &\triangleq \lambda x : A.\, \mathsf{hs}(t) \\ \mathsf{hs}(t\ u) &\triangleq \mathsf{hs}(t)\ \_ \\ \mathsf{hs}(t) &\triangleq t \end{aligned}$$

Definition 3.2 (Weak head normal form)**.** A term t is in *weak head normal form* (whnf) if it is an abstraction or if its head structure is invariant by reduction.

To reduce a term to its whnf, we define the whnf function as (where @ is the concatenation or cons, depending on the argument)

$$\begin{aligned} \mathsf{whnf}(t) &\triangleq \mathsf{add\_args}(u, \boldsymbol{u}) \\ &\quad \text{where } (u, \boldsymbol{u}) = \mathsf{whnf}'(t, \mathsf{nil}) \\ \mathsf{whnf}'(t\ u, \boldsymbol{s}) &\triangleq \mathsf{whnf}'(t, u@\boldsymbol{s}) \\ \mathsf{whnf}'(\lambda x.\, t, u@\boldsymbol{s}) &\triangleq \mathsf{whnf}'(t\{x := u\}, \boldsymbol{s}) \\ \mathsf{whnf}'(\mathtt{f}, \boldsymbol{t}) &\triangleq \begin{cases} (\mathtt{f}, \boldsymbol{t}) & \text{if } \boldsymbol{t} \vdash \mathsf{tree}(\mathtt{f}) \rightsquigarrow \bot \\ (\mathtt{u}, \boldsymbol{u}) & \text{if } \boldsymbol{u} \vdash \mathsf{tree}(\mathtt{f}) \ni \mathsf{Leaf}(u) \rightsquigarrow u \end{cases} \\ \mathsf{add\_args}(t, u@\boldsymbol{v}) &\triangleq \mathsf{add\_args}(t\ u, \boldsymbol{v}) \\ \mathsf{add\_args}(t, \mathsf{nil}) &\triangleq t \end{aligned} \tag{3.1}$$

with $\mathsf{tree}(\mathtt{f})$ the function building the tree associated to the clause matrix containing all rules whose root symbol is $\mathtt{f}$. In the last case of whnf', $\boldsymbol{u}$ is the last value stack. root

We present the algorithm used to reduce a term to its whnf in Appendix B because it is very close to the implementation as well as the theory. The whnf is used extensively in the unification algorithm. def.1.15 p.19

3.2. Implementation concerns

This section groups several remarks concerning the implementation of the procedures described previously. The full code is available on github at `https://github.com/gabrielhdt/lambdapi.git` and is about 1.4k lines of code.

3.2.1. Binders representation

While Dedukti2 used De Bruijn representation of binders, Dedukti3 uses higher order abstract syntax implemented in Bindlib by R. Lepigre in [LR18]. Consequently, binder operations are abstracted. For instance, to check occurrence of a free variable `x : term Bindlib.var` in a term t,

```
let b = lift t in
Bindlib.occur x b
```

where `lift: term -> term Bindlib.box` transforms a term t into a `Bindlib.box`, a datatype used to encapsulate incomplete binders and to manipulate them easily.

Having binder $\lambda x.\, t$, the term t with bound variable x replaced by a free variable is obtained with `Bindlib.unbind t` which returns both the body t and the fresh free variable injected into it. In our case, we want to create variables at compile time and reuse them at evaluation time. For this, variables are created with `Bindlib.new_var mkfree s` where `s` is the name of the variable and `mkfree` is the function used to inject a variable in a term.

3.2.2. Building the substitution

Assume term `f (g o) k` is matched against the following rule

$$\texttt{f}\ (\texttt{g}\ X)\ Y \hookrightarrow Y\ X \tag{3.2}$$

To obtain the right-hand side, a substitution mapping variables from the left-hand side to some subterms of the filtered term must be built. Here, in order to yield `k o`, the substitution is $\sigma = \{X \coloneqq \texttt{o}, Y \coloneqq \texttt{k}\}$.

To build such a substitution, each leaf of the tree contain a structure mapping slots of the array of terms s used during evaluation in inference rules (see Figure 2.3) to the corresponding pattern variable in the right-hand side.

Back on our previous example, given a tree (and a heuristic), the array s of variables can be, when reaching a leaf, `[| o; k |]`. The mapping built at compile time would then be $\sigma = \{X \coloneqq s_0, Y \coloneqq s_1\}$. Finally, X and Y will be substituted by terms at `s.(0)` and at `s.(1)` yielding `k o`.

To go deeper in the implementation, we can explain briefly how the pattern variables are represented, and how the substitution $X \coloneqq s_0$ is performed in practice. The right-hand side is itself a binder (and thus represented as a `Bindlib.binder`), with an environment as an array of terms. Each pattern variable in the left-hand side that is also in the right-hand side has a dedicated slot in that environment. Consequently, the mapping is in practice from slots of the s array filled during evaluation to slots of the environment of the right-hand side.

To build the aforementioned mapping during tree compilation, we use the storage counter n introduced in section 2.4. Let r be a row of the matrix whose first element is a pattern variable. Let i be the slot of that pattern variable in the right-hand side. Then $\{n \coloneqq i\}$ is added to a mapping attached to the rule r.

3.2.3. Sharing

The current code of Dedukti3 already implements sharing; it is important in practice, as explained in the following example.

Example 3.3. Consider the following rewrite system, with $\Sigma = \{\texttt{is\_succ}, \texttt{zero}, \texttt{s}, \texttt{f}, \texttt{t}, \texttt{fact}\}$,

$$\texttt{is\_succ zero} \to \texttt{f}$$
$$\texttt{is\_succ (s \_)} \to \texttt{t}$$

with in addition the function `fact` computing the factorial of a number. If there is no sharing, `is_succ (fact 21)` will call `fact 21` twice,

1. the first time on the first rule, which leads to a matching failure since $21\,! \neq 0$,
2. a second time when trying to match the second rule, which will succeed.

This is a problem since computing factorial is long. Using sharing allows to compute `fact 21` only once. Indeed, when matching the first rule, the computation will be saved into the argument stack, and the second rule will match directly the result of `fact 21` against `s _`.

Sharing is implemented on the terms of the input stack, i.e. the terms that are given as input the normalisation function `whnf`. The goal is to profit from the recursive calls to `whnf` even if the matching fails.

whnf def.3.1 p.43

The `tree_walk t stk` of Appendix B page 69 performs recursive calls to `whnf` on elements of `stk` to match its head structure against tree constructors. In Equation 3.1, the recursive calls are induced by the rewriting step, that is, $\boldsymbol{s} \vdash \mathsf{tree}(s)$. More precisely, these recursive calls appear when attempting to match s against a symbol in a switch case list.

Without sharing, if the matching fails (that is, $\boldsymbol{s} \vdash \mathsf{tree}(\texttt{s}) \rightsquigarrow \bot$ in Equation 3.1 or, in the code in Appendix B, the function `tree_walk` returns `None`), then the stack in is unchanged.

With sharing, calls to `whnf` modify also the terms of the original stack. This way, even if the matching fails, the reduction performed will be saved (into the `stk` variable in the body of `whnf_stk`, as stack $\boldsymbol{t}$ in Equation 3.1).

In practice, this sharing is performed via OCaml references and the main `term` datatype (as written in Appendix B) which contains an additional constructor of type `TRef : term ref -> term`. Therefore, having a term `t`, `TRef(t)` is a term with a reference to term `t`. With this, we are able to transform any application `let t = Appl(Appl(f, u), v)` into `let t = Appl(Appl(f, TRef(u)), TRef(v))`. Any reduction performed on either `u` or `v` will update `t` as well.

3.2.4. Incremental constraint update

To avoid pre-parsing each left-hand-side to fill constraints sets as described in section 2.2.2, they can be filled incrementally during compilation.

Three sets have to be used per row, (N, F, Q) where Q is the set containing pattern variables. This set is required to record non linearity constraints as they involve at least two variables.

3.3. Results

This section presents the performance of our matching algorithm and rewriting engine. We use the notation $T(n)$ to denote the temporal complexity in function of a parameter n. We use the general Knuth-Landau notations $T(n) = O(n)$.

Remark 3.4. During the last two weeks of the internship, a mistake has been found in Dedukti3. The consequence was that the rewriting engine wasn't correct. Fixing this issue in the legacy rewriting engine deteriorates heavily the performance. The issue has been fixed in our decision trees as well, but the performance are unchanged. The computation time do not take into account the bug fix.

3.3.1. On the shape of rules

Care must be taken when comparing methods regarding which rule is used by each method. This question can arise when using symbols with apparently redundant rules like

$$0{+}N \to N \tag{3.3}$$
$$(\mathtt{s}\, M){+}N \to \mathtt{s}\,(M{+}N) \tag{3.4}$$
$$M{+}0 \to M \tag{3.5}$$
$$M{+}(\mathtt{s}\, N) \to \mathtt{s}\,(M{+}N) \tag{3.6}$$

Using the rule that reduces its first argument can be much faster than using the other. For instance define

$$\mathtt{triangle}\, 0 \to 0 \tag{3.7}$$
$$\mathtt{triangle}\,(\mathtt{s}\, N) \to N{+}(\mathtt{triangle}\, N) \tag{3.8}$$

Proposition 3.5. *Let* `triangle` *be defined as above.*

1. *If* $+$ *is defined reducing its rightmost argument, then* $T(\mathtt{triangle}\, n) = O(n^3)$.
2. *If* $+$ *reduces its leftmost argument, then* $T(\mathtt{triangle}\, n) = O(n^2)$.

Proof. 1. $+$ is defined by 3.5 and 3.6. First, $\mathtt{triangle}\, n \to^* n{+}(n{-}1{+}(n{-}2{+}\cdots{+}(1{+}0)\cdots))$ in n operation using 3.8 and 3.7.

The next rule to be used is 3.5 to transform $1+0 \to 1$. The rightmost term is now $2+1$. Rule 3.6 yields $2+1 \to \mathtt{s}\,(2+0) \to \mathtt{s}\, 2$. Using 3.6 $n-2$ times will allow the generated `s` to traverse the term from right to left, resulting in $\mathtt{s}\ (n + (n-1+(n-2+\cdots+(3+2)\cdots)))$. Re iterating twice this last operation will exhaust the rightmost 2. The rightmost term is then 3, which will be exhausted calling 3.6 $3(n-2)$ times.

Rigorously, we should prove by induction that if the rightmost term is j, then reducing this term to zero costs $j(n-j)$ operations approximately.

Assuming this, we deduce that

$$T(\mathtt{triangle}\, n) = O\left(\sum_{j=0}^{n} j(n-j)\right) = O(n^3)$$

2. $+$ is now defined with 3.3 and 3.4. The first rule called is 3.8, $\mathtt{triangle}\, n \to n{+}\mathtt{triangle}\,(n{-}1)$. Next, instead of reducing `triangle` as above, calling 3.4 yields $n{+}\mathtt{triangle}\,(n-1) \to \mathtt{s}\,((n-1){+}\mathtt{triangle}\,(n-1))$. Calling again the same rule $n-1$ times results in

$$\overbrace{\mathtt{s}\,(\mathtt{s}\,(\cdots(\mathtt{s}}^{\times n}\,(\mathtt{triangle}\,(n-1)))\cdots))$$

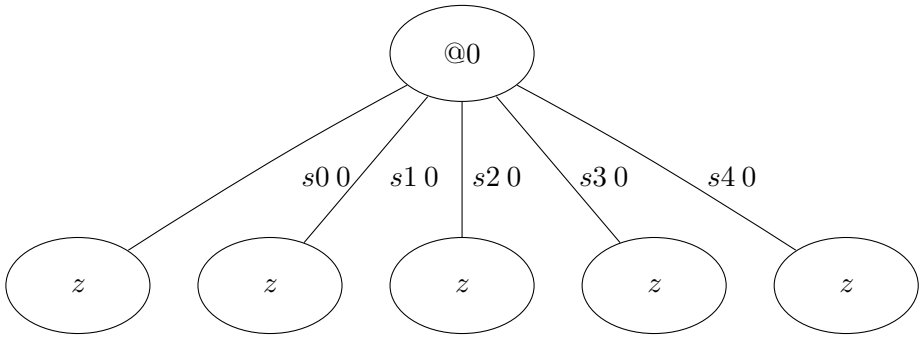

Figure 3.1. : Thump of width 4

Re iterating this procedure $n-1$ times yields the final result in

$$T(\texttt{triangle}\, n) = O\left(\sum_{k=0}^{n} k\right) = O(n^2)$$

□

If the behaviour of the naïve matching algorithm is easy to predict, the rule chosen by the decision tree is not. To avoid the aforementioned phenomenon, the + can be designed such that it does not always deconstruct on the same argument, for instance,

$$\begin{aligned} 0{+}N &\to N \\ M{+}\texttt{s}\, N &\to \texttt{s}\, (N{+}M) \\ M{+}0 &\to M \\ \texttt{s}\, M{+}N &\to \texttt{s}\, (N{+}M) \end{aligned}$$

3.3.2. Toy examples

Some tests are performed on specific examples to evince strengths or weaknesses of decision trees.

Thump

A thump is a tree of depth 1. In this test, a large thump is created to measure the efficiency of selecting a rule among many. The test file is created by a script. The rules are as follows with $\texttt{s}i$ being constants :

(Thump)
$$\begin{aligned} \texttt{thump s0} &\to 0 \\ \texttt{thump s1} &\to 0 \\ &\vdots \\ \texttt{thump s}n &\to 0 \end{aligned}$$

The generated tree can be seen Figure 3.1 page 47. This test allows to measure the branching efficiency of the decision tree engine. The naïve algorithm basically traverses all its rules until the good one is found, and thus $T(\mathrm{Match}_{\mathrm{leg}}(\texttt{thump}\, k)) = \Theta(k)$. The behaviour of our decision tree is similar, except that it traverses its branches. Branches are stored into an OCaml Map. Knowing that OCaml implements maps as balanced tree, the access complexity is $O(\log n)$ for n the number of elements of the map. Figure 3.2 evinces the benefit of using decision trees, the computation times stay smaller, and the difference increases with the width of the thump.

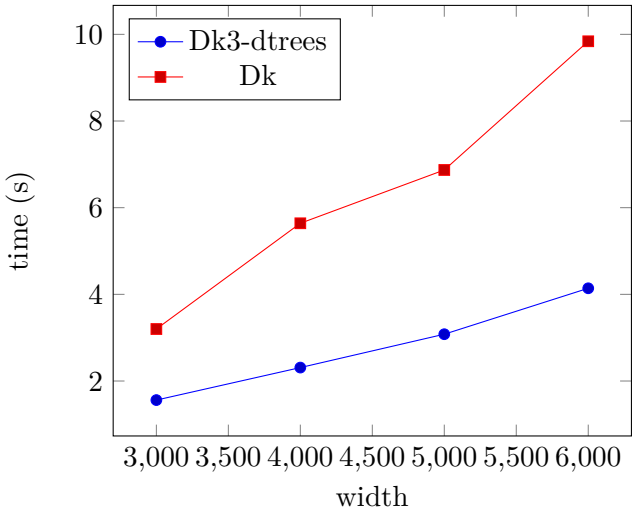

Figure 3.2. : Computation time on thumps

Comb

The principle is to use a deep unbalanced tree. Remind that naturals are represented using a unary notation, i.e. 3 is `s` (`s` (`s` `z`)).

(Comb)
$$\begin{aligned} \texttt{comb}\ 0 &\to 0 \\ \texttt{comb}\ 1 &\to 0 \\ &\vdots \\ \texttt{comb}\ n &\to 0 \end{aligned}$$

The tree generated can be seen Figure 3.3 page 49. Looking at Table 3.1 page 51, decision trees perform better than legacy matching. It might be explained by a quick complexity analysis. Regarding legacy matching, as all rules before the one that match are tested, we can deduce that $T(\text{Match}_{\text{leg}}(\texttt{comb}\,k)) = \sum_{j=0}^{k} T(\text{Match}(k,j))$, where $\text{Match}(k,j)$ is matching unary integer j against unary integer k. As $T(\text{Match}(k,j)) = O(\min(k,j))$, we obtain $T(\text{Match}_{\text{leg}}(\texttt{comb}\,k)) = \sum_{j=0}^{k} j = O(k^2)$.

With decision trees, $\text{Match}_{\text{dtree}}(\texttt{comb}\,k)$ only walks through the tree depicted in Figure 3.3 once, therefore $T(\text{Match}_{\text{dtree}}(\texttt{comb}\,k)) = O(k)$. The Figure 3.4 exhibits the somewhat linear ratio between the complexity using decision trees and the complexity without.

Flagellum

A flagellum designates a degenerate tree, equivalent to a list. The idea is to test the engine on deep trees. A pattern giving birth to a flagellum can be

$$\texttt{flagellum}\ \texttt{m}\ \overbrace{\_\cdots\_}^{\times n} \to 0$$

which creates the tree in Figure 3.5 page 50. The complexity of matching between trees and naïve matching is the same, in $O(n)$, with n being the arity of the flagellum. Figure 3.6 page 50 shows that no algorithm stands particularly better than the other since the difference looks constant.

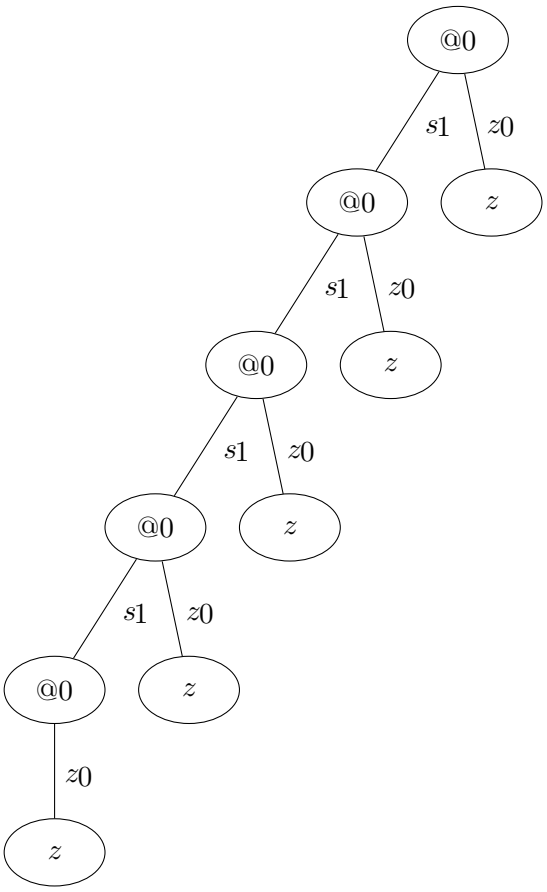

Figure 3.3. : Comb of depth 4

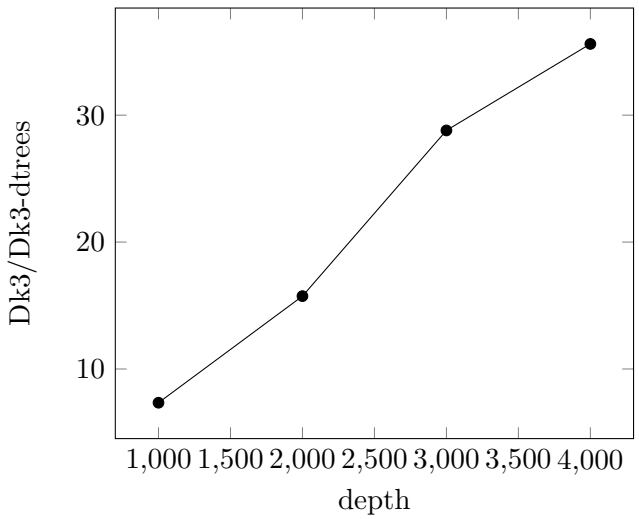

Figure 3.4.: Ratio of computing time without decision trees on computing time using decision trees

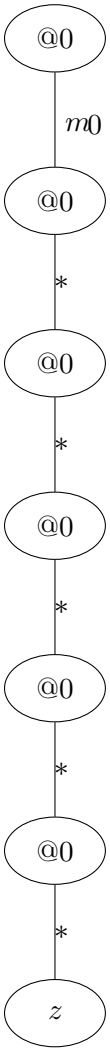

Figure 3.5. : Flagellum of length 6

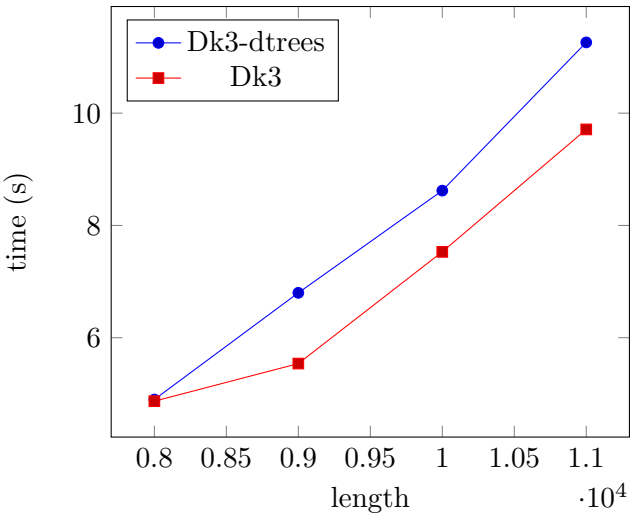

Figure 3.6. : Computation time on a flagellum

File	Param	Dedukti3	Dedukti3 dtrees
Comb	2000	1 min 13	4.4
Thump	4000	5.3	2.3
Flagellum	8000	4.1	3.8
Loop		3.7	4.1
LoopNL		3.8	0.3
Fibonacci		10.5	10.3

Table 3.1. : Comparing performances of rewriting engines, times in seconds

Sudoku	Dedukti2	Dedukti3	Dedukti3 dtrees
easy	0.7	0.5	0.5
medium	7.7	5.2	5.2
hard	8 min 43	5 min 8	5 min 15
empty		3.5	3.5

Table 3.2. : Duration of solving sudokus in seconds

Non linear loop

The following set of rules show that the naïve method can easily carry out useless computations :

(LoopNL)
$$\texttt{loopnl}\ X\ X\ (\texttt{s}\ Y) \to \texttt{loopnl}\ X\ X\ Y$$
$$\texttt{loopnl}\ (\texttt{s}\ X)\ \_\ Z \to \texttt{loopnl}\ X\ X\ 500$$

While the legacy rewriting engine shall each time verify the convertibility constraint, the decision tree looks directly at the third argument and thus can avoid the convertibility check.

3.3.3. Hand written libraries

The github repository `https ://github.com/deducteam/libraries` contains several hand-written Dedukti examples, including a sudoku solver and a SAT solver implementing DPLL.

Sudoku

Three sudoku problems are available. The solving times are in Table 3.2 page 51.

DPLL

Definition 3.6 (CNF)**.** A boolean formula is in *conjunctive normal form* or *clausal normal form*, also written cnf when it is the conjunction of one or more clauses, where each clause is a disjunction of literals.

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is used to decide the satisfiability of propositional logic formulae in conjunctive normal form.

A naive SAT-solver works the following way. Having a formula in cnf, a variable is picked, and a truth value is assigned. If the recursive call of the algorithm on the simplified formula returns true, then the original formula with this assignment is true. Otherwise, a backtrack is performed and the opposite truth value is assigned ; a recursive call is then performed on the simplified formula.

DPLL uses two other rules to cut down the search tree.

DPLL	Dedukti2	Dedukti3	Dedukti3 dtrees
2_ex	2	1	0.2
ok_50x80	4	4.3	0.3

Table 3.3. : Duration of solving SAT problems with DPLL in seconds

- If a clause contains only one variable, then the truth value which satisfies that clause is set.
- If a variable appears always positive, or always negative, then the truth value that makes all clauses true is assigned to this variable.

There are two example files

- `2_ex` with a problem defined as, for any n in $\mathbb{N}$ and v_n a literal,

$$p(0) = v_0; \quad p(n+1) = p(n) \wedge (v_n \neq v_{n+1})$$

- `ok_50x80` is an arbitrary formula with 50 literals.

sharing 3.2.3 p.45

Because of backtracking, DPLL algorithm makes sharing crucial. Each time there is an assignment, the formula is simplified thanks to rewriting rules. Therefore, if there is no sharing, each time a backtracking is performed, all simplifications made on sub problems are lost.

3.3.4. Libraries

Libraries are collection of definitions and theorems along with their proofs. They embody what Dedukti is made for, checking proofs, rather than performing computation. Currently, 6 libraries have been translated into Dedukti.

- dklib
- Focalide, containing proofs from Focalize development ;
- HolIDe, proofs from the standard library of OpenTheory, the common format for HOL proof assistant ;
- Matita, containing files produced by Krajono, a translator from the proof assistant Matita implementing the calculus of inductive constructions to Dedukti ;
- Verine containing files translated from the automated SMT solver veriT ;
- IProverModulo TPTP, containing files generated by IProverModulo from TPTP ;
- ZenonModulo set theory library, proofs generated by the first order theorem prover Zenon.

Statistics of rules The Table 3.5 gives some statistics on some of the libraries, where the arity and heights are computed on the left-hand side of rules. The arity of a rule is the arity of the root symbol, e.g. the arity of `f` $X\ Y \to X$ is 2.

height def.1.14 p.19

Library	Dedukti3	Dedukti3 dtrees
FocaliDe	7.7	7.7
Matita	18 min	25 min
Holide	4 min 20	4 min 30
Verine	1 min 10	1 min 8

Table 3.4. : Libraries type checking time (in seconds)

	symbols	rules	avg. arity	avg. height	non linear
Dklib	361	258	2.5	2.2	1
Focalide	3225	1283	3	17.6	0
Holide	329840	3	1	2	
Verine	157780	7	1	1.4	0

Table 3.5. : Statistics of rewrites rules of libraries

3.3.5. Rewriting engine competition

The Rewriting Engine Competition[1] first appeared in 2009, organised by a university in Illinois with the objective to compare rewriting engines and make people aware of them. F. Duràn and H. Garavel revived the competition in 2018, and another study has been done in 2019 [DG19]. There are 14 rewriting engines tested, among which Haskell's GHC and OCaml.

The problems are written in a specific REC syntax which is then translated into the target languages with Awk scripts. To translate those scripts to Dedukti, I chose to start from Haskell files and to translate them into Dedukti3 files. The script is available at `https://github.com/gabrielhdt/lambdapi/tree/rec/tools/rec_to_lp`.

We have compared Dedukti3 with decision trees with OCaml and Haskell using GHC in Table 3.6 page 53. The column `ocaml` and `runghc` interpret the files while `ocamlopt` and `ghc` compile the source file to machine language.

We observe that our algorithm does in general not scale as well as other tools, even though it performs better in some cases.

1. http ://rec.gforge.inria.fr

	Dedukti3	`ocamlopt`	`ocaml`	`ghc`	`runghc`
`add8`	11.3	N/A	12.5	24.1	N/A
`add16`	1 min 26	N/A	7.6	2 min 32	18.5
`mul8`	2 min 43	2.1	3.2	0.4	4.0
`revnat100`	0.02	0.09	0.06	0.45	0.6
`revnat1000`	2.4	0.1	0.14	0.45	5.0
`garbagecollection`	0.01	0.1	0.06	0.5	0.5

Table 3.6.: Performance on various REC benchmarks in seconds. N/A when the program ran out of memory.

4. Miscellaneous work

This chapter concerns miscellaneous work that I have been assigned to.

4.1. ReSyStanCe

To better understand the performance of the rewriting engine, a tool to evaluate the characteristics of rewrite systems has been developed. It is available on github at `https://github.com/deducteam/resystance`. It uses Dedukti3 as an API, and given a set of rewrite rules, it computes

- the number of symbols,
- the number of rules,
- the number of higher order rules,
- the number of non linear rules,
- the distribution of the aritiy of the root symbols of rules,
- the distribution of the height (as defined in Definition 1.14) of rules.

4.2. Isabelle

Makarius Wenzel, developer of Isabelle, was invited at the LSV from the 17th to the 28th of June. As I have had a PhD grant to work on the translation from Isabelle to Dedukti, I worked with Makarius on this subject. The idea was to create a simple syntactical translator which would output a Dedukti3 file. We have been able to translate axioms, theorems and constants but not the proofs for they are not available in the Isabelle environment. The code is available at `https://github.com/gabrielhdt/isabelle_dedukti`. The translator is an Isabelle plugin written in Scala using Isabelle core data to write Dedukti3 files.

4.3. Logipedia

In february, Deducteam has been allowed to recruit two engineers from Inria to work on the web part of Logipedia. Meanwhile, François Thiré, who developed the initial version of Logipedia is about to leave the team soon as he will defend his PhD. Consequently, as a new PhD student, I have been asked to take over the maintenance of Logipedia along with a colleague.

Our first assignment has been to define a standard format that the engineers can use to automate the creation of the website. As standard format we mean a widely used plain text format that can be parsed efficiently. We chose to use JSON for its integration with OCaml (via `ppx` and the library Yojson). The idea is to communicate between the OCaml world of proofs and logic and the frontend via these files. A Logipedia web page contains the following sections :

```
type id = string
type qid = id list
type var =
  { v_symb: id
  ; v_args: ppterm list }
and const =
  { c_symb: qid
  ; c_args: ppterm list }
and binder =
  { b_symb: id
  ; bound: id
  ; annotation: ppterm option
  ; body: ppterm }
and ppterm =
  | Var of var
  | Binder of binder
  | Const of const
```

Figure 4.1. : Pretty printable terms

general contains global data on the theorem or definition,

- a qualified identifier,
- the original logic system,
- the category of the object (axiom or proof or definition or constant),
- the term,
- the dependencies of the proof,
- the basic axioms and
- the available export logic systems.

export contains data of exported proofs

- the logic into which the element is exported
- a textual representation, in the target system, of the object.

Moreover, displaying of mathematical formulae is a not so trivial problem : the transformation of a complex term representation into a readable string. To address this issue, we chose to separate the pretty printing software from Logipedia using the serialised files. The files would contain an intermediate representation of terms that retain their complex structure, but that is flexible enough to handle the diversity of systems. To this end we created "pretty printable terms", whose OCaml datatype is given in Figure 4.1. The next step is then to write a code that transforms one of these `ppterm` written as JSON into HTML. A prototype that transforms a JSON `ppterm` into LaTeX has been developed (in Prolog), and is available at `https://github.com/gabrielhdt/logippedia.git`. With the help of software like KaTeX[1], the LaTeX code can be embedded into HTML.

1. https://katex.org

Conclusion

This work presents the development of a new rewriting engine for the Dedukti3 software. This rewriting engine is needed mainly for peformance and scalability.

We have described, formalised and evaluated the adaptation of an algorithm for first order matching with algebraic patterns to higher order non algebraic patterns.

The first section started by exposing the theoretical basis of the work, brought some information on the application of such an algorithm (proofs assistants) and finished by introducing the main topic : pattern matching and rule filtering.

As the algorithm performs compilation, the formalism started by describing the source language and its properties and continues on the target language, the language of decision trees. The correctness of the compilation is stated and proved.

Some details of implementation are then informally given, followed by the results. We compared our algorithm with two previous versions of Dedukti as well as with OCaml and Haskell. The results are rather optimistic, with small to massive improvements when compared to Dedukti2 and Dedukti3, but it suffers from a poor scalability when compared to programming languages.

The document is closed by some remarks on miscellaneous work performed for the team, and in particular for Logipedia, the main application of our rewriting engine.

Further work The REC database contains problems using conditional rewriting. Implementing the latter could be done without too much work since the way we handled closedness tests and non linearity is abstract enough to be made general.

Rewriting modulo associativity and commutativity could be implemented, as a next feature. It is needed for instance to translate Coq into Dedukti.

Concerning ReSyStanCe, it would be interesting to allow it to access some features of the rewriting engines to e.g. record rewriting steps, to know how many rewriting have been performed, how many β reductions, &c.

Personal outcome

The courses given at the ENAC provided me a useful background in theoretical computing thanks to the lessons on OCaml, Prolog, constraint programming, symbolic artificial intelligence, theory of language and automata and theory of compilation. Regarding the double diploma with Paul Sabatier, the lessons on modal logics acted as first contact with logics, although this lesson could have been made more formal. On the other hand it would seem relevant that the ENAC provides teaching on proof assistants, considering that aviation uses critical software ; and it would be a visible link between the world of certification, embodied by the OPS section, and the world of computer science.

More personally, this internship allowed me to work in the context of fundamental research, after having carried out a three-months internship in applied research at the University of Westminster. It appears that this last experience is the one I preferred, motivating a PhD in the same team, on the translation of the Isabelle prover into Dedukti.

Bibliography

[ARCT11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *CADE*, 2011.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. In *TPHOLs*, 2009.

[BN99] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *TLCA*, 2007.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2) :56–68, 1940.

[DG19] Francisco Durán and Hubert Garavel. The rewrite engines competitions : A rectrospective. In *TACAS*, 2019.

[dt18] Coq development team. The coq proof assistant manual : version 8.9. 2018.

[Egi15] Satoshi Egi. Egison : Non-linear pattern-matching against non-free data types. *ArXiv*, abs/1506.04498, 2015.

[Eke96] 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.

[EN18] Satoshi Egi and Yuichi Nishiwaki. Non-linear pattern matching with backtracking for non-free data types. *ArXiv*, abs/1808.10603, 2018.

[HHP87] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40 :143–184, 1987.

[How69] William A. Howard. The formulae-as-types notion of construction. 1969.

[LDF+13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Jonathan Protzenko, and Jérôme Vouillon. The ocaml system release 4.06 : Documentation and user's manual. 2013.

[LR18] Rodolphe Lepigre and Christophe Raffalli. Abstract representation of binders in ocaml using the bindlib library. 2018.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. 2008.

[Mid19] Aart Middeldorp. *Term Rewriting.* 2019.

[Mil90] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming. Springer Lecture Notes in Artificial Intelligence*, 1990.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PN94] Lawrence C. Paulson and Tobias Nipkow. Isabelle a generic theorem prover. 1994.

[Ros11] 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.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3) :222–262, 1908.

[Sai15] Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice*. Theses, Ecole Nationale Supérieure des Mines de Paris, September 2015.

[Sch09] Carsten Schürmann. The twelf proof assistant. In *TPHOLs*, 2009.

Index

A. OCaml guide

We provide here a short guide on how to read OCaml code.

A.1. Types

OCaml is a strongly typed language, that is, each expression is typed at compile time. There are basic types such as

- `bool` booleans
- `string`
- `int` integers
- `float`
- `unit`

and type constructor which can be used to build other types, such as `list`, `array`, &c. Type constructors are usually used in postfix notation, meaning that the type of a list of integers is noted `int list`. Such type constructors are said *polymorphic*, meaning that they can take any type as argument. Arguments of type constructors are usually noted `'a`, `'b` &c. and read as α, β &c.

We can now mention the essential type constructors,

- `'a -> 'b` type of functions from `'a` to `'b`;
- `'a * 'b` Cartesian product type (also known as tuple);
- `'a list` type of lists.

We give some examples of types,

```
(bool * bool) list
(** List of tuples of booleans. *)

'a -> 'a
(** Identity function. *)

('a * 'b) list -> 'a -> 'b
(** Function searching for a key of type 'a and returning the
    corresponding element of type 'b. *)
```

A.2. Expressions

Given an expression `e`, the notation `e:t` means that `e` is of type `t`. This is called an *ascription*.

A.2.1. Basic values

- Booleans : `true` and `false`.
- Integers : `0`, `1`, …
- Lists : the empty list `[]`; a list with some elements is noted `[a; b; c]`. The infix cons operator `::` is of type `'a -> 'a list -> 'a list` and puts its left argument on top of its right one. Therefore, `[0; 1; 2] = 0 :: [1; 2] = 0 :: (1 :: (2 :: []))`.

- Tuples : the expression `(a, b, c)` returns a tuple of three elements of type `'a * 'b * 'c` given that `a : 'a`, `b : 'b` and `c : 'c`.
- Unit : `()`.

The `unit` type has only one value and is usually used as the `void` type in C.

A.2.2. Compound expressions

More complex expressions can be built by applying functions. The function application is noted as in the λ calculus, juxtaposing terms. If `f : 'a -> 'b` and `x : 'a`, then `f x : 'b`. The application is left associative, that is, `f x y z = (((f x) y) z)`.

To bind expressions into other expressions, one can use `let ... in` constructs. For instance `let x = [0; 1] in 2 :: x` yields an expression of type `int list` which equals `[2; 0; 1]` as `x` stands for `[0; 1]` in the expression `2 :: x`.

A.2.3. Function declaration

`fun x y -> x + y` creates a function that sums its two arguments. To name this function, one can therefore use a `let in` construction,

```
let plus = fun x y -> x + y in ...
```

A.3. User defined types

A.3.1. Union types

One can define its own types using the `type` keyword. To define a union type,

```
type int_or_bool = Int of int | Bool of bool
```

which define a type containing either booleans or integers. `Bool` and `Int` are constructors. A value of type `int_or_bool` can be created using `Int(x)` given that `x : int` or `Bool(b)` with `b : bool`.

A.3.2. Records

A record is equivalent to a Cartesian product with labels. The following example shows how to use a record.

```
type complex = { real : float
               ; imaginary : float }
(** The null complex. *)
let zero_c : complex = { real = 0.0 ; imaginary = 0.0 }

(** [sq_module z] computes the square of the module of [z]. *)
let sq_module : complex -> float = fun z ->
  z.real * z.real + z.imaginary * z.imaginary
```

A.3.3. Pattern matching

Given a compound term, pattern matching allows to bind efficiently subterms. Pattern matching can be performed on function definition, in a `let` construction or with the special keyword `match t with`. As the name suggests, the term is matched against a pattern composed of constructors and variables. For instance, given a list `[1; 2]`, to retrieve the values 1 and 2, one can use the pattern `[x; y]` to have `x = 1` and `y = 2`. The pattern matching is done that way :

```
let [x; y] = [1; 2] in ...
match [1; 2] with [x; y] -> ...
```

The `match ... with` allows to cover several cases. Typically, if `xs` is a list, it can be either empty or of the form `x :: xs`. To cover both cases, one can use

```
match xs with
| []      -> e
| x :: tl -> e'
```

A.4. Modules

Code can be segmented into modules. Given a module `A`, a function `f` defined into it is accessed using `A.f`.

B. Weak head normalisation

We present here the code that transforms a term into its whnf. Let's define the terms,

```
type term =
  Vari of term Bindlib.var
  (** Variable *)
  | Abst of (term, term) Bindlib.mbinder
  (** Abstraction as a [Bindlib] binder *)
  | Symb of sym
  (** A user defined symbol ([sym] is a record) *)
  | Appl of term * term
  (** Application *)
```

we omit the definition of `sym`. All we have to know is that it is a record with a field `sym_tree` containing the decision tree compiled from the rewriting rules which have this symbol as root symbol. The application is binary, thus, `f` $X\ Y\ Z$ is encoded as `Appl(Appl(Appl(f, X), Y), Z)`.

We define an auxiliary function,

```
let rec add_args : term -> term list -> term = fun h s ->
  match s with
  | head :: tail -> add_args (Appl(h, head)) tail
  | []           -> h
```

which builds a term as defined above from a symbol and a stack of arguments.

Let `tree_walk : dtree -> term list -> term * stack` be the function such that `tree_walk tree stk` matches stack `stk` against tree `tree` (using the inference rules in Figure 2.3).

The code of the function defined in Equation 3.1 page 43 is then

```
let rec whnf : term -> term = fun t ->
  let (head, stk) = whnf_stk t [] in
  add_args head stk
and whnf_stk : term -> term list -> term * term list = fun t stk ->
  match t, stk with
  (* Push argument to the stack. *)
  | (Appl(f,u), stk   ) ->
  whnf_stk f (u::stk)
  (* Beta reduction. *)
  | (Abst(f)  , u::stk) ->
      let t = Bindlib.subst f u in
      whnf_stk t stk
  (* Try to rewrite. *)
  | (Symb(s)  , stk   ) ->
      begin match tree_walk s.sym_tree stk with
      (* If no rule is found, return the original term *)
      | None        -> t, stk
      | Some(t,stk) -> whnf_stk t stk
      end
  (* In head normal form. *)
  | (_         , _    ) -> t, stk
```