A semantic interpretation of tree automata

Charles Grellois
joint work with Paul-André Melliès

PPS & LIAFA

October 11th 2013

Introduction

- A problem in verification : model-checking of MSO over trees produced by higher-order recursion schemes
- It is decidable (Ong 2006)
- Many connections with semantics appear in this result
- Our aim: obtaining this decidability result by semantic means

Introduction

- A problem in verification : model-checking of MSO over trees produced by higher-order recursion schemes
- It is decidable (Ong 2006)
- Many connections with semantics appear in this result
- Our aim: obtaining this decidability result by semantic means

Introduction

- A problem in verification : model-checking of MSO over trees produced by higher-order recursion schemes
- It is decidable (Ong 2006)
- Many connections with semantics appear in this result
- Our aim: obtaining this decidability result by semantic means

Higher-order recursion schemes

- Models of recursive programs used in verification since the 60's
- Informally : we have a ranked alphabet Σ, non-terminals, variables, an axiom and parametrized rewriting rules
- Example : $\Sigma = \{a : 2,\ b : 1,\ c : 0\}$.

Higher-order recursion schemes : an example

$$\begin{array}{c} S \rightarrow F\ c \\ F\ x \rightarrow a\ x\ (F\ (b\ x)) \end{array}$$

generates

S

Higher-order recursion schemes : an example

$$S \rightarrow F\ c$$
$$F\ x \rightarrow a\ x\ (F\ (b\ x))$$

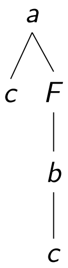
generates

$$\begin{array}{c} F \\ | \\ c \end{array}$$

Higher-order recursion schemes : an example

$$S \to F\ c$$
$$F\ x \to a\ x\ (F\ (b\ x))$$

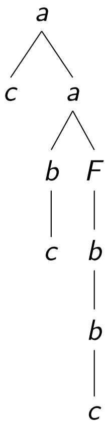
generates



Higher-order recursion schemes : an example

$$S \to F\ c$$
$$F\ x \to a\ x\ (F\ (b\ x))$$

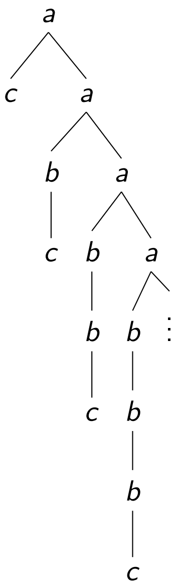
generates



Higher-order recursion schemes : an example

$$S \to F\ c$$
$$\color{red}{F\ x \to a\ x\ (F\ (b\ x))}$$

generates



λ-calculus

- λ-calculus is a calculus of functions
- It is built from variables and constants by abstraction and application
- Grammar : $t \ ::= \ x \mid \lambda x.t \mid t_1 \, t_2$

Example :

$$\lambda f.\, f\,(\,f\,(\,a\,)\,)$$

is a program taking a function f as input and applying it twice to a constant a.

λ-calculus

- λ-calculus is a calculus of functions
- It is built from variables and constants by abstraction and application
- Grammar : $t \ ::= \ x \mid \lambda x.t \mid t_1 \, t_2$

Example :

$$\lambda f.\, f\,(\,f\,(\,a\,)\,)$$

is a program taking a function f as input and applying it twice to a constant a.

λ-calculus

- λ-calculus is a calculus of functions
- It is built from variables and constants by abstraction and application
- Grammar : $t \; ::= \; x \mid \lambda x.t \mid t_1 \, t_2$

Example :

$$\lambda f.\, f\,(\,f\,(\,a\,)\,)$$

is a program taking a function f as input and applying it twice to a constant a.

Simply-typed λ-calculus

It is the fragment of the λ-calculus typable by the following rules :

Axiom $$\frac{}{\Gamma\,,\,x : A \;\vdash\; x : A}$$

Application $$\frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Abstraction $$\frac{\Gamma, x : A \;\vdash\; M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

With these rules we can type the non-terminals of recursion schemes, taking an appropriate context (in our example, $a : o \Rightarrow o \Rightarrow o \in \Gamma$).

Types = formulas of minimal logic

Simply-typed λ-calculus

It is the fragment of the λ-calculus typable by the following rules :

$$\text{Axiom} \qquad \frac{}{\Gamma\,,\, x : A \;\vdash\; x : A}$$

$$\text{Application} \qquad \frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\text{Abstraction} \qquad \frac{\Gamma, x : A \;\vdash\; M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

With these rules we can type the non-terminals of recursion schemes, taking an appropriate context (in our example, $a : o \Rightarrow o \Rightarrow o \in \Gamma$).

Types = formulas of minimal logic

Simply-typed λ-calculus

It is the fragment of the λ-calculus typable by the following rules :

Axiom $$\overline{\Gamma\,,\, x : A \ \vdash \ x : A}$$

Application $$\frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Abstraction $$\frac{\Gamma, x : A \ \vdash \ M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

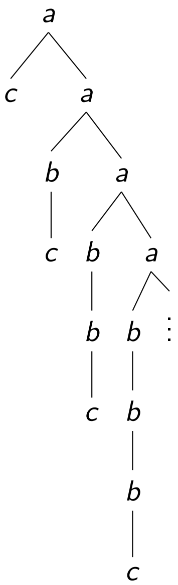
With these rules we can type the non-terminals of recursion schemes, taking an appropriate context (in our example, $a \, : \, o \Rightarrow o \Rightarrow o \in \Gamma$).

Types = formulas of minimal logic

Higher-order recursion schemes : back to the example

$$S \rightarrow F\ c$$
$$F\ x \rightarrow a\ x\ (F\ (b\ x))$$

generates

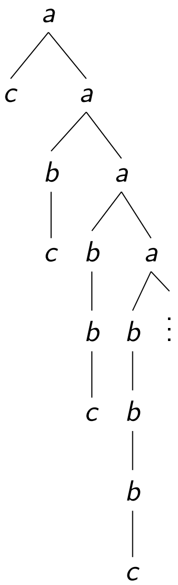


Higher-order recursion schemes : back to the example

$$S \to F\ c$$
$$F \to \lambda x.\, a\ x\ (F\ (b\ x))$$

where $F\ :\ o \Rightarrow o$

generates



Back to our verification problem

- We use recursion schemes to build trees corresponding to all the possible behaviours of a progam (= a λ-term)
- We want to express MSO properties over them
- Over such trees :

$$\text{MSO} \Leftrightarrow \text{modal } \mu\text{-calculus} \Leftrightarrow \text{alternating parity tree automata}$$

Here we will not discuss the parity issue.

Back to our verification problem

- We use recursion schemes to build trees corresponding to all the possible behaviours of a progam ($=$ a λ-term)
- We want to express MSO properties over them
- Over such trees :

$$\text{MSO} \Leftrightarrow \text{modal } \mu\text{-calculus} \Leftrightarrow \text{alternating parity tree automata}$$

Here we will not discuss the parity issue.

Back to our verification problem

- We use recursion schemes to build trees corresponding to all the possible behaviours of a progam (= a λ-term)
- We want to express MSO properties over them
- Over such trees :

$$\text{MSO} \Leftrightarrow \text{modal } \mu\text{-calculus} \Leftrightarrow \text{alternating parity tree automata}$$

Here we will not discuss the parity issue.

Back to our verification problem

- We use recursion schemes to build trees corresponding to all the possible behaviours of a progam ($=$ a λ-term)
- We want to express MSO properties over them
- Over such trees :

$$\text{MSO} \Leftrightarrow \text{modal } \mu\text{-calculus} \Leftrightarrow \text{alternating parity tree automata}$$

Here we will not discuss the parity issue.

Alternating tree automata

- A kind of (top-down) tree automata...
- ...whose run-trees have a very special shape
- During a run, an automaton can duplicate or drop a subtree
- In semantics, we know such a behaviour: the one of the exponential of linear logic !

Alternating tree automata

- A kind of (top-down) tree automata…
- …whose run-trees have a very special shape
- During a run, an automaton can duplicate or drop a subtree
- In semantics, we know such a behaviour: the one of the exponential of linear logic !

Alternating tree automata

- A kind of (top-down) tree automata. . .
- . . . whose run-trees have a very special shape
- During a run, an automaton can duplicate or drop a subtree
- In semantics, we know such a behaviour: the one of the exponential of linear logic !

Alternating tree automata

- A kind of (top-down) tree automata. . .
- . . . whose run-trees have a very special shape
- During a run, an automaton can duplicate or drop a subtree
- In semantics, we know such a behaviour: the one of the exponential of linear logic !

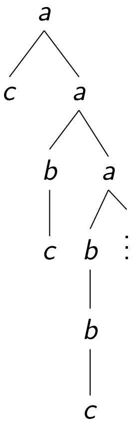
Alternating tree automata

δ maps a state and a Σ symbol to a conjunction of states to label each son :

Example : $\delta(q_0, a) \;=\; (1, q_1) \wedge (2, q_0) \wedge (2, q_2)$

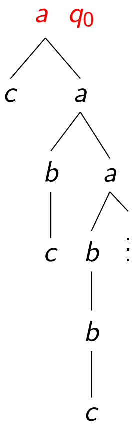
The automaton duplicates the right son, runs with state q_0 on a copy and q_2 on the other, and runs with q_1 on the left son.

Alternating tree automata: example of an execution



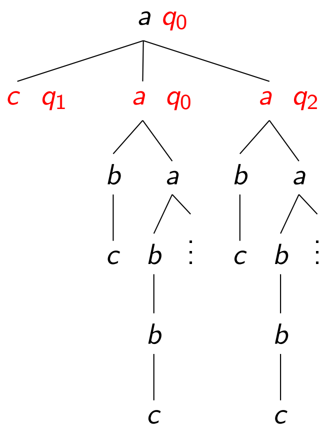
We start from state q_0.

Alternating tree automata: example of an execution



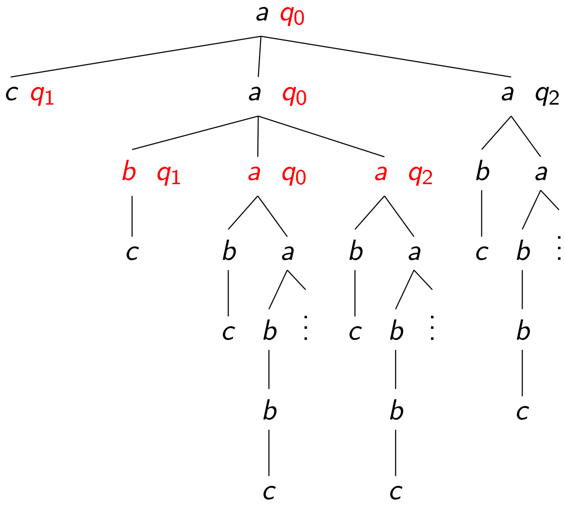
$$\delta(q_0, a) \;=\; (1, q_1) \wedge (2, q_0) \wedge (2, q_2)$$

Alternating tree automata: example of an execution



$$\delta(q_0, a) \;=\; (1, q_1) \wedge (2, q_0) \wedge (2, q_2)$$

Alternating tree automata: example of an execution



$$\delta(q_0, a) \;=\; (1, q_1) \wedge (2, q_0) \wedge (2, q_2)$$

Alternating tree automata and typing

Another point of view : δ gives a way to type the symbols of Σ.

We had $a : o \Rightarrow o \Rightarrow o$.

We will have $a : q_1 \Rightarrow (q_0 \wedge q_2) \Rightarrow q_0$: we refine types by interpreting the base type o with Q.

This extends to terms, and thus to HORS rules, for example :

$$\lambda f. \lambda x. f\ x\ x \quad : \quad (q_0 \Rightarrow q1 \Rightarrow q_2) \Rightarrow (q_0 \wedge q_1) \Rightarrow q_2$$

Alternating tree automata and typing

Another point of view : δ gives a way to type the symbols of Σ.

We had $a\ :\ o \Rightarrow o \Rightarrow o$.

We will have $a\ :\ q_1 \Rightarrow (q_0 \wedge q_2) \Rightarrow q_0$: we refine types by interpreting the base type o with Q.

This extends to terms, and thus to HORS rules, for example :

$$\lambda f.\lambda x.\, f\ x\ x \quad : \quad (q_0 \Rightarrow q1 \Rightarrow q_2) \Rightarrow (q_0 \wedge q_1) \Rightarrow q_2$$

Alternating tree automata and typing

Another point of view : δ gives a way to type the symbols of Σ.

We had $a : o \Rightarrow o \Rightarrow o$.

We will have $a : q_1 \Rightarrow (q_0 \wedge q_2) \Rightarrow q_0$: we refine types by interpreting the base type o with Q.

This extends to terms, and thus to HORS rules, for example :

$$\lambda f.\lambda x.\, f\ x\ x \quad : \quad (q_0 \Rightarrow q1 \Rightarrow q_2) \Rightarrow (q_0 \wedge q_1) \Rightarrow q_2$$

Alternating tree automata and typing

Another point of view : δ gives a way to type the symbols of Σ.

We had $a : o \Rightarrow o \Rightarrow o$.

We will have $a : q_1 \Rightarrow (q_0 \wedge q_2) \Rightarrow q_0$: we refine types by interpreting the base type o with Q.

This extends to terms, and thus to HORS rules, for example :

$$\lambda f.\lambda x.\, f\ x\ x \quad : \quad (q_0 \Rightarrow q1 \Rightarrow q_2) \Rightarrow (q_0 \wedge q_1) \Rightarrow q_2$$

Alternating tree automata and typing

Kobayashi (2009): a type system where

typability of a recursion scheme

=

its value tree is accepted by a given automaton

Model checking thus amounts to:

- Computing the biggest type for every non-terminal, which has to be consistent with rewriting
- Checking that S has type the initial state

Alternating tree automata and typing

Kobayashi (2009): a type system where

typability of a recursion scheme

$=$

its value tree is accepted by a given automaton

Model checking thus amounts to:

- Computing the biggest type for every non-terminal, which has to be consistent with rewriting
- Checking that S has type the initial state

Alternating tree automata and typing

$$\text{Axiom} \qquad \frac{\vdash \ \tau_j :: \kappa \qquad (\forall j \in J)}{\Gamma\,,\, x : \bigwedge_{j \in J} \tau_j :: \kappa \ \vdash \ x : \tau_j :: \kappa}$$

$$\text{App} \qquad \frac{\Gamma \vdash M : (\bigwedge_{j \in J} \tau_j) \rightarrow \sigma :: \kappa \rightarrow \kappa' \qquad \Gamma \vdash N : \tau_j :: \kappa \qquad (\forall j \in J)}{\Gamma \vdash MN : \sigma :: \kappa'}$$

$$\text{Lambda} \qquad \frac{\Gamma, x : \bigwedge_{j \in J} \tau_j :: \kappa \ \vdash \ M : \sigma :: \kappa'}{\Gamma \vdash \lambda x.M : (\bigwedge_{j \in J} \tau_j) \rightarrow \sigma :: \kappa \rightarrow \kappa'}$$

Refines usual λ-calculus typing with intersection types. δ types Σ symbols.

Note the duplication of typing proofs for N in the App rule.

Alternating tree automata and typing

$$\text{Axiom} \qquad \frac{\vdash \ \tau_j :: \kappa \qquad (\forall j \in J)}{\Gamma\,,\, x : \bigwedge_{j\in J} \tau_j :: \kappa \ \vdash \ x : \tau_j :: \kappa}$$

$$\text{App} \qquad \frac{\Gamma \vdash M : (\bigwedge_{j\in J} \tau_j) \to \sigma :: \kappa \to \kappa' \qquad \Gamma \vdash N : \tau_j :: \kappa \qquad (\forall j \in J)}{\Gamma \vdash MN : \sigma :: \kappa'}$$

$$\text{Lambda} \qquad \frac{\Gamma, x : \bigwedge_{j\in J} \tau_j :: \kappa \ \vdash \ M : \sigma :: \kappa'}{\Gamma \vdash \lambda x.M : (\bigwedge_{j\in J} \tau_j) \to \sigma :: \kappa \to \kappa'}$$

Refines usual λ-calculus typing with intersection types. δ types Σ symbols.

Note the duplication of typing proofs for N in the App rule.

Linear logic and intersection types

- Linear logic comes from the study of models of programming languages
- It is a logic of resources
- An hypothesis must be used exactly once – the modality ! relaxes this

The point for us :

$$A \Rightarrow B \; = !A \multimap B$$

Linear logic and intersection types

- Linear logic comes from the study of models of programming languages
- It is a logic of resources
- An hypothesis must be used exactly once – the modality ! relaxes this

The point for us :

$$A \Rightarrow B \;= !A \multimap B$$

Linear logic and intersection types

$$A \Rightarrow B \;=\; !A \multimap B$$

Meaning : a (general) function has the power to duplicate or drop every of its arguments, before using all the copies linearly.

Point of view of resource management.

If we refine o with Q, an element of $!o$ is a *collection* of states.

An element of $!o \multimap o$	$=$	several states producing exactly one state.
	$=$	an intersection of states returning a state
	$=$	the alternating behaviour of a unary symbol

Linear logic and intersection types

$$A \Rightarrow B \;=\; !A \multimap B$$

Meaning : a (general) function has the power to duplicate or drop every of its arguments, before using all the copies linearly.

Point of view of resource management.

If we refine o with Q, an element of $!o$ is a *collection* of states.

An element of $!o \multimap o$	$=$	several states producing exactly one state.
	$=$	an intersection of states returning a state
	$=$	the alternating behaviour of a unary symbol

Linear logic and intersection types

$$A \Rightarrow B \ = !A \multimap B$$

Meaning : a (general) function has the power to duplicate or drop every of its arguments, before using all the copies linearly.

Point of view of resource management.

If we refine o with Q, an element of $!o$ is a *collection* of states.

An element of $!o \multimap o$	$=$	several states producing exactly one state.
	$=$	an intersection of states returning a state
	$=$	the alternating behaviour of a unary symbol

Linear logic and intersection types

In the Application rule - as in the alternating runs - we actually need several copies (or none) of some terms/subtrees.

So the intersection looks like the ! modality

What if we consider no longer types in minimal logic, but types in linear logic instead ?

This idea leads us to equivalent type systems, in which typing amounts to interpreting terms in models of linear logic.

Linear logic and intersection types

In the Application rule - as in the alternating runs - we actually need several copies (or none) of some terms/subtrees.

So the intersection looks like the ! modality

What if we consider no longer types in minimal logic, but types in linear logic instead ?

This idea leads us to equivalent type systems, in which typing amounts to interpreting terms in models of linear logic.

Models, terms and types

- In a model, a type is interpreted as a mathematical object
- For example, in a qualitative model,

$$!o \multimap o \; = \mathcal{P}_{fin}(Q) \times Q$$

- A term is then interpreted as a subset of (the interpretation of) its type.
- Here : we can think of the interpretation of a term as the set of all its possible intersection types.
- And they give all the possible δ accepting the term !

Models, terms and types

- In a model, a type is interpreted as a mathematical object
- For example, in a qualitative model,

$$!o \multimap o \; = \mathcal{P}_{fin}(Q) \times Q$$

- A term is then interpreted as a subset of (the interpretation of) its type.
- Here : we can think of the interpretation of a term as the set of all its possible intersection types.
- And they give all the possible δ accepting the term !

Models, terms and types

- In a model, a type is interpreted as a mathematical object
- For example, in a qualitative model,

$$!o \multimap o \ = \mathcal{P}_{fin}(Q) \times Q$$

- A term is then interpreted as a subset of (the interpretation of) its type.
- Here : we can think of the interpretation of a term as the set of all its possible intersection types.
- And they give all the possible δ accepting the term !

Models, terms and types

- In a model, a type is interpreted as a mathematical object
- For example, in a qualitative model,

$$!o \multimap o \;= \mathcal{P}_{fin}(Q) \times Q$$

- A term is then interpreted as a subset of (the interpretation of) its type.
- Here : we can think of the interpretation of a term as the set of all its possible intersection types.
- And they give all the possible δ accepting the term !

Linear typing systems

We obtained two different linear typing systems for HORS.

They differ on their answer to the question: what is a *collection* of states ?

- A (coherent) set of elements of Q ?
- A multiset of elements of Q ?

The choice of the answer has an impact on the properties of the intersection of types.

Linear typing systems

We obtained two different linear typing systems for HORS.

They differ on their answer to the question: what is a *collection* of states ?

- A (coherent) set of elements of Q ?
- A multiset of elements of Q ?

The choice of the answer has an impact on the properties of the intersection of types.

Linear typing systems

We obtained two different linear typing systems for HORS.

They differ on their answer to the question: what is a *collection* of states ?

- A (coherent) set of elements of Q ?
- A multiset of elements of Q ?

The choice of the answer has an impact on the properties of the intersection of types.

Linear typing systems

The two possible answers correspond to two different kinds of models:

Set-based interpretation	=	idempotent intersection types
	=	focus on states used in the computation without counting them
	=	qualitative models

Multiset-based interpretation	=	non-idempotent intersection types
	=	states and their multiplicities
	=	quantitative models

Linear typing systems

The two possible answers correspond to two different kinds of models:

Set-based interpretation	=	idempotent intersection types
	=	focus on states used in the computation without counting them
	=	qualitative models

Multiset-based interpretation	=	non-idempotent intersection types
	=	states and their multiplicities
	=	quantitative models

Semantics of alternating automata

We obtained that in the qualitative model ScottL :

an alternating TA runs successfully run over the tree produced by a term

$=$

the intersection types given by δ belong to the semantics of the term

So that we can perform "semantic" model-checking by :

1. Computing the semantics of a term in the model
2. Translating δ into a set of intersection types
3. Checking whether this set belongs to the semantics of the term

Semantics of alternating automata

We obtained that in the qualitative model ScottL :

an alternating TA runs successfully run over the tree produced by a term

=

the intersection types given by δ belong to the semantics of the term

So that we can perform "semantic" model-checking by :

1. Computing the semantics of a term in the model
2. Translating δ into a set of intersection types
3. Checking whether this set belongs to the semantics of the term

Semantics of alternating automata

We have obtained the same result in the quantitative model Rel.

Moreover, with our type system reflecting Rel, we can describe a run over a term only by a derivation in (indexed) linear logic, which corresponds to the resource management of the automaton.

We can thus forget the term and only study an object which reflects how its behaviour affects states of an automaton.

Our next concerns

First of all, we need to deal properly with recursion and generation of infinite trees.

Then we will study the connection between:

- Rel, which reflects the (infinitary) execution of the automaton
- ScottL, the finitary model in which we can decide the existence of an alternating run

Ehrhard (2012) : "ScottL is Rel where you do not count the multiplicities"

How do this semantically explain the decidability result ?

Another point: dealing with the parity condition (= all MSO).

Our next concerns

First of all, we need to deal properly with recursion and generation of infinite trees.

Then we will study the connection between:

- Rel, which reflects the (infinitary) execution of the automaton
- ScottL, the finitary model in which we can decide the existence of an alternating run

Ehrhard (2012) : "ScottL is Rel where you do not count the multiplicities"

How do this semantically explain the decidability result ?

Another point: dealing with the parity condition (= all MSO).

Our next concerns

First of all, we need to deal properly with recursion and generation of infinite trees.

Then we will study the connection between:

- Rel, which reflects the (infinitary) execution of the automaton
- ScottL, the finitary model in which we can decide the existence of an alternating run

Ehrhard (2012) : "ScottL is Rel where you do not count the multiplicities"

How do this semantically explain the decidability result ?

Another point: dealing with the parity condition (= all MSO).

Our next concerns

First of all, we need to deal properly with recursion and generation of infinite trees.

Then we will study the connection between:

- Rel, which reflects the (infinitary) execution of the automaton
- ScottL, the finitary model in which we can decide the existence of an alternating run

Ehrhard (2012) : "ScottL is Rel where you do not count the multiplicities"

How do this semantically explain the decidability result ?

Another point: dealing with the parity condition (= all MSO).