Model Checking and Functional Program Transformations*

Axel Haddad

LIAFA (Université Paris Diderot / CNRS)
LIGM (Université Paris Est / CNRS)

Abstract

We study a model for recursive functional programs called higher order recursion schemes (HORS). We give new proofs of two verification related problems: reflection and selection for HORS. The previous proofs are based on the equivalence between HORS and collapsible pushdown automata and they lose the structure of the initial program. The constructions presented here are based on shape preserving transformations, and can be applied on actual programs without losing the structure of the program.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Higher-order recursion schemes, Model checking, Tree automata

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2013.115

1 Introduction

Recursion schemes were introduced in the early 70s as a model of computation, describing the syntactical part of a functional program [18, 6, 7, 8]. Originally they only handled order-0 (constants) or order-1 (functions on constants) expressions, but not higher-order types. Higher-order versions of recursion schemes were later introduced to deal with functions taking functions as arguments [13, 9, 10].

Recently, the focus came back on higher-order recursion schemes when considering them as generators of possibly infinite trees [14, 19, 12]. Indeed, roughly speaking a recursion scheme is a deterministic rewriting system on typed terms that generates ad infinitum a unique infinite tree. As the trees they generate are very general and as they can capture the computation tree of (higher-order) functional programs, studying their logical properties leads to very natural and challenging problems.

The most popular one is *(local) model-checking*: for a given recursion scheme and a formula *e.g.* from monadic second order logic (MSO) or μ-calculus, decide whether the tree generated by the scheme satisfies the formula. Following partial decidability results for the subclass of *safe* recursion schemes [14, 5], Ong proved, using the notion of traversals, the decidability of MSO model-checking for the whole class of trees generated by recursion schemes [19]. Since then, other proofs of this result have been obtained using different approaches: Hague, Murawski, Ong and Serre established the equivalence of schemes and higher-order collapsible pushdown automata (CPDA), and then showed the MSO decidability by reduction to parity games on collapsible pushdown automata [12]; following ideas from [1], Kobayashi and Ong [17] developed the type system of [15] to obtain a new proof. Finally, Salvati and Walukiewicz used Krivine machines to establish the MSO decidability of λ-Y-calculus, which is a typed lambda calculus with recursion, equivalent to higher order recursion schemes [21].

* This work was supported by the project AMIS (ANR 2010 JCJC 0203 01 AMIS).

33rd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013).
Editors: Anil Seth and Nisheeth K. Vishnoi; pp. 115–126

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Another important problem is *global model-checking*: for a given recursion scheme and a formula, provide a finite representation of the set of nodes in the tree generated by the scheme where the formula holds. Broadbent, Carayol, Ong and Serre answered this question using a so-called endogenous approach to represent the set of nodes: they showed how to construct another scheme generating the same tree as the original scheme except that now those nodes where the formula holds are marked [3]. They refer to this property as the *logical reflection*. The technique they used relies on the equivalence between schemes and CPDA.

Going further with the idea of marking a set of nodes, Carayol and Serre considered in [4] the following problem called *logical selection* and generalising global model-checking: for a given recursion scheme and an MSO formula $\varphi[X]$ with one free set variable, provide (if it exists) a finite representation of a set S of nodes in the tree generated by the scheme such that $\varphi[S]$ holds. They show that one can construct another scheme generating the same tree as the original scheme except that now those nodes are marked and describe a set S with the previous property. Again, they relied on the equivalence with CPDA.

One interest for logical reflection and logical selection is that they can be used to modify a scheme in a useful way. Indeed, assume some (syntax tree of a) program is described by a scheme and that it contains bad behaviours: using logical reflection, one can get a new scheme where those bad behaviours are marked and this latter scheme can then easily be modified to remove these behaviours. Using logical selection, one can *e.g.* select branches in the syntax tree so that a given property is satisfied in the resulting subtree. A drawback of the approach in [3, 4] that goes back and forth between schemes and CPDA is that the scheme that is finally obtained is structurally very far from the original one (the names of the non terminals as well as the shape of the rewriting rules have been lost): hence this is a serious problem if one is interested in doing automated correction of programs or even synthesis.

Our main contribution in this paper is to provide proofs of both logical reflection and selection without appealing to CPDA as we only reason on schemes. Our constructions avoid the loss of the structure, *i.e.* the solution scheme is obtained only by duplicating and annotating some parts of the initial scheme and the transformation is easily reversible. Our constructions are based on the type system and the game used by Kobayashi and Ong in their proof of the model-checking decidability [17]. There is no known correspondence between these proofs and the former ones. In order to prove the logical reflection, we introduce the notion of morphism inspired by denotational semantics, and we give a morphism for MSO. In order to prove the logical selection, we embed carefully a winning strategy of the model-checking game into the scheme.

In Section 2 we give the basic definitions and in Section 3 we introduce the two problems we are looking at in the paper. In Section 4 we introduce morphisms, explain how to "embed" a morphism into a scheme, and give some possible applications. In Section 6, we show how to use morphisms to obtain a new proof of logical reflection. In Section 7, we deal with logical selection. In Section 5, we present a simple example that describes how we can use a morphism describing a property to transform a scheme. An extended version of this work is available online: `http://hal.archives-ouvertes.fr/hal-00865682`.

2 Preliminaries

In this section we give the definition of a higher order recursion scheme, which is a deterministic rewriting system that produces an infinite tree. It handles terms of typed symbols, i.e. each symbol is a constant or a function that can have functions as arguments. Terms may represent expressions of higher order programming languages, for example the term **`Apply Copy File`**

means "apply the function `Copy` to the file". In this example an intuitive rewrite rule for the function `Apply` would be the term where the first argument is applied on the second argument, written: `Apply` $\varphi\ x \to \varphi\ x$.

Types. *Types* are defined by the grammar $\tau ::= o \mid \tau \to \tau$ and o is called the *ground type*. Considering that $\to$ is associative to the right (i.e. $\tau_1 \to (\tau_2 \to \tau_3)$ can be written $\tau_1 \to \tau_2 \to \tau_3$), any type τ can be written uniquely as $\tau_1 \to \cdots \to \tau_k \to o$. The integer k is called the *arity* of τ. We define the *order of a type* by $\text{order}(o) = 0$ and $\text{order}(\tau_1 \to \tau_2) = \max(\text{order}(\tau_1) + 1, \text{order}(\tau_2))$. For instance $o \to o \to o \to o$ is a type of order 1 and arity 3, $(o \to o) \to (o \to o)$, that can also be written $(o \to o) \to o \to o$ is a type of order 2.We let $\tau^\ell \to \tau'$ be a shorthand for $\underbrace{\tau \to \cdots \to \tau}_{\ell \text{ times}} \to \tau'$.

Terms. Let Γ be a finite set of typed symbols, and Let Γ^τ denote the set of symbols of type τ. For all type τ, we define the set of *terms* of type τ, $\mathcal{T}^\tau(\Gamma)$ as the smallest set containing the symbols of type τ and the application of a term of type $\tau' \to \tau$ to a term of type τ' for all τ'; formally: $\Gamma^\tau \subseteq \mathcal{T}^\tau(\Gamma)$ and $\bigcup_{\tau'} \{t\ s \mid t \in \mathcal{T}^{\tau' \to \tau}(\Gamma), s \in \mathcal{T}^{\tau'}(\Gamma)\} \subseteq \mathcal{T}^\tau(\Gamma)$. We shall write $\mathcal{T}(\Gamma)$ for the set of terms of any type, and $t : \tau$ if t has type τ. The arity of a term t, $arity(t)$, is the arity of its type. Remark that any term t can be uniquely written as $t = \alpha\ t_1 \dots t_k$ with $\alpha \in \Gamma$ and $0 \leq k \leq \text{arity}(\alpha)$. We say that α is the *head* of the term t. For instance, let $\Gamma = \{F : (o \to o) \to o \to o\ ,\ G : o \to o \to o\ ,\ H : (o \to o)\ ,\ \mathtt{a} : o\}$. Then $F\ H$ and $G\ \mathtt{a}$ are terms of type $o \to o$; $F(G\ \mathtt{a})\ (H\ (H\ \mathtt{a}))$ is a term of type o; $F\ \mathtt{a}$ is not a term since F is expecting a first argument of type $o \to o$ while $\mathtt{a}$ has type o.

A set of symbols of order at most 1 (i.e. each symbol has type o or $o \to o \to \cdots \to o$) is called a *signature*. In the following, we use the letters t, r, s to denote terms, and given a tuple of term $\vec{t} = (t_1, \dots, t_k)$ we may use the shorthand $s\ \vec{t}$ to denote $s\ t_1 \dots t_k$.

Contexts. Let $t : \tau$, $t' : \tau'$ be two terms, $x : \tau'$ be a symbol of type τ', then we write $t_{[x \mapsto t']} : \tau$ for the term obtained by substituting all occurrences of x by t' in the term t. A *τ-context* is a term $C[\bullet^\tau] \in \mathcal{T}(\Gamma \uplus \{\bullet^\tau : \tau\})$ containing exactly one occurrence of $\bullet^\tau$; it can be seen as an application turning a term into another, such that for all $t : \tau$, $C[t] = C[\bullet^\tau]_{[\bullet^\tau \mapsto t]}$. In general we will only talk about ground type contexts where $\tau = o$ and we will omit to specify the type when it is clear. For instance, if $C[\bullet] = F \bullet\ (H\ (H\ a))$ and $t' = G\ \mathtt{a}$ then $C[t'] = F\ (G\ \mathtt{a})\ (H\ (H\ \mathtt{a}))$.

Rewrite Rules. Given two disjoint sets of symbols $\Gamma, \mathcal{V}$ where $\mathcal{V}$ is called a set of *variables*, we define a (fully applied) *rewrite rule* on Γ and $\mathcal{V}$ as a pair of terms of $\mathcal{T}(\Gamma \uplus \mathcal{V})$ of type o written $F\ x_1 \dots x_k \to e$ with $F \in \Gamma$, $x_1, \dots, x_k \in \mathcal{V}$ such that for all $i \neq j$ $x_i \neq x_j$, and $e \in \mathcal{T}(\Gamma \uplus \{x_1, \dots, x_k\})$. Given a set of rewrite rules $\mathcal{R}$, we define the *rewriting relation* $\to \in\ \mathcal{T}(\Gamma)^2$ as $t \to t'$ iff there exists a context $C[\bullet]$, a rewrite rule $F\ x_1 \dots x_k \to e$, and a term $F\ t_1\ \dots\ t_k : o$ such that $t = C[F\ t_1 \dots t_k]$ and $t' = C[e_{[x_1 \mapsto t_1] \dots [x_k \mapsto t_k]}]$. We call $F\ t_1\ \dots\ t_k : o$ a *redex*. We define $\to^*$ as the reflexive and transitive closure of $\to$. Finally we say that a set of rewrite rules is *deterministic*[1] if for all $F \in \Gamma$ there exists at most one rule of the form $F\ x_1 \dots x_k \to e$.

Trees. Let Σ be a finite signature, m be the maximum arity in Σ and $\perp : o$ be a fresh symbol. A (ranked) *tree* t over $\Sigma \uplus \perp$ is a mapping $t : dom^t \to \Sigma \uplus \perp$, where dom^t is a prefix-closed subset of $\{1, \dots, m\}^*$ such that if $u \in dom^t$ and $t(u) = a$ then $\{j \mid u \cdot j \in dom^t\} = \{1, \dots, arity(a)\}$.

Given a node $u \in dom^t$, we define the subtree of t rooted at u as the tree t_u such that $dom^{t_u} = \{j \mid u \cdot j \in dom^t\}$ and for all $v \in dom^{t_u}$, $t_u(v) = t(u \cdot v)$. Given $a : o^k \to o$ and

[1] Although the rules are deterministics, there may be several possible rules to apply in a term.

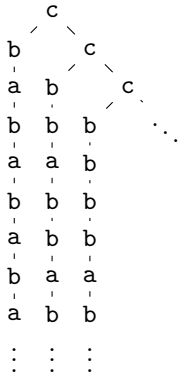
some trees $t_1, \ldots, t_k$ we use the notation $a\ t_1 \ldots t_k$ to denote the tree t' whose domain is $dom^{t'} = \bigcup_i i \cdot dom^{t_i}$, $t'(\varepsilon) = a$ and $t'(i \cdot u) = t_i(u)$. Note that there is a direct bijection between ground terms of $\mathcal{T}^o(\Sigma \uplus \bot)$ and finite trees. Hence we will freely allow ourselves to treat ground terms over $\Sigma \uplus \bot$ as trees. We define the partial order $\sqsubseteq$ over trees as $t \sqsubseteq t'$ if $dom^t \subseteq dom^{t'}$ and for all $u \in dom^t$, $t(u) = t'(u)$ or $t(u) = \bot$. Given a (possibly infinite) sequence of trees $t_0, t_1, t_2, \ldots$ such that $t_i \sqsubseteq t_{i+1}$ for all i, the set of all t_i has a supremum that is called the *limit tree* of the sequence.

Higher Order Recursion Schemes. A *higher order recursion scheme (HORS)* $\mathcal{G} = \langle \mathcal{V}, \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ is a tuple such that: $\mathcal{V}$ is a finite set of typed symbols called *variables*; Σ is a finite signature, called the *set of terminals*; $\mathcal{N}$ is a finite set of typed symbols called *non-terminals*; $\mathcal{R}$ is a deterministic set of rewrite rules on $\Sigma \uplus \mathcal{N}$ and $\mathcal{V}$, such that there is exactly one rule per non terminal and no rule for terminals; $S \in \mathcal{N}$ is the *initial non-terminal*.

We define inductively the $\bot$*-transformation* $(\cdot)^\bot : \mathcal{T}^o(\mathcal{N} \uplus \Sigma) \to \mathcal{T}^o(\Sigma \uplus \{\bot : o\})$ turning a ground term into a finite tree as: $(F\ t_1\ \ldots\ t_k)^\bot = \bot$ for all $F \in \mathcal{N}$ and $(a\ t_1\ \ldots\ t_k)^\bot = a\ t_1^\bot \ldots t_k^\bot$ for all $a \in \Sigma$. We define a *derivation*, as a possibly infinite sequence of terms linked by the rewrite relation, and we will mainly look at derivations where the first term of the sequence is equal to the initial non terminal. Let $t_0 = S \to t_1 \to t_2 \to \cdots$ be a derivation, then one can check that $(t_0)^\bot \sqsubseteq (t_1)^\bot \sqsubseteq (t_2)^\bot \sqsubseteq \ldots$, hence it admits a limit. One can prove[2] that the set of all such limit trees has a greatest element that we denote $\|\mathcal{G}\|$ and refer to as the *value tree* of $\mathcal{G}$. Note that $\|\mathcal{G}\|$ is the supremum of $\{t^\bot \mid S \to^* t\}$. Given a term $t : o$, we denote by $\mathcal{G}_t$ the scheme obtained by transforming G such that it starts derivations with the term t, formally, $\mathcal{G}_t = \langle \mathcal{V}, \Sigma, \mathcal{N} \uplus \{S'\}, \mathcal{R} \uplus \{S' \to t\}, S' \rangle$. One can prove that if $t \to t'$ then $\|\mathcal{G}_t\| = \|\mathcal{G}_{t'}\|$. We call $\|\mathcal{G}_t\|$ the value tree of t in $\mathcal{G}$.

Parallel derivation. Intuitively, the *parallel rewriting* from a term t is obtained by rewriting all the redexes in t simultaneously. Formally, given a term $t = \alpha\ t_1 \ldots t_k$ with $\alpha \in \Sigma \uplus \mathcal{N}$, we define inductively the parallel rewriting $t^\star$ of t as if $\alpha \in \Sigma$ or $k < \text{arity}(\alpha)$, then $t^\star = \alpha\ t_1^\star \ldots t_k^\star$, if $\alpha \in \mathcal{N}$ and $k = \text{arity}(\alpha)$, let $\alpha\ x_1 \ldots x_k \to e$ be the rewrite rule associated to α, we have $t^\star = e_{[\forall i\ x_i \mapsto t_i^\star]}$. Given t, t', we write $t \Rightarrow t'$ for the relation "$t' = t^\star$". Notice that if $t \Rightarrow t'$ and $t' \to^* t''$ then $t \to^* t''$ (in particular $t \to^* t'$). Furthermore the derivation $S \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \ldots$ leads to $\|\mathcal{G}\|$.

▶ **Example 1.** Let $\mathcal{G} = \langle \Sigma, \mathcal{V}, \mathcal{N}, S, \mathcal{R} \rangle$ with $\Sigma = \{\mathtt{a}, \mathtt{b} : o \to o, c : o \to o \to o\}$, $\mathcal{V} = \{x : o, \varphi : o \to o\}$, $\mathcal{N} = \{S : o, I, F : (o \to o) \to o, D, A : (o \to o) \to o \to o\}$ and $\mathcal{R} = \{S \to F\ \mathtt{b},\ \ D\ \varphi\ x \to \varphi\ (\varphi\ x),\ \ A\ \varphi\ x \to \varphi\ (\mathtt{a}\ x),\ \ I\ \varphi \to \varphi\ (I\ \varphi),\ \ F\ \varphi\ \to\ \mathtt{c}\ (I\ (A\ \varphi))\ (F\ (D\ \varphi))\}$.
Remark the rewrite rule associated to D: it means that for any function t, $D\ t$ is simply the function obtained by composing t with itself. The rule associated to I is also interesting: for any function t, $I\ t$ leads to the infinite iteration of t. For example the term $I\ \mathtt{a}$ can be derived to obtain $\mathtt{a}\ (\mathtt{a}\ (\mathtt{a}\ (\ \ldots\ ))))$. The tree $\|\mathcal{G}\|$ is depicted on the left, its branches are labelled by $\mathtt{c}^\omega$ or $\mathtt{c}^n \cdot (\mathtt{b}^{2^{(n-1)}} \cdot \mathtt{a})^\omega$ for all $n \geq 1$.



Parity tree automaton. A *non-deterministic max parity automaton* (we will just refer to them as automata in the following) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ with Σ a finite signature, Q a finite set called the *set of states*, $\delta \subseteq \{q \xrightarrow{a} (q_1, \ldots, q_{\text{arity}(a)}) \mid a \in \Sigma, q, q_1, \ldots, q_{\text{arity}(a)} \in Q\}$

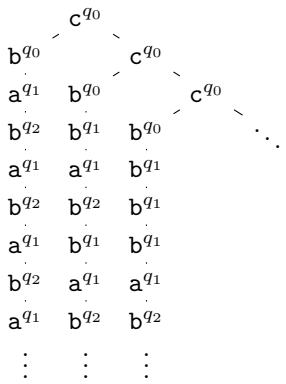
[2] The existence of a value tree is a consequence the confluence property of HORS.

is called the *transition relation*, $q_0 \in Q$ the *initial state*, $\Omega : Q \to \{1, \dots, m_{\max}\}$ for some $m_{\max} \in \mathbb{N}$ the *colouring function*.

Given an infinite tree t on Σ we define a *run* r of $\mathcal{A}$ on t as a tree on $\Sigma \times Q = \{a^q : o^k \to o \mid a : o^k \to o, q \in Q\}$ such that $dom^r = dom^t$, for all $u \in dom^t$, if $r(u) = a^q$ then $t(u) = a$ and there exists $q \xrightarrow{a} q_1, \dots, q_k \in \delta$ such that for all i, $r(u \cdot i) = t(u \cdot i)^{q_i}$, and $r(\varepsilon) = t(\varepsilon)^{q_0}$.

We say that the automaton $\mathcal{A}$ accepts the tree t, written $t \models \mathcal{A}$, if there exists a run r on t such that for every infinite branch $b = (a_0, q_0) \cdot j_0 \cdot (a_1, q_1) \cdot j_1 \cdot \dots$ in r, the greatest colour seen infinitely often in $\Omega(q_0), \Omega(q_1), \Omega(q_2), \dots$ is even. We define $\mathcal{A}_q$ as the automaton obtained by changing in $\mathcal{A}$ the initial state to q: $\mathcal{A}_q = \langle \Sigma, Q, \delta, q, \Omega \rangle$, and we say that $\mathcal{A}$ accepts the tree t from state q, if $t \models \mathcal{A}_q$.

▶ **Example 2.** Let $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \Omega \rangle$ be an automaton with $\Sigma = \{\mathtt{a}, \mathtt{b} : o \to o, c : o \to o \to o\}$, $Q = \{q_0, q_1, q_2\}$, $\Omega = \{q_0 \mapsto 0, q_1 \mapsto 1, q_2 \mapsto 2\}$ and $\delta = \{q_0 \xrightarrow{\mathtt{c}} (q_0, q_0), q_0 \xrightarrow{\mathtt{b}} q_1, q_1 \xrightarrow{\mathtt{a}} q_2, q_1 \xrightarrow{\mathtt{b}} q_1, q_2 \xrightarrow{\mathtt{a}} q_2, q_2 \xrightarrow{\mathtt{b}} q_1\}$.
The automaton $\mathcal{A}$ accepts the trees whose branches are labelled by c^ω or by $c^* \cdot b \cdot (b^* \cdot \mathtt{a})^\omega$. An accepting run of $\mathcal{A}$ on the tree $\|\mathcal{G}\|$ of Example 1 is depicted on the right.



3 Logical reflection and logical selection

In this section we formalise the notion of annotated tree and we define the problems of logical reflection and logical selection we are interested in.

Given a signature Σ, a set X and a tree t on the signature Σ, we define the signature $\Sigma \times X = \{(a, x) : o^k \to o \mid a : o^k \to o \in \Sigma, x \in X\}$ and we say that the tree t' on the signature $\Sigma \times X$ is an *X-annotation* of t, if $dom^t = dom^{t'}$ and if for all $u \in dom^t$, $t'(u) = (t(u), x)$ for some $x \in X$. For example a run of an automaton over a tree t is a Q-annotation of t, with Q being the set of states of the automaton. Furthermore, for any tree t' on the signature $\Sigma \times X$, we define $Unlab(t')$ as the tree on Σ obtained by turning all nodes (a, x) of t' into a *i.e.* the tree obtained by removing the annotated part of the tree.

Given a set of nodes S in a tree t, we define an *S-marking of t* as a $\{\mathtt{0}, \mathtt{1}\}$-annotation t' of t such that for all nodes u, $u \in S$ if and only if $t'(u) = (t(u), \mathtt{1})$. Given a μ-calculus formula φ, we say that t' is a *φ-reflection* of t if it is a marking of the set of nodes u such that the subtree of t rooted at u satisfies the formula φ. Given a monadic second-order logic (MSO) formula $\varphi[x]$ with a first order free variable, we say that t' is a *$\varphi[x]$-reflection* of t if it is a marking of the set of nodes u satisfying $\varphi[u]$. Given an MSO formula $\varphi[X]$ with a second order free variable, we say that t' is a *$\varphi[X]$-selection* of t if it is a marking of a set of nodes S satisfying $\varphi[S]$.

Finally, we define the *μ-calculus reflection*, MSO *reflection*, and MSO *selection* as follow. Given a class R of generators of trees (*i.e.* a set of finitely described elements such that to each $g \in R$ we associate a unique tree $\|g\|$), we say that R is (effectively) *reflective with respect to the μ-calculus* (*resp.* MSO) if for any μ-calculus formula φ (*resp.* any MSO formula $\varphi[x]$) and any tree generator $g \in R$, one can construct another generator g' such that the $\|g'\|$ is a φ-reflection (resp. $\varphi[x]$-reflection) of $\|g\|$. We say that R is (effectively) *selective with respect to* MSO if for any MSO formula $\varphi[X]$ and any generator $g \in R$ such that there

exists a subset S of nodes of $\|g\|$ satisfying $\varphi[S]$, one can construct another generator g' such that $\|g'\|$ is a $\varphi[X]$-selection.

The main contribution of this paper is to give new proofs of the fact that schemes have these properties. In order to do so we use the equivalence results between logic and automata, and we rather prove automata reflexivity and automata selectivity defined as follows.

Given a tree t on the signature Σ and an automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \Omega \rangle$, we define an *$\mathcal{A}$-reflection* of $\mathcal{A}$ on t as a 2^Q-annotation t' of t, such that for all nodes u, $t'(u) = (t(u), Q')$ with $Q' \subseteq Q$ being the set of states q such that $\mathcal{A}_q$ accepts the subtree of t rooted on u; we define an *$\mathcal{A}$-selection* of $\mathcal{A}$ on t as an accepting run of $\mathcal{A}$ on t. We say that a class R of generators of trees is *reflective with respect to automata* if for any automaton $\mathcal{A}$ and any generator $g \in R$, one can construct another generator g' such that $\|g'\|$ is an $\mathcal{A}$-reflection of $\|g\|$. We say that R is *selective with respect to automata* if for any automaton $\mathcal{A}$ and any generator $g \in R$ such that $\|g\| \models \mathcal{A}$, one can construct another generator g' such that $\|g'\|$ is an $\mathcal{A}$-selection of $\|g\|$.

From the equivalence between logics and automata [20] we have that schemes are reflective with respect to the μ-calculus iff they are reflective with respect to automata, and they are selective with respect to MSO iff they are reflective with respect to automata. Furthermore, it is shown in [3] that μ-calculus reflection for schemes implies MSO reflection.

4 Morphisms

4.1 Definitions

In the following we fix a scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{V}, S, \mathcal{R} \rangle$. We define a *typed domain* $\mathcal{D}$ as a set such that each element is typed, and to each element $d : \tau_1 \to \tau_2 \in \mathcal{D}$ is associated a *partial* mapping f_d from $\mathcal{D}^{\tau_1}$ to $\mathcal{D}^{\tau_2}$, where $\mathcal{D}^\tau = \{d \in \mathcal{D} \mid d : \tau\}$. We write $d\ d'$ the element $f_d(d')$.

We define a *morphism* $[\![\cdot]\!] : \mathcal{T}(\Sigma \uplus \mathcal{N}) \to \mathcal{D}$ from terms on $\Sigma \uplus \mathcal{N}$ to the domain $\mathcal{D}$ as a mapping such that (1) if $t : \tau$ then $[\![t]\!] : \tau$, (2) if $t_0 : \tau_1 \to \tau_2$ and $t_1 : \tau_1$, then $[\![t]\!]\ [\![t']\!]$ is defined and equal to $[\![t\ t']\!]$. In the following, we refer to $[\![t]\!]$ as the $\mathcal{D}$-value of the term t.

▶ **Example 3.** Let $\mathcal{D} = \bigcup_\tau \{\bot^\tau : \tau, \top^\tau : \tau\}$ such that for all τ_1, τ_2: $\top^{\tau_1 \to \tau_2}\ \top^{\tau_1} = \top^{\tau_2}$; $\top^{\tau_1 \to \tau_2}\ \bot^{\tau_1} = \top^{\tau_2}$; $\bot^{\tau_1 \to \tau_2}\ \top^{\tau_1} = \top^{\tau_2}$; If $\tau_2 = o$ then $\bot^{\tau_1 \to \tau_2}\ \bot^{\tau_1} = \top^{\tau_2}$, otherwise $\bot^{\tau_1 \to \tau_2}\ \bot^{\tau_1} = \bot^{\tau_2}$. For $t : \tau$, we define $[\![t]\!]$ as $[\![t]\!] = \top^\tau$ if t contains a ground subterm and $[\![t]\!] = \bot^\tau$ otherwise. Then $[\![\cdot]\!]$ is a morphism since $t_1\ t_2$ contains a ground subterm iff t_1 contains a ground subterm, or t_2 contains a ground subterm, or $t_1\ t_2$ is ground.

Note that a morphism is entirely defined by its value on $\Sigma \uplus \mathcal{N}$, i.e. from those values one can compute $[\![t]\!]$ for any term t. Also remark that given a context $C[\bullet]$ and two terms t and t', if $[\![t]\!] = [\![t']\!]$ then $[\![C[t]]\!] = [\![C[t']]\!]$. We say that a morphism $[\![\cdot]\!]$ is *stable by rewriting* if for $t, t' \in \mathcal{T}$ such that $t \to t'$, $[\![t]\!] = [\![t']\!]$.

Finally, given a set of terms $\mathcal{T}' \subseteq \mathcal{T}(\Sigma \uplus \mathcal{N})$, we say that the morphism $[\![\cdot]\!]$ *recognises* $\mathcal{T}'$ if there exists a subset $\mathcal{D}'$ of $\mathcal{D}$ such that $t \in \mathcal{T}'$ if and only if $[\![t]\!] \in \mathcal{D}'$. In Example 3, the morphism recognises the set of terms containing a ground term as a subterm.

4.2 Embedding a morphism into a scheme

We fix a scheme $\mathcal{G} = \langle \mathcal{V}, \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ and a morphism $[\![\cdot]\!] : \mathcal{T}(\Sigma \uplus \mathcal{N}) \to \mathcal{D}$ on $\mathcal{G}$, stable by rewriting, such that for all type τ, $\mathcal{D}^\tau$ is finite. We transform $\mathcal{G}$ into $\mathcal{G}' = \langle \mathcal{V}', \Sigma', \mathcal{N}', \mathcal{R}', S \rangle$ which, while it is producing a derivation, evaluates $[\![t']\!]$ for any subterm t' of the current term and annotates the term with all these $\mathcal{D}$-values. The new symbols of $\mathcal{G}'$ are symbols

of $\mathcal{G}$ annotated with elements of the domain $\mathcal{D}$, and we define a transformation $(\cdot)^+$ from terms of $\mathcal{G}$ to terms of $\mathcal{G}'$ such that the transformation t^+ of t will be annotated with the $\mathcal{D}$-values of the subterms of t. The tree generated by $\mathcal{G}'$ will be annotated and when one removes these annotations, one retreives back the tree generated by $\mathcal{G}$. More precisely we show the following.

▶ **Theorem 4** (Embedding a morphism). *Given two terms $t, t' : o \in \mathcal{T}(\Sigma \uplus \mathcal{N})$, if $t \Rightarrow_{\mathcal{G}} t'$, then $t^+ \Rightarrow_{\mathcal{G}'} t'^+$. In particular $Unlab(\|\mathcal{G}'_{t^+}\|) = \|\mathcal{G}_t\|$ (where Unlab is the function that removes the annotations).*

▶ Remark. This transformation keeps the structure of the original scheme *i.e.* the new symbols are simple labelings of the original ones, new rewrite rules are obtained by duplicating some subterms and labeling the symbols, a very simple transformation allows to get back to the original scheme, and there is a direct correspondence between derivations of the two schemes.

4.3 Applications

Embedding properties of subterms during a derivation, or properties of subtrees of the value tree, can be useful for program analysis: instead of saying "There will be a forbidden behaviour in the program" reflection allows to say during the execution of a program "Here, the forbidden behaviour will appear in this subexpression, but the rest of the program is valid". Furthermore, once a property is embedded into a scheme, one can add some new rewrite rules that deal explicitly with wether the property is valid or not. For example one could replace all forbidden subtrees of the value tree by a special symbol FORBIDDEN, as illustrated in Section 5.

Some morphisms generated by type systems are used in [15] to model check a subclass of trivial acceptance condition automata (i.e. automata where there is no colouring function, and the acceptance of a tree simply asks if there exists a run of the automaton on the tree). Then the construction allows one to reflect the accepting states of the automaton (as defined in Section 3). This result has been improved in [22] to deal with the whole class of trivial acceptance condition automata. In Section 6 we extend this result to show that for any parity tree automaton one can create a morphism that reflects the acceptance of the automaton on the value tree.

One can create a model (for example in [2]) to decide whether or not a ground term t would be productive or not (i.e. $\|t\| \neq \bot$). Reflecting the productivity of terms into a scheme $\|\mathcal{G}\|$ allows one to create a scheme $\mathcal{G}'$ on the signature $\Sigma \uplus \{\bot\}$ such that $\|\mathcal{G}'\| = \|\mathcal{G}\|$, but such that no derivation will create some non productive terms. In [11] we developed this idea to compare evaluation policies.

5 An example of scheme transformation

In this section, we present a simple example that describes how one can use the embedding procedure to transform a program.

Let $\mathrm{Map}(\{0,1\})$ be the typed domain inductively defined by $\mathrm{Map}(\{0,1\})^o = \{0,1\}$, $\mathrm{Map}(\{0,1\})^{\tau \to \tau'}$ is the set of total functions from $\mathrm{Map}(\{0,1\})^\tau$ to $\mathrm{Map}(\{0,1\})^{\tau'}$. And given $f : \tau \to \tau'$ and $h : \tau$ in $\mathrm{Map}(\{0,1\})$, $f \cdot h = f(h)$.

Let $\mathcal{G} = \langle \mathcal{V}, \Sigma, \mathcal{N}, S, \mathcal{R} \rangle$ be defined by $\mathcal{V} = \{y : o \to o, x : o\}$, $\Sigma = \{\mathtt{a} : o^2 \to o, \mathtt{b} : o, \mathtt{c} : o\}$,

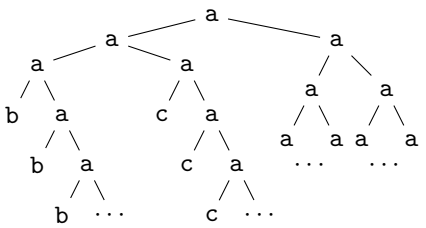


Figure 1 The value tree of the scheme of Section 5.

$\mathcal{N} = \{S : o, H : o \to o, J : o \to o, F : (o \to o) \to o\}$, $\mathcal{R}$ contains the following rewrite rules:

$$\begin{array}{lcllcl} S & \to & \mathtt{a}\ (F\ H)\ (F\ J) & H\ x & \to & \mathtt{a}\ x\ (H\ x) \\ J\ x & \to & \mathtt{a}\ (J\ x)\ (J\ x) & F\ y & \to & \mathtt{a}\ (y\ \mathtt{b})\ (y\ \mathtt{c}). \end{array}$$

The value tree of $\mathcal{G}$ is (partially) depicted in Figure 1. We write $[u, v]$ the mapping $f : \{0, 1\} \to \{0, 1\}$ such that $f(0) = u$ and $f(1) = v$ for all u, v. We define the morphism φ as follows:

$$\begin{array}{llll} \varphi(\mathtt{b}) = 0 & \varphi(\mathtt{c}) = 1 & \varphi(S) = 1 & \varphi(\mathtt{a})\ u\ v = u \vee v \\ \varphi(H)\ u = u & \varphi(J)\ u = 0 & \varphi(F)\ [u, v] = u \vee v. & \end{array}$$

One can check that the morphism φ is stable by rewrite. Furthermore it recognises the property "t has type o, and its value tree contains a $\mathtt{c}$", with the subset $A' = \{1\}$.

We construct a scheme $\mathcal{G}' = \langle \mathcal{V}', \Sigma', \mathcal{N}', S, \mathcal{R} \rangle$ that consists of an embedding of the morphism φ inside the scheme $\mathcal{G}$. $\mathcal{V}' = \{x : o,\ y^0, y^1 : o \to o\}$, $\Sigma' = \{\mathtt{a}^{0,0}, \mathtt{a}^{0,1}, \mathtt{a}^{1,0}, \mathtt{a}^{1,1} : o^2 \to o,\ \mathtt{b}, \mathtt{c} : o\}$, $\mathcal{N}' = \{S : o,\ H^0, H^1, J^0, J^1 : o,\ F^{[0,0]}, F^{[0,1]}, F^{[1,0]}, F^{[1,1]} : (o \to o)^2 \to o\}$, $\mathcal{R}'$ contains the following rewrite rules:

$$\begin{array}{lcl} S & \to & \mathtt{a}^{1,0}\ (F^{[0,1]}\ H^0\ H^1)\ (F^{[0,0]}\ J^0\ J^1) \\ H^0\ x & \to & \mathtt{a}^{0,0}\ x\ (H^0\ x) \\ H^1\ x & \to & \mathtt{a}^{1,1}\ x\ (H^0\ x) \\ J^0\ x & \to & \mathtt{a}^{0,0}\ (J^0\ x)\ (J^0\ x) \\ J^1\ x & \to & \mathtt{a}^{0,0}\ (J^1\ x)\ (J^1\ x) \\ F^{[0,0]}\ y^0\ y^1 & \to & \mathtt{a}^{0,0}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}) \\ F^{[0,1]}\ y^0\ y^1 & \to & \mathtt{a}^{0,1}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}) \\ F^{[1,0]}\ y^0\ y^1 & \to & \mathtt{a}^{1,0}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}) \\ F^{[1,1]}\ y^0\ y^1 & \to & \mathtt{a}^{1,1}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}). \end{array}$$

Let us explain how the rewrite rule related to $F^{[0,1]}$ has been produced. Recall the original rule: $F\ y \to \mathtt{a}\ (y\ \mathtt{b})\ (y\ \mathtt{c})$. The first occurrence of y is applied to $\mathtt{b}$, therefore it should be annotated with $\varphi(\mathtt{b}) = 0$. Similarly, the second occurrence of y should be annotated with $\varphi(\mathtt{c}) = 1$. This justifies the occurrence of y^0 and y^1 on the left hand part of the rule.

The annotation $[0, 1]$ means that this rule will be applied to an argument whose evaluation is the mapping $[0, 1]$, thus the evaluation of $y\ \mathtt{b}$ is equal to $[0, 1]\ \varphi(\mathtt{b}) = [0, 1]\ 0 = 0$ and by a similar reasoning the evaluation of $y\ \mathtt{c}$ is equal to 1. Therefore the occurrence of $\mathtt{a}$ should be annotated with $(0, 1)$.

We want to transform the scheme in order to forbid the derivation of a subterm when its associated value tree will not include any $\mathtt{c}$. For instance, the occurrence of a $\mathtt{c}$ would correspond to a completed service, and thus such a situation witnesses a useless derivation. In the embedded scheme, this can be detected by applying the evaluation of the head over

its annotation. For instance $F^{[0,0]}t_1\ t_2$ is the annotation of a term $F\ t$ whose value is $\varphi(F)\ [0,0] = 0$. Therefore we might turn the rule associated to $F^{[0,0]}$ into $F^{[0,0]}y_1\ y_2 \to$ FORBIDDEN, where FORBIDDEN : o is a new terminal added to the scheme. Here is the whole set of rewrite rules transformed this way.

$$
\begin{array}{lcl}
S & \to & \mathtt{a}^{1,0}\ (F^{[0,1]}\ H^0\ H^1)\ (F^{[0,0]}\ J^0\ J^1) \\
H^0\ x & \to & \text{FORBIDDEN} \\
H^1\ x & \to & \mathtt{a}^{1,1}\ x\ (H^0\ x) \\
J^0\ x & \to & \text{FORBIDDEN} \\
J^1\ x & \to & \text{FORBIDDEN} \\
F^{[0,0]}\ y^0\ y^1 & \to & \text{FORBIDDEN} \\
F^{[0,1]}\ y^0\ y^1 & \to & \mathtt{a}^{0,1}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}) \\
F^{[1,0]}\ y^0\ y^1 & \to & \mathtt{a}^{1,0}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}) \\
F^{[1,1]}\ y^0\ y^1 & \to & \mathtt{a}^{1,1}\ (y^0\ \mathtt{b})\ (y^1\ \mathtt{c}).
\end{array}
$$

6 Logical reflection

In the following we present a morphism based on [17] that recognises the acceptance of a parity tree automaton. Using the construction introduced in Section 4.2, one can construct a scheme that reflects the accepting states of the automaton, which is equivalent to reflect the subtrees accepted by a formula of the μ-calculus. In [3], μ-calculus reflection (and MSO reflection) on schemes is already proven, but this construction uses the equivalence between schemes and collapsible pushdown automata, and the successive transformations (scheme $\to$ pushdown automaton $\to$ reflective pushdown automaton $\to$ reflective scheme) lose the structure of the scheme. In our construction, the structure of the scheme is the preserved, in the sense of Remark 4.2. Since our proof of the logical reflection, as well as the one of logical selection in Section 7, is build on top of the MSO model checking proof of Kobayashi and Ong [17], we first recall their construction and then we explain how to use this result to obtain the logical reflection.

6.1 Kobayashi-Ong result

We fix a non deterministic parity tree automaton $\mathcal{A} = \langle \Sigma, Q, q_I, \delta, \Omega \rangle$ and a scheme $\mathcal{G} = \langle \Sigma, \mathcal{N}, \mathcal{V}, S, \mathcal{R} \rangle$. We let $\text{arity}_{\max}$, $\text{order}_{\max}$, and $m_{\max}$ be the maximum arity in $\Sigma \uplus \mathcal{N}$, order in $\Sigma \uplus \mathcal{N}$, colour in $\Omega(Q)$. The idea of the result is to define a type system, and to use this type system in the construction of a two player parity game, such that Eve wins the game if and only if the automaton accepts the value tree of the scheme.

The type system. Kobayashi and Ong introduced a set of *judgement rules* that allow to type a term by an element of the typed set Map, called the set of *mappings*. Mappings of type o are the states Q, and mappings of type $\tau \to \tau'$ are of the form $(\theta_1, m_1) \wedge \ldots \wedge (\theta_k, m_k) \to \theta$ with for all i θ_i is a mapping of type τ, m_i is a color, and θ is a mapping of type τ'. The judgements are of the form $\Gamma \vdash t \triangleright \theta$ meaning that under the environment Γ, one can judge t with the mapping θ. The environment Γ associates some mapping and colors to non terminal and variables, and gives some restriction on the judgements one can make. Terminals are judged according to the transition of the automaton, *i.e.* $a : o^k \to o \in \Sigma$ may be judged as $\emptyset \vdash a \triangleright (q_1, m_1) \to \ldots \to (q_k, m_k) \to q$ with for all i, $m_i = \max(\Omega(q_i), \Omega(q))$ and $q \xrightarrow{a} q_1, \ldots, q_k \in \delta$. This type system keeps track of the colours in order to know exactly what colour has been seen along the term. It is given formally in the extended version.

The game. Now we define a game $\mathbb{G}_{\mathcal{A}}$ in which Eve's states will be triples made of a non terminal, mapping and a colour, and Adam's states will be environments. Eve chooses an environment that can judge the rewrite rule of the current nonterminal with the current atomic mapping, while Adam picks one binding in the current environment. Intuitively, Eve tries to show a well-typing of the terms with respect to the rewrite rules, that would induce a well-coloured run of the automaton, and Adam tries to show that she is wrong. From the state (F, θ, m), Eve has to find an environment Γ such that she can prove $\Gamma \vdash r_F : \theta$, then Adam picks a F and θ' in Γ and asks Eve to prove that θ' is chosen correctly according to the rewrite rule of F. If at some point of a play, Eve cannot find a correct environment, she loses the play; if she can choose the empty environment, Adam would have nothing to choose then she wins the play; if the play is infinite, Eve wins if and only if the greatest colour seen infinitely often is even. The game is also given formally in the extended version.

▶ **Theorem 5** (Kobayashi, Ong 09). *The tree $\|\mathcal{G}\|$ is accepted by $\mathcal{A}$ from the state q if and only if Eve has a winning strategy from the vertex $(S, q, \Omega(q))$ in the game $\mathbb{G}_{\mathcal{A}}$.*

6.2 A morphism for automata reflection

From the Eve's winning strategy, we define a morphism $[\![\cdot]\!] : \mathcal{T}(\Sigma \uplus \mathcal{N}) \to \mathcal{D}$: the domain $\mathcal{D}$ contains the sets of mappings of the same type: for all τ, $\mathcal{D}^\tau = 2^{\mathrm{Map}^\tau}$. Given a nonterminal F, $[\![F]\!] = \{\theta \mid \exists m \text{ } \textit{Eve wins from } (F, \theta, m)\}$, given $a \in \Sigma$, $[\![a]\!] = \{\theta \mid \emptyset \vdash a : \theta\}$, and given $d : \tau_1 \to \tau_2 \in \mathcal{D}$ and $d' : \tau_1 \in \mathcal{D}$ $d\ d' = \{\theta \mid \exists (\theta_1, m_1) \wedge \ldots \wedge (\theta_k, m_k) \to \theta \in d\ \forall i\ \theta_i \in d'\}$.

▶ **Theorem 6.** *The morphism $[\![\cdot]\!]$ recognises the states of the automaton, i.e. for each state $q \in Q$ of the automaton, it recognises the set $\mathcal{T}_q = \{t : o \mid \|\mathcal{G}_t\| \models \mathcal{A}_q\}$ which is the set of ground terms whose associated value tree is recognised by the automaton from state q. Furthermore, it is stable by rewriting.*

Using the construction of Section 4.2, we have the following result.

▶ **Corollary 7** (Automata Reflection). *Higher order recursion schemes are reflective with respect to automata (hence they are reflective with respect to μ-calculus, and to* MSO*).*

7 Selection

Given a signature Σ, we recall the selection problem: given an MSO formula $\varphi[X]$ having one free monadic second order variable X, and a scheme $\mathcal{G}$ such that $\|\mathcal{G}\|$ satisfies the formula $\exists X\ \varphi[X]$, produce another scheme $\mathcal{G}'$ on the signature $\Sigma \times \{0, 1\}$ such that their exists a set S satisfying $\varphi[S]$ and $\|\mathcal{G}'\|$ is a S-marking of $\|\mathcal{G}\|$. Note that MSO selection implies MSO reflection. Indeed, being able to mark the nodes u satisfying the formula $\varphi[x]$ is equivalent to being able to mark a (unique) set satisfying $\psi[X] = \forall x\ x \in X \Leftrightarrow \varphi[x]$.

As mentioned in Section 3, MSO selection is equivalent to automata selection; we show a construction of a scheme annotating itself with an accepting run of the automaton. Since we cannot embed this problem into a morphism, we define another construction, similar to the one of Section 4.2. The construction is also based on the Kobayashi-Ong result presented in Section 5. The main difference between reflexivity and selectivity is that to construct a reflection of an automaton we embedded the winning region of the game into the scheme, while here we will embed the winning strategy of the game to prove the selection.

▶ **Theorem 8** (Automata Selection). *Higher order recursion schemes are selective with respect to automata (hence to MSO).*

Proof sketch. In the following, we give an informal glimpse on the proof, the full (technical) proof can be found in the extended version. Take a scheme $\mathcal{G}$ and an automaton $\mathcal{A}$ such that $\|\mathcal{G}\| \models \mathcal{A}$. We want to construct a scheme $\mathcal{G}'$ such that $\|\mathcal{G}'\|$ is an accepting run of $\mathcal{A}$ on $\|\mathcal{G}\|$. The first observation is that there are some great similarities between the structure of the proof trees in the type system system and the definition of a term. Indeed, one can type proofs and one can apply proofs to one another to get new proofs. For example take two terms $t_0 : o \to o$ and $t_1 : o$ and assume that we have a proof $\mathcal{P}_0$ of $t_0 \triangleright (\theta_1, m_1) \to \theta$ and a proof $\mathcal{P}_1$ of $t_1 \triangleright \theta_1$ under some environments. Then one can put together $\mathcal{P}_0$ and $\mathcal{P}_1$ to obtain a proof of $\vdash t_0\ t_1 \triangleright \theta$. We can see this proof as $\mathcal{P}_0\ \mathcal{P}_1$: the application of $\mathcal{P}_1$ to $\mathcal{P}_0$.

In the actual construction, the transformed scheme will not deal with such proofs, but we use this similarity between proofs and terms to create annotations of terms. Given a term t and a proof $\mathcal{P}$ of a judgement $\Gamma \vdash t \triangleright \theta$, we will define the annotated term $t^{\mathcal{P}}$ where each symbol is annotated by an atomic mapping and a color, verifying that if a non terminal F is annotated by (θ, m) then Γ assiciate F to (θ, m). The term $t^{\mathcal{P}}$ is somehow a trace of the proof $\mathcal{P}$. Now given a rewrite rule $F\ x_1 ... x_k \to e$ in the original scheme, we want to define for all annotated versions $F^{(\theta,m)}$ an associated rewrite rule in the transformed scheme. If (F, θ, m) is a winning vertex of the game, then Eve can choose an environment Γ such that $\Gamma \vdash e \triangleright \theta$. We take a proof $\mathcal{P}$ of this judgement and we define $F^{(\theta,m)} x_1 \dots x_k \to e^{\mathcal{P}}$. Since Eve has chosen the environment Γ with respect to her winning strategy, then for all annotated non terminals $H^{(\theta',m')}$ appearing in $e^{\mathcal{P}}$, (H, θ', m') is winning in the game. In particular, if the initial non terminal of the transformed scheme is $S^{(\theta,m)}$ with (S, θ, m) winning in the game (as it will be), any non terminal in any term of any derivation will be annotated in order to represent a winning vertex in the game. Therefore we do not need to care about which rewrite rule is chosen for $F^{(\theta,m)}$ when (F, θ, m) is not winning.

As we said, the initial non terminal is $(S, q_0, \Omega(q_0))$ which is winning in the game since $\mathcal{A}$ accepts $\|\mathcal{G}\|$ from state q_0. Any terminal a will be annotated by some $(q_1 \to \dots \to q_k \to q, m)$. Then to obtain elements of $\Sigma \times Q$, we just turn any $a^{(q_1 \to \dots \to q_k \to q, m)}$ into a non terminal and we add a rewrite rule transforming it into a^q.

The intuition of why this construction works is the following, based on the proof of the soundness of Kobayashi and Ong construction. To a derivation in the transformed scheme we associate a tree of plays in the game. The terms will be labeled by some $F^{(\theta,m)}$ that Eve has chosen in the environments she picked, and each time Adam choose one such $F^{(\theta,m)}$, it is rewritten according to Eve's strategy. Due to the colour constraints in the type systems, we can show that from the point where $F^{(\theta,m)}$ is created to the point where it is on the head of a redex, the maximum colour that has been seen is m. Furthermore, we have that along an infinite branch, there is an infinite sequence of nonterminals that are rewritten such that each non terminal is created when the previous one is rewritten. This means that we can map an infinite branch to an infinite play in the game. Furthermore the greatest colour seen infinitely often along this branch is equal to the greatest colour seen infinitely often in the sequence of maximum colours appearing between the non terminals of the sequence. And this is equal to the greatest colour seen infinitely often in the play. Since Eve wins in the game, this colour is even, then for any branch of the value tree of $\mathcal{G}'$ the greatest colour seen infinitely often is even, hence it is an accepting run. ◀

8 Conclusion

We have given new shape preserving constructions for logical reflection and logical selection using a scheme-only approach, which can be useful for correction or synthesis of programs.

The complexity is the same as in the solutions proposed so far, *i.e.* the problem is n-EXPTIME complete, and the size of the new scheme is n-EXP the size of the original one n being the order of the scheme. As possible continuation of these work, we may be interesting to see if these results scale for actual program verification, and if they can be included in tools like T-RecS [16], a model-checker for HORS.

References

1 Klaus Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. In *CSL'06*, volume 4207 of *LNCS*, pages 104–118, 2006.
2 R. M. Amadio and P.-L. Curien. *Domains and Lambda-Calculi.* CTTCS, 1998.
3 Christopher H. Broadbent, Arnaud Carayol, C.-H. Luke Ong, and Olivier Serre. Recursion schemes and logical reflection. In *LICS'10*, pages 120–129, 2010.
4 Arnaud Carayol and Olivier Serre. Collapsible pushdown automata and labeled recursion schemes. In *LICS'12*, pages 165–174, 2012.
5 Didier Caucal. On infinite terms having a decidable monadic theory. In *MFCS'02*, volume 2420 of *LNCS*, pages 165–176, 2002.
6 Bruno Courcelle. A representation of trees by languages I. *TCS*, 6:255–279, 1978.
7 Bruno Courcelle. A representation of trees by languages II. *TCS*, 7:25–55, 1978.
8 Bruno Courcelle and Maurice Nivat. The algebraic semantics of recursive program schemes. In *MFCS'78*, volume 64 of *LNCS*, pages 16–30, 1978.
9 Werner Damm. Higher type program schemes and their tree languages. In *Theoretical Computer Science, 3rd GI-Conference*, volume 48 of *LNCS*, pages 51–72, 1977.
10 Werner Damm. Languages defined by higher type program schemes. In *ICALP'77*, volume 52 of *LNCS*, pages 164–179, 1977.
11 Axel Haddad. IO vs OI in higher-order recursion schemes. In *FICS'12*, pages 23–30, 2012.
12 Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *LICS'08*, pages 452–461, 2008.
13 Klaus Indermark. Schemes with recursion on higher types. In *MFCS'76*, volume 45 of *LNCS*, pages 352–358, 1976.
14 Teodor Knapik, Damian Niwiński, and Pawel Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, volume 2303 of *LNCS*, pages 205–222, 2002.
15 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL'09*, pages 416–428, 2009.
16 Naoki Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FOSSACS'11*, pages 260–274, 2011.
17 Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS'09*, pages 179–188, 2009.
18 M. Nivat. On the interpretation of recursive program schemes. In *Symp. Mat.*, 1972.
19 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS'06*, pages 81–90, 2006.
20 M.O. Rabin. Decidability of second-order theories and automata on infinite trees. In *Trans. Amer. Math. Soc.*, 1969.
21 Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. In *ICALP'11*, pages 162–173, 2011.
22 Sylvain Salvati and Igor Walukiewicz. Using models to model-check recursive schemes. In *Typed Lambda Calculi and Applications*, pages 189–204. Springer, 2013.