Model-Checking Higher-Order Recursion Schemes

Ligia Nicoleta Nistor

Submitted in partial fulfillment of the requirements
of the degree of Master in Computer Science

Supervisor: Prof.Luke Ong

Department of Computer Science
University of Oxford

August 21, 2009

Abstract

The verification of functional programs is an area that has seen growing research interest in recent years, due to theoretical advancements. A functional program can be transformed to a higher order recursion scheme(HORS) that generates a tree representing all the possible event sequences of the program, and then the HORS is model-checked. The verification framework of a higher order program is sound and complete and can be used to check temporal properties. We aim to implement a type based verification algorithm for HORS's, using the OCaml language. The algorithm can verify only a subset of modal μ-calculus properties, expressible by trivial tree automata. We have applied our implementation to HORS's obtained from functional programs, and have tested the problem of resource usage analysis. According to our experiments, the prototype model-checker can automatically verify a number of small, but complicated examples. After small optimizations, our model-checker will be able to verify bigger examples and could be the starting point of a more practical approach to the verification of functional programs.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Prof. Luke Ong, for offering me this interesting project to work on and for his constant guidance. I have also been helped by Steven Ramsay, to which I am grateful for the insightful talks we had and for the patient explanations he has given me.

I thank my boyfriend, family and friends for encouraging me while I was working on the project.

Contents

1 Introduction **5**

2 Theoretical Background **8**

2.1 Higher-Order Recursion Schemes(HORS) 8
2.2 The Value Tree of a HORS 9
2.3 Model Checking Recursion Schemes 11
2.4 Verification of Functional Programs 13
2.4.1 Resource Usage Verification Problem 14
2.4.2 Target Language 15
2.4.3 The Complete Verification Framework 17
2.5 Intersection Types 19

3 The Model-Checking Algorithm **22**

3.1 The Original Type System 23
3.2 The Improved Type System 25
3.3 Step 1: Constructing the Configuration Graph 28
3.4 Step 2: Extracting Type Information 30
3.5 Step 3: Deciding if $\mathcal{G}$ is Well-typed 33
3.6 Optimizations 33
3.7 Correctness of the Algorithm 34

4 Implementation Features **35**

4.1 Processing the input file 36
4.2 Implementation of Step 1 38
4.3 Implementation of Step 2 42
4.4 Implementation of Step 3 46
4.5 Tests and evaluation 48

5 Conclusions and Future Work **56**

5.1 Conclusions 56
5.2 Future Work 57

List of Figures

2.1 Value Tree 11
2.2 Run Tree 14
2.3 Verification Framework 17

3.1 Type Based Verification of HORS 22
3.2 Configuration Graph 30
3.3 A Configuration Graph Containing an Error Node 31

List of Tables

3.1 The Model Checking Algorithm 23
3.2 Exponential growth of number of intersection types 28

4.1 Experimental results 48

Chapter 1

Introduction

Hardware and software are used in applications where failure is unacceptable: e-commerce, air-traffic control, medical instruments, etc. The need for reliable systems is critical. We need formal methods to increase our confidence in such systems. Model checking is a technique for automatically verifying finite state programs. The procedure uses an exhaustive search of the state space of the system to determine if some specification is true or not. According to [1], the process of model checking consists of three tasks:

•*Modeling* The first task is to convert a design into a formalism accepted by a model checking tool. This can be a compilation task or it may require the use of abstraction to eliminate irrelevant detail.

•*Specification* It is necessary to express the properties that the design must satisfy. The specification is usually given using temporal logic, which can assert how the behaviour of the system evolves over time. Through model checking, one can see if the design satisfies a given specification.

• *Verification* Ideally the verification is completely automatic. However, in practice, it often involves manual activity, such as the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analysing the error trace may require a modification to the system and reapplication of the model checking algorithm.

Extensive research has been done in software model checking and the advancement of propositional SAT solvers has allowed more applicability in industry, but mainly imperative programming languages have been observed. This is because the majority of software developing companies in the world write their programs using imperative languages. The current style of imperative programming does not separate different functionalities in totally independent functions, but instead there are a lot of dependencies between the functions of an imperative program. The dependecies do not allow for compositional model checking to be properly applied. The problem of function dependency does not appear in functional programs. Thus, if we want correct programs, we may need to change our style of programming, by making it more strict, possibly even by emphasising correctness over performance. A step in this direction is the use of higher-level programming languages, such as OCaml, in place of languages such as C++ or Java. OCaml is especially adequate for scientific programming, as it can model easily and correctly scientific problems.

As described in [2], some advantages offered by the OCaml language are:

•*Safety* OCaml programs are strictly checked at compile-time such that they are proven to be entirely safe to run.
•*Functional* Functions may be nested, passed as arguments to other functions and stored in data structures as values.
•*Statically typed* Any typing errors in a program are detected at compile-time by the compiler, instead of at run-time as in other languages. From our experience, if an OCaml program compiles, then almost certaintly it will run.
•*Type inference* The types of values are automatically inferred during compilation by the context in which they occur.
•*Polymorphism* In the cases where any of several different types may be valid, any such type can be used. This makes it easier to write generic, reusable code.
•*Modules* Programs can be structured by grouping their data structures and related functions into modules.
•*Separate compilation* Source files can be compiled separately into object files which are then linked together to form an executable. When linking, object files are automatically type checked and optimized before the final executable is created.

Thus, there is scope to investigate the correctness of functional programs, especially the ones written in OCaml. This project aims to build an application written in the OCaml language which model-checks higher-order recursion schemes. We will implement a recently published verification algorithm for temporal properties of higher order functional programs. To apply this method, a program is first transformed to a higher order recursion scheme (HORS)- a grammar for describing an infinite tree. This model is checked with a type based algorithm, that we have implemented. The algorithm can deal only with the safety fragment of modal μ-calculus, or equivalently the acceptance problem of Buchi nondeterministic automata, with a trivial acceptance condition. The goal of the project is to tackle the problem of resource usage verification for higher order functional languages(such as OCaml). The verification method based on recursion schemes is sound and complete for an extension of the simply-typed λ-calculus with recursion and finite data domains.

We have implemented a model-checker for recursion schemes based on the algorithm proposed in [3]. We have tested it successfully on a number of small but tricky examples. These examples have been obtained from verification problems of functional programs by using Naoki Kobayashi's translation from [7].

This report starts with a compelling theoretical background, describing notions such as Higher Order Recursion Schemes (HORS), the value tree of a HORS, how to model-check a HORS, the resource usage verification problem and its target language, the complete verification framework, intersection types. In Chapter 3, the type-based model-checking algorithm is explained in detail. This algorithm uses a type system which could be improved for future optimizations, so we presented both the type-system that we have used in our current implementation and the type system with subtyping which can be used when doing the optimizations. Following this, the implementation features are discussed and the commented code of representative functions is provided. We have then showed the output of running our implementation on a number of examples. Finally, we move on to talk about some inefficient parts of the implementation and suggest directions for future work.

Chapter 2

Theoretical Background

2.1 Higher-Order Recursion Schemes(HORS)

As described in [3],a higher-order recursion scheme is a grammar for generating a (maybe infinite) tree. A word grammar uses the alphabet of a formal language to generate strings syntactically valid within the language. In the same way, a tree grammar generates a tree language, except that the basic elements are trees, therefore terminals and non-terminals may have arity bigger than 0. According to [4], a tree grammar $G = (S, N, F, \mathcal{R})$ is composed of an axiom S(also called the start symbol), a set N of non-terminal symbols such that $S \in N$, a set F of terminal symbols, a set $\mathcal{R}$ of production rules of the form $\alpha \rightarrow \beta$ where α, β are trees of $T = (F \cup N \cup X)$, X is a set of dummy variables and α contains at least one non-terminal. Moreover we require that $F \cap N = \emptyset$, that each element of $N \cup F$ has a fixed arity and that the arity of the axiom S is 0.

The set of kinds, denoted by k, is defined as:

$$k ::= o|k_1 \rightarrow k_2$$

The kind o represents a tree. The kind $k_1 \rightarrow k_2$ represents a function that takes an argument of type k_1 and returns an value of type k_2. More details on this type system approach will be given later on. Two functions characterize each kind: order and arity. The order of k measures how deeply nested the type is on the left of the arrow constructor and is given by the formula:

$$order(o) := 0;\ order(k_1 \rightarrow k_2) := max(order(k_1) + 1, order(k_2))$$

The arity of a kind k gives the number of arguments requested by a symbol of type k. The arity is given by the formula:

$$arity(o) := 0;\ arity(k_1 \rightarrow k_2) := arity(k_2) + 1$$

Both these definitions rely on the fact that in the simply typed lambda calculus, nested arrow types associate to the right, unless otherwise bracketed.

A (deterministic) higher-order recursion scheme is a quadruple$(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where:

•Σ is a ranked alphabet, each symbol in the alphabet has an associated arity which determines the number of children a node labeled with this symbol will have. I.e., Σ is a map from a finite set of symbols called *terminals* to kinds of order 0 or 1. The kind of a non-terminal can only be o or $o \rightarrow o \rightarrow ... \rightarrow o$ (a finite enumeration of o's , where the bracketing is always the implicit one, associative to the right).
•$\mathcal{N}$ is a map from a finite set of symbols called *non-terminals* to kinds. The only requirement here is that $\mathcal{N}(S) = o$.
•$\mathcal{R}$ is a map from the set of non-terminals to terms of the form $\lambda\tilde{x}.t$, where $\tilde{x}$ is a list of variables and t is a term of kind o. This restriction on t is needed for the model checking algorithm to work. Even if terms in the lambda calculus can be variables, abstractions or application, a term in this project is regarded as either a symbol or an application. By convention application associates to the left, so that $t_1t_2 \ldots t_n$ is a shorthand for $(((t_1t_2)t_3)\ldots)t_n)$ for $n \geq 2$. Thus, a symbol (a terminal, non-terminal or variable) of kind k is a term of kind k. If terms t_1 and t_2 have kinds $k_1 \rightarrow k_2$ and k_1 respectively, then t_1t_2 is an application term of kind k_2. When $R(F) = \lambda\tilde{x}.t$, the requisite is that $F\tilde{x}$ must be a term of kind o. The set of applicative terms generated by Σ, $T(\Sigma)$, is the least set X containing Σ that is closed under the rule: if $s : \alpha \rightarrow \beta$ and $t : \alpha$ are in X then $(st) : \beta$ is in X.
•S is a non-terminal called the *start symbol.*

2.2 The Value Tree of a HORS

In the following, we will give a formal definition of the tree generated by a recursion scheme, namely the *value tree.* There are a number of preliminary notions that need to be described, as in [6].

First, we define the rewriting relation(a one-step reduction relation) $\rightarrow_{\mathcal{G}}$ inductively:

$$Ft_1...t_n \rightarrow_{\mathcal{G}} e[t_1/x_1, ..., t_n/x_n]$$

where $Fx_1 \ldots x_n \rightarrow e$ is a rewrite rule of $\mathcal{G}$.

If $t \rightarrow_{\mathcal{G}} t'$ then $(st) \rightarrow_{\mathcal{G}} (st')$ and $(ts) \rightarrow_{\mathcal{G}} (t's)$.

Intuitively, $s \rightarrow_{\mathcal{G}} s'$ only if s' comes from s by substituting the head symbol(non-terminal) F by the right-hand side of its rewrite rule in which all the formal variables are substituted by the actual parameters.

Next, **Σ-labelled ranked trees** are partial functions t from $T = \{1, 2 \ldots n\}^*$ to $dom(\Sigma)$ such that $dom(t)$ is prefix closed. We assume that Σ is a ranked alphabet and n is the largest arity of symbols in Σ. A subset S of T is prefix closed if for any element $e \in S$, every prefix of e is also in S. Recursion schemes are generators of Σ-labelled trees and we call the generated trees-*value trees*. The value tree $[\mathcal{G}]$ of a recursion scheme $\mathcal{G}$ is a possibly infinite applicative term (which does not contain abstractions) constructed from the terminals in Σ, obtained by fair rewriting of the grammar rules infinitely many times, performing substitutions of the formal variables each time, starting from the non-terminal S. For an applicative term to be a proper representation of a tree, it has to contain only terminals. Thus, we need a procedure to eliminate each non-terminal and its arguments, present in the term. This is done by the map $(\cdot)^{\perp} : T(\Sigma \cup N) \rightarrow T(\Sigma)$. In the example below, f ranges over Σ-symbols and F over non-terminals in N.

$$f^{\perp} = f$$

$$F^{\perp} = \perp$$

$$(st)^{\perp} = \begin{cases} \perp & \text{if } s^{\perp} = \perp; \\ (s^{\perp} t^{\perp}) & \text{otherwise.} \end{cases}$$

For example, $(f(Fab)c)^{\perp} = f \perp c$.

As in [7], we need to define a partial order $\sqsubseteq$ on $dom(\Sigma) \cup \perp$ (the set of terminals and the $\perp$ symbol): $\forall a \in dom(\Sigma), we\ have\ \perp \sqsubseteq a$. We need to extend it to the partial order on trees using this equivalence: $t \sqsubseteq s$ iff $\forall w \in dom(t).(w \in dom(s) \wedge t(w) \sqsubseteq s(w))$. For example $f \perp \perp \sqsubseteq f \perp b \sqsubseteq fab$. For a set T of trees, we use the notation $\bigsqcup T$ for the least upper bound of elements of T, with respect to the partial order $\sqsubseteq$. The value tree of $\mathcal{G}$, $[\mathcal{G}]$, is defined by:

$$[\mathcal{G}] = \bigsqcup \{t^{\perp} | S \rightarrow^{*}_{\mathcal{G}} t\}.$$

A couple of notes have to be made: firstly, $[\mathcal{G}]$ above is well defined, because there is exactly one rewriting rule for each non-terminal. Secondly, the value trees of the recursion schemes in this project do not contain $\perp$. This is because these recursion schemes are generated by the translation algorithm of functional programs in [7], which will be briefly mentioned. In the general case, if we are given a recursion scheme $\mathcal{G}$ and a formula ψ, we can transform them into $\mathcal{G}'$ and ψ' such that:

(1) $[\mathcal{G}']$ satisfies ψ' if and only if $[G]$ satisfies ψ
(2) $[\mathcal{G}']$ does not contain $\perp$. This result is mentioned in [6].

EXAMPLE 2.1. We consider the recursion scheme $G_0 = (\Sigma, N, R, S)$, where:

$$\Sigma = \{a : o \to o \to o, b : o \to o, c : o\}$$

$$\mathcal{N} = \{S : o, F : o \to o\}$$

$$\mathcal{R} = \{S \to F(Fc), F \to \lambda x.ax(b(Fx))\}$$

S can be reduced as follows(we do not use rightmost or leftmost derivation techniques, but instead at each step we expand a certain non-terminal so that we can represent the illustrative value tree in Figure 2.1):
$S \to F(Fc)$
$\to a(Fc)(b(F(Fc)))$
$\to a(ac(b(Fc)))(b(F(Fc)))$
$\to a(ac(b(Fc)))(b(a(Fc)(b(F(Fc)))))$
$\to \ldots$

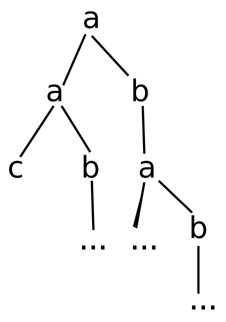

Figure 2.1: Value Tree

2.3 Model Checking Recursion Schemes

The practical advancements in model checking functional programs have all been possible due to the recent theoretical results. We will state the most important theorem so far and define some basic concepts needed to grasp the meaning of the theorem. In 2006, Prof. Luke Ong proved the decidability of the modal μ-calculus model checking of recursion schemes [6]. The modal μ-calculus is a class of temporal logics with a defining feature: use of fixpoint operators.

THEOREM 2.1. *Let $\mathcal{G}$ be a recursion scheme of order n, and ψ be a modal μ-calculus formula. The problem of checking whether $[\mathcal{G}]$ satisfies ψ is n-EXPTIME-complete.*

According to [8], in computational complexity theory, the complexity class n-EXPTIME is the set of all decision problems solvable by a deterministic Turing machine in O(x), where x is a tower formed from recursively raising 2 to the power of 2, for n times. All is then raised to the power $p(m)$, a polynomial function of n. For example, for 2-EXPTIME, this means $O(2^{2^{p(m)}})$. A decision problem is n-EXPTIME-complete if it is in n-EXPTIME, and every problem in n-EXPTIME has a polynomial-time many-one reduction to it. I.e. there is a polynomial-time algorithm that transforms instances of one to instances of the other with the same answer.

In [6] it is mentioned that the question whether $[\mathcal{G}]$ satisfies ψ is equivalent to deciding if a given alternating parity tree automaton B has an accepting run-tree over $[\mathcal{G}]$ [Emerson and Jutla 1991]. Alternating automata on trees are a generalization of non-deterministic tree automata. For a finite set X, $B^+(X)$ is a set of positive Boolean formulas over X, i.e. for x ranging over X, a boolean formula $\theta \in B^+(X)$ can be $true|false|x|\theta \wedge \theta|\theta \vee \theta$. For a set $Y \subseteq X$ and a formula $\theta \in B^+(X)$, Y satisfies θ only if assigning true to elements in Y and false to elements in $X \backslash Y$ makes θ true. It means that **true** is satisfied by all subsets of X, and **false** by none. Because $\theta \in B^+(X)$ is positive, if a set Y satisfies θ, so does every superset of Y.

An alternating parity automaton over Σ-labelled trees is a tuple $A = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ where:

-Σ is the ranked input alphabet
-Q is a finite set of states, and $q_0 \in Q$ is the initial state
-$\delta : Q \times \Sigma \rightarrow B^+(Dir(\Sigma) \times Q)$ is the transition function where, for each $f \in \Sigma$ and $q \in Q$, we have $\delta(q, f) \in B^+(Dir(f) \times Q)$. In the preceding, we assume that each symbol $f \in \Sigma$ is assigned a finite set $Dir(f)$ of exactly $arity(f)$ directions, and we define $Dir(\Sigma) = \bigcup_{f \in \Sigma} Dir(f)$. Every element in $(Dir(f) \times Q)$ is a pair of a direction and a state, thus $\delta(q, f)$ is a positive boolean formula formed from these pairs, giving possible ways of advancing in the tree.
-$\Omega : Q \rightarrow \aleph$ is the priority function.

A run-tree of an alternating parity automaton over a Σ-labelled ranked tree t is a $(Dom(t) \times Q)$-labelled unranked tree r. r is a function having the domain a D-tree, where D is a finite set of directions, and the codomain is

$(Dom(t) \times Q)$. (If D is a set of directions, a D-tree is just a prefix-closed subset of D^*, the free monoid of D.) A run-tree is accepting if all its infinite paths $\beta_0\beta_1\beta_2\ldots$ through $Dom(r)$ satisfy the parity acceptance condition: the largest number that occurs infinitely often in the numeric sequence $\Omega'(\beta_0)\Omega'(\beta_1)\Omega'(\beta_2)\ldots$ is even.

In this project, we implement the model checking algorithm only for a subset of properties: safety properties (meaning that a bad thing never happens, when a tree is traversed from the root). These properties are described by deterministic Büchi automata with a trivial acceptance condition. We present some basic definitions, as in [7].

A deterministic Büchi automaton with a trivial acceptance condition B is a quadruple $M = (Q, \Sigma, q_0, \Delta)$, where Q is a finite set of states, Σ is a ranked alphabet, $q_0 \in Q$ is the initial state, $\Delta \subseteq \bigcup_k (Q \times dom(\Sigma) \times Q^k)$, such that if $(q, a, q_1, \ldots, q_k) \in \Delta$, then $arity(\Sigma(a)) = k$ (this way, the arity of a is respected).

A run of M on the tree t over Σ is a tree ρ over Q, such that $\rho(\epsilon) = q_S$ and $(\rho(w), t(w), \rho(w_0), \ldots, \rho(w_{(k-1)})) \in \Delta$ for any $w \in dom(t)$ and $k = \Sigma(t(w))$. As opposed to a Büchi automaton, which has a set of accepting states, a trivial automaton has no such set. Hence, M accepts t if there is a run of M on t. This means that M accepts t if every time when reading a terminal while being in state q, there is always a tuple of future states to move to, specified by the transition function δ. I.e. the automaton never blocks on an input terminal.

EXAMPLE 2.2. Let B_0 be the automaton $(\Sigma, \{q_0, q_1\}, \delta, q_0)$, where:

$$\Sigma = \{a \to o \to o \to o, b \to o \to o, c \to o\},$$
$$\delta(q_0, a) = q_0 q_0 \qquad \delta(q_0, b) = q_1 \qquad \delta(q_1, b) = q_1$$
$$\delta(q_0, c) = \epsilon \qquad \delta(q_1, c) = \epsilon$$

The run tree of B_0 over the tree generated by the recursion scheme in EXAMPLE 2.1 is depicted in Figure 2.2. B_0 does not accept the value tree in Figure 2.1, because it blocks on the rule $\delta(q_1, a)$.

2.4 Verification of Functional Programs

In this project, the problem of model checking recursion schemes is derived from the resource usage verification problem. At first, we are presented with a skeleton of a functional language. This is a simply-typed functional language with resources and non-deterministic branches that has the same expressive power as the call-by-value λ-calculus with primitives described

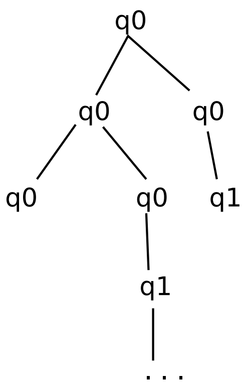

Figure 2.2: Run Tree

in [9]. Then, the problem of resource usage is analysed, related to the former language. In order to solve this problem, the source program is transformed to a HORS and the model checking is applied, by using a type system. Our implementation deals only with the model checking phase, but we feel it is important to understand the entire framework.

2.4.1 Resource Usage Verification Problem

A program is correct only if it accesses its resources in a valid manner. For example, a file that has been opened should be closed after use. In concurrent programming, when working with shared resources, a lock should be acquired before such a resource is accessed. After the lock has been acquired and the resource used, the lock should be released. Assertion-based model-checking problems(like "X¡0 holds at program point p")can also be reformulated as the resource verification problem, by thinking about an assertion failure as an access to a global resource. For example, instead of `‘‘assert(b)’’` , we can write `‘‘if b then skip else fail’’`, where `fail` is a function that accesses the global resource. Thus, the problem of checking lack of assertion failures is reduced to the resource usage verification problem of checking if the `fail` action occurs.

There are different types of primitives for accessing resources, depending on each case (initialization, read, write, close, deallocation, etc.). We want to be sure that those primitives are applied in a correct order. Thus, the resource usage analysis question is closely related to flow analysis, paying attention to the order of resource access. Intuitively, we don't want any invalid resource access to occur(such as closing a file before it has been opened) and all required accesses must take place before the termination of the program. If these two conditions are met, the given program is resource-safe.

We will give a brief account of the formalization of the resource usage analysis problem, similar to the formalization of the flow analysis problem, as in [9]. We annotate each expression of a program with a label and let L be the set of labels. For the standard flow analysis question for λ-calculus, we need to obtain a function $flow \in L \to 2^L$, where $flow(l) = \{l_1, \ldots, l_n\}$ means that the value to which an expression labeled with l evaluates to is the same value generated by an expression labeled with one of $l_1, \ldots, l_n$. This can be reformulated in this way: we need to obtain a function $flow^{-1} \in L \to 2^L$, where $flow^{-1}(l) = \{l_1, \ldots, l_n\}$ says that the expressions labeled with $l_1, \ldots, l_n$ are the only ones that can evaluate to the value generated by an expression labeled with l. From $flow^{-1}$, we deduce what kind of access may occur in the case of each resource. We shall analyse the following line of a functional-like program:

EXAMPLE 2.3.

$$\textbf{let } x = \textbf{fopen}(s)^l \; in \ldots \textbf{fread}(M^{l_r}) \ldots fclose(N^{l_c}) \ldots$$

Here, *fopen* opens a file named s and returns a file pointer for further access, and *fread, fclose* take a file pointer as an input and read, respectively close the file. If we define $flow^{-1}(l) = \{l_r\}$, then we know that the file opened at l may be read, but may not be closed. If we want to be allowed to close the file, then we must define $flow^{-1}(l) = \{l_r, l_c\}$. In this latter case, we do not have any information about the order of resource accesses. In fact, a flow function does not provide this kind of information: we don't know if the file created at l is closed after it has been read or is read after having been closed. We need to introduce a second notion to provide the order of accesses: a function $use \in L \to 2^{L^*}$, where L^* is the set of finite sequences of labels and $l_1 \ldots l_n \in use(l)$ signifies that a value generated by an expression labeled with l may be accessed by primitives labeled with $l_1, \ldots l_n$, in this order. Now, the usage analysis problem becomes the problem of computing the function *use* and checking whether $use(l)$ contains only correct sequences. When reconsidering the above line of code, if $use(l) = \{l_r l_c, l_r l_r l_r l_c\}$, we know that the file opened at l may be closed after being read once or three times. These two are the only uses of l that are permitted. If we had defined $use(l) = \{l_r l_c, l_c l_r\}$, the file could be read after having been closed.

2.4.2 Target Language

The resource usage verification problem is applied to a skeleton functional language, described in [7]. Programs written in this language have only top-level definition functions. This is because we consider that programs are already in continuation passing style (CPS transformation and λ-lifting having been already applied). These tranformations (that will be discussed later) do not change the resource access sequences.

A program D is a set of function definitions $\{F_1\tilde{x_1} = e_1, \ldots F_n\tilde{x_n} = e_n\}$, where F_i is a function symbol and e ranges over the set of expressions, defined by:

$$e ::= *|F|e_1e_2|\mathbf{if} * e_1e_2|\mathbf{new}^L e|\mathbf{acc}_a xe$$

We assume that the symbols $F_1, \ldots, F_n$ are different from each other, and that any program D contains exactly one definition for S (which is considered the main function), of the form $S = e$. We can already see the similarities with the $\mathcal{R}$ set of a HORS.

The expression * is the unit-value. The expression e_1e_2 means that the function e_1 is applied to e_2, and $\mathbf{if} * e_1e_2$ non-deterministically executes e_1 or e_2. The expression $\mathbf{new}^L e$ creates a resource that should be used according to the specification L, and passes it to e (which is a function that takes a resource as an argument). This is the same mechanism as in Example 2.2, where a pointer to a file was created and then passed to the functions *close, read* as a way to access the file. As above, L is the set of access sequences that may occur until the program terminates. For example, the specification for read-only files is the infinite set $\{c, rc, rrc, \ldots\}$. Every element in the set has to have c as last element, because files must be closed before the program terminates. Thus, the element r is not in the set. L is assumed to be a regular language. If we replace the class of regular languages with that of context free languages, the verification problem becomes undecidable. $\mathbf{acc}_a\ x\ e$ applies an operation of name a to the resource x, and then evaluates the expression e. We mention that a ranges over a finite set of the names of resource access primitives. We write $[e_1/x]e_2$ for the expression obtained by replacing all the occurrences of x in e_2 with e_1.

EXAMPLE 2.4. The next program accesses a read-only file and is written in an OCaml-like language.

```
let rec f x =if b then close(x)
   else begin read(x); f x end in
let r=open_in linestxt in f r
```

The corresponding program, written in the language of this section, is:

$S = \mathbf{new}^{r^*c} C$

$Cx = F\ x\ \star$

$Fxk = if\ *\ (acc_c x)\ (acc_r x(Fxk))$

The specification r^*c means that the file can be read for as many times as wanted, and then it needs to be closed.

The general complete model-checking method is detailed next.

2.4.3 The Complete Verification Framework

The resource usage problem is decidable for the language having only resources and functions, but the problem becomes undecidable if we add infinite value domains (such as integers). In this latter case, we can use techniques from Software Verification, like predicate abstraction and counter-example guided abstraction refinement. In Figure 2.3, one can see the entire cycle needed for finding out if the program is resource safe or not(although the cycle may go on forever).

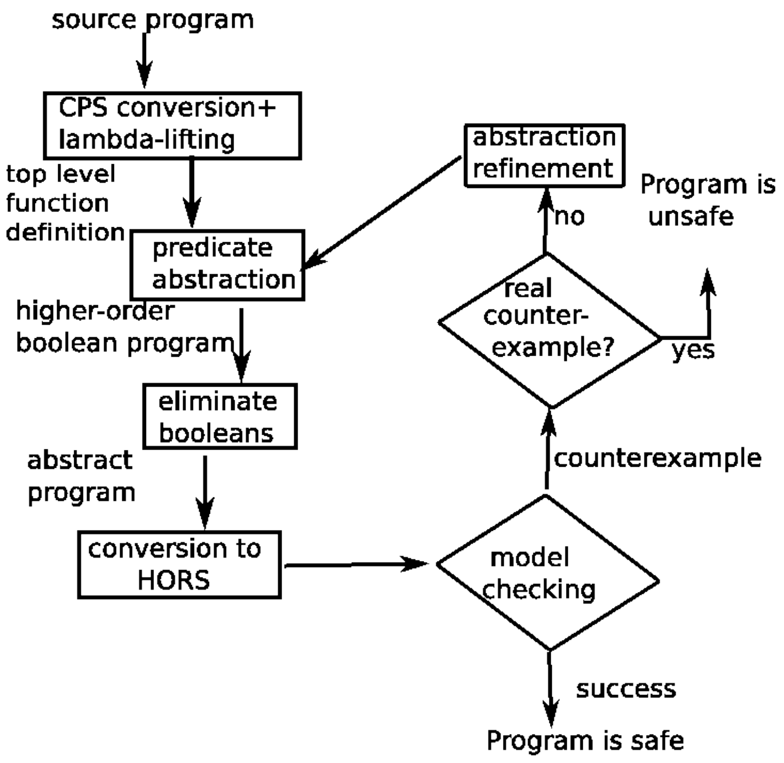

Figure 2.3: Verification Framework

The language of the source program has resources, functions and infinite value domains. Through CPS conversion and λ-lifting, the source program is transformed to a set of top-level function definitions.

A program is in CPS if every function takes a continuation as a parameter, and as its last action calls its continuation, passing it the function's 'result'. The parameter is in fact a function which is meant to receive the result of the computation performed within the original function. CPS is synonym with static single assignment and both make some things explicit, which are otherwise implicit in the imperative style: procedure returns, which become clear as being calls to a continuation; intermediate values, which are all given different names, etc. After transforming the program to CPS, we need to apply λ-lifting, that consists of the following, according to [10]: we need to eliminate the free variables from local function definitions of the program and then replace every local function definition that has no free variables with an identical global function. After applying these steps a number of times, all free variables and local functions will be eliminated. At this point, we will have a set of top-level function definitions representing the original source program. Because the language was extended with infinite value domains, we can use predicate abstraction. We can also use counter-example guided abstraction refinement, an iterative method to compute a sufficiently precise abstraction. When computing the existential abstraction of programs, we need to refine the predicates (which can be null in the beginning) and one way of doing this is by using Craig Interpolants.

After having applied predicate abstraction, we get a higher order boolean program, for which we have to eliminate the booleans. For every boolean value, we can express true and false by 1 and 0 respectively. If a function F takes a boolean b as an argument, we eliminate the boolean by using function F_1 for $b = 1$ and function F_2 for $b = 2$. In general, we can represent a function F of type $\mathbf{bool} \to \mathbf{bool} \to \ldots \to \mathbf{bool} \to \tau$ (where τ is not of the form $\mathbf{bool} \to \sigma$) by a finite set of functions $F_{b_1 \ldots b_n}$, where $F_{b_1 \ldots b_n}$ has the same behaviour as $F(b_1, \ldots, b_n)$. After eliminating all the booleans, we have an abstract program of the language in section 2.4.2, which has to be converted to a HORS.

In [7],Kobayashi gives a method of transforming the program D into a pair consisting of a HORS $H(D)$ and a tree automaton $M(D)$ (for infinite trees), such that D is resource-safe if and only if $[H(D)]$ is accepted by $M(D)$. We will give a short description of such a $H(D)$, for the reader to understand what is the origin of the future HORS examples.

If D is the program written in the language of section 2.4.2, let $\mathbf{Fnames}(D)$, $\mathbf{Anames}(D)$ and $\mathbf{Specs}(D)$ be the set of defined function symbols, access primitive names (such as *read*,*close*), specifications(the L of new^L) respectively. The transformation from D to the HORS H is:

$H(D) = (\Sigma, N, R, S)$

where:

$\Sigma = \mathbf{Anames}(D) \cup \{\nu^L | L \in \mathbf{Specs}(D)\} \cup \{call, t, br\}$

$N = \mathbf{Fnames}(D) \cup \{\mathbf{new}^L | L \in \mathbf{Specs}(D)\}$
$\quad \cup \{\mathbf{acc}_a | a \in \mathbf{Anames}(D)\} \cup \{\mathbf{if}*, I, K, \star\}$

$R = \{F\ \tilde{x} \to \mathbf{call}(e) | F\tilde{x} = e \in D\}$

$\cup\{\mathbf{acc}_a\ x\ y \to x\ a\ y | a \in \mathbf{Anames}(D)\}$
$\cup\{\mathbf{new}^L x \to br(xK)(\nu^L(xI)) | L \in \mathbf{Specs}(D)\}$
$\cup\{\mathbf{if}*xy \to br\ x\ y, Ixy \to x(y), Kxy \to y, \star \to t(\star)\}$

The types of the terminals are:
$a, t, call, \nu^L : o \to o; br : o \to o \to o$
The types of the non-terminals are:
$\star : o; I, K : (o \to o) \to o \to o; \mathbf{if}* : o \to o \to o;$
$\mathbf{new}^L : (((o \to o) \to o \to o) \to o) \to o;$
$\mathbf{acc}_a : ((o \to o) \to (o \to o)) \to o \to o.$

This definition guarantees that the value tree of $H(D)$ does not contain $\perp$. The symbol $\star$ represents a termination and is reduced to the infinite tree $t(t(t(\ldots)))$ because all the paths of the value tree have to be infinite.

The resulting HORS is verified by using a type-based algorithm. Our implementation regards only this algorithm, having a HORS and a trivial automaton as input and giving as output "yes" if the automaton accepts the value tree of the HORS and "no" otherwise. If the answer is "yes", the verification has succeeded and we can conclude that the source program is resource-safe. On the other hand, if the answer is "no", we will construct a counterexample using the verification algorithm and return to the big verification framework. Because the idea of the abstraction is to reduce the size of the original model by removing irrelevant detail (as in Software Verification), it might turn out that the counterexample is spurious. When we check, if the counterexample proves to be real, then we can say that the program is unsafe. If it is spurious, we will refine the predicates and apply again predicate abstraction. This cycle is repeated until the program is found to be safe or unsafe. Because the problem is undecidable, the cycle might loop indefinitely, in which case we will not have an answer to the question "is this program safe or not?".

2.5 Intersection Types

The model checking of a HORS makes use of intersection types. This is the last notion that we need to present for an easy understanding of the following chapters. We will use the definitions and explanations from [12], and occasionally from [13].

Simple type systems for the λ-calculus are used to prohibit "unsafe" operations. In the case of programming languages, these are operations where one format of data is used in a way intended for another - type mismatching. There are many type systems, for example Alonzo Church's system or Haskell Curry's system. In both of these systems, the combinator

$s = \lambda x.xx$ is not typable, i.e. there is no context Γ and type A such that $\Gamma \vdash s : A$. We mention that a type context is a finite(maybe empty) set of pairs $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$, where each (distinct) x_i is a term variable and each A_i is a type.

We introduce the Coppo-Dezani (CD) Type-Assignment System $TA_\lambda(\wedge, \omega)$. In this system, the statement "M has type τ" describes the computational behaviour of the term M as an operator in some significant way. Each term will be assigned a set of types that describes its behaviour. We will give a general presentation of this type system, for the reader to have an intuition about intersection types.

We are given an infinity of type-variables a,b,c,..., and one type-constant, ω, called the universal-type. The rules for building the set of types **Typ** are:

(1) The type-variables and ω are types (called atoms).

(2)If σ, τ are types, then so are $(\sigma \to \tau)$, $(\sigma \wedge \tau)$.

If parenthesis are omitted from types, types associate to the right, as opposed to terms, which associate to the left. For example

$$\rho \wedge \sigma \wedge \tau \equiv (\rho \wedge (\sigma \wedge \tau))$$

$$\rho \to \sigma \to \tau \equiv (\rho \to (\sigma \to \tau))$$

where $\equiv$ means"is identical to" and ρ,σ and τ are types.

Each type denotes a set. Thus, $\sigma \wedge \tau$ denotes the intersection of sets σ and τ; ω is the universe of all objects; $\sigma \to \tau$ denotes the set of partial operators which, when they are applied to an input from the set σ, produce an output from τ. It can also be the case that no output is produced at all: if the input is of type σ, the output need not be in τ.

As for any other type system, there is an axiom-scheme. In fact, there is an infinite set of axioms, one for each term M, summarized in this axiom scheme:

$$M : \omega \qquad (\omega)$$

$$\frac{M : (\sigma \to \tau) \quad N : \sigma}{(MN) : \tau} \qquad (\to E)$$

$$\frac{[x : \sigma] \quad M : \tau}{(\lambda x.M) : (\sigma \to \tau)} \qquad (\to I)$$

$$\frac{M : (\sigma_1 \wedge \sigma_2)}{M : \sigma_1} \qquad \frac{M : (\sigma_1 \wedge \sigma_2)}{M : \sigma_2} \qquad (\wedge E)$$

$$\frac{M : \sigma_1 \quad M : \sigma_2}{M : (\sigma_1 \wedge \sigma_2)} \tag{$\wedge I$}$$

$$\frac{(\lambda x.Mx) : \sigma}{M : \sigma}(if\ x\ is\ not\ free\ in\ M) \tag{η_1}$$

There are some conditions to be met when using $(\rightarrow I)$, but we will not go into more detail here. The interested reader should consult [12].

The deduction trees are built, using these rules, having the assumptions at the top of branches and the conclusions at the bottom. If Γ is a context $\{x_1 : \sigma_1, x_2 : \sigma_2, \ldots\}$ and $M : \tau$ is a type assignment statement, $\Gamma \vdash M : \tau$ if and only if there is a deduction tree having $M : \tau$ at the bottom, with all non-axiom assumptions cancelled except those in the context Γ. If Γ is empty, then we say that the deduction is a proof and that M has type τ, or $\vdash M : \tau$.

We will show the role of $\wedge$ in $TA_\lambda(\wedge, \omega)$ by means of an example.

EXAMPLE 2.5. We consider the term $\lambda x.xx$. Although in the Curry and Church type systems, this term is untypable, in the CD system, the term is given a type by the proof below.

$$\dfrac{\dfrac{\dfrac{[x : (a \wedge (a \rightarrow b))]}{x : a \rightarrow b}(\wedge E) \qquad \dfrac{[x : (a \wedge (a \rightarrow b))]}{x : a}(\wedge E)}{xx : b}(\rightarrow E)}{(\lambda x.xx) : (a \wedge (a \rightarrow b)) \rightarrow b}(\rightarrow I) cancel\ x : (a \wedge (a \rightarrow b))$$

As we can see, in the CD system, the self-application $\lambda x.xx$ is not just dismissed as untypable. It has the type given at the bottom of the tree, if we can accept that in this system the object x is both a function and an argument at once. This is a major difference between simple type systems for λ-calculus and systems containing intersection types. This difference will be visible when we present the model checking algorithm. We mention that although Kobayashi's type system uses intersection types, it is different from the CD system in the following respects: it does not have a type-constant, ω, called the universal-type, and it contains no typing rule stating that:

$$M : \omega \tag{ω}$$

Chapter 3

The Model-Checking Algorithm

We have implemented a type-based verification of HORS, using type refinement. This algorithm corresponds to the HORS model-checking part in the general framework from Figure2.3. A more detailed presentation of the verification cycle of a HORS is given in Figure3.1 below.

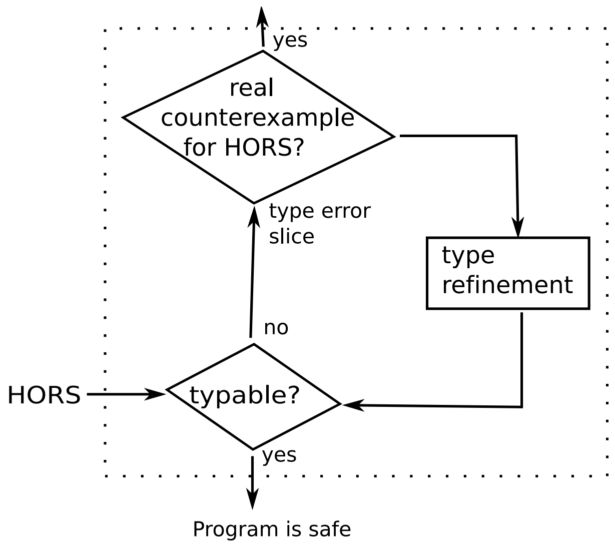

Figure 3.1: Type Based Verification of HORS

The pseudocode for the algorithm is given below. The idea of this pseudocode and the explanation of the steps is taken from [3], while the correction of Steps 2 and 3 is inspired from [14].

```
Init:
 $C$:=the initial configuration graph;
 goto Step 1;

Step 1:
 count:=0;
 while(count<MAX and an open node exists) do
 { N:=an open node;
  if $C$ can be expanded with respect to $N$ then {
   $C$:=expand($C$,$N$);
   count:=count+1}
  else {
   error_path:=the path from the root to $N$;
   raise PROPERTY_VIOLATED(error_path)}
};
 goto Step 2;

Step 2:
 $\Gamma$:=ElimTE($\Gamma_C$);
 goto Step 3;

Step 3:
 while ($\Gamma \neq F(\Gamma)$) do $\Gamma := F(\Gamma)$;
 if $S : q_0 \in \Gamma$ then
  raise PROPERTY_SATISFIED($\Gamma$)
 else
  goto Step 1;
```

Table 3.1: The Model Checking Algorithm

3.1 The Original Type System

Naoki Kobayashi has shown in [7] that the model checking problem of deciding if the tree generated by a recursion scheme is accepted by a trivial automaton can be reduced to a type-checking problem. His idea makes extensive use of intersection types. We have paid special attention to the definitions of the notions in [7] and have tried to implement them similarly in the OCaml language. The recursive definitions in the paper have been very helpful and easy to implement, because OCaml programs rely heavily on the use of recursion.

We will describe now the type system used in the type checking algorithm. Let B be a deterministic trivial automaton (Σ, Q, δ, q_0). The set of **atomic types**, ranged over by τ, is given by:

$$\tau ::= q | (\bigwedge\{\tau_1, \dots, \tau_n\}) \rightarrow \tau$$

where q can be any state in Q. We can also write $\bigwedge_{i=1}^{n}\tau_i$ for $\bigwedge\{\tau_1, \dots, \tau_n\}$. We will use $\top$ (Top) for $\bigwedge\emptyset$. $\top$ is treated as the unit on intersection, i.e. $\bigwedge\{\tau_1, \dots, \tau_n, \top\} = \bigwedge\{\tau_1, \dots, \tau_n\}$. The symbol $\wedge$ has a higher precedence than $\rightarrow$, so we will write $\bigwedge_{i=1}^{n}\tau_i \rightarrow \tau$ instead of $(\bigwedge\{\tau_1, \dots, \tau_n\}) \rightarrow \tau$. Intuitively, the type q is a refinement of the kind o: o describes trees and q describes trees accepted by the automaton from state q. The type $q_1 \wedge q_2 \rightarrow q$ is a refinement of the kind $o \rightarrow o$: it describes functions that take, as input, a tree accepted from both states q_1 and q_2, and return a tree accepted from state q.

In [3], a **type environment** is defined as a finite set of bindings of the form $x : \tau$, where x is a symbol and τ is a type; in the implementation, the variables of the environment will be terminals, non-terminal and variables. One can see that the concept "type environment" is synonym to "context" from Section 2.5, with the exception that now Γ may contain more than one bindings for each symbol. A type judgement for terms is of the form $\Gamma \vdash t : \tau$ and is synonym to the notion of "deduction" we have previously presented. We reproduce the typing rules from [3]:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\delta(q, a) = q_1 ... q_n}{\Gamma \vdash a : q_1 \rightarrow \dots \rightarrow q_n \rightarrow q}$$

$$\frac{\Gamma \vdash t_0 : \bigwedge_{i=1}^{n}\tau_i \rightarrow \tau \qquad \Gamma \vdash t_i : \tau_i \text{ (for each i=1,...,n)}}{\Gamma \vdash t_0 t_1 : \tau}$$

$$\frac{\Gamma, x : \tau_1, \dots, x : \tau_n \vdash t : \tau \qquad \text{x does not occur in } \Gamma}{\Gamma \vdash \lambda x.t : \bigwedge_{i=1}^{n}\tau_i \rightarrow \tau}$$

Kobayashi has shown(2009) that the type system is sound and complete. We will put forward a couple of theorems and the necessary details for understanding Step 3 of the algorithm. We will follow the same flow of ideas as in [3].

Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a recursion scheme. If $\Gamma \vdash \mathcal{R}(F) : \tau$ holds for every $F : \tau \in \Gamma$, then we write $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$.

A recursion scheme $\mathcal{G}$ is **well-typed**, written $\vdash_{\mathcal{B}} \mathcal{G}$, only if there exists Γ such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ and $S : q_0 \in \Gamma$. We say that $\mathcal{G}$ **is well-typed under** Γ, for such Γ.

THEOREM 3.1. *(Kobayashi (2009)). Let $\mathcal{B}$ be a deterministic trivial automaton and $\mathcal{G}$ be a recursion scheme having the same terminal symbols as $\mathcal{B}$. Then $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ if and only if $[\mathcal{G}]$ is accepted by $\mathcal{B}$.*

The next theorem outlines the type checking algorithm given by Kobayashi(2009).

THEOREM 3.2. *Let $\mathcal{F}$ be the function on type environments defined by:*

$$\mathcal{F}(\Gamma) = \{F : \tau \in \Gamma | \Gamma \vdash \mathcal{R}(F) : \tau\}$$

Let Γ_{max} be the type environment:

$$\{F : \tau | F \in dom(\mathcal{N}) \text{ and } \tau \text{ has kind } \mathcal{N}(F)\}.$$

If $\mathcal{F}^n(\Gamma_{max}) = \mathcal{F}^{n+1}(\Gamma_{max})$, for some n, then $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ if and only if $S : q_0 \in \mathcal{F}^n(\Gamma_{max})$.

Γ_{max} is finite because $dom(\mathcal{N})$ is finite and since Q is finite, for each $F \in dom(\mathcal{N})$, there are a finite number of types τ which have kind $\mathcal{N}(F)$. Because of the finiteness of Γ_{max} and the monotonicity of $\mathcal{F}$, there always exists an n in the theorem above such that $\mathcal{F}^n(\Gamma_{max}) = \mathcal{F}^{n+1}(\Gamma_{max})$.

If we suppose that the sizes of kinds and $|Q|$ are bounded above by a constant, then the number of bindings on each symbol in Γ_{max} is not only finite as above, but also bounded by a constant. This means that the algorithm is quadratic in the size of the recursion scheme. In general, the size of Γ_{max} is n-EXPONENTIAL, making the above algorithm impractical. The algorithm has n-EXPTIME complexity in the worst case and the best case.

For the special case of deterministic trivial automata, Kobayashi's algorithm can be optimized to run in $(n-1)$-EXPTIME, for $n \geq 2$. However, the algorithm always has $n-1$-EXPTIME complexity.

3.2 The Improved Type System

The model checking algorithm from [3] relies on the subtyping relation on intersection types, even if it does so only in the optimization part of the algorithm. Thus, for the optimized version of thealgorithm to be complete, a subtyping relation must be incorporated into the definition of the typing judgement. The need to add support for full subtyping to the type system has been detected in [14], by means of a counterexample. In [14], the first system that is thought to satisfy our requirements is van Bakel's essential type sytem. The subtyping relation $\leq_E$ defined by van Bakel is given by the following inductive clauses:

(1)$\forall i.1 \leq i \leq n \Rightarrow \sigma_1 \wedge \ldots \wedge \sigma_n \leq_E \sigma_i$

(2)$(\forall i.1 \leq i \leq n \Rightarrow \sigma \leq_E \sigma_i) \Rightarrow \sigma \leq_E \sigma_1 \wedge \ldots \wedge \sigma_n$

(3)$\sigma \leq_E \tau$ and $\tau \leq_E \rho \Rightarrow \sigma \leq_E \rho$

(4)$\rho \leq_E \sigma$ and $\tau \leq_E \mu \Rightarrow \sigma \rightarrow \tau \leq_E \rho \rightarrow \mu$

This relation is a very natural notion of subtyping for intersection types. The rule that needs to be added to Kobayashi's system is (bearing in mind that the typing judgement is here written $\vdash_E$):

$$\frac{}{\Gamma, x : \sigma \vdash_E x : \tau}\ \sigma \leq_E \tau$$

Although the subtyping rule is only applicable to variables, the following rule is admissible in the system (M is a term):

$$\frac{\Gamma \vdash_E M : \sigma}{\Gamma \vdash_E M : \tau}\ \sigma \leq_E \tau$$

Although van Bakel's system fulfills our requisites, the computational cost associated with it is too big. A major component of Kobayashi's algorithm is type checking the recursion rules, so we need an efficient procedure to decide the typing relation. In [14], an alternative formulation of van Bakel's system is given, that is more adequate to a recursive algorithm of type checking. This alternative system leads to an efficient recursive algorithm for type checking, which will be put forward below.

For simplicity, we assume that the types of the terminals are bound in the environment. These types are determined by the transition rules of the automaton, so for every transition $(a, q) \rightarrow q_1, \ldots, q_n$ we add the binding $a : q_1 \rightarrow \ldots \rightarrow q_n \rightarrow q$ to Γ. Our final goal is to obtain the non-terminals that have a binding $F : \tau$ in the environment and $\Gamma \vdash \mathcal{R}(F) : \tau$. In any such derivation $\Gamma \vdash \mathcal{R}(F) : \tau$, the abstraction rule is only used once. $\mathcal{R}$ is a map from the set of non-terminals to terms of the form $\lambda\tilde{x}.t$, where t is a ground-kind, purely applicative term. Hence, in each derivation, the abstraction rule is instantiated at most once, at the root of the proof tree.

The alternative intersection type system with subtyping, that we will subsequently use, is:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{Id})$$

$$\frac{\begin{array}{c}\bigvee T_i \leq \bigvee S_i (\text{for each i}) \\ \Gamma \vdash \xi : \bigvee S_1 \to \ldots \to \bigvee S_n \to \tau \\ \Gamma \vdash t_i : \theta \text{ for each } \theta \in T_i, \text{ for each } i\end{array}}{\Gamma \vdash \xi t_1 \ldots t_n : \tau} \text{(L-Apps)}$$

$$\frac{\Gamma, x_1 : \bigvee S_1, \ldots, x_n : \bigvee S_n \vdash t : \tau}{\Gamma \vdash \lambda x_1 \ldots x_n . t : \bigvee S_1 \to \ldots \to \bigvee S_n \to \tau} \text{(L-Abs)}$$

We make some comments related to this system. We only allow subtyping at application, and not at the axiom. We defined a long abstraction clause to reflect long-application. We mention that in each instance of the long-application rule, there is always a choice of type for ξ and choice of types for each t_i that are both subformulae of types in the goal judgement and correctly related by the subtyping order. The notation $x : \bigvee S$ stands for $x : s_1, \ldots, x : s_m$, in the type environment, where $S = \{x_1, \ldots, x_m\}$.

In [14], it is proven that this system preserves types under substitution and reduction. We reproduce from the same paper a deterministic algorithm for type checking, that we have used in our implementation.

Given a type environment Γ and a purely applicative term t, we compute the set of types $\mathcal{T}_\Gamma(t)$ derivable for t under Γ, using the following rules, in a recursive manner.

$$\mathcal{T}_\Gamma(\xi) := \{\tau | \xi : \tau \in \Gamma\} \tag{3.1}$$

$$\mathcal{T}_\Gamma(\xi t_1 \ldots t_n) := \{\sigma | \xi : \bigwedge S_1 \to \ldots \to \bigwedge S_n \to \sigma \in \Gamma \wedge \forall i . \bigwedge \mathcal{T}_\Gamma(t_i) \leq \bigwedge S_i\} \tag{3.2}$$

The first rule is a special case of the second one and will act as the base case for the recursive type checking algorithm. The two rules give indeed the set of types derivable for a given term, according to [14] .

LEMMA 3.3. *Given type environment* Γ *and purely applicative term* t*,* $\mathcal{T}_\Gamma(t) = \{\tau | \Gamma \vdash t : \tau\}$

If an efficient procedure for deciding subtyping is found, type-checking is also efficient. Assuming that the time to check the subtype relation is given, in the worst case, by the constant C, then the worst-case time complexity for constructing $\mathcal{T}_\Gamma(t)$ is $\mathcal{O}(|t|A|\Gamma|C)$, where A is the largest arity of any binding in Γ.

This optimised algorithm with subtyping is proved to be correct in Theorem3.4 [14] , by making only minor corrections to the correctness proof of Kobayashi for the unoptimised algorithm in [3]. We will also discuss Kobayashi's correctness theorem in Chapter 4.

THEOREM 3.4. *(Correctness). Given a recursion scheme $\mathcal{G}$ and a deterministic, trivial, tree automaton $\mathcal{A}$, the algorithm eventually terminates. Moreover, its output is Γ just if $[\mathcal{G}] \in \mathcal{L}(\mathcal{A})$ and is an error path otherwise.*

3.3 Step 1: Constructing the Configuration Graph

The details of all the steps in the algorithm are presented in [3], and we will adhere to the same presentation here. The purpose of Steps 1 and 2 is to find a feasible candidate for the type environment, which should be passed to Step 3. A visible candidate is Γ_{max} described in Section 3.1, but that is not the best choice, because the size of Γ_{max} is n-exponential. For example, assume that the number of automaton states is 2, the states being q_0 and q_1. Then the number of intersection types for each kind grows up very quickly.

kinds	the number of intersection types
$o \to o$	$2^2 \times 2 = 8$
$(o \to o) \to o$	$2^8 \times 2 = 512$
$((o \to o) \to o) \to o$	$2^{512} \times 2 = 2^{513}$

Table 3.2: Exponential growth of number of intersection types

For example, for the kind $o \to o$, the first o is an intersection type with symbols from $S = \{q_0, q_1\}$, hence it can have as many values as the power set of S has(2^2 values). The last o can have 2 values, so finally $o \to o$ could be one of 8 values.

In [7], another type-checking algorithm has been advanced, but it would be impractical even for order-3 recursion schemes. The current algorithm relies on a key observation: the usage patterns of each function are limited, so the size of the type environment passed to Step 3 can be much smaller than the size of Γ_{max}. To construct such a good type environment, in Step 1, we expand the given recursion scheme a finite number of steps (the constant MAX in the pseudocode in Table 3), and construct a configuration graph. This graph represents the traces of the execution and in Step 2 it is used to extract type information from its nodes.

A **configuration graph** is a labeled directed graph, constructed by applying the expansion rules defined below. The initial configuration graph consists of a single node(the root node), labeled by $\langle S, q_0, open\rangle$. Let C be a configuration graph, and N is a node of C labeled by $\langle t, q, open\rangle$. Then, the result of expanding C with respect to N is the graph C', which is obtained from C, by replacing the flag of N with *closed* and adding nodes and directed edges, in the following way:

(1) If $t = at_1 \dots t_m$ and $\delta(q, a) = q_1 \dots q_m$:

If a node labeled with $\langle t_i, q_i, f_i\rangle$(it does not matter if f_i is open or closed) does not exist, add a new node $\langle t_i, q_i, \mathit{open}\rangle$. Add directed edges from N to the nodes labeled by $\langle t_i, q_i, f_i\rangle$ and label the edge with i, for $i \in \{1, \dots, n\}$.

If $t = a$, where a has arity 0, we only need to change the flag of node N to closed, as this node cannot be further expanded.

(2)If $t = Ft_1 \dots t_m$ and $R(F) = \lambda x_1 \dots x_m.u$:

If a node labeled with $\langle [t_1/x_1, \dots, t_m/x_m]u, q, f\rangle$ does not exist, add a new node $\langle [t_1/x_1, \dots, t_m/x_m]u, q, \mathit{open}\rangle$. Add a directed edge from N to the node labeled by $\langle [t_1/x_1, \dots, t_m/x_m]u, q, f\rangle$ and label the edge with 0.

The configuration graph of a recursion scheme is obtained from the initial configuration graph(containing one node) by applying the expansion operations, possibly an infinite number of times. A configuration graph is finitely expanded if it is obtained by a finite number of expansions, and it is closed if it contains no open nodes. For a given recursion scheme, a closed configuration graph is uniquely determined(up to an isomorphism) and it may be an infinite graph. When representing a graph, we will use open brackets to denote an open node and closed brackets to denote a closed node.

Figure 3.2 shows a configuration graph for the recursion scheme G and the automaton B in EXAMPLE 3.1.

EXAMPLE 3.1. Let G be the recursion scheme (Σ, N, R, S), where:

$\Sigma = \{b \to o \to o \to o, r \to o \to o, c \to o \to o, e \to o\}$,
$N = \{S \to o, F \to o \to o\}$,
$R = \{S \to Fe, F \to \lambda k.b(ck)(r(Fk))\}$.

Let B be the automaton $(\Sigma, \{q_0, q_1\}, \delta, q_0)$, where:

$\Sigma = \{b \to o \to o \to o, r \to o \to o, c \to o \to o, e \to o\}$,
$\delta(q_0, b) = q_0q_0 \qquad \delta(q_1, b) = q_1q_1 \qquad \delta(q_1, e) = \epsilon$
$\delta(q_0, r) = q_0 \qquad \delta(q_0, c) = q_1$

This is a finitely expanded, closed configuration graph. The edge label 0 is omitted in the figure.

A graph cannot be expanded with respect to N if N is labeled by $\langle at_1 \dots t_m, q, \mathit{open}\rangle$ but $\delta(q, a)$ is undefined. In that case, the property is not satisfied and the path from the root to the node is output as an error

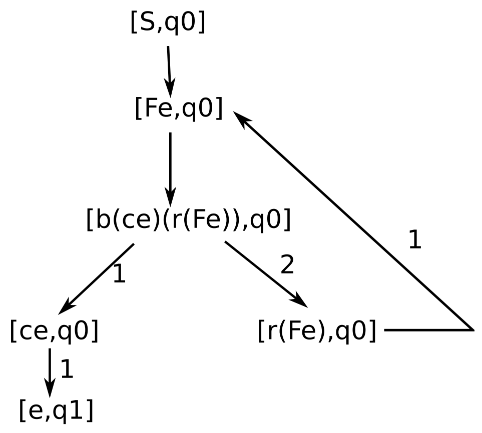

Figure 3.2: Configuration Graph

path. The selection of node N on line 3 of Step 1 is fair, i.e. every open node is eventually selected. This is achieved by using a FIFO queue of open nodes in the implementation. We will reproduce from [3] a configuration graph that contains an error path.

EXAMPLE 3.2. Consider the recursion scheme G_0 in EXAMPLE 2.1 and the automaton B_0 in EXAMPLE 2.2. The configuration graph obtained by several expansions is shown in Figure 3.3. The node at the bottom of the figure cannot be expanded, because $\delta(q_1, a)$ is undefined. The error path is 0:0:2:1:0 (where ":" means string concatenation). This path 2:1 (obtained by ignoring 0) of the corresponding value tree in Figure 2.1 is labeled by aba, which violates the property expressed by B_0, that a should not occur after b. This is also why the run tree in Figure 2.2 blocks when it reads a in state q_1.

3.4 Step 2: Extracting Type Information

Step 2 is the most important part of the algorithm: we are now extracting type information from a configuration graph.

The following algorithm for extracting type information has been inspired from the proof of the completeness of Naoki Kobayashi and Luke Ong's type system for the modal μ-calculus model checking of recursion schemes (Kobayashi and Ong 2009). A term t' is called a **prefix** of t if t is of the form $t'\tilde{s}$, where $\tilde{s}$ is a possibly empty sequence of terms. So a prefix of t consists of the first symbol of t (may it be a terminal or non-terminal), followed by zero, one or more of the arguments given to this first symbol, present in the

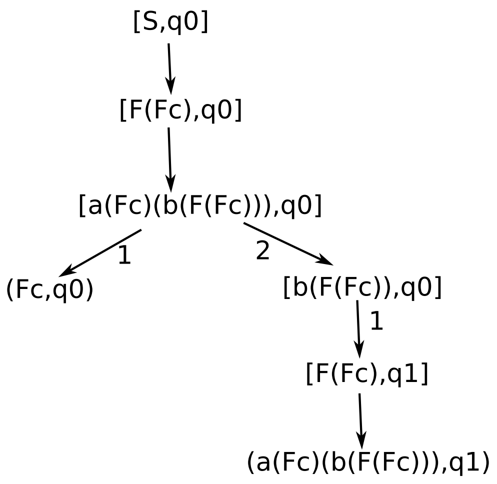

Figure 3.3: A Configuration Graph Containing an Error Node

term t. For each node N labeled with $\langle t, q, f\rangle$, we assign a type $\tau_{t,N}$ to each prefix of t. The type $\tau_{t,N}$ is defined by induction on the kind of t in this way:

(1)If t has kind o, N must have a label of the form $\langle t, q, f\rangle$, so we define $\tau_{t,N} := q$.

(2)If t has kind $k_1 \to k_2$, N must have a label of the form $\langle ts_0\tilde{s}, q, f\rangle$, where s_0 and ts_0 have kinds k_1 and k_2 respectively. Let $\{N_1, \dots, N_m\}$ be the set of nodes that are reachable from N, have labels of the form $\langle s_0\tilde{u}, q', f\rangle$(i.e. s_0 is a prefix of the term in the node) and s_0 originates from that of the node N. If s_0 occurs in an open node reachable from N(including N itself), then let $S = \{\alpha, \tau_{s_0,N_1}, \dots, \tau_{s_0,N_m}\}$, where α is a fresh type variable. Otherwise, if s_0 appears only as a prefix of terms in closed nodes, let $S = \{\tau_{s_0,N_1}, \dots, \tau_{s_0,N_m}\}$. Finally, we define $\tau_{t,N} := (\bigwedge S) \to \tau_{s_0,N}$. To obtain the types of all prefixes, this algorithm needs to be applied recursively.

If we are given a configuration graph C, we define the type environment Γ_C by:

$\Gamma_C := \{F : \tau_{F,N} | N$ has a label of the form $\langle Ft_1 \dots t_m, q, f\rangle\}$.

EXAMPLE 3.3. Consider the configuration graph C of Figure 3.2. Let N_0, N_1 and N_2 be the nodes labeled by $[S, q0]$, $[Fe, q0]$ and $[e, q1]$ respectively. Then,

$$\tau_{S,N_0} = q0$$

$$\tau_{F,N_1} = \bigwedge\{\tau_{e,N_2}\} \to \tau_{Fe,N_1} = q1 \to q0$$

Γ_C is given by:

$$\{S : q0, F : q1 \to q0\}$$

The theorem underlying the algorithm of Step2 is the following:

THEOREM 3.5. *Suppose that* $[G]$ *is accepted by* B*. Let* C *be a configuration graph of* G *over* B*. If* C *is closed, then* G *is well-typed under* Γ_C*.*

In general, infinitely many expansions are needed to get a closed configuration graph C, so the theorem above cannot be directly used in the algorithm. Still, from the definition of Γ_C, we see that if we expand the graph a finite number of times, we get an approximation of Γ_C.

LEMMA 3.6. *Let* C' *be a finitely expanded configuration graph of* G *over* B*, and* C *be the closed configuration graph. Then, there is a substitution* θ *for type variables such that* $\theta\Gamma_{C'} = \Gamma_C$*.*

In Step2 of the algorithm, Γ_C is first computed from the current(finitely expanded) configuration graph C. The type variables (α_i, that denote an unknown type, but a type which may become known if we further expand the graph) are then removed from Γ_C by the operation *ElimTE* defined below.

We first define the function *Elim*, which takes an atomic type as input and returns a set of closed atomic types.

$Elim(q) = \{q\}$
$Elim(\alpha) = \emptyset$
$Elim(\bigwedge_{i=1}^{m} \tau_i \to \tau = \bigwedge_{i=1}^{n} \tau'_i \to \tau' | \tau'_i \in Elim'(\tau_i), \tau' \in Elim(\tau)\}$

$$Elim'(\tau) = \begin{cases} Elim(\tau) \cup \{\top\} & \text{if } \tau \text{ contains a type variable;} \\ Elim(\tau) & \text{otherwise.} \end{cases}$$

The auxiliary function $Elim'$ may return a set consisting of closed atomic types and $\top$. In the definition of $Elim(\bigwedge_{i=1}^{m} \tau_i \to \tau)$, if an argument type τ_i contains a type variable, *Elim* may choose $\top$ from $Elim'(\tau_i)$. Intuitively, this is because τ_i may express incomplete information that should be ignored.

EXAMPLE 3.4. $Elim((q_0 \wedge \alpha \to q_1) \wedge q_2 \to q)$
$= \{\tau_1 \wedge \tau_2 \to q | \tau_1 \in Elim'(q_0 \wedge \alpha \to q_1), \tau_2 \in Elim'(q_2)\}$
$= \{\tau_1 \wedge q_2 \to q | \tau_1 \in \{q_0 \to q_1, \top\}\}$
$= \{(q_0 \to q_1) \wedge q_2 \to q, q_2 \to q\}$

$ElimTE(\Gamma)$ is a pointwise extension of the operation *Elim*, defined by:

$$ElimTE(\Gamma) = \bigcup_{F:\tau \in \Gamma} \{F : \tau' | \tau' \in Elim(\tau)\}$$

We mention, as in [3], that the completeness would be lost if we simply replaced all the type variables in Γ with $\top$, instead of applying the operation *ElimTE*. For example, suppose $\Gamma = \{(q_0 \wedge \alpha \to q_1) \wedge q_2 \to q\}$. If we replace α with $\top$, we obtain the type environment $\{(q_0 \to q_1) \wedge q_2 \to q\}$. However, the actual type of F in the recursion scheme may be $q_2 \to q$.

Recall the type environment Γ_C from EXAMPLE 3.2. Because it contains no type variables, after applying *ElimTE*, we obtain:

$$ElimTE(\Gamma_C) = \Gamma_C = \{S : q0, F : q1 \to q0\}$$

3.5 Step 3: Deciding if $\mathcal{G}$ is Well-typed

Step 3 takes a type environment Γ as an input, and checks if there exists a subset Γ' of Γ such that $\vdash_B \mathcal{G} : \Gamma'$ and $S : q_0 \in \Gamma'$. In the pseudo code algorithm, F is the function on type environments defined in THEOREM 3.2.

The following theorem guarantees the corectness of the algorithm.

THEOREM 3.7. *Let F be the function on type environments as defined in THEOREM 3.2. Then $F^0(\Gamma)(=\Gamma), F^1(\Gamma), F^2(\Gamma), \ldots$ is a decreasing sequence, i.e.*

$$F^0(\Gamma) \supseteq F^1(\Gamma) \supseteq F^2(\Gamma) \supseteq \ldots$$

and there exists n such that $F^n(\Gamma) = F^{n+1}(\Gamma)$. Furthermore, $\Gamma' = F^n(\Gamma)$ is the largest set such that $\vdash_B G : \Gamma'$ and $S : q_0 \in \Gamma'$.

To find out if $\vdash_B G : \Gamma'$, we apply the algorithm from Section 3.2, with the mention that for the unoptimized version of the algorithm we do not need the notion of subtyping and the rules 3.1 and 3.2 become (the only change is in fact in the second rule):

$$\mathcal{T}_\Gamma(\xi) := \{\tau | \xi : \tau \in \Gamma\} \tag{3.3}$$

$$\mathcal{T}_\Gamma(\xi t_1 \ldots t_n) := \{\sigma | \xi : \bigwedge S_1 \to \ldots \to \bigwedge S_n \to \sigma \in \Gamma \wedge \forall i. \mathcal{T}_\Gamma(t_i) \supseteq S_i\} \tag{3.4}$$

3.6 Optimizations

The operation *Elim* from Step 2 takes an atomic type as input and, by using a cartesian product of two sets, it returns a set of closed atomic types. This could cause a combinatorial explosion of the number of atomic types.

In [3], Kobayashi presents some optimizations that reduce the number of atomic types. They are the following:

(1) Use the canonical representation of function types. A function type $\bigwedge_{i=1}^{n} \tau_i \rightarrow \tau$ is canonical if for each i, there is no $j \neq i$ such that $\tau_j \leq \tau_i$. The subtype relation $\tau_j \leq \tau_i$ means that τ_j represents a more general type than τ_i, so that a value of type τ_j may be used as a value of type τ_i.

(2) In the definition of $Elim(\bigwedge_{i=1}^{n} \tau_i \rightarrow \tau)$ choose $\top$ as τ_i' only if τ_i is subsumed by τ_j for some j, i.e. if there is a substitution θ such that $\theta\tau_i = \tau_i'$ for some j.

Unfortunately, this optimization does not preserve the completeness of the algorithm. This is proved by means of a counterexample in [14]. The problem appeared because the canonical form of function types depends on the subtyping relation, but the usual subtyping relation is not incorporated in the typing judgement. The problem is solved by adding support for full subtyping to the type system. This is why we should use the improved type system and the algorithm from Section 3.2 instead of the one in Section 3.1.

3.7 Correctness of the Algorithm

The soundness and completeness of the unoptimized algorithm has been proven in [3].

THEOREM 3.8. *Given a recursion scheme $\mathcal{G}$ and a deterministic trivial automaton $\mathcal{B}$, the algorithm eventually terminates. Furthermore, if the algorithm outputs a type environment Γ, then $\mathcal{G}$is well-typed under Γ (hence $[\mathcal{G}]$ is accepted by $\mathcal{B}$). If the algorithm reports an error path, then $[\mathcal{G}]$ is not accepted by $\mathcal{B}$.*

Chapter 4

Implementation Features

We have implemented a model checker for recursion schemes. The source code can be found at http://bitbucket.org/ligianicoletanistor/MScProject/. The implementation is mostly based on the algorithm in Chapter 3, except the following points:

•The implementation uses trees for representing configuration graphs. Thus, the same node may be expanded multiple times. We have used trees for ease of implementation and because we think that there is not much to be gained by representing the configuraton graph as a graph, as opposed to a tree. Our thought is that the computation is likely to terminate (i.e. exit the loop) before the advantage of a graph representation is realised. These are only guesses though that will need to be proved in the future. In Chapter5 we have proposed an alternative representation of trees-Colin Stirling's pointer machine, which might improve the performance of the algorithm.

•The number of iterations in Step 1(the value of MAX in the algorithm) is set to 50.

•The two optimizations discussed in Section 3.6 are not applied, but they should be implemented in a future version of the algorithm. We have implemented the subtype relation, to decide if type τ_1 is a subtype of type τ_2, but we still need to decide/prove which is the role of $\top$ in the subtype relation. This is left for future work.

The files of the implementation are: **hors.ml**, **automata.ml**, **mygraph.ml**, **myset.ml**, **horsLexer.mll**, **horsParser.mly** and **calc.ml**. Each of the first four files is a module which groups specific data structures and related functions together, related to higher order recursion schemes, to automata, to the configuration graph and to the binding environments (regarded as sets) respectively. The next two files are used for reading and parsing the input. The last file, **calc.ml**, is where all the modules interact, to form the algorithm. **calc.ml** can be thought of as the skeleton of the algorithm, the main part of the implementation.

The input, a higher order recursion scheme and an automaton, is given in a file. The tokens appearing in this file are identified using **horsLexer.ml**. We have used the tool *ocamllex* for generating the file **horsLexer.ml** from **horsLexer.mll**. The list of tokens is then parsed using **horsParser.ml** and **horsParser.mli**. We have used the tool *ocamlyacc* to generate the files **horsParser.ml** and **horsParser.mli** from **horsParser.mly**. After having processed the input file, we have the structures representing a HORS and an automaton and we can start the algorithm.

4.1 Processing the input file

The structure of the input file is: Σ-the map from terminals to kinds; $\mathcal{N}$-the map from non-terminals to kinds; $\mathcal{R}$-the map from non-terminals to λ-terms; the number of states in the automaton; the transition rules of the automaton. Every mapping is written on a new line and every transition is also written on a separate line. Terminals and variables are represented by lowercase letters, non-terminals by uppercase letters and automaton states by integers. This information is used when separating the input into tokens. The rules for the formation of tokens, taken from **horsLexer.mll** are given below:

```
rule token = parse
  |[' ' '\t']{ token lexbuf }
  |['\n'] {EOL}
  |['0'-'9']+ as lxm { INT(int_of_string lxm) }
  |['a'-'n']|['p'-'z'] as s{ VAR_TERMINAL(s)}
  |['A'-'Z'] as t{NONTERMINAL(t)}
  |':'{OF_TYPE}
  |'-'{REWRITES_TO}
  |'.'{BOUNDED_TO}
  |'('{LBRACKET}
  |')'{RBRACKET}
  |'o'{OMICRON}
  |eof {raise Eof}
```

After this, the tokens are parsed. Now, we have to make the connection with the “*.ml“ files, to be able to access in our implementation of the algorithm the resulting structures given by the parser. For example, in **horsParser.mly**,we create a list of pairs of the form (terminal, type) that will be used to form a map from terminals to types. The (selective) code is below. The first line tells *ocamlyacc* which is the starting point of the parsing. We only need to look at 'main', as the other starting points are for reading the rest of the input file. When constructing the rule for 'myType', we had to pay attention that types are associative to the right. What is seen in the curly brackets on each line is the type attributed to the specific rule, the type that is returned when that rule is applied.

```
%start main main2 main3 autnr delta
%type <Hors.pair_list>  main
main:
  |sigma {$1};
sigma:
  |VAR_TERMINAL OF_TYPE myType EOL
     sigma{Hors.C($1)::Hors.K($3)::$5}
  |EOL {[]};
myType:
  |OMICRON {Hors.Omicron}
  |OMICRON myType {Hors.Arrow(Hors.Omicron,$2)}
  |LBRACKET myType RBRACKET myType {Hors.Arrow($2,$4)}
  |LBRACKET myType RBRACKET {$2};
```

After having run the parser, we will have two lists of pairs for Σ and $\mathcal{N}$, one list of triples for $\mathcal{R}$, an integer for the number of states of the automaton and an element of type **Automata.d** for the transition rules of the automaton. The type of the elements returned by the parser is seen in the following lines from **horsParser.mly**:

```
%type <Hors.pair_list>  main
%type <Hors.pair_list> main2
%type <Hors.triple_list> main3
%type <int> autnr
%type <Automata.d> delta
```

The function **reading** below is the part from **calc.ml** where the reading is implemented.

```
let reading () =
try
  let in_channel = open_in "examples/twofiles.txt" in
  let lexbuf = Lexing.from_channel in_channel in
    while true
    do
      let result = HorsParser.main HorsLexer.token lexbuf in
      let resultfinal=Hors.get_pair_list result in
```

```
          List.iter hsadd resultfinal;
        let result2 = HorsParser.main2 HorsLexer.token lexbuf in
        let resultfinal=Hors.get_pair_list result2 in
          List.iter hnadd resultfinal;
        let result3 = HorsParser.main3 HorsLexer.token lexbuf in
        let resultfinal=Hors.get_triple_list result3 in
          List.iter hradd resultfinal;
        let result4 = HorsParser.autnr HorsLexer.token lexbuf in
          nrs:=result4;
          alfa:=result4+1;
        let result5 = HorsParser.delta HorsLexer.token lexbuf in
        let resultfinal= Automata.get_delta_list result5 in
          List.iter dadd resultfinal;
      done
with HorsLexer.Eof -> output_string handle "␣"
```

4.2 Implementation of Step 1

As previously stated, we have used a tree to implement the configuration graph. However, technically our structure is a directed graph, but we treat it as if it was a tree (i.e. all the directed edges go forward). We have used the graph library for OCaml, namely **ocamlgraph**.

According to [15], **ocamlgraph** provides an easy to use graph data structure, but also persistent(imutable) and imperative(mutable) graph implementations. It has implementations for graphs that are directed or not, that have labels for vertices or for edges, that have abstract types for vertices, etc. These implementations are written as functors: one gives the types of vertices labels, edge labels, etc. and one gets the data structure as a result. We have chosen to use **ocamlgraph** because it provides several classic operations and algorithms over graphs, hopefully more efficient than if we had written them ourselves.

In the project, we have used

module G=Imperative.Digraph.ConcreteLabeled(V)(E).

This is a mutable directed graph, for which the labels of vertices and edges are concrete types (**node** and **integer** respectively). The type **node** is defined

type node={f:Hors.term; t:literm; q:int; nnode:int; oc:int ref}

where **Hors.term** is a type defined in the file **hors.ml**, **literm** is a list of terms, **q** represents the state of the automaton, **nnode** is the unique number of the current node, and **oc** is 0 if the node is open, or 1 if the node is closed.

The type **node** is exactly the implementation of a node from Section 3.3. In the file **hors.ml**, a type **term** is defined:

```
type term=
    Symbol of char*llint
  | App of term*term;;
```

Every symbol of a term has a list of pairs of integers attached to it. This list represents the history of the symbol. This history is formed when the configuration graph is expanded. The history can be modified only when expanding a node the term of which starts with a non-terminal, because this is when new symbols are introduced.

The rules for creating the history of a symbol are: if the character c appears more than once in the term t of the node numbered with n, then to the list of the first appearance of c the pair (1,n) will be added, and to the list of the second appearance, the pair (2,n) will be added, and so on. If the character c appears only once, then its list is unchanged. Whenever a new character (that was not in the term before) coming from the application of a rule in $\mathcal{R}$ is introduced in the node numbered with n, its initial list is [(0,n)]. When we are expanding a node of which the term starts with a terminal, we simply copy the history of each symbol to the resulting nodes. The history of the symbols is very important in Step 2 because it tells us if a prefix of the term in node M originates from the term of node N. We will write about this when discussing Step 2.

The function **expandT** expands the node **n**, which contains a term starting with a terminal. It is used when expanding the configuration graph, Case 1. **g** is the graph, **lonodes** and **lcnodes** are the current lists of open/closed nodes, **lstates** will be used when calling the function **create_nodes**. The nodes resulting from this expansion will be inserted in the graph.

```
let expandT g lonodes lcnodes n lstates =
let lterms= n.t in
let list_nodes=create_nodes lterms lstates in
if (List.length list_nodes)>0 then
 let finsert_node (son,nr)=
     insert_node n g lonodes lcnodes son nr in
 List.iter finsert_node list_nodes;;
```

The function **create_nodes** returns a list of nodes created from the list $terms_i$,$i \in \{1 \ldots m\}$ and the list $states_i$,$i \in \{1 \ldots m\}$. Every such node is open and is of the form $\langle t_i, q_i, open\rangle$, as in Case 2 of Step 1.

```
let create_nodes terms states =
let rec loop lterms lstates i=
match lterms with
  te:: lrest->
    (match lstates with
       qe::lsrest->
          let tt=decompose te in
          ({fn=(first_term te);tn=tt;qn=qe},i)::
            (loop lrest lsrest (i+1))
       |[]->[];)
|[]->[]
in loop terms states 1;;
```

The function **expandN** below expands the node **n**, which contains a term starting with a non-terminal. It is used when expanding the configuration graph, Case 2. **g** is the graph, **lonodes** and **lcnodes** are the current lists of open/closed nodes, **lvars** and **te** will be used when calling the function **compose_nodes**. The node resulting from this expansion will be inserted in the graph.

```
let expandN g lonodes lcnodes n lvars te =
let lterms = n.t in
let myt = compose_nodes (lvars,te) lterms in
let k=(!num_node +1) in
let newterm=Hors.compute_history myt k in
let newnode={fn=(first_term newterm);
       tn=(decompose newterm);qn=n.q} in
(*The node 'newnode' is inserted in the graph*)
  insert_node n g lonodes lcnodes newnode 0 ;;
```

The function **compose_nodes** takes as arguments a pair representing a λ-term and a list **l** of terms with which we have to substitute the variables in the λ-term, in the right order. This function will return the new term, after the substitution has been made.

```
let rec compose_nodes (var,ter) l=
match l with
  le::lrest->
    (match var with
       vare::varrest->
         let newterm=compose_node (vare,ter) le  in
         compose_nodes (varrest,newterm) lrest
       |[] -> ter;)
  |[]-> ter;;
```

The fragment corresponding to Step 1 in **calc.ml** is written below. The graph is expanded 50 times. We keep a FIFO list of open nodes, from which we take the next node to be expanded. We check if the first symbol of the term in this node is a terminal or a non-terminal and we call **expandT** or **expandN** accordingly. In case an error node is reached(a node which cannot be expanded anymore) we use the Dijkstra algorithm implemented in **ocamlgraph** to print the path from the root to the node, as an error path. In this case, the program exits.

```
  let count=ref 0 in
    while ((!count)<50 &&
        ((List.length (!lonodes)) >0))
    do
      let n=List.hd (!lonodes) in
        lonodes:= List.tl (!lonodes);
      let myn=M.first_symbol n.M.f in
(*if myn is lowercase, then it is a terminal*)
        if Char.lowercase myn==myn
        then
          try
            let lstates=Delta.find
                {ST.i=n.M.q;ST.c=myn} (!ad) in
             M.expandT g lonodes lcnodes n lstates
(*an error node has been reached*)
          with Not_found ->
            let i2=
            MyG.V.create({M.f=n.M.f;M.t=n.M.t;
                   M.q=n.M.q;M.nnode=0;M.oc=ref 0})in
            let (h2,ljj)=
              (Dijkstra.shortest_path g i1 i2) in
            let print_edge ed=
              let temp_node=(MyG.E.dst ed) in
              output_string handle
               ("["^Hors.string_of_terms temp_node.M.t^"]␣␣")
            in List.iter print_edge h2;
            exit 0
(*character myn is uppercase, so it is a non-terminal*)
        else
          let res= HorsR.find myn (!hr) in
            M.expandN g lonodes lcnodes
                 n (res.Hors.v) (res.Hors.lambdat) ;
          count:=(!count)+1
    done;
```

4.3 Implementation of Step 2

This is the step when type information is extracted from the configuration graph. The function that extracts this type information is **assignTypePrefix** and is found in the file **mygraph.ml**. Regarding the arguments given to this function, the prefix **t** of node **n** in the graph **g** is given a type. The pair (prefix, type) is introduced in the map **taumap**, if it is not already there. **alfa** is an integer used for identifying each new type variables and **ok** records if the following term after the prefix appears in an open node. The function **assignTypePrefix** implements exactly the theory presented in Section 3.4, up to the point when the *Elim* function is applied. In the second case of the type assignment algorithm, when t has kind $k_1 \to k_2$, if $\{N_1, \ldots, N_m\}$ is the set of nodes that are reachable from N and these nodes have labels of the form $\langle s_0\tilde{u}, q', f\rangle$, we must be sure that s_0 originates from that of the node N. This is done by using the history of each symbol. When we have found s_0 as a prefix of the term in one of the nodes N_i, we want to check that this s_0 comes from the one in node N. We will denote the s_0 in node N_i by $s_0^{N_i}$ and the one in the original node N by s_0^N. $s_0^{N_i}$ originates from s_0^N if all the symbols of the former originate from the symbols of the latter. But we have recorded a history for each symbol when we have constructed the configuration graph. This is where we will use it. A symbol c, having attached the list of pairs of integers(its history) l, originates from a symbol c', having attached its history l', if and only if $c == c'$ and the history of c' is a prefix of the history of c. For example, $('a', [(1,2);(2,3)])$ originates from the symbol $('a', [(1,2);(2,3);(2,5)])$. On the other hand, $('b', [(0,4);(2;6)])$ does not originate from the symbol $('b', [(0,4);(3,7)])$ because even though they contain the same character, the list $[(0,4);(2;6)]$ is not a prefix of the list $[(0,4);(3,7)]$. The implementation of the type assignment procedure is reproduced below.

```
let rec assignTypePrefix g n t taumap alfa ok =
let mytype=ref( is_in_map taumap n t) in
if (value_mytype !mytype ) then
(*the type of t was not in taumap*)
  if (List.length t==((List.length n.t) +1) ) then
(*t is the whole term on n*)
  begin
    taumap:= TauMap.add
```

```
        {TermNode.taut=(n.f)::(n.t) ;
        TermNode.taun=n.nnode} (I(n.q)) (!taumap);
      mytype:=I(n.q);
    end
    else
(*t is a proper prefix *)
    begin
    let lent=(List.length t)-1 in
    let s0=List.nth n.t lent in
    let li=ref [] in
    let reach=ref [] in
    reachableNodes g n reach;
    let len=(List.length !reach)-1 in
    for i=0 to len
    do
      let elem=(List.nth !reach i) in
      if (!(elem.oc)==1) then
      begin
(*checks if s0 is a prefix of the term in elem*)
        let mystart=(starts_with_s0 elem s0) in
        if mystart>(-1) then
        begin
        let newt=(first_term s0)::(decompose s0 ) in
        if (is_s0 n (term_of_list t) ) then
        begin
          mytype:=I(n.q);
          if (is_type_in_list !mytype !li)==false then
              li:=(!mytype)::(!li);
        end
        else
        begin
        let type2check=
        (assignTypePrefix g elem newt taumap  alfa (ref 0) ) in
          if (is_type_in_list type2check !li)==false then
                li:=type2check::(!li);
          end
        end
      end
      else
(*here, elem is an open node*)
        if (contains_s0 elem s0 ) then ok:=1;
```

```
   done;
   let lastt=List.append t [s0] in
   let ki=(assignTypePrefix g n lastt taumap  alfa (ref 0) ) in
   if (!ok==1)||(!(n.oc)==0) then
   begin
(*we add a type variable*)
      alfa:=!alfa+1;
      li:=I(!alfa)::(!li);
   end;
   if (List.length !li)==0 then mytype:=A([I(-1)],ki)
   else  mytype:=A(!li,ki);
   ok:=0;
   taumap:= TauMap.add
   {TermNode.taut=t ;TermNode.taun=n.nnode} (!mytype) (!taumap);
end;
!mytype;;
```

The function above is applied to all prefixes of all nodes. We perform a breath first search in the configuration graph and gather all the nodes. We then successively apply the function **assignTypePrefix** to all the prefixes of the nodes. Knowing that the term in any node is of the form $st_1t_2 \dots t_n$, there is a right way to do this, described next. We have to remember that S, the start symbol of the input HORS is of ground type. Because of type conservation when applying the rewriting rules of the HORS (which is actually a higher order grammar) , any term in any node is also of ground type. This argument helps us understand that the term in any node is composed from the first symbol (a terminal or a non-terminal), followed by the right number (all) of arguments. So, if the node is $\langle st_1t_2 \dots t_n, q_i, f\rangle$, where f could be open or closed, the first thing that we can say is that $st_1t_2 \dots t_n$ has type q_i. Then, we apply the function **assignTypePrefix** to the prefix $st_1t_2 \dots t_{n-1}$, then to $st_1t_2 \dots t_{n-2}$, and so on. The last prefix for which we will find the type is s itself. This way, after having examined all the nodes of the configuration graph, we will know the types (which will contain type variables) for all the non-terminals. To eliminate the type variables, we apply the *Elim* function from the paper. This function needs to be optimized in the future, for it to properly work on bigger input example. The implementation of the *Elim* function is:

```
 let rec elim mytype nr=
match mytype with
  M.I(i)->
         if i>nr then  SetTypes.empty
```

```
        else
        begin
          let st=ref SetTypes.empty in
          st:=SetTypes.add mytype (!st);
          !st
        end
  |M.A(li,ty)->
        let len=List.length li in
        let lnew_elim=ref [] in
        for i=0 to (len-1)
        do
          let st=ref SetTypes.empty in
          let inter_type=(List.nth li i) in
          st:=SetTypes.union (elim inter_type nr) (!st);
          if (is_without_variables inter_type nr)==false then
              st:=SetTypes.add (M.I(-1)) (!st);
          lnew_elim:=List.append (!lnew_elim) [!st];
        done;
        let st=elim ty nr in
             lnew_elim:=List.append (!lnew_elim) [st];
        let l=ref SetListTypes.empty in
              l:=SetListTypes.add [] !l;
        for j=0 to (List.length !lnew_elim)-1
        do
          l:=cartesian_product !l (List.nth !lnew_elim j);
        done;
        let st=ref SetTypes.empty in
        let f x=
          st:=SetTypes.add (atomictype_of_listtypes x) !st in
        SetListTypes.iter f !l;
        !st;;
```

After the function **elim** is applied, we need to remove the element $\top$ from the types where it appears too many times. $\top$ is treated as the unit on intersection, so it has to appear in the final type only when the intersection type is the empty set. The elimination of $\top$ and then the replacing of empty lists representing empty intersection types, with $\top$, has been done in the functions **removeT** and **replaceEmptyList**, which can be found in the file **myset.ml**.

At the end of Step 2, we have the environment Γ_C, containing the bindings for all the non-terminals in the configuration graph expanded so far.

4.4 Implementation of Step 3

In this step, the major challenge has been to check if the HORS $\mathcal{G}$ is well-typed. We have applied the rules 3.3 and 3.4. The function **tGamma** returns the set of types derivable for the argument term **t**. For each binding *(F:non-terminal,typ:type)* in the environment Γ_C, we will search for the λ-term $\lambda\tilde{x}.t$ corresponding to F in the map $\mathcal{R}$ of the HORS $\mathcal{G}$. We will give t as an argument to the function **tGamma**. We will thus obtain the set of derivable types for the purely applicative term t. If *typ* is in this set, it means that we add the binding *(F:non-terminal,typ:type)* to the new set $\mathcal{F}(\Gamma)$. We repeat this procedure until we reach a fixed point, i.e. when $\Gamma = \mathcal{F}(\Gamma)$.

The next thing that we need to check is if the binding $S : q_0$ is found in Γ. If this is true, then the property is satisfied and we exit the program. If not, it means we do not have enough information about the types of the non-terminals, so we need to expand more nodes of the configuration graph. Because of this, we need to go back to Step 1; this is done by putting all the 3 steps in a while loop. This loop does not have any termination condition (it is a **while true do** loop) because the program will eventually exit in one of two ways: either the property is violated or it is satisfied (in both cases, we call **exit 0**). We have reproduced below the trickiest part of Step 3, the implementation of the $\mathcal{T}_\Gamma$ procedure.

```
let rec tGamma envir char_arityF t=
let ch=M.first_symbol t in
let ar= ref 0 in
let bound=(List.length char_arityF) -1 in
for num=0 to bound
do
  let (a,b)=(List.nth char_arityF num) in
  if (a==ch) then ar:=b
done;
(*ar is the arity od the first symbol*)
if (!ar==0) then
(*we check if there is a binding in Gamma for c*)
begin
  let t_types=ref SetTypes.empty in
  let g x=(if x.Binding.c==ch then
              t_types:=SetTypes.add (x.Binding.ty) !t_types)
  in SetBindings.iter g envir;
  !t_types;
end
```

```
else
let t_types=ref SetTypes.empty in
let li=M.decompose t in
let lenli=List.length li  in
if lenli==0 then
begin
  let t_types=ref SetTypes.empty in
  let g x=(if x.Binding.c==ch then
             t_types:=SetTypes.add (x.Binding.ty) !t_types)
           in SetBindings.iter g envir;
  !t_types;
end
else
begin
let f x=
(if x.Binding.c==ch then
    begin
    let l=ref [] in sets_of_type x.Binding.ty l lenli;
    let ok=ref 1 in
    for i=0 to (lenli -1)
      do
      let si=(List.nth !l i) in
      let emptyset=ref SetTypes.empty in
      emptyset:=SetTypes.add (M.I(-1)) !emptyset;
      if (SetTypes.equal !emptyset si)== false then
        let tG=tGamma envir char_arityF (List.nth li i) in
        if (SetTypes.subset si tG)==false
         then ok:=0;
     done;
     if (!ok==1) then
      begin
       let listforunion=List.nth !l lenli in
       t_types:=SetTypes.union listforunion !t_types;
      end
          end) in
SetBindings.iter f envir;
!t_types;
end;;
```

4.5 Tests and evaluation

Even though the HORS model checking is n-EXPTIME in general, this algorithm is linear in the size of HORS, assuming the sizes of types and specifications are bounded by a constant. We have tested the model checker for a number of small but tricky examples. All the examples are taken from [3] or from Naoki Kobayashi's home page. Table 4.5 shows the results. The columns "order", "rules", "size" and "states" show the order of the recursion scheme, the number of rewriting rules, the size of rewriting rules (which are measured by the number of occurrences of symbols in the right-hand side of the rewriting rules) and the number of automaton states respectively. The column "result" shows whether the property is satisfied or not. If the property is satisfied, the model checker outputs a type environment and if the property is not satisfied, the model checker outputs an error trace. The column "time" shows the running time. The time is given in seconds, and it includes the time of compiling the source code, of lexing and parsing the input file, of running the algorithm and of writing the result to the file "result.txt". For each of the 5 examples below, we will present the input file and the output file "result.txt"

Programs	order	rules	size	states	result	time
example2-3.txt	1	6	13	1	YES	0.8
file.txt	1	2	8	2	YES	1.3
flow.txt	3	7	16	1	YES	0.8
lock1.txt	4	12	38	3	YES	0.8
twofiles.txt	4	12	56	5	YES	10
example3-1.txt	1	2	8	1	NO	0.8

Table 4.1: Experimental results

Example 1
Input file example2-3.txt

```
b:o o o
t:o o
f:o

S:o
F:o o
E:o o
H:o o
G:o o
```

```
T: o

S  −. b  (F  T)  (E  T)
F  −k.  H  k
E  −k  .  H  k
H  −k  .  k
G  −k  .  f
T  −.  t  T

1
0  b  0  0
0  t  0
```

Output file:

```
 PROPERTY SATISFIED
E( q0  −>q0  )
F( q0  −>q0  )
H( q0  −>q0  )
Sq0
Tq0
```

Example 2 Input file:

```
b: o  o  o
r: o  o
c: o  o
e: o

S: o
F: o  o

S−.F  e
F−k.b  (c  k)  (r  (F  k))

2
0  b  0  0
1  b  1  1
1  e
0  r  0
0  c  1
```

Output file:

```
 PROPERTY SATISFIED
F(q1 ->q0 )
Sq0
```

Example 3 Input file flow.txt

```
e:o
f:o o

S:o
A:o o
B:o o o
C:o o
L:o o
M:o o
I:o o o

S -. A I
A -i . i L (B i)
B -i u . i M C
C -x .x e
L -x . f x
M -x .x
I -x k . k x

1
0 e
```

Output file:

```
 PROPERTY SATISFIED
A(((q0 ->q0 )->(((q0 ->q0 )->q0 )->q0 ))
   ^(Top->((Top->q0 )->q0 ))->q0 )
B(((q0 ->q0 )->(((q0 ->q0 )->q0 )->q0 ))
   ->(Top->q0 ))
C((q0 ->q0 )->q0 )
I((q0 ->q0 )->(((q0 ->q0 )->q0 )->q0 ))
I(Top->((Top->q0 )->q0 ))
M(q0 ->q0 )
Sq0
```

Example 4 Input file lock1.txt

```
b:o o o
```

```
n:o o
l:o o
u:o o
e:o

S:o
F:o o o
G:o o o
C:o o o
D:o o o
N:o o
I:o o o
K:o o o
L:o o o
U:o o o

S - .b (N (F e)) (N (G e))
F -k x . C x k
G -k x .L x (D x k)
C -x k . k
D -x k . U x k
N -k . b (n (k I)) (k K)
I -x y . x y
K -x y . y
L -x k . x l k
U -x k . x u k

3
0 b 0 0
1 b 1 1
2 b 2 2
0 n 1
1 l 2
2 u 1
0 e
1 e
```

Output file:

```
 PROPERTY SATISFIED
C(Top->(q0 ->q0 ))
C(Top->(q1 ->q1 ))
```

```
D(((q1 ->q2 )->(q1 ->q2 ))->(q1 ->q2 ))
D((Top->(q0 ->q0 ))->(q0 ->q0 ))
F(q0 ->(Top->q0 ))
F(q1 ->(Top->q1 ))
G(q0 ->((Top->(q0 ->q0 ))->q0 ))
G(q1 ->(((q1 ->q2 )->(q1 ->q2 ))
   ^((q2 ->q1 )->(q2 ->q1 ))->q1 ))
I((q1 ->q2 )->(q1 ->q2 ))
I((q2 ->q1 )->(q2 ->q1 ))
K(Top->(q0 ->q0 ))
L(((q2 ->q1 )->(q2 ->q1 ))->(q2 ->q1 ))
L((Top->(q0 ->q0 ))->(q0 ->q0 ))
N((((q1 ->q2 )->(q1 ->q2 ))^((q2 ->q1 )->
   (q2 ->q1 ))->q1 )^((Top->(q0 ->q0 ))->q0 )->q0 )
N((Top->q1 )^(Top->q0 )->q0 )
Sq0
U(((q1 ->q2 )->(q1 ->q2 ))->(q1 ->q2 ))
U((Top->(q0 ->q0 ))->(q0 ->q0 ))
```

Example 5 Input file twofiles.txt

```
b:o o o
n:o o
m:o o
r:o o
w:o o
c:o o
e:o

S:o
A:o o
B:o o o
F:o o o o
I:o o o
K:o o o
N:o o
M:o o
C:o o o
R:o o o
W:o o o

S -. N A
```

```
A −x . M (B x)
B −x y . F x y e
F −x y k . b (C x (C y k)) (R x (W y (F x y k)))
I −x y . x y
K −x y . y
N −k . b (n (k I)) (k K)
M −k . b (m (k I)) (k K)
C −x k . x c k
R −x k . x r k
W −y k . y w k

5
0 b 0 0
1 b 1 1
2 b 2 2
3 b 3 3
0 n 1
1 r 1
1 c 4
0 m 2
2 w 2
2 c  4
2 n 3
1 m 3
3 r 3
3 w 3
3 c  3
4 e
0 e
3 e
```

Output file:

```
 PROPERTY SATISFIED
A(((q4 −>q1 )−>(q4 −>q1 ))^((q1 −>q1 )−>(q1 −>q1 ))
   ^((Top−>q3 )−>(Top−>q3 ))^((q3 −>q3 )−>(q3 −>q3 ))−>q1 )
A((Top−>(q0 −>q0 ))^(Top−>(q2 −>q2 ))−>q0 )
B(((Top−>q3 )−>(Top−>q3 ))^((q3 −>q3 )−>(q3 −>q3 ))
   −>(((q3 −>q3 )−>(q3 −>q3 ))−>q3 ))
B(((q4 −>q1 )−>(q4 −>q1 ))^((q1 −>q1 )−>(q1 −>q1 ))
   −>((Top−>(q4 −>q4 ))^(Top−>(q1 −>q1 ))−>q1 ))
B((Top−>(q0 −>q0 ))−>((Top−>(q0 −>q0 ))−>q0 ))
```

```
B((Top->(q2 ->q2 ))->(((q4 ->q2 )->(q4 ->q2 ))
   ^((q2 ->q2 )->(q2 ->q2 ))->q2 ))
C(((q3 ->q3 )->(q3 ->q3 ))->(q3 ->q3 ))
C(((q4 ->q1 )->(q4 ->q1 ))->(q4 ->q1 ))
C(((q4 ->q2 )->(q4 ->q2 ))->(q4 ->q2 ))
C((Top->(q0 ->q0 ))->(q0 ->q0 ))
C((Top->(q2 ->q2 ))->(q2 ->q2 ))
C((Top->(q4 ->q4 ))->(q4 ->q4 ))
F(((Top->q3 )->(Top->q3 ))^((q3 ->q3 )->(q3 ->q3 ))
   ->(((q3 ->q3 )->(q3 ->q3 ))->(q3 ->q3 )))
F(((q4 ->q1 )->(q4 ->q1 ))^((q1 ->q1 )->(q1 ->q1 ))
   ->((Top->(q4 ->q4 ))^(Top->(q1 ->q1 ))->(q4 ->q1 )))
F((Top->(q0 ->q0 ))->((Top->(q0 ->q0 ))->(q0 ->q0 )))
F((Top->(q2 ->q2 ))->(((q4 ->q2 )->(q4 ->q2 ))^
   ((q2 ->q2 )->(q2 ->q2 ))->(q4 ->q2 )))
I((Top->q3 )->(Top->q3 ))
I((q1 ->q1 )->(q1 ->q1 ))
I((q2 ->q2 )->(q2 ->q2 ))
I((q3 ->q3 )->(q3 ->q3 ))
I((q4 ->q1 )->(q4 ->q1 ))
I((q4 ->q2 )->(q4 ->q2 ))
K(Top->(q0 ->q0 ))
K(Top->(q1 ->q1 ))
K(Top->(q2 ->q2 ))
K(Top->(q4 ->q4 ))
M((((q3 ->q3 )->(q3 ->q3 ))->q3 )^((Top->(q4 ->q4 ))
   ^(Top->(q1 ->q1 ))->q1 )->q1 )
M((((q4 ->q2 )->(q4 ->q2 ))^((q2 ->q2 )->(q2 ->q2 ))->q2 )
   ^((Top->(q0 ->q0 ))->q0 )->q0 )
N((((q4 ->q1 )->(q4 ->q1 ))^((q1 ->q1 )->(q1 ->q1 ))^
   ((Top->q3 )->(Top->q3 ))^((q3 ->q3 )->(q3 ->q3 ))->q1 )^
   ((Top->(q0 ->q0 ))^(Top->(q2 ->q2 ))->q0 )->q0 )
R(((q1 ->q1 )->(q1 ->q1 ))->(q1 ->q1 ))
R(((q3 ->q3 )->(q3 ->q3 ))->(q3 ->q3 ))
R((Top->(q0 ->q0 ))->(q0 ->q0 ))
R((Top->(q2 ->q2 ))->(q2 ->q2 ))
Sq0
W(((q2 ->q2 )->(q2 ->q2 ))->(q2 ->q2 ))
W(((q3 ->q3 )->(q3 ->q3 ))->(q3 ->q3 ))
W((Top->(q0 ->q0 ))->(q0 ->q0 ))
W((Top->(q1 ->q1 ))->(q1 ->q1 ))
```

Example 6
Input file example3-1.txt

```
a:o o o
b:o o
c:o

S:o
F:o o

S -. F (F c)
F -x .a x (b (F x))

2
0 a 0 0
0 b 1
1 b 1
0 c
1 c
```

Output file:

```
 THE ERROR PATH:
(Fc)   (Fc)(b(F(Fc)))   (F(Fc))   (Fc)   (Fc)(b(F(Fc)))
```

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We have implemented a prototype model-checker of higher order recursion schemes, that works on a number of small, but complicated examples. This has been possible due to the recent advances in the theory of verification of higher order programs and the implementation is a first step towards a practical approach. The experiments show that we were able to verify even a HORS of order 4, which would be impossible with previous model checking algorithms. The verification of a HORS is $(n-1)$-EXPTIME complete in general, but in typical programs the usage patterns of each higher-order function is limited. This makes it clear to us that is is worthwhile to further optimize our implementation for a successful use in practice.

While working on this interesting project, I have learned many new things. I have had the opportunity to develop my research skills. Without having previous experience in functional programming, I have learned how to program in OCaml. I have been impressed by the concisiveness and clarity of this style of programming. I have fully understood the theory behind the verification of higher order recursion schemes, and in this dissertation I have tried to present it such that the reader will also find it easy to grasp. I have applied the knowledge acquired in the MSc course, such as the verification of software, lambda calculus and types, the theory of automata.

In conclusion, the project has fulfilled all its objectives in terms of technical aims and the development of soft skills.

5.2 Future Work

The next important step is to optimize this implementation. The optimizations can be done in a number of places. Firstly, we need to use the type system presented in Section 3.2 , for being able to apply the two optimizations of the function **Elim** detailed in Section 3.6. This will greatly improve performance.

Another improvement is to use a proper graph structure for representing the configuration graph. However, there is an optimizations that we can make, even if we are using a tree representation. Thus, we can construct a tree that has "forward pointers" (i.e. from a node to one of its descendants), but also "backward pointer“. The "forward pointers“ are the pointers that constitute the dependency relationship underpinning the construction of types. The "backward pointers“ denote the origin of the terms and are similar to the notion of "history“ from our implementation. This can be achieved by appealing to the notion of pointer machines, as introduced by Colin Stirling. Intuitively, a pointer machine can be understood as an implementation of a higher order grammar. A stack β is a non-empty list $[n_1, n_2, \ldots, n_m]$ containing nodes of the configuration graph, that have already been expanded. Each n_i is referred to as an item. Each item in the stack may have several pointers emanating from it, apart from n_1, which has no pinters. The pointer represents the origin of that respective term and is a physical link connecting an item n_i to some item n_j, for $j \leq i$. This results in savings not just in space, which is obvious, but also in time.

Another optimization, also proposed by Kobayashi in [3] is a more tight integration of the three steps. For example, when Step 3 fails(i.e. $S : q_0 \in \Gamma$ does not hold), we should be able to use the computed type environment Γ in Step 1 for effectively deciding which open node should be expanded (to get more type information), instead of the current FIFO queue implementation.

As a long term goal in this line of research, we could implement a software model checkers for OCaml on top of the model checker for HORS. For doing this, we need to combine the model checker with predicate abstractions and CEGAR.

Bibliography

[1] Edmund M. ClarkeJr., Orna Grumberg, Doron A. Peled *Model Checking*, The MIT Press, 1999

[2] Dr Jon D. Harrop *OCaml for scientists*, Flying Frog Consutancy Ltd., 2005

[3] Model-Checking Higher-Order Functions *Naoki Kobayashi*. In proceedings of PPDP 2009 (to appear).

[4] Tree Automata Techniques and Applications *Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, Marc Tommasi*, 2008

[5] A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher Order Recursion Schemes *N. Kobayashi, C.-H.L.Ong* In proceedings of LICS 2009 (to appear).

[6] On Model-Checking Trees Generated by Higher-Order Recursion Schemes *C.-H.L.Ong*. In proceedings of LICS 2006.

[7] Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs *Naoki Kobayashi* In proceedings of POPL 2009

[8] http://en.wikipedia.org/wiki/EXPTIME

[9] Resouce Usage Analysis *Atsushi Igarashi, Naoki Kobayashi* In proceedings of POPL 2002 .

[10] http://en.wikipedia.org/wiki/Continuation-passing_style

[11] http://en.wikipedia.org/wiki/Lambda_lifting

[12] Types with Intersection: An Introduction *J.Roger Hindley*. In Formal Aspects of Computing, 1992

[13] Lambda Calculus and Types *Andrew D. Ker* Course Lecture Notes, Oxford Univeristy, Hilary Term 2009

[14] Subtyping for Model Checking Recursion Schemes *C.-H.L.Ong ,S.J.Ramsay* July 2009 (unpublished)

[15] http://ocamlgraph.lri.fr/

[16] http://caml.inria.fr/