Regular type inference into the OCaml Interpreter - M1 Project Report

Bastien ROUSSEAU (ENS Rennes)
supervised by Thomas GENET (CELTIQUE - IRISA Rennes)

1 Introduction

1.1 The project

This project builds upon Mathieu Poirier's master project. It consists of carrying out the integration of Timbuk - a lightweight formal verification tool for functional program - into the OCaml interpreter.

The purpose of Timbuk is to allow non-experts developers to have a formal verification tool. Indeed, most of the existing tools for the formal verification like proof assistants (as Coq or Isabelle/HOL) or annotation systems (as "Liquid Types") needs a high level of expertise. Thus, the objective is to equip the OCaml interpreter with a verification mechanism as automatic as testing device, but with formal guarantees.

The first part of my work during the project was to dive into the understanding of Timbuk, the different parts of his inputs. The aim was at have an overview of the tool. The second part of my work was to focus on a particular issue of the link between the OCaml interpreter and Timbuk: the transformation of a rewriting system with priority order into a rewriting system without priority order. This transformation is called disambiguation. To fulfill this second task, I read papers for two different methods ([Kra08; CM19]) and implemented an OCaml prototype for both of them. Finally, I implemented the best method into the OCaml interpreter, in the continuation of Mathieu Poirier's project.

1.2 Contents

In section [2], I will present an overview of Timbuk and develop the link between Timbuk and the OCaml interpreter. I will recall in more details some important definitions over the term rewriting system which is the input of Timbuk. In section [3], I will give an intuition of the Timbuk's algorithm used to verify properties. Finally, in section [4], I will focus on the disambiguation problem. I'll compare the two methods I studied and give more details on the one we decided to implement.

2 Timbuk4

Timbuk4 [HGJ20] is a formal verification tool based on term rewriting systems and tree automata. It allows to prove properties on functional program with high-order functions. In this section, we will focus on the following example:

Example 1. We define the type *ab* with `type ab = A | B`. We want to prove that the function *remove e l* delete all the occurrences of the element *e* of type *ab* in the list *l*. *l* is a list of type *ab list*. Thus, the property we want to prove is:

```
forall (e:ab) (l:ab list). not (member e (remove e l))
```

2.1 From the interpreter to Timbuk

In the OCaml's interpreter, the user defines types and functions. He can also define properties working on types and functions he has previously defined. All the expressions evaluated by the interpreter are intercepted and transformed to perform the verification with Timbuk.

Timbuk input's are:

1. A term rewriting system (TRS)
2. A type automaton
3. A property to prove (Pattern)

Figure 1 shows how the OCaml interpreter interacts with Timbuk.

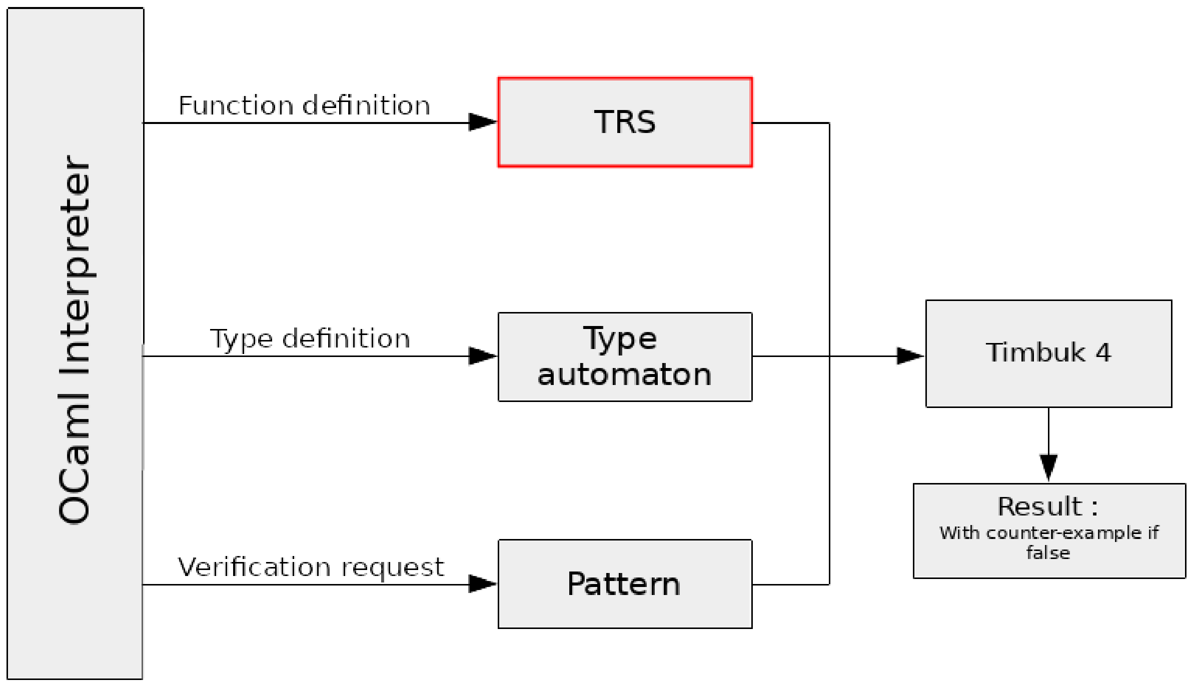


Fig. 1. Timbuk Inputs.

Remarks *In the following report, we will mainly focus on the TRS. We will not detail the type automaton, which is the data structure used to represent types.*

Since Timbuk works with purely functional program, we firstly detail the language and the features we consider for the project.

2.2 VerySimpleOCaml

`VerySimpleOCaml` is the subset of OCaml considered for the project. `VerySimpleOCaml` is inspired by $OCaml_{LIGHT}$ [Owe], a purely functional part of OCaml. So that, it excludes module and objects. Moreover, it is even more restrictive than $OCaml_{LIGHT}$. Informally, we can describe the supported features of `VerySimpleOCaml` with:

– definitions
  - variant data types (e.g., `type t = I of int | C of char`)
  - parametric type constructors (e.g., `type 'a t = C of 'a`)
  - recursive and mutually recursive combinations of the above
  - values

– lists, boolean, tuples

– control structures: $if \ldots then \ldots$ and $if \ldots then \ldots else \ldots$
– let-binding
– functions (recursive, mutually recursive, anonymous functions)
– pattern-matching (including nested pattern-matching)

Table 1 presents a short summary of the features supported or not by `VerySimpleOCaml`, in comparison with $OCaml_{LIGHT}$.

Present	Missing
Native types with constructors (list, bool)	Native types without constructors (int, floats, ...)
Tuples	Record types
Parametric and Recursive User defined types	Sequencing
Functions	Mutable references
Pattern-Matching	Exceptions
Polymorphic equality	$with$, $while$, for, $assert$, try, $raise$ expressions

Table 1. Functionalities of VerySimpleOCaml

2.3 TRS

TRS is the semantic representation of the program. Each OCaml function definition is translated into a set of rewriting rules. The union of all theses rules is called **term rewriting system** (TRS).

Definition 1. *Let $\mathcal{F}$ a finite alphabet and $\mathcal{X}$ a set finite set of variables. We note $\mathcal{T}$ ($\mathcal{F}$, $\mathcal{X}$) the set of terms.*

– *A **rewriting rule** is a pair $l \to r$ where $l, r \in \mathcal{T}$ ($\mathcal{F}$, $\mathcal{X}$).*
– *A **term rewriting system** $\langle \mathcal{F}, \mathcal{R} \rangle$ is a pair where $\mathcal{R}$ is a set of rewriting rule on $\mathcal{F}$.*
– *A **position** on a term t is a word on $\mathbb{N}$ which refers to a sub-term of t. The set of all positions of the term t, noted $Pos(t)$, is defined as follow:*

$$Pos(x) = \{\epsilon\},\ x \in \mathcal{X}$$

$$Pos(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \le i \le n \wedge p \in Pos(t_i)\},\ t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$$

– *For $p \in Pos(t)$, we denote $t|_p$ the sub-term of t at the position p and $t[s]_p$ the term t where the sub-term at the position p has been replaced by s.*
– *A **substitution** $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ is a map from variables to terms.*
– *Let $s, t \in \mathcal{T}$ ($\mathcal{F}$, $\mathcal{X}$. If it exists a rule $l \to r \in \mathcal{R}$, a position $p \in Pos(s)$ and a substitution σ such that $l\sigma = s|_p$ and $t = t[r\sigma]_p$, then we can **apply** the rule. We write $s \to_{\mathcal{R}} t$ this **rewriting relation** and $s \to^*_{\mathcal{R}} t$ the **reflexive-transitive closure** of the relation.*

Informally, if s matches with l for a certain position and a certain substitution, we can apply the rewriting rule $l \to r$ of the rewriting system $\mathcal{R}$, and obtain the resultant term t.

Example 2. Let $\mathcal{F} = \{remove, nil, cons, ite, A, B, true, false\}$ and $\mathcal{X} = \{E, L, X, Y\}$. $remove\ E\ nil \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is a term and $nil \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is also a term. So that, we can create the following rewriting rule: $remove(E, nil) \to nil$. Figure 2 encodes the $remove$ function in OCaml. Its corresponding TRS is described on the Figure 3

Let's apply some rewriting rules on the term $remove\ A\ cons(A,\ nil)$.

1. By the rule (2), we obtain the term $ite(eq(A, A),\ remove(A, nil),\ ns(A,\ remove(A, nil)))$.
2. By the rule (5), the next term is $ite(true,\ remove(A, nil),\ cons(A,\ remove(A, nil)))$.
3. By the rule (3), the next term is $remove(A, nil)$.
4. By the rule (1), the final term is nil.

We will now focus on the third input of OCaml, the pattern.

```
let rec remove e l =
  match l with
  |  [] -> []
  | h::t -> if (x=e)
            then (remove e r)
            else x::(remove e r)
```

Fig. 2. Function *remove* with OCaml

$$remove(E, nil) \to nil \tag{1}$$
$$remove(E, cons(X, L)) \to ite(eq(X, E), remove(E, L), cons(X, remove(E, L))) \tag{2}$$
$$ite(true, X, Y) \to X \tag{3}$$
$$ite(false, X, Y) \to Y \tag{4}$$
$$eq(A, A) \to true \tag{5}$$
$$eq(B, B) \to true \tag{6}$$
$$eq(A, B) \to false \tag{7}$$
$$eq(B, A) \to false \tag{8}$$

Fig. 3. Term rewriting system encoding the *remove* function previously defined

2.4 Pattern

To prove a property, we need to define it. Actually, the property is a term, called **pattern**. *Proving a property* means to type correctly the pattern.

Example 3. Suppose that we already have the predicate `(member e l)` which says if `e` belongs to list `l`.
Considers the following verification request:

```
(forall (e:ab) (l:ab list). not(member e (remove e l)))
```

Timbuk will try to prove that the term

```
forall (e:ab) (l:ab list). not(member e (remove e l))
```

can only be typed with *true*. In other words, forall *ab list l*, if we remove the occurrences of an element *e* with the function $remove$, then the element e no longer belongs to the list l. This is the expected behavior for the function $remove$.

Since we have defined the input of Timbuk, it is now possible to discuss the algorithm.

3 Timbuk - Type Inference Algorithm

In this section, I give an intuition of the algorithm used by Timbuk4 [HGJ20] to type a pattern.

3.1 Abstract interpretation

Proving a property about a pattern is like computing the set of reachable terms w.r.t the given TRS $\mathcal{R}$. However, this set is generally infinite. Thus, it is not computable. To tackle the problem, Timbuk uses an abstract interpretation such that

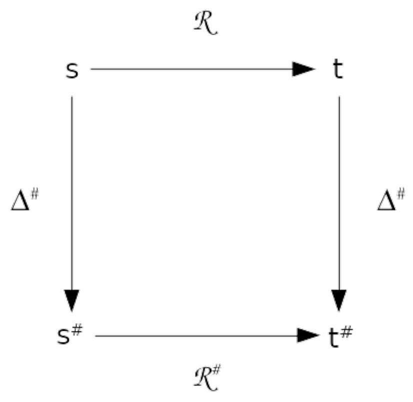
– the concrete domain is the set of terms

- the abstract domain $\Sigma^{\#}$ is the set of regular language on the concrete terms
- the abstraction function $\Delta^{\#}$ is a TRS which defines how the concrete terms are rewritten into abstract terms
- the abstract semantic $\mathcal{R}^{\#}$ is a TRS on the abstract terms. $\mathcal{R}^{\#}$ is extracted from $\mathcal{R}$.

Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s^{\#}, t^{\#} \in \Sigma^{\#}$. Suppose that $s \rightarrow_{\Delta^{\#}} s^{\#}$. The abstract semantic $\mathcal{R}^{\#}$ must respect the following property:

$$s \rightarrow_{\mathcal{R}} t \Rightarrow s^{\#} \rightarrow_{\mathcal{R}^{\#} \cup \Delta^{\#}} t^{\#}$$

In other words, if the concrete semantic $\mathcal{R}$ rewrites the terms s into t and if $s^{\#}$ is the abstract term of s, then we can rewrite $s^{\#}$ into $t^{\#}$ (which is the abstract term of t) w.r.t the abstract semantic $\mathcal{R}^{\#}$. Figure 4 summarizes the property.



$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"] \arrow[d, "\Delta^{\#}"'] & t \arrow[d, "\Delta^{\#}"] \\ s^{\#} \arrow[r, "\mathcal{R}^{\#}"'] & t^{\#} \end{tikzcd}$$

Fig. 4. Abstract interpretation of rewriting sequences.

As said previously, the abstract terms are regular languages. In fact, if $s \rightarrow_{\Delta^{\#}} s^{\#}$, we can say that $s^{\#}$ is a regular language which contains s. An example of $\Delta^{\#}$ and $s^{\#}$ is given below.

Since Timbuk is a fully automated verification system, $\Sigma^{\#}$, $\Delta^{\#}$ and $\mathcal{R}^{\#}$ are not known and need to be inferred. To do that, Timbuk uses a type system with type annotation. Let $s^{\#}$ be the abstract term of s again. Let X be a variable in a pattern. We note $X : s^{\#}$ if X is one of the concrete term of $s^{\#}$. Finally, to determine which $s^{\#}$ is the correct abstraction for X, Timbuk will infer, modularly (ie. function by function), $\mathcal{R}^{\#}$ and $\Delta^{\#}$ such that they respect the previous property. In reality, types are represented by tree automaton, and the inference algorithm is based on tree automata completion algorithm ([GHJ18; HGJ20]).

To give a better intuition of the inference procedure, we detail an example of the Timbuk output.

3.2 Example with Timbuk4

Timbuk uses term rewriting system and type automaton to types the given pattern.

Example 4. Take back the example of the function *remove* on *ab list*. We want to prove that the following pattern can only be typed with the language `#true` which only contains $true$:

```
not (member A(remove A l))
```

Figure 5 is the current output given by Timbuk4 for the pattern (for the sake of readability, some rules were removed and variables were renamed).

Let's explain how we can interpret the output, and how Timbuk constructs $\Delta^{\#}$ and $\mathcal{R}^{\#}$:

Value abstraction $\Delta^{\#}$:
```
true → #true
false → #false
A → #a
B → #b
nil → #b_list
cons(#b, #b_list) → #b_list
cons(#b, #ab_list) → #ab_list
cons(#a, #ab_list) → #ab_list
cons(#a, #b_list) → #ab_list
```

TRS abstraction $\mathcal{R}^{\#}$:
```
not(#false) → #true
member(#a, #b_list) → #false
remove(#a, #ab_list) → #b_list
ite(#true, #b_list, #ab_list) → #b_list
ite(#true, #b_list, #b_list) → #b_list
ite(#false, #ab_list, #b_list) → #b_list
ite(#false, #b_list, #b_list) → #b_list

eq(#a, #b) → #false
```

Typed patterns:
```
not(member(A:#a, remove(A:#a, '2:#ab_list):#b_list):#false):#true
```

Fig. 5. Output of Timbuk4

1. Timbuk wants to separate possible types for a certain variable `X` in the pattern `not(X)`. Here, the possible types for `X` are only *#true* and *#false*. Since the goal is to prove that `not(X):#true`, Timbuk creates a new abstract type *#false* (to understand, the regular language which describes the concrete values $\{false\}$). Timbuk discovers that *#true* is the only possible type to prove the goal.
2. Thus, Timbuk infers that `member A (remove A l)` can only be type with *#false*. We can write `member A (remove A l) : #false`
3. Now, Timbuk tries to separate possible types for the term `member A Y`. Timbuk creates a new type $\#a$ for A. Then, Timbuk creates a new abstract type *#b_list* whose language is the set of ab_list with no A. Then, if `Y : #b_list`, we are sure that `(member A:#a Y:#b_list)` has type $\#false$.
4. Thus, Timbuk infers that the term `remove A l` must be typed with $\#false$
5. Timbuk iterate this procedure on each terms, and so on ...

4 Disambiguation

We have discussed the formal verification tool Timbuk. We will now focus on an issue about the link between the OCaml interpreter and Timbuk, more precisely with the TRS.

4.1 Motivation

Generally, The OCaml pattern-matching transformation into its corresponding term rewriting system cannot be direct. In a TRS, it is possible to apply any rule since the term matches with the left-side of the rule. In the OCaml pattern matching, only one rule is applied: the first rule of the list which matches with the term. The following example illustrates the issue.

Example 5. We can assume that we have a `type nat = Z | S of nat`. We can also assume that we have a function `is_zero` defined by OCaml pattern-matching as follow:

```
let rec is_zero n =
 match n with
 | Z -> true
 | _ -> false
```

Fig. 6. Function *is_zero* with OCaml pattern-matching

If we naively transform this pattern-matching into a term rewriting system, the resultant TRS is:

$$is\_zero(Z) \to true \qquad \text{(zero)}$$
$$is\_zero(n) \to false \qquad \text{(all\_nat)}$$

Fig. 7. TRS corresponding to the naive transformation of the function *is_zero*

However, in the TRS of the Figure 7, the term $is\_zero(Z)$ can be either rewritten in

– $true$, by application of the rule (zero)
– $false$, by application of the rule (all_nat)

Thus, we want to transform the pattern-matching into a term rewriting system in which each term has a **unique** applicable rule. Using the same pattern-matching definition of `is_zero`, the term rewriting system we want to generate is the following:

$$is\_zero\ Z \to true \qquad \text{(zero)}$$
$$is\_zero\ (S\ n) \to false \qquad \text{(succ)}$$

Fig. 8. TRS corresponding to a non-ambiguous transformation of the function *is_zero*

With the new TRS of the Figure 8, it exists only one valid rule to rewrite the term $is\_zero(Z)$, the rule (zero).

To transform the naive TRS into a disambiguated TRS, we studied two different methods that we compare in the next section.

4.2 Comparison

The two studied method are:

1. Pattern Minimization Over Recursive Data Types, proposed by Krauss [Kra08]
2. Generic Encoding of Constructor Rewriting Systems, proposed by Cirstea and Moreau [CM19]

These two methods tackles the problem with a different approach. With the aim of highlighting the differences between the approaches, we first explain the concepts of the transformation for both methods. Then, we perform the calculation. Finally, we explain which method we selected for implementation, and why.

Overviews

The first method is proposed by *Krauss* in [Kra08]. The algorithm works only on a set of patterns we want to disambiguate. The input of the algorithm is the set of the left-hand side of the rewriting rules. The first step consists of computing a set of patterns called **minterms**. The set of minterms is a set of patterns which partitionates the set of all reachable terms w.r.t the initial set of patterns. According to this procedure, minterms do not overlaps. Actually, minterms solve the disambiguation problem, but the computed set can be very large. Thus, there is a second step to minimize this set. The second step consists of generalizing minterms as much as possible while preserving the partition of the initial set of patterns.This set is called the **essential prime implicants**.

The second method is proposed by *Cirstea,Moreau* in [CM19]. It consists of a transformation of a list of rewriting rules, with priority order, into a set of rewriting rules, without priority order. For this purpose, the list of rewriting rules, using normal patterns, is transformed into a set of rewriting rules, using extended patterns. Extended patterns include addition of patterns and complement of patterns. We need to introduces **pure pattern**, which are patterns with only variables, constructor patterns and additions. The method consists of a transformation of **complementary patterns** into a **pure pattern**, thanks to a dedicated TRS.

Calculation To compare the methods, we will do the calculations for the previous example is_zero. For the sake of clarity, detailed calculations are available in appendix.

Krauss

Example 6. We follow the algorithm presented by *Krauss* ([Kra08] Section 4.4).

1. Since the method works with patterns, we keep only the left-hand side of the rules, which are $is\_zero(Z)$ and $is\_zero(n)$. Because is_zero is the function, this is a common head term for each pattern. Hence, we can consider that the set of pattern to disambiguate is $\{Z, *\}$.
2. Compute positive and negative minterms:
   $M_P^+ = \{Z, S(*)\}$ and $M_P^- = \{\}$.
   M_P^+ represents a disambiguate partition of ground terms with the same semantic of the given input. M_P^- represents a disambiguate partition of ground terms which not matches with a pattern of the given input. [1] For this particular example, the set of positive minterms is already minimal. But, in general, the generated set is huge and we need to minimize it.
3. Compute the set of the essential prime implicants, which is $E = \{Z\}$.
4. The third step consists of computing the prime implicants and finding a covering of $\overline{M}$ with them. This set is $N' = \{S(*)\}$.
5. Finally, the output of the algorithm is $E \cup N' = \{Z, S(*)\}$.

The details of the calculations are available in appendix A.

We may notice that only the left-hand side is treated with this method. Therefore, the right-hand side is not treated. In particular, modifying variables in the left-hand side is needed to find the correct substitution for the right-hand side. In this example, the original rule $is\_zero(n) \to false$ may be transform into $is\_zero(S(n_1)) \to false$. Thus, we need to find the substitution σ such that $\sigma(n) = S(n_1)$, where n_1 is a fresh variable.

Cirstea,Moreau

Example 7. We follow the algorithm presented by *Cirstea,Moreau* ([CM19] Section 5.3).

1. We have the following list of rewriting rules with priority $\mathcal{L} = [Z \to true,\ x \to false]$. We transform this list into a set of extended rules without priority: $\{Z \to true,\ x \setminus Z \to false\}$ where $x \setminus Z$ denotes the pattern which recognizes all terms except Z.

[1] In practice, since the pattern-matching is complete, the set of negative minterms is always void.

2. Thanks to a specific TRS, we compute the normal form of the left-hand side for each rule. So that, we have $\mathcal{I}^{<}(\mathcal{L}) = \{Z \to true,\ x@S(z_1) \to false\}$. This is a set of **pure pattern**. They are disambiguate rules, but it remains aliased pattern.
3. Finally, we remove the aliased pattern and get the output of the algorithm:

$$\{Z \to true,\ S(z_1) \to false\}$$

   This final step manages the substitution of variables.

A more detailled calculation is available in appendix B.

We may notice that the method manages the left-hand side and the right-hand side of the rewriting rules.

Outcome Now we have seen example about the two methods, we can compare them.

Some crucial steps of *Krauss'* algorithm (pattern minimization) are based on the complex Quine-McCluskey algorithm. As a result, *Krauss'* algorithm technique is not intuitive and seems difficult to implement efficiently. On the contrary, the second method uses TRS to transform extended patterns into additive patterns. Each step is very intuitive.

Moreover, where *Krauss'* algorithm focuses only on the left-hand side, *Cirstea,Moreau* transforms a list of left and right-hand sides of rewriting rules (with priority order) into a disambiguate and minimal set of patterns (without priority order). The latter technique also handles right-hand sides and we do not have to add a new step to substitute variables. Furthermore, after implementation and handmade verification, we found an example which doesn't work with *Krauss'* algorithm (appendix C).

Thus, we decided to implement *Cirstea,Moreau*. We give more details on this method.

4.3 Generic Encoding of Constructor Rewriting Systems

Definitions and notations Firstly, we need to give some important notations and definitions.

Definition 2. *Considers $\mathcal{C}$ a set of constructor symbols and $\mathcal{X}$ a set of variable symbols.*

- *a **constructor pattern** $p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ is a linear term, a term where every variable occurs at most one time.*
- *a **value** $v \in \mathcal{T}(\mathcal{C})$ is a ground term, a term without any variable*

Extended patterns An **extended pattern** p is defined as follow:

$$p := \mathcal{X} \mid f(p_1, \ldots, p_n) \mid p_1 + p_2 \mid p_1 \setminus p_2 \mid q@p \mid \bot$$

with $f \in \mathcal{C}$, $q \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

We call $p = p_1 \setminus p_2$ a **complement pattern**. We call $p = p_1 + p_2$ an **addition pattern**. We call $p = p_1@p_2$ an **aliased pattern**. We call p an **additive pattern**, an extended pattern, which contains no \. We call p a **pure pattern**, an extended pattern, which contains no $\bot$.

Definition 3. *We define the **ground semantic** of patterns as follows:*

- $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}(\mathcal{C})\},\ p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$
- $[\![x]\!] = \mathcal{T}(\mathcal{C}),\ x \in \mathcal{X}$
- $[\![p_1 + p_2]\!] = [\![p_1]\!] \cup [\![p_2]\!]$
- $[\![p_1 \setminus p_2]\!] = [\![p_1]\!] \setminus [\![p_2]\!]$
- $[\![q@p]\!] = [\![q]\!] \cap [\![p]\!],\ q \in \mathcal{T}(\mathcal{C}, \mathcal{X})$
- $[\![\bot]\!] = \emptyset$

Notice that, in practice, $q \in \mathcal{X}$. Thus, $[\![q@p]\!] = [\![p]\!]$

Definition 4. *Given a list of pattern $P = [p_1, \dots, p_n]$, we defined the* ***disambiguation problem*** *as finding sets of patterns $P_1, \dots, P_n$ such that, for each $i \in [1, \dots, n]$, $[\![P]\!] = \bigcup_{i=1}^{n} [\![p_i]\!] \setminus \cup_{j=1}^{i-1} [\![p_j]\!]$.*

In other words, for each pattern of the list, we want to find a pattern which recognizes the same terms without terms from the previous patterns of the list. As a consequence, a list of equations can be replaced by a semantically equivalent set of patterns, without priority order.

Definition 5. *Let p be a constructor pattern and v a value. We can define the* ***instance relation*** *between extended patterns, noted $p \preceq q$, if it exists a substitution σ such that $q = \sigma(p)$. We say that "q matches with p". Since p is linear, we can inductively define the* ***instance relation*** *as following:*

$$x \preceq v, \ x \in \mathcal{X}$$
$$c(p_1, \dots, p_n) \preceq c(v_1, \dots, v_n), \ \textit{iff} \ \wedge_{i=1}^{n} p_i \preceq v_i, c \in \mathcal{C}$$

Algorithm

Intuition Let $\mathcal{L}$ be an ordered list of rewriting rules. To transform $\mathcal{L}$ into an unordered set of rewriting rules, we apply the following transformation:

$$\mathcal{I} = \mathcal{I}^{@} \circ \mathcal{I}^{<}$$

where

- $\mathcal{I}^{<}$ encodes the priority order and return **pure additive** patterns, possibly with aliases
- $\mathcal{I}^{@}$ removes alias from pure additive patterns

In fact, we can decompose $\mathcal{I}^{<}$ in 2 steps:

1. For each rule $p_i \to q_i$ of the list $\mathcal{L}$, the left-hand side p_i is transformed into an extended pattern $p_i \setminus (p_1 + \dots + p_{i-1})$. So that, if a term matches with $p_1, \dots, p_{i-1}$, this term do not matches with $p_i \setminus (p_1 + \dots + p_{i-1})$. This step transforms the list of *normal* rewriting rules (with priority) into a set of *extended* rewriting rules (without priority).
2. For each left-hand side, the new extended pattern is transformed into a **pure additive** pattern, which is the normal form w.r.t the TRS $\mathcal{R}_{@}$ (defined in *Cirstea,Moreau* [CM19]).
   Finally, if $p_i \setminus (p_1 + \dots + p_{i-1}) \downarrow \mathcal{R}_{@}\, t_1 + \dots + t_n$ (ie. the normal form of $p_i \setminus (p_1 + \dots + p_{i-1})$), the rewriting rule $p_i \to q_i$ is transformed into the following rules $t_1 \to q_i, \dots, t_m \to q_i$.

It is important to notice that each rule of $R_{@}$ preserves the ground semantic of the pattern.

Focus on some rules The full TRS $\mathcal{R}_{@}$ is available in [CM19], Figure 3. Nevertheless, we can focus on some rules and explain them.

The first rule we can focus on is

$$\overline{V} \setminus g(t_1, \dots, t_n) \Rightarrow \overline{V} @ (\sum_{c \in \mathcal{C}} c(z_1, \dots, z_m) \setminus g(t_1, \dots, t_n)) \tag{M4'}$$

where $\overline{V}$ is a pattern variable, g is a constructor symbol, t_i are pure additive patterns, $m = arity(c)$ and z_i are fresh variables. This is the rule which complements a constructor pattern from a variable. Since the semantic of a variable is the set of all ground constructor patterns, it is possible to obtain the set with the union of all constructors, which is represented as the sum of theses patterns. The alias guarantees that the variables of a complement pattern are not lost in the transformation (and keep the trace of the transformation for the substitution step $\mathcal{I}^{@}$).

The second rule we can focus on is

$$g(v_1, \dots, v_n) \setminus g(t_1, \dots, t_n) \Rightarrow g(v_1 \setminus t_1, \dots, v_n) + \dots + g(v_1, \dots, v_n \setminus t_n) \tag{M7}$$

where v_i is an additive pattern. This is the rule which complements two patterns with the same constructor pattern in head. It corresponds to the set difference of cartesian products. The other rules are more intuitive. Thus, we can see some optimizations of this TRS, which are really important.

Optimizations They are two majors optimizations for this TRS. However, before explaining them, we need to define a new concept.

Definition 6. *We say that a pattern p* ***is subsumed by*** *a set of patterns $P = \{p_1, \ldots, p_n\}$ if $p \setminus (p_1 + \ldots + p_n) \downarrow \mathcal{R}_\setminus = \bot$.*

The optimizations are the following:

– **Cut Useless Choices:** for the rule (M7), $f(v_1, \ldots, v_n) \setminus f(t_1, \ldots, t_n)$, if it exists k such as $v_k \setminus t_k = v_k$, then, we know that the term $f(v_1, \ldots, v_n)$ subsumes all the other. Thus, we can reduce the rule to this term.
– **Sort Encoding:** for the rule (M4′), the rule may produce ill-formed terms. It is then possible to filter them and only keep well-formed terms.

Our results shows that theses optimizations are essentials. Without them, the normalization produced a very large set of rules. This huge set cannot be handled by the minimization step in reasonable time, which is the last step.

Minimization The idea of the minimization algorithm of a set of pattern P is the following. For each pattern of P:

– If the pattern is not subsumed by all the others, we need to keep it. The pattern is absolutely essential.
– If the pattern is subsumed by all the others, then we iterate the algorithm on the other pattern of P, with and without this pattern. We keep the set with minimal number of patterns.

4.4 Implementation

I implemented a prototype for both method:

– *Krauss* method: ~ 800 lines of OCaml code
– *Cirstea,Moreau* method: ~ 1000 lines of OCaml code

Moreover, the implementation into the interpreter needed some modifications: find the right place to integrate the disambiguation algorithm into the existing code, attach type information with constructor terms and compile normal patterns into extended patterns. It represents ~ 200 lines of code.

We tested the implementation with the same examples used in [Kra08] and [CM19] and had the same results for the number of patterns generated by the disambiguation:

– *Interp* produced 25 rules.
– *Red Black Tree Balance* produced 91 rules.
– *Numadd* produced 256 rules.

5 Conclusion

This project was a great opportunity to introduce the domain of the formal verification. I discovered the formalism of term rewriting system and learned about pattern-matching. Moreover, I could implement the disambiguation algorithm into the interpreter. Even if some features have not been implemented in this project, this is a big step toward a demo tool for Timbuk4 on `VerySimpleOCaml`.

References

[Kra08] Alexander Krauss. “Pattern minimization problems over recursive data types”. In: *ACM SIGPLAN Notices* 43.9 (Sept. 2008), pp. 267–274. DOI: `10.1145/1411203.1411242`. URL: `https://doi.org/10.1145/1411203.1411242`.

[GHJ18] Thomas Genet, Timothée Haudebourg, and Thomas Jensen. “Verifying Higher-Order Functions with Tree Automata”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 565–582. DOI: `10.1007/978-3-319-89366-2_31`. URL: `https://doi.org/10.1007/978-3-319-89366-2_31`.

[CM19] Horatiu Cirstea and Pierre-Etienne Moreau. “generic encodings of constructor rewriting systems”. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. ACM, Oct. 2019. DOI: `10.1145/3354166.3354173`. URL: `https://doi.org/10.1145/3354166.3354173`.

[HGJ20] Timothée Haudebourg, Thomas Genet, and Thomas Jensen. “Regular language type inference with term rewriting”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (Aug. 2020), pp. 1–29. DOI: `10.1145/3408994`. URL: `https://doi.org/10.1145/3408994`.

[Owe] Scott Owens. “A Sound Semantics for OCaml light”. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 1–15. DOI: `10.1007/978-3-540-78739-6_1`. URL: `https://doi.org/10.1007/978-3-540-78739-6_1`.

A Full calculation steps for *Krauss*

A.1 Minterms

We have the following set of patterns: $P = \{Z, *\}$.

$$MT(P) = \{Z\} \cup \{S(MT(\prod_{S,1}(P)))\}$$

The second set is computed thanks to:

$$\prod_{S,1}(P) = \{*\}$$

$$MT(\{*\}) = \{*\}$$

So that, we have:

$$MT(P) = \{Z,\ S(*)\}$$

Finally, we can partition $MT(P)$ as $M_P^+ = \{Z, S(*)\}$ and $M_P^- = \{\}$.

A.2 Prime implicants

For all $m, m' \in M_P^+$, compute $\lceil \sup(m, m') \rceil$.

$$\begin{aligned} \lceil \sup(Z, Z) \rceil &= Z \\ \lceil \sup(S(*), S(*)) \rceil &= S(*) \\ \lceil \sup(Z, S(*)) \rceil &= * \end{aligned}$$

We can filter them to keep only implicants. Then, we have $N = \{Z, S(*)\}$

A.3 Essential prime implicants

For all $m \in M_P^+$, compute $R_{M_P^+}(m)$.

Firstly,

$$R_{M_P^+}(Z) = \sup\{g \in G(Z) \,|[\,g] \subseteq [M_P^+]\}.$$

$$G(Z) = \{*, Z\}$$

Since $[*] \subsetneq [M_P^+]$,

$$R_{M_P^+}(Z) = \sup\{Z\} = Z$$

Secondly,

$$R_{M_P^+}(S(*)) = \sup\{g \in G(S(*)) \,|[\,g] \subseteq [M_P^+]\}.$$

$$G(S(*)) = \{*, S(G(*))\}$$

But, $G(*) = \{\}$. So that, $G(S(*)) = \{*\}$. Since $[*] \nsubseteq [M_P^+]$, there is no $R_{M_P^+}(S(*))$.
Finally, we can conclude that $E = \{Z\}$.

A.4 Conclusion

Finally, we easily compute $\overline{M} = \{S(*)\}$ and $N' = \{S(*)\}$. The disambiguate and minimized set of pattern is the following: $E \cup N' = \{Z, S(*)\}$.

B Full calculation steps for *Cirstea,Moreau*

Let's compute $x \setminus Z \downarrow \mathcal{R}_{\setminus}^{@}$.

By the rule (M4') $\rightarrow x@(Z \setminus Z + S(z_1) \setminus Z)$
By the rule (S2) $\rightarrow x@(Z \setminus Z) + x@(S(z_1) \setminus Z)$
By the rule (M7) $\rightarrow x@\bot + x@(S(z_1) \setminus Z)$
By the rule (M8) $\rightarrow x@\bot + x@S(z_1)$
By the rule (E2) $\rightarrow \bot + x@S(z_1)$
By the rule (A1) $\rightarrow x@S(z_1)$

Finally, we have $x \setminus Z \downarrow \mathcal{R}_{\setminus}^{@} = x@S(z_1)$.

C Wrong calculation for *Krauss*

For the example, we want to disambiguate $P = \{\langle Suc(0), *\rangle, \langle *, 0\rangle\}$.

C.1 Minterms

Let's compute the minterms of P.

$$MT(P) = \langle MT(\prod_{\langle\rangle,1}(P)),\ MT(\prod_{\langle\rangle,2}(P))\rangle$$
$$\prod_{\langle\rangle,2}(P) = \{*, 0\}$$
$$MT(\{*, 0\}) = \{0, Suc(*)\}$$
$$\prod_{\langle\rangle,1}(P) = \{Suc(0), *\}$$
$$MT(\{Suc(0), *\}) = \{0\} \cup Suc(MT(\prod_{Succ,1}(\{Suc(0), *\})))$$
$$MT(\{Suc(0), *\}) = \{0\} \cup Suc(MT(\{0, *\}))$$
$$MT(\{Suc(0), *\}) = \{0\} \cup Suc(\{0, Suc(*)\})$$
$$MT(\{Suc(0), *\}) = \{0, Suc(0), Suc(Suc(*))\}$$
$$MT(P) = \{\langle 0, 0\rangle, \langle 0, Suc(*)\rangle, \langle Suc(0), 0\rangle, \langle Suc(0), Suc(*)\rangle, \langle Suc(Suc(*)), 0\rangle, \langle Suc(Suc(*)), Suc(*)\rangle\}$$

We can partition $MT(P)$ as

- $M^+ = \{\langle 0, 0\rangle, \langle Suc(0), 0\rangle, \langle Suc(0), Suc(*)\rangle, \langle Suc(Suc(*)), 0\rangle\}$
- $M^- = \{\langle 0, Suc(*)\rangle, \langle Suc(Suc(*)), Suc(*)\rangle\}$

C.2 Prime implicants

Let's compute the prime implicants For all $m, m' \in M_P^+$, compute $\lceil \sup(m, m')\rceil$.
$M_P^+ = \{\langle 0, 0\rangle, \langle Suc(0), 0\rangle, \langle Suc(0), Suc(*)\rangle, \langle Suc(Suc(*)), 0\rangle\}$, we obtain:

- $\langle 0, 0\rangle$
- $\langle *, 0\rangle$
- $\langle *, *\rangle$
- $\langle Suc(0), 0\rangle$
- $\langle Suc(0), *\rangle$
- $\langle Suc(*), 0\rangle$
- $\langle Suc(0), Suc(*)\rangle$
- $\langle Suc(*), *\rangle$
- $\langle Suc(Suc(*)), 0\rangle$

Next, we filter them to keep those which are implicants, that is $\forall p \in M_P^-.\ p \wedge \lceil \sup(m, m')\rceil = \bot$, and those which are not instances of other patterns of the set. Thus, prime implicants are $N = \{\langle *, 0\rangle, \langle Suc(*), *\rangle\}$

C.3 Essential prime implicants

Let's compute the set of essential prime implicants. For each positive minterms $m \in M_P^+$, we compute its neighbourhood term $R_P(m)$. We recall that

- $P = \{\langle Suc(0), *\rangle, \langle *, 0\rangle\}$
- $M_P^+ = \{\langle 0, 0\rangle, \langle Suc(0), 0\rangle, \langle Suc(0), Suc(*)\rangle, \langle Suc(Suc(*)), 0\rangle\}$

1. $m = \langle 0, 0\rangle$

- Generalisation $G(m) = \{*, \langle *, 0\rangle, \langle 0, *\rangle\}$
- Filtering $\{g \in G(m) \mid [g] \subseteq [P]\} = \{\langle *, 0\rangle\}$
- Sup $R_P(m) = \langle *, 0\rangle$

2. $m = \langle Suc(0), 0\rangle$
   - Generalisation $G(m) = \{*, \langle *, 0\rangle, \langle Suc(*), 0\rangle, \langle Suc(0), *\rangle\}$
   - Filtering $\{g \in G(m) \mid [g] \subseteq [P]\} = \{\langle *, 0\rangle, \langle Suc(*), 0\rangle, \langle Suc(0), *\rangle\}$
   - Sup $R_P(m) = \langle *, *\rangle$
3. $m = \langle Suc(0), Suc(*)\rangle$
   - Generalisation $G(m) = \{*, \langle *, Suc(*)\rangle, \langle Suc(*), Suc(*)\rangle, \langle Suc(0), *\rangle\}$
   - Filtering $\{g \in G(m) \mid [g] \subseteq [P]\} = \{\langle Suc(0), *\rangle\}$
   - Sup $R_P(m) = \langle Suc(0), *\rangle$
4. $m = \langle Suc(Suc(*)), 0\rangle$
   - Generalisation $G(m) = \{*, \langle *, 0\rangle, \langle Suc(*), 0\rangle, \langle Suc(Suc(*)), *\rangle\}$
   - Filtering $\{g \in G(m) \mid [g] \subseteq [P]\} = \{\langle *, 0\rangle, \langle Suc(*), 0\rangle\}$
   - Sup $R_P(m) = \langle *, 0\rangle$

After filtering the $R_P(m)$ which are not implicants, set of essentials prime implicants is the following : $E = \{\langle *, 0\rangle, \langle Suc(0), *\rangle\}$

C.4 Conclusion

Finally, $E \cup N' = \{\langle *, 0\rangle, \langle Suc(0), *\rangle\}$ This is the initial set of patterns P, which is ambiguous because $\langle Suc(0), 0\rangle$ is an instance of both $\langle *, 0\rangle$ and $\langle Suc(0), *\rangle$.