Master research Internship

Bibliographic report

Towards a Lightweight Formal Verification for OCaml$_{\text{light}}$

Programming Languages

Author:
Timothée Haudebourg

Supervisor:
Thomas Genet
Celtique

Abstract: From a Term Rewriting System (TRS) representing a functional program logic and a language describing the set of possible inputs, it is possible to compute an over approximation of the set of possible outputs. We can use this approximation to verify some regular properties on the considered program. We call this technique "Lightweight Formal Verification" because the user does not need either to annotate the program or to write complicated proof to verify simple properties on the program. Our objective is to use this technique to build an abstract interpreter for $\text{OCaml}_{\text{light}}$, a subset of OCaml, that could be used to interactively verify $\text{OCaml}_{\text{light}}$ programs. This requires finding a way to translate any $\text{OCaml}_{\text{light}}$ program into a TRS, and adapt the theory behind LFV to handle higher-order functional programs.

Contents

1 Introduction **1**

2 Related Work **2**
2.1 Refinement Types . . . 3
2.2 Pattern-Matching Recursion Schemes . . . 4

3 Lightweight Formal Verification **5**
3.1 Preliminaries . . . 5
3.1.1 Terms . . . 6
3.1.2 Term Rewriting systems . . . 7
3.1.3 Tree automata . . . 8
3.2 Tree Automaton Completion . . . 9
3.2.1 Problem formalization . . . 9
3.2.2 Completion Algorithm . . . 9
3.2.3 Termination Criteria . . . 11
3.3 Extracting the Output Terms . . . 12

4 Future Contribution **13**
4.1 Application to $\text{OCaml}_{\text{light}}$ . . . 13
4.2 Handling higher-order in a TRS . . . 13

5 Conclusion **14**

1 Introduction

Static analysis consists in analyzing the text of a program, looking for potential bugs or malfunctions, *before* it's execution. Typing is already a way to verify that all operations performed in the program are legal. However, even if in most cases it ensures that the program will not crash, *good typing* rarely means *good result*. A well-typed program can still be not correct.

Verifying the correctness of a program (ensuring that it computes what it was designed for) is a harder problem. It requires the designer to reason on the program semantics. The best way to do it is to *prove* the program correct using a proof assistant such as [Coq, 2016] for instance. However proving a program can be hard and time consuming, especially if the program is *not* correct. It also requires a high level of expertise to define the property of correctness, and to achieve the proof. We are interested here in *the verification of functional programs* with an alternative technique, stronger that simple type checking, but lighter than a formal proof. Some techniques have already been devlopped for this purpose, from which we may cite Refinement Types [Vazou et al., 2013, Vazou et al., 2015] that uses a richer definition of types to check a program, or Pattern-Matching Recursion Schemes [Ong and Ramsay, 2011] used to model-check functional programs. We are particularly interested in a new technique called *Lightweight Formal Verification* (LFV). Within it, program inputs and values are modeled with a finer representation than simple types, using *Tree Automata* [Comon et al., 2008]. It then uses the program logic, represented as a Term Rewriting System [Baader and Nipkow, 1998] (TRS), to approximate the set of output values of the program [Genet, 2016]. This output can then be used to check some regular properties on the program, as a first step to prove it's correctness.

Our goal is to take advantage of the LFV technique to design an abstract interpreter for $\text{OCaml}_{\text{light}}$ [Genet et al., 2015]. The idea is to use this abstract interpreter to interactively verify the correctness of $\text{OCaml}_{\text{light}}$ programs. In this interpreter, the user would write the program using the $\text{OCaml}_{\text{light}}$ syntax, and call the defined functions not with a single parameter, but with a term representing all the possible values of this parameter.

Example 1. *Let's imagine that the user defines in the interpreter the following (wrong) recursive function, trying to remove every input element x from a list:*

```
let rec delete x = function
        | [] -> []
        | y::l -> if x = y then l else y::(delete x l);
```

We can try this function on the OCaml interpreter and get:

```
# delete 1 [0; 1; 2];;
- : int list = [0; 2]
```

At a first glance, the function seems to works fine. Now let's note by the a regular expression `[(a|b)*]` *all the lists of an arbitrary number of a and b in any order. Our goal is to have an abstract interpreter where we can try:*

```
# delete b [(a|b)*];;
- : abst list = [(a|b)*]
```

...and see that the result is not what we would expect. Because this function is supposed to remove all occurrences of `b`*, we were excepting the result to be* `[a*]`*, all the lists of* `a`*s.*

In this previous example, because the lightweight verification only gives us a over approximation of the outputs, one could say that it is not a proof that the function is wrong, however the LFV

technique gives us some guarantees over the approximation [Genet, 2016]. It at least forces the user to revise the implementation.

Example 2. *In the previous example, the recursive call to* `delete` *is missing in the* **`then`** *branch. Here is the corrected version of the function:*

```
let rec delete x = function
        | [] -> []
        | y::l -> if x = y then (delete x l) else y::(delete x l);
```

In that case, we expect the abstract interpreter to return `[a*]`*.*

```
# delete b [(a|b)*];;
- : abst list = [a*]
```

Because the abstract interpreter gives an over approximation, we now know for sure that all `b` *values are removed from the list.*

Event if the Lightweight Formal Verification technique is already functional, building an abstract interpreter for $\text{OCaml}_{\text{light}}$ based on this technique requires some additional work. Our contribution will be divided into two main parts:

- First, LFV needs the program to be represented as a Term Rewriting System. Our first work will be to define the semantics of $\text{OCaml}_{\text{light}}$ as a TRS. To do that, we will base ourselves on a precisely defined semantics of $\text{OCaml}_{\text{light}}$ proposed by [Owens, 2008].
- Second, for now LFV has been designed to handle first-order functional programs. However $\text{OCaml}_{\text{light}}$ supports higher order program definition. For our abstract interpreter to work with the full semantics of $\text{OCaml}_{\text{light}}$, we will need to adapt the theory behind LFV to handle higher order functional programs. Once again this will be achieved based on existing work [Jones and Muchnick, 1979, Ong and Ramsay, 2011, Vazou et al., 2015].

The rest of this paper aims to give a general view of the state of the art in the domain, and an idea of the work remaining to achieve this two objectives. It contains the following: An introduction to the work related to functional programs verification (Section 2). A more detail description of the Lightweight Formal Verification technique we will be using in our abstract interpreter (Section 3). This description includes the required definitions to understand how the program logic and the program inputs will be modeled in our interpreter. Finally, we will give a more detailed view of the remaining work, and some idea proposed to achieve it (Section 4). Those are the basics on which we will build our abstract interpreter.

2 Related Work

We focuses here on two main verification techniques for higher-order functional programs. Both could later be an alternative to the LFV technique, or at least help us adapt the theory behind LFV. The first one is a type-based program analysis using refinement types [Vazou et al., 2015] to prove properties on higher-order programs. The second one is able to over-approximate the set of outputs of a higher-order program using Pattern-Matching Recursion Schemes [Ong and Ramsay, 2011].

2.1 Refinement Types

Refinement Types have been introduced by [Vazou et al., 2013] and extended in [Vazou et al., 2015] to perform a type-based analysis of higher-order programs. We said earlier that simple types are not enough check the correctness of a program. Their idea is to use a richer notion of types in the definition of functions (for instance $\{x : int \mid x > 0\}$ the set of positive integers). A SMT solver is then used to check the validity of the type system.

Example 3. *Let define the following function* `filter`*:*

```
let rec filter p = function
        | x::l -> if (p x) then x::(filter p l) else (filter p l)
        | [] -> [];;
```

This function uses a predicate p, given as a parameter, to remove from the input list the elements that violate this predicate. We want to verify that this function respects the property:

$$\forall p, \forall l, \forall x, \quad x \in (filter\ p\ l) \Rightarrow (p\ x)$$

In other words, all non-filtered elements respects the property p. Let's see how to prove this property by giving the appropriate signature to `filter` *using refinement types. One may try:*

```
filter : (int -> bool) -> int list -> int<p> list
```

In this syntax, `int<p>` *is equivalent to* `{n : int | p n}` *"an integer satisfying p". If the type checker accept the program, it means that the function* `filter` *is correct. Sadely, from a pure type-checker point of view, the recursive call* `x::(filter p l)` *violates this type signature since* `x` *is of type* `int`*, and not of type* `int<p>`*. The problem here is that the call to* `(p x)` *in the condition returns a boolean that gives no information on* `x`*. This is called "boolean blindness". We need to find a way to tell the type-checker that* `(p x) = true` implies *that* `x` *satisfies p, and thus* `x : int<p>`*.*

Bounded Refinement Types have been introduced by [Vazou et al., 2015] to overcome the "boolean blindness" seen with the `filter` function. It adds the notion of *bound* on the refined types. A bound can be used to add a "condition" in a type signature.

Example 4. *Let's go back to the previous example. The idea is that we will use a bound to relate the boolean returned by* `p x` *to the property verified by* `x`*. The predicate function will have the signature* `(x:int -> bool<w x>)`*, with* `bool<w x> = {b : bool | w x b}` *"a boolean that witnesses x". We now need to* bind *the property w (on the boolean) to the property p (on* `x`*). To do that, we define the following bound:*

```
bound witness p w = forall x b, b => w x b => p x
```

This declares the property w as a witness for p: "For all variable x and boolean b, if b is true and is a witness for x, then x satisfies p". We can now use this bound in the signature of `filter`*:*

```
filter : (witness p w) => (x:int -> bool<w x>) -> int list -> int<p> list
```

Now the type checker knows that in the **`then`** *branch of the condition,* `(p x)` *has returned the value* `true` *of type* `bool<w x>`*. Thanks to the bound, this implies that* `x` *satisfies* `p`*. So we have* `x : int<p>`*, and the call to* `x::(filter p l)` *does not violates the signature of* `filter` *anymore.*

We have proven with Bounded Refinement Types that `filter` *is correct.*

Bounded Refinement Types is a very powerful tool to prove properties over higher-order programs using types. It is successfully able to prove the correctness of `filter`. However it is for now unusable in the context of *lightweight* verification since the user needs to express the property he wants to prove using types. Encoding the property, and giving a right signature to each function to allow the type checker to check the type system requires a high level of expertise. In some more recent work, [Castagna et al., 2014, Castagna et al., 2015] are able to perform a type inference with refined types. However this is still a theoretical work and not yet mature enough to be used in this context.

2.2 Pattern-Matching Recursion Schemes

Pattern-Matching Recursion Schemes (or PMRS) is a technique developed by [Ong and Ramsay, 2011] as an extension of HORS to model check higher-order functional programs. HORS can be seen has higher-order grammars describing the set of terms produced from a start symbol. In the current context, it can be used to represent both the input terms and the program logic in a single structure.

Definition 1 (Higher-Order Recursion Schemes)**.** *Let's $\mathcal{X}$ be a set of variables. A HORS is a quadruple $\mathcal{P} = \langle \Sigma, \mathcal{N}, \mathcal{R}, Start \rangle$ where Σ is an alphabet (terminal symbols), $\mathcal{N}$ a set of non terminal symbols with $Start \in \mathcal{N}$ a start symbol and $\mathcal{R}$ a set of rules of the form*

$$S \to t \quad or$$
$$F\ x_1 \dots x_n \to t$$

where $S \in \mathcal{N}$, $F \in \Sigma, x_1 \dots x_n \in \mathcal{X}$ and t is a term *composed of terminals (Σ), non-terminals ($\mathcal{N}$) and variables ($\mathcal{X}$).*

The fact that $x_1 \dots x_n$ are not terms but variables makes HORS unable to model functional programs with infinite data structures such that algebraic data-types (a list, for instance). PMRS however, is an evolution of HORS that can. It introduces the notion of *patterns*, and allows some rules to have an extra parameter for pattern-matching:

$$F\ x_1 \dots x_n\ p \to t$$

Where p is a pattern composed of terminals and variables.

Example 5. *We can use a PMRS to model our `filter` example function. Let $\mathcal{P} = \langle \Sigma, \mathcal{N}, \mathcal{R}, Start \rangle$ be such a PMRS. $\mathcal{R}$ is defined by:*

$$\begin{aligned} Start &\to filter\ nz\ L \\ N &\to 0 \mid s\ N \\ L &\to N :: L \mid [\,] \end{aligned}$$

$$\begin{aligned} filter\ p\ [\,] &\to [\,] \\ filter\ p\ (x :: l) &\to \textbf{\textit{if}}\ p\ x\ \textbf{\textit{then}}\ x :: (filter\ p\ l)\ \textbf{\textit{else}}\ (filter\ p\ l) \end{aligned}$$

with $\Sigma = \{filter, nz, ::, [\,]\}$, $\mathcal{X} = \{p, x, l\}$ and $\mathcal{N} = \{Start, L\}$. Here the non terminal N represents a natural number, L represents a list of natural numbers, and $Start$ calls the function $filter$ with the predicate nz (non-zero, not defined here).

For a given property φ and a program (a PMRS), the authors tries to find an output that does not satisfy the property. The problem is to decide which term can be reach from the start non-terminal *Start*, to find a counter example. In a sense this can be compared to the LFV technique which computes the set of output terms of a program. However in both cases, the problem being *undecidable*, the answer cannot be *exact*. To approximate the reachable terms, the authors have chosen to first approximate the giver PMRS $\mathcal{P}$ into a weak-PMRS (wPMRS) $\mathcal{P}'$. Then they are able to reason on $\mathcal{P}'$ to compute an over-approximation of the output terms.

A wPMRS is just like a regular PMRS but with the additional condition that, in each rule, none of the variables appearing in the pattern can be in the right side of the rule. In other words, in a wPMRS, for a rule $F\ x_1 \dots x_n\ p \to t$, p and t cannot share any variable $x \in \mathcal{X}$. One way to build a wPMRS from a regular PMRS is to replace every pattern variable in t by a non-terminal representing its possible values.

Example 6. *From the PMRS $\mathcal{P}$ of the previous example 5, we can build the associated weak-PMRS $\mathcal{P}'$ by replacing the two $filter$ rules by:*

$$filter\ p\ [\,] \to [\,]$$
$$filter\ p\ (x :: l) \to \textbf{\textit{if}}\ p\ N\ \textbf{\textit{then}}\ N :: (filter\ p\ L)\ \textbf{\textit{else}}\ (filter\ p\ L)$$

Where on the right sides, the variables x and l has been replaced respectively by N and L. The major side effect is that the filtering operation becomes totally inoperative.

From the raw wPMRS approximation, the authors are able to find a possible counter example to the property φ. It is then tested on the initial PMRS. If the counter example also works with the initial PMRS, it means that φ is for sure not verified by the program. However, if the counter example does not work on the initial PMRS, it means that the approximation given by the wPMRS is too rough. It is then possible to *refine* the rules that gave rise to the wrong counter example.

Example 7. *In the previous example, we had to replace some variable by non-terminals. However, instead of using non-terminals we can* refine *$\mathcal{P}'$ by using constants to differentiate* 0 *from the positive integers.*

$$filter\ p\ (0 :: l) \to \textbf{\textit{if}}\ p\ 0\ \textbf{\textit{then}}\ 0 :: (filter\ p\ L)\ \textbf{\textit{else}}\ (filter\ p\ L)$$
$$filter\ p\ ((s\ N) :: l) \to \textbf{\textit{if}}\ p\ (s\ N)\ \textbf{\textit{then}}\ (s\ N) :: (filter\ p\ L)\ \textbf{\textit{else}}\ (filter\ p\ L)$$

Refinement can be used to iteratively refine the approximation of a wPMRS looking for a counter example for φ, by unrolling the structure of a variable to a certain (limited) depth.

3 Lightweight Formal Verification

This section focuses on the Lightweight Verification Technique which will be used in our abstract interpreter to approximate the set of output values of a program. This technique is mainly based on the Tree Automaton Completion algorithm [Genet and Rusu, 2010] which is able to compute a tree automaton that represents the computations steps of the program.

3.1 Preliminaries

This section describes how *terms*, *rewriting systems* and *tree automata* can be used to model both a program input, and a program logic. Most of the following definition comes from [Baader and Nipkow, 1998] and [Comon et al., 2008].

3.1.1 Terms

If we go back to our previous example, where we wrote in our interpreter `delete 1 [0; 1; 2];;`. We need something to model both the input parameters `1` and `[0; 1; 2]`, and the function call itself `delete x y`. To do that, we will be using *terms*, defined on a specific *alphabet*.

Definition 2 (Alphabet)**.** *An alphabet $\mathcal{F}$ is a finite set of symbols, associated to an* arity *function:*

$$ar_{\mathcal{F}} : \mathcal{F} \to \mathbb{N}$$

□

In our case, a symbol can be seen as a function name, associated to an arity. A symbol f such as $ar_{\mathcal{F}}(f) = 0$ is called a *constant*. For the sake of simplicity, we also note $f \in \mathcal{F}^n$ when $f \in \mathcal{F}$ and $ar_{\mathcal{F}}(f) = n$.

Definition 3 (Terms)**.** *All alphabet $\mathcal{F}$ and finite set of* variables $\mathcal{X}$ *derives a set of* terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *such as:*

$$\begin{aligned} x \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \quad &\Leftarrow \quad x \in \mathcal{X} \\ f\ t_1 \dots t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \quad &\Leftarrow \quad f \in \mathcal{F}^n \text{ and } t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \end{aligned}$$

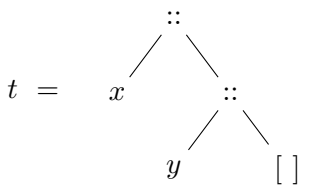
If t is a term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, then $Var(t)$ is the set of all variable $x \in \mathcal{X}$ occurring in t. □

Note the analogy with the functional application, where f is the function and $t_1, \dots, t_n$ the parameters. A term is *closed* if it contains no variable. The set of all the closed terms is noted $\mathcal{T}(\mathcal{F})$. In this definition, a term is always *prefixed* with the function name. However in $\text{OCaml}_{\text{light}}$ some operators are infix, such as `1 + 2`, or `1::2::[]` (which denote the list `[1; 2]`). In the following, we will allow ourselves to use the infix notation.

Example 8. *Here is an example where terms are used to model $OCaml_{light}$ lists. $\mathcal{F}$ contains lists constructors, and $\mathcal{X}$ the bound variables.*

$$\begin{aligned} \mathcal{X} &= \{x, y\} \\ \mathcal{F} &= \{::, [\,]\} \end{aligned}$$

According to the $OCaml_{light}$ syntax, `[]` is the empty list, and `::` is the list construction operator. This is our alphabet $\mathcal{F}$. By convention a term can be represented as a tree where each node is labeled by an element of $\mathcal{F} \cup \mathcal{X}$. For instance the term $t =$ `x::y::[]` (also noted `[x; y]`), by:



We say that a term t is *linear* if the multiplicity of each variable in t is at most 1. The term used in the example 8 is linear.

Definition 4 (Positions)**.** *A* position *on a term t is a word of $\mathbb{N}$ pointing to a* sub-term *of t. $Pos(t)$ is the set of all positions associated to t, one for each sub-term of t. $Pos(t)$ is inductively defined by:*

$$Pos(t) = \{\epsilon\} \quad \Leftarrow \quad t \in \mathcal{X}$$
$$Pos(f\ t_1 \dots t_n) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \wedge p \in Pos(t_i)\} \quad \Leftarrow \quad f \in \mathcal{F}^n, t_1 \dots t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$$

Where "." in $i.p$ is the concatenation *operator.* □

This definition comes with some operations on *positions*. Let $p \in Pos(t)$. We note $t|_p$ the sub-term of t on position p. $t[s]_p$ is the term t where the sub-term on position p has been replaced by s.

Example 9. *On the previous example 8, $t =$* `x::y::[]`*. Let p the position s.t. $p = 2.1$, then $t|_p =$* `y` *and $t[$* `[]` *$]_p =$* `x::[]::[]`*.*

Definition 5 (Substitution)**.** *A substitution σ is an application of $\mathcal{X} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$, mapping a variable to a term. For the sake of simplicity we conventionally extend it to the endomorphism $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$. σt is the result of the application of the substitution σ to the term t.* □

Example 10. *Let's consider the term $t =$* `x::y`*, and the substitution $\sigma = \{x \mapsto 0, y \mapsto 1 ::: [\,]\}$. Then $t\sigma =$* `0::1::[]`*.*

Now that terms are defined, we are able to represent a value, such as a list. We now need something to represent a program and it's logic, to be able, *in fine*, to compute the possible outputs and write our abstract interpreter.

3.1.2 Term Rewriting systems

Rewriting systems are a convenient way to define a program and it's logic, using terms. It can be seen as a set of rules, each of them defining one step of computation.

Definition 6 (Term Rewriting System)**.** *A rewriting system is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is an alphabet and $\mathcal{R}$ a set of rewriting rules of the form $l \to r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$.* □

Definition 7 (Linear Rule)**.** *A rewriting rule $l \to r$ is said left-linear if the term l is linear. In the same way, it is said right-linear if the term r is linear. A* linear *rule is left and right-linear.* □

In the rest of this paper, we will note $\mathcal{R}$ a rewriting system $\langle \mathcal{F}, \mathcal{R} \rangle$ if there is no ambiguity on $\mathcal{F}$. Additionally, in the context of representing a program with terms, we will also do the distinction between the *constructor* terms $\mathcal{T}(\mathcal{C}) \subseteq \mathcal{T}(\mathcal{F})$ representing the *values* from the rest of the terms.

Example 11. *The following rewriting system can be used to represent the* `delete` *function defined in the example 2. Let's define $\mathcal{C} = \{::, [\,]\}$ the constructors, $\mathcal{F} = \{\textbf{\textit{if}}, delete, =\} \cup \mathcal{C}$, and $\mathcal{R}$ as:*

$$delete\ x\ [\,] \to [\,] \tag{1}$$
$$delete\ x\ (y :: l) \to \textbf{\textit{if}}\ x = y\ \textbf{\textit{then}}\ (delete\ x\ l)\ \textbf{\textit{else}}\ y :: (delete\ x\ l) \tag{2}$$

$$\textbf{\textit{if}}\ true\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ y \to x \tag{3}$$
$$\textbf{\textit{if}}\ false\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ y \to y \tag{4}$$

It is supposed here that the predicate $x = y$ is already defined.

A rewriting system $\mathcal{R}$ cannot be used *as is*, since the relation $\rightarrow$ defined by $\mathcal{R}$ only relates terms in their general forms. We need to extend this relation to include terms that match those patterns.

Definition 8 (Rewriting relation)**.** *Any rewriting system $\mathcal{R}$ derives a rewriting relation $\rightarrow_{\mathcal{R}}$ where for all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $s \rightarrow_{\mathcal{R}} t$ if it exists a rule $l \rightarrow r \in \mathcal{R}$, a position $p \in Pos(s)$ and a substitution σ such that:*

$$l\sigma = s|_p \text{ and } t = s[r\sigma]_p$$

The transitive closure of $\rightarrow_{\mathcal{R}}$ is noted $\rightarrow^*_{\mathcal{R}}$. □

Example 12. *The rewriting system introduced in the previous example also derives a rewriting relation $\rightarrow \mathcal{R}$ where*

$$delete\ 1\ 0 :: 1 :: [\,] \rightarrow^*_{\mathcal{R}} 0 :: [\,]$$

No rule can be applied on the last term `0::[]` cannot be applied an other rule. It is irreductible*, and can be considered as the result of the function call. Note that it is a constructor term (member of $\mathcal{T}(\mathcal{C})$).*

So far we are able to represent a value using terms, and represent/execute a program using term rewriting systems. However our goal is to do an abstract interpretation of the program. This means that representing and executing the program on a *single* value is not enough. The program needs to be executed on a *set* of (possibly infinite) input values.

3.1.3 Tree automata

Tree automata are a convenient way to represent a set of terms. We will see in this section the definition of a tree automata, and how to use them to represent the inputs of a program. The following definitions mainly come from [Comon et al., 2008].

Definition 9 (Tree Automaton)**.** *A tree automaton $\mathcal{A}$ is a quadruplet $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where:*

- *$\mathcal{F}$ is an alphabet;*
- *$\mathcal{Q}$ is a finite set of symbols of arity 0, called* states*. $\mathcal{Q} \cap \mathcal{F} = \emptyset$;*
- *$\mathcal{Q}_f$ is a set of* final states*, $\mathcal{Q}_f \subseteq \mathcal{Q}$;*
- *Δ is a rewriting system on $\mathcal{F} \cup \mathcal{Q}$, where each rule is of the form $l \rightarrow q$, $q \in \mathcal{Q}$ and l is either a state ($\in \mathcal{Q}$), or a* configuration *of the form $f\ q_1 \dots q_n$ with $f \in \mathcal{F}, q_1 \dots q_n \in \mathcal{Q}$.*

A term t is recognized *by $\mathcal{A}$ if it exists a state $q \in \mathcal{Q}_f$ such that $t \rightarrow^*_{\Delta} q$. The set of all terms recognized by $\mathcal{A}$ is noted $L(\mathcal{A})$.* □

Tree automatons can recognize *regular languages* (sets of regular terms). In our case, we use them to represent the set of possibles inputs and outputs of a program.

Example 13. *In the example 2 where we define the `delete` function, our analysis supposes that the input list contains `a` and `b` values. The following automaton $\mathcal{A} = \langle \mathcal{C}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ describes such lists. $\mathcal{C} = \{::, [\,]\}, \mathcal{Q} = \{q_0, q_1\}, \mathcal{Q}_f = \{q_1\}$, and Δ:*

$$b \rightarrow q_0 \qquad\qquad q_0 :: q_1 \rightarrow q_1$$
$$a \rightarrow q_0 \qquad\qquad [\,] \rightarrow q_1$$

$\mathcal{A}$ recognizes the regular language of `ab`-lists.

Regular languages are closed by union and intersection and there exists some algorithm to compute the union and intersection of tree automata.

3.2 Tree Automaton Completion

From a program represented as a *rewriting system* $\mathcal{R}$ and it's input represented as a *tree automaton* $\mathcal{A}$, we want to compute the corresponding output (or at least, a good approximation of it). One way to do it is to "complete" the input tree automaton $\mathcal{A}$ using $\mathcal{R}$. The resulting tree automaton $\mathcal{A}^*$ recognizes every terms of $L(\mathcal{A})$ derived by $\mathcal{R}$. It includes all intermediate computations, and in particular, the *output terms*. It is then possible to extract the output terms from $L(\mathcal{A}^*)$.

3.2.1 Problem formalization

The completion algorithm aims to compute a tree-automaton able to recognize the set of all *reachable terms* of a language.

Definition 10 (Reachable Terms)**.** *Let L be a set of terms, and $\mathcal{R}$ be a rewriting system. $\mathcal{R}^*(L)$ is the set of terms reached by rewriting terms in L using $\mathcal{R}$.*

$$\mathcal{R}^*(L) = \{t \mid \exists s \in L, s \to_{\mathcal{R}}^* t\}$$

□

Note that $L \subseteq \mathcal{R}^*(L)$. If L represents the input language of a program, and $\mathcal{R}$ the program's logic, then $\mathcal{R}^*(L)$ contains every term derived from L after an arbitrary number of computation steps. Usually, L is *regular* and is represented as a tree-automaton $\mathcal{A}$. In this case, computing $\mathcal{R}^*(L)$ is equivalent to find a new tree-automaton $\mathcal{A}^*$ such that $\mathcal{R}^*(L) = L(\mathcal{A}^*)$. We call "completion of $\mathcal{A}$" the computation of $\mathcal{A}^*$, because it consists in adding in $\mathcal{A}^*$ all the states and transitions missing in $\mathcal{A}$ to recognize completely $\mathcal{R}^*(L)$.

Definition 11 (Tree Automaton Completion)**.** *Given a tree automaton $\mathcal{A}$ and a left-linear TRS $\mathcal{R}$, tree automaton completion produces a new tree automaton $\mathcal{A}^*$ that recognizes or over approximates $\mathcal{R}^*(L(\mathcal{A}))$:*

$$L(\mathcal{A}^*) \supseteq \mathcal{R}^*(L(\mathcal{A}))$$

□

Note that in most cases, completing exactly to $\mathcal{R}^*(L(\mathcal{A}))$ is impossible due to Rice's theorem. We will see in section 3.2.3 how to workaround this limitation by over approximating the set of reachable terms in a controlled manner.

3.2.2 Completion Algorithm

Let's assume that we dispose of a tree automaton $\mathcal{A}_0 = \langle F, Q, Q_f, \Delta_0 \rangle$ and a left-linear TRS $\mathcal{R}$. Tree automaton compltetion iteratively compute $\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$ such that $\forall i, i \geq 0$:

$$L(\mathcal{A}^i_{\mathcal{R}}) \subseteq L(\mathcal{A}^{i+1}_{\mathcal{R}})$$
$$s \in L(\mathcal{A}^i_{\mathcal{R}}) \;\Rightarrow\; s \to_{\mathcal{R}} t \;\Rightarrow\; t \in L(\mathcal{A}^{i+1}_{\mathcal{R}})$$

This means that the language grow at each iteration, and if a term t is reachable in one step from $\mathcal{A}^i_\mathcal{R}$, then it is recognized by the automaton $\mathcal{A}^{i+1}_\mathcal{R}$. The process continues during $k+1$ iterations until we get the automaton $\mathcal{A}^k_\mathcal{R}$ such that $L(\mathcal{A}^k_\mathcal{R}) = L(\mathcal{A}^{k+1}_\mathcal{R})$. We note this fixpoint automaton $\mathcal{A}^k_\mathcal{R} = A^*$. The construction of $\mathcal{A}^{i+1}_\mathcal{R}$ from $\mathcal{A}^i_\mathcal{R}$ consists in finding and *completing* the *critical pairs* of $\mathcal{A}^i_\mathcal{R}$.

Definition 12 (Critical Pair)**.** *Given a tree automaton $\mathcal{A}$ provided with a rewriting system $\mathcal{R}$, a critical pair is a triple $\langle l \to r, \sigma, q\rangle$ where $l \to r \in \mathcal{R}$, σ is a substitution, and $q \in \mathcal{Q}$ such that $l\sigma \to^*_\Delta q$:*

$$\begin{array}{ccc} l\sigma & \xrightarrow[\mathcal{R}]{} & r\sigma \\ {\scriptstyle\Delta}\downarrow & {\scriptstyle *} & \\ q & & \end{array}$$

□

A critical pair describes a term $r\sigma$ that *should* be recognized by the completed automaton by q (because it is derived from a term $l\sigma$ recognized by q). If $r\sigma \not\to^*_\Delta q$, then we need to complete the automaton in order to also have

$$r\sigma \to^*_\Delta q$$

Showing how to find the critical pairs is not the purpose of this paper. The corresponding algorithm is detailed in [Feuillade et al., 2004]. In the following, we consider that we already dispose of an algorithm able to find the critical pairs $CP(\mathcal{R}, \mathcal{A})$ of a tree-automaton $\mathcal{A}$. Once we dispose of $CP(\mathcal{R}, \mathcal{A}^i_\mathcal{R})$, for each critical pair $\langle l \to r, q, \sigma\rangle$, we need to *solve* the critical pair by adding the nessessary transitions to have $r\sigma$ recognized by q. A simple solution proposed in [Genet and Rusu, 2010] is to add a new state q' such that:

$$\begin{array}{ccc} l\sigma & \xrightarrow[\mathcal{R}]{} & r\sigma \\ {\scriptstyle\mathcal{A}^i_\mathcal{R}}\downarrow & & \downarrow{\scriptstyle\mathcal{A}^{i+1}_\mathcal{R}} \\ q & \xleftarrow[\mathcal{A}^{i+1}_\mathcal{R}]{} & q' \end{array}$$

However we need to be more careful, first because $r\sigma$ may not be of the form $f\ q_1 \dots q_n$ required in a tree automaton, because $r\sigma$ could be a state itself, and because q' may be a duplicated state. We want to transform the transition $r\sigma \to q'$ to make sure it is of the right form, and reuse already existing states. This step is called *normalization* of the transition. Once it is normalized, the transition is added to $\mathcal{A}^{i+1}_\mathcal{R}$. Finally for some i, if $CP(\mathcal{R}, \mathcal{A}^i_\mathcal{R}) = \emptyset$ then $\mathcal{A}^{i+1}_\mathcal{R} = \mathcal{A}^i_\mathcal{R}$, the algorithm terminates.

Example 14. *Let's consider the following TRS $\mathcal{R}$ defined by:*

$$first\ x :: l \to x \tag{5}$$

$\mathcal{R}$ defines a (partial) function that returns the first element of a list. Let $\mathcal{A} = \langle \mathcal{F}, Q, Q_f = \{q_f\}, \Delta\rangle$ be a tree a automaton, where Δ is defined by:

$$\begin{array}{lll} a \to q_0 & [\,] \to q_1 & q_0 :: q_1 \to q_2 \\ b \to q_0 & q_0 :: q_1 \to q_1 & first\ q_2 \to q_f \end{array}$$

Note that q_2 recognizes any non-empty list, so that $\mathcal{A}$ recognizes any call of the function `first` *on a non-empty list. We now complete the automaton $\mathcal{A}$ using $\mathcal{R}$.*

To compute $\mathcal{A}^1_{\mathcal{R}}$, we first need to find the critical pairs of $\mathcal{A}$. Notice that if we take the substitution $\sigma = \{x \mapsto q_0, l \mapsto q_1\}$, we have "$first\ q_0 :: q_1 \to^*_\Delta q_f$", and "$first\ q_0 :: q_1 \to_R q_0$" thanks to the rule (5). We have found the critical pair $\langle first\ x :: l \to x, \sigma, q_f \rangle$, which gives us a new transition $q_0 \to q_f$ already normalized. There is no more critical pairs to complete in $\mathcal{A}^1_{\mathcal{R}}$ so the completion is done in one step. Since q_0 recognizes the values a and b, and we have $q_0 \to^* q_f$ we know that the function `first` *returns a or b.*

3.2.3 Termination Criteria

So far we have seen a completion algorithm that tries to compute a tree automaton that recognize *exactly* $R^*(L(\mathcal{A}_0))$. This worked fine in the example 14, however in most cases completion is not guaranteed to terminate. For instance, if $R^*(L(\mathcal{A}_0))$ is not regular it cannot be represented by a tree automaton. In that case, the completion never terminates. This section shows how to guarantee the termination of the completion by over-approximating $R^*(L(\mathcal{A}_0))$.

Example 15. *Let's see an example where the completion never terminates even on a regular language. Let $\mathcal{A} = \langle \mathcal{F}, Q, Q_f = \{q_0\}, \Delta \rangle$ be a tree automaton such that Δ is defined by:*

$$f\ q_0 \to q_0 \qquad a \to q_1$$
$$g\ q_1 \to q_0$$

and the rewriting system $\mathcal{R}$ defined by:

$$f\ (g\ x) \to g\ (f\ x)$$

The language recognized by $\mathcal{A}$ is $L(\mathcal{A}) = \{f^*\ (g\ a)\}$, and one can see that $R^*(L(\mathcal{A})) = \{f^*\ (g\ (f^*\ a))\}$ is regular. If we apply one step of completion, we find the critical pair defined by $f\ (g\ q_1) \to^*_\Delta q_0$ and $f\ (g\ q_1) \to_\mathcal{R} g\ (f\ q_1)$. The algorithm then adds the new normalized transitions $g\ q_2 \to q_0$ and $f\ q_1 \to q_2$ where q_2 is a new state to have $g\ (f\ q_1) \to^*_\Delta q_0$. The new automaton $\mathcal{A}^1_{\mathcal{R}}$ recognizes the set of reachable terms after at most one step of $\to_\mathcal{R}$. We can quickly see that on the next step, the two new transitions will give rise to a new critical pair defined by $f\ (g\ q_2) \to^*_\Delta q_0$ and $f\ (g\ q_2) \to_\mathcal{R} g\ (f\ q_2)$, introducing a new state q_3 and so on... At each step a new state is added, and the algorithm never terminates.

To overcome this problem, it is possible to *simplify* the completed automaton with a set of *equations* E. This operation aims to over-approximate languages that cannot be recognized *exactly* using completion. The idea is to find the $E-$equivalent terms in a tree automaton $\mathcal{A}$ recognized by different states, and merge those states into one.

Definition 13 (Equation set)**.** *An equation set E is a set of equations of the form $l = r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. From E is derived the relation $=_E$ as the* smallest congruence *such that for all terms l, r and substitution σ we have:*

$$l = r \in E \quad \Rightarrow \quad l\sigma =_E r\sigma$$

The set of equivalence classes defined by $=_E$ on $\mathcal{T}(\mathcal{F})$ is noted $\mathcal{T}(\mathcal{F})/_{=E}$.

The completion algorithm uses the equation set E to merges states recognizing E-equivalent terms in such a way that for all term $s =_E t$, if $s \to^*_{\Delta'} q$ and $t \to^*_{\Delta'} q'$ then $q = q'$. This results in an over-approximation of the language originally recognized by the automaton.

Example 16. *To illustrate how using an equation set over-approximates an automaton we use an example inspired by [Genet, 2016]. Let* $E = \{x :: x :: [\,] = x :: [\,]\}$ *and* $\mathcal{A}$ *be the tree automaton with* $Q_f = \{q_2\}$ *and the set of transitions* $\Delta = \{a \to q_0, [\,] \to q_1, q_0 :: q_1 \to q_3, q_0 :: q_3 \to q_2\}$. *Hence* $L(\mathcal{A}) = \{a :: a :: [\,]\}$. *Notice that we have the following:*

$$\begin{array}{ccc} q_0 :: q_0 :: q_1 & \overset{E}{=\!=} & q_0 :: q_1 \\ \Delta \downarrow * & & \Delta \downarrow * \\ q_2 & & q_3 \end{array}$$

Thus we can merge the states q_2 *and* q_3 *into a single state* q_2. *The resulting set of transitions* $\Delta' = \{a \to q_0, [\,] \to q_1, q_0 :: q_1 \to q_2, q_0 :: q_2 \to q_2\}$. *The resulting automaton* $\mathcal{A}'$ *recognizes any non-empty list so we have* $L(\mathcal{A}') \supseteq L(\mathcal{A})$.

It has been proved that if $\mathcal{T}(\mathcal{F})/_{=E}$ is *finite*, then the completion algorithm terminates. However, proving that $\mathcal{T}(\mathcal{F})/_{=E}$ is finite is an *undecidable* problem. Later, [Genet, 2016] has proved that it is sufficient to have $\mathcal{T}(\mathcal{C})/_{=E}$ finite (where $\mathcal{T}(\mathcal{C})$ are the constructor terms as defined in the section 3.1.2), with some constraints on E and R for the completion to terminate.

Lemma 1 (Completion Termination)**.** *Let* R *be a TRS* sufficiently complete*, where "sufficiently complete" means that all initial term can be reduced into a constructor term:*

$$\forall s \in \mathcal{T}(\mathcal{F}), \quad \exists t \in \mathcal{T}(\mathcal{C}) \text{ s.t. } s \to^*_{\mathcal{R}} t$$

Let $E = E^r \cup E_{\mathcal{R}} \cup E_c$ *be an equation set such that*

- $E^r = \{f\ x_1 \dots x_n = f\ x_1 \dots x_n \mid f \in \mathcal{F}^n\}$
- $E_{\mathcal{R}} = \{l = r \mid l \to r \in \mathcal{R}\}$
- E_c *a set of* contracting *equations left to define.*

If $\mathcal{T}(\mathcal{C})/_{=E_c}$ *is finite, then the completion algorithm terminates.*

The idea behind the proof is that E makes the completion algorithm to generate a finite number of states, which implies that a fixpoint automaton $\mathcal{A}^k_{\mathcal{R}}$ will be reached some day. However, we will see later that this proof does not hold when applied to higher-order programs.

3.3 Extracting the Output Terms

Thanks to the completion algorithm, it is possible to compute the reachable terms of a program input. This includes the output terms, but also every intermediate computation of the program. We now need to find a way to isolate the output terms. Let's note $\mathcal{A}^*_{\mathcal{R}}$ the tree automaton $\mathcal{A}$ completed by $\mathcal{R}$. The output terms are the only terms irreducible by $\mathcal{R}$ (because there is nothing left to compute on them). One can notice that because the output terms are *values*, they are exclusively composed of constructors symbols (on $\mathcal{C}$). Gladly, $\mathcal{T}(\mathcal{C})$ is regular, and as we have seen in the section 3.1.3, regular languages are closed by intersection. Thus, it is possible to compute a tree automaton that recognizes $L(\mathcal{A}^*_{\mathcal{R}}) \cap \mathcal{T}(\mathcal{C})$, which correspond to the set of output terms.

4 Future Contribution

The Lightweight Formal Verification technique let us approximate the set of output terms of a program, to then test some regular properties on them. Our goal is to use the LFV technique to build an abstract interpreter for $\text{OCaml}_{\text{light}}$ to interactively verify the good behavior of a program. However, a lot of work remains to be able to use this technique with $\text{OCaml}_{\text{light}}$.

4.1 Application to $\text{OCaml}_{\text{light}}$

$\text{OCaml}_{\text{light}}$ is a subset of the OCaml language that supports the definition of variant data types (e.g. `type list = Cons of int | Nil`), records, parametric type constructors (e.g. `type 'a list = Cons of 'a | Nil`), exceptions, values. It also supports expressions, polymorphism, mutually-recursive function definition (with `let rec`), pattern-matching and more. A small-step operational semantics has been defined by [Owens, 2008] for this language. Our first task will be to use this operational semantics to find a way to translate any $\text{OCaml}_{\text{light}}$ program into a Term Rewriting System which can be used with the LFV technique. We will also need to translate any "abstract" input expression into a Tree Automaton. Both would then be used to compute the set of output terms of the program, translated back to an OCaml-like syntax. Note that using a TRS is a classic approach to represent the semantics of a functional languages [Clavel et al., 2009], and we've seen through all the examples that it works well with $\text{OCaml}_{\text{light}}$.

4.2 Handling higher-order in a TRS

Term rewriting systems are primarily not designed to represent higher-order programs. In a TRS, a variable can be replaced by a symbol only if it is a constant (symbol of arity 0). Thankfully some methods have been developped to work around this problem. One extends the definition of TRS allowing higher-order variables [Jouannaud et al., 2004] but will not be developed here, because it is not compatible with first order formalism of tree automata and their completion. The other one solves the problem by introducing a new application symbol [Jones and Muchnick, 1979]. The idea is to only consider symbol of arity 0, even for the symbols of functions. This makes sense in the context of higher-order programs, where a function is manipulated as any other value. However, to be able to *call* a function, a new *application* symbols @ of arity 2 is introduced, where @ f x represents the (potentially partial) application of the function f with the parameters x.

Example 17. *Let's define the* `filter` *function with* @*.*

$$\begin{aligned} @\ (@\ filter\ p)\ [\,] &\to [\,] \\ @\ (@\ filter\ p)\ (x :: l) &\to \textbf{\textit{if}}\ (@\ p\ x)\ \textbf{\textit{then}}\ (x :: (@\ (@\ filter\ p)\ l))\ \textbf{\textit{else}}\ (@\ (@\ filter\ p)\ l) \end{aligned}$$

Note that this time, $ar(filter) = ar(p) = 0$.

This method allows us to represent higher-order programs, including the manipulation of *partial applications*, which can cause some troubles to the completion algorithm. In some cases, partial applications can prevent the completion algorithm to terminate, even if the function itself is terminating. This comes from the fact that one of our termination criteria seen in the section 3.2.3 is for the considered TRS to be *sufficiently complete*, which is not always the case when allowing partial applications.

Example 18. *Let define the TRS $\mathcal{R}$ with the rules:*

$$\begin{aligned} @\ (@\ g\ x)\ y &\to y \\ @\ (@\ f\ x)\ [\,] &\to [\,] \\ @\ (@\ f\ x)\ (y :: l) &\to @\ (@\ f\ (@\ g\ x))\ l \end{aligned}$$

Let's consider the tree automaton $\mathcal{A}$ recognizing the terms $@\ (@\ f\ a)\ l$ where l is an arbitrary list. Here the function f is terminating since the input list is decreasing at each call. However the completion $\mathcal{A}$ does not terminate because $\mathcal{R}$ is not sufficiently complete since the term $(@\ g\ x)$. During the completion, at each iteration we would need a new state to represent the partial application term $(@\ g\ x)$, $(@\ g\ (@\ g\ x))$, $(@\ g\ (@\ g\ (@\ g\ \ldots)))$, etc. We would need an infinite number of states to terminate the completion because there is no way to reduce this partial application into a constructor term using $\mathcal{R}$.

We have seen earlier a similar problem when we were considering indefinitely growing constructor terms. To work around this problem, we used the equation set E^c to over-approximate the problematic terms by merging some state in the completed automaton (c.f. example 16). However, E^c is only porting on *constructor terms*. Our goal is precisely to be able to find a develop a similar technique to obtain a terminating completion algorithm handling higher-order functions.

5 Conclusion

The Lightweight Formal Verification technique seems very promising in our quest to build an abstract interpreter to interactivelly verify $\text{OCaml}_{\text{light}}$ programs. It allows us to use a TRS representation of a program to over-approximate the output terms of this program. However, we will first need to find a way to translate any $\text{OCaml}_{\text{light}}$ program into a Term Rewriting System to be able to use this technique. Morever the completion algorithm is not guarenteed to terminates with higher-order programs. Thus we will also need to adapt the completion algorithm to handle partial application terms, or refine the termination criterias on higher-order programs to prevent any non-terminating run of the completion algorithm.

References

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

[Castagna et al., 2015] Castagna, G., Nguyen, K., Xu, Z., and Abate, P. (2015). Polymorphic functions with set-theoretic types: part 2: local type inference and type reconstruction. In *POPL'15*. ACM.

[Castagna et al., 2014] Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., and Padovani, L. (2014). Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL'14*. ACM.

[Clavel et al., 2009] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. (2009). Maude homepage. http://maude.cs.uiuc.edu.

[Comon et al., 2008] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., and Tommasi, M. (2008). Tree automata techniques and applications. `http://tata.gforge.inria.fr`.

[Coq, 2016] Coq (2016). The coq proof assistant reference manual: Version 8.6.

[Feuillade et al., 2004] Feuillade, G., Genet, T., and Viet Triem Tong, V. (2004). Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383.

[Genet, 2016] Genet, T. (2016). Termination criteria for tree automata completion. *Journal of Logical and Algebraic Methods in Programming*, 85(1):3–33.

[Genet et al., 2015] Genet, T., Kordy, B., and Vansyngel, A. (2015). Vers un outil de vérification formelle légere pour ocaml. In *AFADL*, page 6.

[Genet and Rusu, 2010] Genet, T. and Rusu, V. (2010). Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597.

[Jones and Muchnick, 1979] Jones, N. D. and Muchnick, S. S. (1979). Flow analysis and optimization of lisp-like structures. In *POPL*, pages 244–256.

[Jouannaud et al., 2004] Jouannaud, J.-P., van Raamsdonk, F., and Rubio, A. (2004). Higher-order rewriting with types and arities. In *HOR 2004 2nd International Workshop on Higher-Order Rewriting*, page 89.

[Ong and Ramsay, 2011] Ong, C.-H. L. and Ramsay, S. J. (2011). Verifying higher-order functional programs with pattern-matching algebraic data types. *ACM SIGPLAN Notices*, 46(1):587–598.

[Owens, 2008] Owens, S. (2008). A sound semantics for ocaml light. In *European Symposium on Programming*, pages 1–15. Springer.

[Vazou et al., 2015] Vazou, N., Bakst, A., and Jhala, R. (2015). Bounded refinement types. In *ACM SIGPLAN Notices*, volume 50, pages 48–61. ACM.

[Vazou et al., 2013] Vazou, N., Rondon, P., and Jhala, R. (2013). Abstract Refinement Types. In *ESOP'13*, volume 7792 of *LNCS*. Springer.