A semantic study of higher-order model-checking

Charles Grellois (joint work with Paul-André Melliès)

PPS & LIAFA — Université Paris 7

March 3rd, 2015

Model-checking higher-order programs

A well-known approach in verification: model-checking.

- Construct a model of a program
- Specify a property in an appropriate logic
- Make them interact in order to determine whether the program satisfies the property.

Interaction is often realized by translating the formula into an equivalent automaton, which then runs over the model.

Need to balance expressivity vs. complexity in the choice of the model and of the logic.

Prologue: finite automata theory

A very naive model-checking problem

Consider the most naive possible model-checking problem where:

- Actions of the program are modelled by a finite word
- The property to check corresponds to a finite automaton

A very naive model-checking problem

A word of actions :

$$open \cdot (read \cdot write)^2 \cdot close$$

A property to check: is every *read* immediately followed by a *write* ?

Corresponds to an automaton with two states: $Q = \{q_0, q_1\}$.
q_0 is both initial and final.

A very naive model-checking problem

A word of actions :

$$open \cdot (read \cdot write)^2 \cdot close$$

A property to check: is every *read* immediately followed by a *write* ?

Corresponds to an automaton with two states: $Q = \{q_0, q_1\}$.
q_0 is both initial and final.

A very naive model-checking problem

A word of actions :

$$open \cdot (read \cdot write)^2 \cdot close$$

A property to check: is every *read* immediately followed by a *write* ?

Corresponds to an automaton with two states: $Q = \{q_0, q_1\}$.
q_0 is both initial and final.

A type-theoretic intuition

The transition function may be seen as a typing of the letters of the word, seen as function symbols.

For example,

$$\delta(q_0,\ read) \quad = \quad q_1$$

corresponds to the typing

$$read \ : \ q_1 \rightarrow q_0$$

Note that the order is reversed.

The idea is that the type of a word is a state from which the word is accepted.

A type-theoretic intuition

The transition function may be seen as a typing of the letters of the word, seen as function symbols.

For example,

$$\delta(q_0, \mathit{read}) \quad = \quad q_1$$

corresponds to the typing

$$\mathit{read} \;:\; q_1 \rightarrow q_0$$

Note that the order is reversed.

The idea is that the type of a word is a state from which the word is accepted.

A type-theoretic intuition

The transition function may be seen as a typing of the letters of the word, seen as function symbols.

For example,

$$\delta(q_0, \mathit{read}) \;=\; q_1$$

corresponds to the typing

$$\mathit{read} \;:\; q_1 \rightarrow q_0$$

Note that the order is reversed.

The idea is that the type of a word is a state from which the word is accepted.

A type-theoretic intuition: a run of the automaton

$$\frac{}{\vdash \ open \ \cdot \ (read \cdot write)^2 \cdot close \ : \ q_0}$$

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash open \ : \ q_0 \to q_0 \qquad \dfrac{}{\vdash \ (read \cdot write)^2 \cdot close \ : \ q_0}}{\vdash open \ \cdot \ (read \cdot write)^2 \cdot close \ : \ q_0}$$

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash \mathit{read} \ : \ q_1 \rightarrow q_0 \qquad \dfrac{}{\vdash \ \mathit{write} \cdot \mathit{read} \cdot \mathit{write} \cdot \mathit{close} \ : \ q_1}}{\dfrac{\vdash \ (\mathit{read} \cdot \mathit{write})^2 \cdot \mathit{close} \ : \ q_0}{\vdots}}$$

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash read \;:\; q_1 \rightarrow q_0 \qquad \dfrac{\vdash write \;:\; q_0 \rightarrow q_1 \qquad \dfrac{}{\vdash read \cdot write \cdot close \;:\; q_0}}{\vdash write \cdot read \cdot write \cdot close \;:\; q_1}}{\dfrac{\vdash (read \cdot write)^2 \cdot close \;:\; q_0}{\vdots}}$$

and so on.

Note that the set of constructors' typings define δ.

And that typing naturally extends to programs computing words.

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash read : q_1 \to q_0 \qquad \dfrac{\vdash write : q_0 \to q_1 \qquad \dfrac{}{\vdash read \cdot write \cdot close : q_0}}{\vdash write \cdot read \cdot write \cdot close : q_1}}{\dfrac{\vdash (read \cdot write)^2 \cdot close : q_0}{\vdots}}$$

and so on.

Note that the set of constructors' typings define δ.

And that typing naturally extends to programs computing words.

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash read \;:\; q_1 \to q_0 \qquad \dfrac{\vdash write \;:\; q_0 \to q_1 \qquad \dfrac{}{\vdash \; read \cdot write \cdot close \;:\; q_0}}{\vdash \; write \cdot read \cdot write \cdot close \;:\; q_1}}{\dfrac{\vdash \; (read \cdot write)^2 \cdot close \;:\; q_0}{\vdots}}$$

and so on.

Note that the set of constructors' typings define δ.

And that typing naturally extends to programs computing words.

A type-theoretic intuition: a run of the automaton

$$\dfrac{\vdash read \;:\; q_1 \rightarrow q_0 \qquad \dfrac{\vdash write \;:\; q_0 \rightarrow q_1 \qquad \dfrac{}{\vdash read \cdot write \cdot close \;:\; q_0}}{\vdash write \cdot read \cdot write \cdot close \;:\; q_1}}{\dfrac{\vdash (read \cdot write)^2 \cdot close \;:\; q_0}{\vdots}}$$

and so on.

Note that the set of constructors' typings define δ.

And that typing naturally extends to programs computing words.

Automata and recognition

Recall that, given a language $L \subseteq A^*$,

there exists a finite automaton $\mathcal{A}$ recognizing L

if and only if

there exists a finite monoid M, a subset $K \subseteq M$
and a homomorphism $\phi : A^* \to M$ such that $L = \phi^{-1}(K)$.

Roughly speaking: there exists a finite algebraic structure in which the language is interpreted.

Automata and recognition

Recall that, given a language $L \subseteq A^*$,

there exists a finite automaton $\mathcal{A}$ recognizing L

if and only if

there exists a finite monoid M, a subset $K \subseteq M$
and a homomorphism $\phi : A^* \to M$ such that $L = \phi^{-1}(K)$.

Roughly speaking: there exists a finite algebraic structure in which the language is interpreted.

A very naive model-checking problem

Now the model-checking problem can be solved by:

- computing the interpretation of a word (its denotation)
- and check whether it belongs to M

This is reminiscent of interpretations in logical models – which would allow to model-check terms as well.

Typings and interpretations. A choice for M is the one of the transition monoid of the automata. Note that it can be computed from the data of all constructors' types.

Somehow, typings compute the denotations.

A very naive model-checking problem

Now the model-checking problem can be solved by:

- computing the interpretation of a word (its denotation)
- and check whether it belongs to M

This is reminiscent of interpretations in logical models – which would allow to model-check terms as well.

Typings and interpretations. A choice for M is the one of the transition monoid of the automata. Note that it can be computed from the data of all constructors' types.

Somehow, typings compute the denotations.

A very naive model-checking problem

In this talk, we explore these ideas of typing and of interpretation in logical models, in a more general case than the one of finite words and of finite automata.

A very naive model-checking problem

A more elaborate problem: what about ultimately periodic words and Büchi automata ?

We would need some model extending the monoid's behaviour with some notion of recursion (for periodicity) which would model the Büchi condition.

Alternatively, we can do this syntactically over type derivations: we get infinite-depth derivations, over which we can check whether a final state occurs infinitely.

Ideas from the typing approach may help to define an appropriate model.

A very naive model-checking problem

A more elaborate problem: what about ultimately periodic words and Büchi automata ?

We would need some model extending the monoid's behaviour with some notion of recursion (for periodicity) which would model the Büchi condition.

Alternatively, we can do this syntactically over type derivations: we get infinite-depth derivations, over which we can check whether a final state occurs infinitely.

Ideas from the typing approach may help to define an appropriate model.

A very naive model-checking problem

A more elaborate problem: what about ultimately periodic words and Büchi automata ?

We would need some model extending the monoid's behaviour with some notion of recursion (for periodicity) which would model the Büchi condition.

Alternatively, we can do this syntactically over type derivations: we get infinite-depth derivations, over which we can check whether a final state occurs infinitely.

Ideas from the typing approach may help to define an appropriate model.

A very naive model-checking problem

A more elaborate problem: what about ultimately periodic words and Büchi automata ?

We would need some model extending the monoid's behaviour with some notion of recursion (for periodicity) which would model the Büchi condition.

Alternatively, we can do this syntactically over type derivations: we get infinite-depth derivations, over which we can check whether a final state occurs infinitely.

Ideas from the typing approach may help to define an appropriate model.

Higher-order model-checking

Model-checking higher-order programs

This work is concerned with the verification of higher-order functional programs (Haskell, OCaML, Scala, Python, Javascript, C++).

A function may take a function as input.
Examples: the `map` function, or `compose` $\phi\ x\ =\ \phi(\phi(x))$

A model for such programs is higher-order recursion schemes (HORS), generating trees describing all the potential behaviours of a program.

Properties will be expressed in MSO or modal μ-calculus (equi-expressive over trees).

Their automata counterpart is given by alternating parity automata (APT).

Model-checking higher-order programs

This work is concerned with the verification of higher-order functional programs (Haskell, OCaML, Scala, Python, Javascript, C++).

A function may take a function as input.
Examples: the `map` function, or `compose` $\phi\ x\ =\ \phi(\phi(x))$

A model for such programs is higher-order recursion schemes (HORS), generating trees describing all the potential behaviours of a program.

Properties will be expressed in MSO or modal μ-calculus (equi-expressive over trees).

Their automata counterpart is given by alternating parity automata (APT).

Model-checking higher-order programs

This work is concerned with the verification of higher-order functional programs (Haskell, OCaML, Scala, Python, Javascript, C++).

A function may take a function as input.
Examples: the `map` function, or `compose` $\phi\ x\ =\ \phi(\phi(x))$

A model for such programs is higher-order recursion schemes (HORS), generating trees describing all the potential behaviours of a program.

Properties will be expressed in MSO or modal μ-calculus (equi-expressive over trees).

Their automata counterpart is given by alternating parity automata (APT).

Model-checking higher-order programs

This work is concerned with the verification of higher-order functional programs (Haskell, OCaML, Scala, Python, Javascript, C++).

A function may take a function as input.
Examples: the `map` function, or `compose` $\phi\ x\ =\ \phi(\phi(x))$

A model for such programs is higher-order recursion schemes (HORS), generating trees describing all the potential behaviours of a program.

Properties will be expressed in MSO or modal μ-calculus (equi-expressive over trees).

Their automata counterpart is given by alternating parity automata (APT).

Model-checking higher-order programs

This model-checking problem is decidable:

- Ong 2006 (game semantics)
- Hague-Murawski-Ong-Serre 2008 (game semantics + collapsible higher-order pushdown automata)
- Kobayashi-Ong 2009 (intersection types)
- Salvati-Walukiewicz 2011 (interpretation with Krivine machines)
- Carayol-Serre 2012 (collapsible higher-order pushdown automata)
- Tsukada-Ong 2014 (game semantics)
- Salvati-Walukiewicz 2015 (interpretation in finite models)
- Grellois-Melliès 2015

Our aim was to deepen the semantic understanding we have of this result, using existing relations between alternating automata, intersection types, (linear) logic and its models – game-based as well as denotational.

Model-checking higher-order programs

Is it possible to extend to this situation the setting for finite automata ?

We would like to interpret the recursion scheme in an algebraic structure, so that

acceptance by the automata

of the tree of behaviours it generates would reduce to

checking whether some element belongs to the semantics

of the term.

Or, using an associated type system, to

check whether the term has an appropriate type.

Higher-order recursion schemes

Higher-order recursion schemes

Idea: it is a kind of grammar whose parameters may be functions and which generates ranked trees labelled by elements of a ranked alphabet Σ.

Alternatively, it is a formalism equivalent to λY calculus with uninterpreted constants of order at most one.

A very simple functional program

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen } x & = & \texttt{if } \textit{end} \texttt{ then } x \texttt{ else Listen (data } x\texttt{)} \end{array}$$

With a recursion scheme we can model this program and produce its tree of behaviours.

Note that constants are not interpreted: in particular, a recursion scheme does not evaluate a boolean conditional `if ... then ... else ...`

A very simple functional program

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen } x & = & \texttt{if } \mathit{end} \texttt{ then } x \texttt{ else Listen (data } x\texttt{)} \end{array}$$

With a recursion scheme we can model this program and produce its tree of behaviours.

Note that constants are not interpreted: in particular, a recursion scheme does not evaluate a boolean conditional `if ... then ... else ...`

A very simple functional program

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen } x & = & \texttt{if } \textit{end} \texttt{ then } x \texttt{ else Listen (data } x\texttt{)} \end{array}$$

formulated as a recursion scheme:

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L } x & = & \texttt{if } x \,(\texttt{L } (\texttt{data } x\,)\,) \end{array}$$

or, in λ-calculus style :

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L} & = & \lambda x.\texttt{if } x \,(\texttt{L } (\texttt{data } x\,)\,) \end{array}$$

(this latter representation is a regular grammar – equivalently, a λY-term)

A very simple functional program

$$\begin{array}{rcl} \texttt{Main} & = & \texttt{Listen Nil} \\ \texttt{Listen } x & = & \texttt{if } \mathit{end} \texttt{ then } x \texttt{ else Listen (data } x\texttt{)} \end{array}$$

formulated as a recursion scheme:

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L } x & = & \texttt{if } x \, (\texttt{L } (\texttt{data } x\,)\,) \end{array}$$

or, in λ-calculus style :

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L} & = & \lambda x.\texttt{if } x \, (\texttt{L } (\texttt{data } x\,)\,) \end{array}$$

(this latter representation is a regular grammar – equivalently, a λY-term)

A very simple functional program

$$
\begin{array}{rcl}
\texttt{Main} & = & \texttt{Listen Nil} \\
\texttt{Listen } x & = & \texttt{if } \textit{end} \texttt{ then } x \texttt{ else Listen (data } x\texttt{)}
\end{array}
$$

formulated as a recursion scheme:

$$
\begin{array}{rcl}
\texttt{S} & = & \texttt{L Nil} \\
\texttt{L } x & = & \texttt{if } x \texttt{ (L (data } x \texttt{))}
\end{array}
$$

or, in λ-calculus style :

$$
\begin{array}{rcl}
\texttt{S} & = & \texttt{L Nil} \\
\texttt{L} & = & \lambda x.\texttt{if } x \texttt{ (L (data } x \texttt{))}
\end{array}
$$

(this latter representation is a regular grammar – equivalently, a λY-term)

Value tree of a recursion scheme

$$\begin{array}{ccc} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{array}$$

generates:

S

Value tree of a recursion scheme

$$\begin{array}{rcl} \texttt{S} & = & \texttt{L Nil} \\ \texttt{L}\ x & = & \texttt{if}\ x\ (\texttt{L}\ (\texttt{data}\ x\ )\ ) \end{array}$$

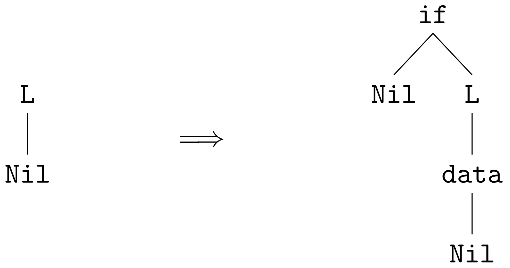
generates:

$\texttt{S} \Longrightarrow$

Value tree of a recursion scheme

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{array}$$

generates:

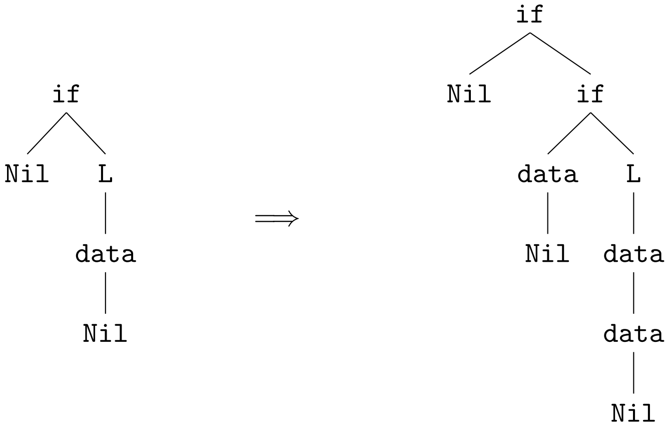


Notice that substitution and expansion occur in one same step.

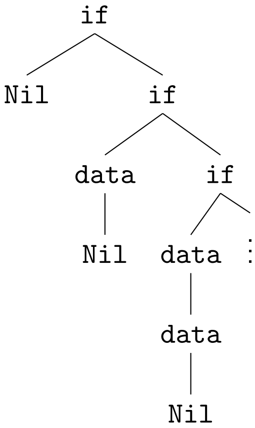
Value tree of a recursion scheme

$$\begin{array}{rcl} \mathtt{S} & = & \mathtt{L\ Nil} \\ \mathtt{L}\ x & = & \mathtt{if}\ x\ (\mathtt{L}\ (\mathtt{data}\ x\ )\ ) \end{array}$$

generates:

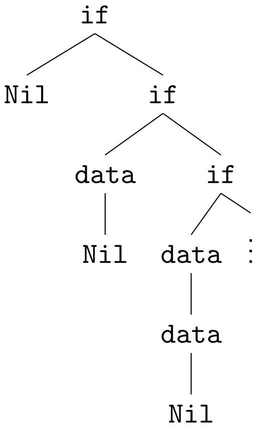


Value tree of a recursion scheme



Very simple program, yet it produces a tree which is not regular. . .

Value tree of a recursion scheme



Very simple program, yet it produces a tree which is not regular. . .

Representation of recursion schemes

The only finite representation of such a tree is actually the scheme itself — even for this very simple, order-1 recursion scheme.

This suggests that we should interpret the associated λY-term in an algebraic structure suitable for higher-order interpretations: a logical model (a domain).

Recursion schemes and models of the λY-calculus

Note that a recursion scheme may be translated to a λ-term with a recursion operator Y, and conversely (for appropriate terms).

Important consequence: recursion schemes can be interpreted in models of the λY-calculus — that is, models of the λ-calculus together with a recursion operator.

Logical specification

Alternating parity tree automata

Over trees we may use several logics: CTL, MSO,...

We focus on MSO, which is equivalent to modal μ-calculus over trees. Its automata companion model is alternating parity tree automata (APT).

APT are non-deterministic tree automata whose transitions may duplicate or drop a subtree.

Example: $\delta(q_0, \texttt{if}) \; = \; (2, q_0) \wedge (2, q_1)$.

Alternating parity tree automata

Over trees we may use several logics: CTL, MSO,. . .

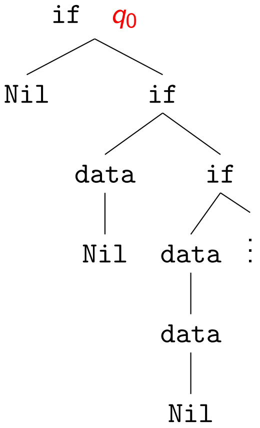
We focus on MSO, which is equivalent to modal μ-calculus over trees. Its automata companion model is alternating parity tree automata (APT).

APT are non-deterministic tree automata whose transitions may duplicate or drop a subtree.

Example: $\delta(q_0, \texttt{if}) \;=\; (2, q_0) \wedge (2, q_1)$.

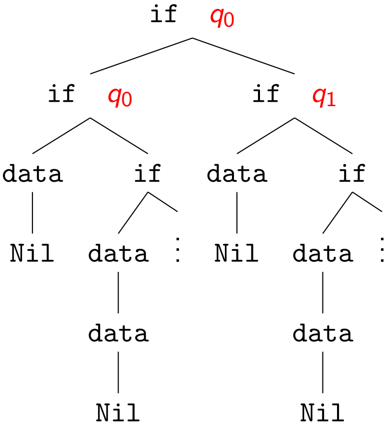
Alternating parity tree automata

$$\delta(q_0, \texttt{if}) \; = \; (2, q_0) \wedge (2, q_1).$$



Alternating parity tree automata

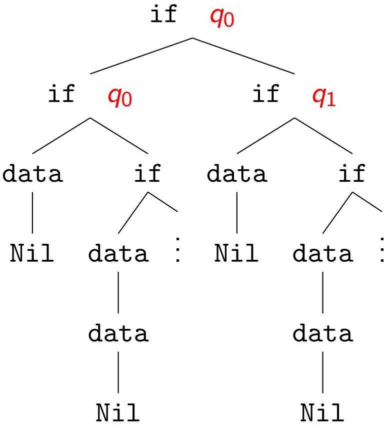
$\delta(q_0, \texttt{if}) \; = \; (2, q_0) \wedge (2, q_1).$



and so on. This gives the notion of run-tree. They are unranked.

Alternating parity tree automata

$\delta(q_0, \texttt{if}) \;=\; (2, q_0) \wedge (2, q_1).$



and so on. This gives the notion of run-tree. They are unranked.

Alternation and intersection types

Alternating parity tree automata and intersection types

A key remark (Kobayashi 2009): if $\delta(q, a) \;=\; (1, q_0) \wedge (1, q_1) \wedge (2, q_2) \ldots$

then we may consider that a has a refined intersection type

$$(q_0 \wedge q_1) \Rightarrow q_2 \Rightarrow q$$

Alternating parity tree automata and intersection types

A key remark (Kobayashi 2009): if $\delta(q, a) \; = \; (1, q_0) \wedge (1, q_1) \wedge (2, q_2) \ldots$

then we may consider that a has a refined intersection type

$$(q_0 \wedge q_1) \Rightarrow q_2 \Rightarrow q$$

Alternating parity tree automata and intersection types

This remark is very important, because unlike automata, typing lifts to higher-order.

So we may type a recursion scheme with the states of an automaton to verify if the property it expresses is satisfied.

Very important consequence: remember even very simple program models can be not regular. But schemes always are finite — and most of the time rather small.

This is a way to prove the decidability of the higher-order model-checking problem.

Alternating parity tree automata and intersection types

This remark is very important, because unlike automata, typing lifts to higher-order.

So we may type a recursion scheme with the states of an automaton to verify if the property it expresses is satisfied.

Very important consequence: remember even very simple program models can be not regular. But schemes always are finite — and most of the time rather small.

This is a way to prove the decidability of the higher-order model-checking problem.

A type-system for verification: without colours

Axiom $\dfrac{}{x\,:\,\bigwedge_{\{i\}}\ \theta_i\ ::\ \kappa\ \vdash\ x\,:\,\theta_i\ ::\ \kappa}$

δ $\dfrac{\{\,(i, q_{ij})\mid 1 \leq i \leq n, 1 \leq j \leq k_i\}\quad \text{satisfies}\quad \delta_A(q,a)}{\emptyset \vdash a\,:\,\bigwedge_{j=1}^{k_1}\ q_{1j} \rightarrow \ldots \rightarrow \bigwedge_{j=1}^{k_n}\ q_{nj} \rightarrow q\ ::\ \perp \rightarrow \cdots \rightarrow \perp}$

App $\dfrac{\Delta \vdash t : (\ \theta_1\ \wedge \cdots \wedge\ \theta_k) \rightarrow \theta\ ::\ \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u\ :\ \theta_i\ ::\ \kappa}{\Delta + \Delta_1 + \ldots + \Delta_k\ \vdash\ t\,u\ :\ \theta :: \kappa'}$

λ $\dfrac{\Delta\,,\,x\,:\,\bigwedge_{i \in I}\ \theta_i\ ::\ \kappa\ \vdash\ t\,:\,\theta\ ::\ \kappa' \qquad\qquad I \subseteq J}{\Delta\ \vdash\ \lambda x\,.\,t\ :\ \left(\bigwedge_{j \in J}\ \theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$

fix $\dfrac{\Gamma \vdash \mathcal{R}(F)\ :\ \theta\ ::\ \kappa}{F\ :\ \theta\ ::\ \kappa \vdash\ F\ :\ \theta\ ::\ \kappa}$

A type-system for verification: without colours

Axiom $\qquad \dfrac{}{x \,:\, \bigwedge_{\{i\}} \;\theta_i \,::\, \kappa \;\vdash\; x \,:\, \theta_i \,::\, \kappa}$

$$\delta \qquad \frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\}\quad \text{satisfies} \quad \delta_A(q,a)}{\emptyset \vdash a \,:\, \bigwedge_{j=1}^{k_1} \; q_{1j} \,\rightarrow\, \dots \,\rightarrow\, \bigwedge_{j=1}^{k_n} \; q_{nj} \rightarrow q \,::\, \bot \rightarrow \cdots \rightarrow \bot}$$

$$\text{App} \qquad \frac{\Delta \vdash t : (\,\theta_1 \;\wedge \cdots \wedge\; \theta_k) \rightarrow \theta \,::\, \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u \,:\, \theta_i \,::\, \kappa}{\Delta + \Delta_1 + \ldots + \Delta_k \;\vdash\; t\,u \,:\, \theta :: \kappa'}$$

$$\lambda \qquad \frac{\Delta\,,\, x \,:\, \bigwedge_{i \in I} \;\theta_i \,::\, \kappa \;\vdash\; t \,:\, \theta \,::\, \kappa' \qquad\qquad I \subseteq J}{\Delta \;\vdash\; \lambda x\,.\,t \,:\, \left(\bigwedge_{j \in J} \;\theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$$

$$\mathit{fix} \qquad \frac{\Gamma \vdash \mathcal{R}(F) \,:\, \theta \,::\, \kappa}{F \,:\, \theta \,::\, \kappa \vdash F \,:\, \theta \,::\, \kappa}$$

A type-system for verification: without colours

Axiom
$$\frac{}{x : \bigwedge_{\{i\}} \ \theta_i :: \kappa \ \vdash \ x : \theta_i :: \kappa}$$

δ
$$\frac{\{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a : \bigwedge_{j=1}^{k_1} \ q_{1j} \rightarrow \ldots \rightarrow \bigwedge_{j=1}^{k_n} \ q_{nj} \rightarrow q :: \perp \rightarrow \cdots \rightarrow \perp}$$

App
$$\frac{\Delta \vdash t : (\ \theta_1 \ \wedge \cdots \wedge \ \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta + \Delta_1 + \ldots + \Delta_k \ \vdash \ t\, u : \theta :: \kappa'}$$

λ
$$\frac{\Delta, x : \bigwedge_{i \in I} \ \theta_i :: \kappa \ \vdash \ t : \theta :: \kappa' \qquad I \subseteq J}{\Delta \ \vdash \ \lambda x . t : \left(\bigwedge_{j \in J} \ \theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$$

fix
$$\frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \theta :: \kappa \vdash F : \theta :: \kappa}$$

A type-system for verification: without colours

Axiom
$$\frac{}{x \,:\, \bigwedge_{\{i\}} \;\theta_i \,::\, \kappa \;\vdash\; x \,:\, \theta_i \,::\, \kappa}$$

δ
$$\frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\,\} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a \,:\, \bigwedge_{j=1}^{k_1} \; q_{1j} \rightarrow \ldots \rightarrow \bigwedge_{j=1}^{k_n} \; q_{nj} \rightarrow q \,::\, \bot \rightarrow \cdots \rightarrow \bot}$$

App
$$\frac{\Delta \vdash t : (\;\theta_1 \;\wedge \cdots \wedge\; \theta_k) \rightarrow \theta \,::\, \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u \,:\, \theta_i \,::\, \kappa}{\Delta + \Delta_1 + \ldots + \Delta_k \;\vdash\; t\,u \,:\, \theta :: \kappa'}$$

λ
$$\frac{\Delta\,, x \,:\, \bigwedge_{i \in I} \;\theta_i \,::\, \kappa \;\vdash\; t \,:\, \theta \,::\, \kappa' \qquad\qquad I \subseteq J}{\Delta \;\vdash\; \lambda\, x\,.\, t \,:\, \left(\bigwedge_{j \in J} \;\theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$$

fix
$$\frac{\Gamma \vdash \mathcal{R}(F) \,:\, \theta \,::\, \kappa}{F \,:\, \theta \,::\, \kappa \vdash F \,:\, \theta \,::\, \kappa}$$

A type-system for verification: without colours

Axiom $$\frac{}{x : \bigwedge_{\{i\}} \ \theta_i :: \kappa \ \vdash \ x : \theta_i :: \kappa}$$

δ $$\frac{\{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a : \bigwedge_{j=1}^{k_1} \ q_{1j} \rightarrow \ldots \rightarrow \bigwedge_{j=1}^{k_n} \ q_{nj} \rightarrow q :: \bot \rightarrow \cdots \rightarrow \bot}$$

App $$\frac{\Delta \vdash t : (\ \theta_1 \ \wedge \cdots \wedge \ \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta + \Delta_1 + \ldots + \Delta_k \ \vdash \ t\,u : \theta :: \kappa'}$$

λ $$\frac{\Delta, x : \bigwedge_{i \in I} \ \theta_i :: \kappa \ \vdash \ t : \theta :: \kappa' \qquad\qquad I \subseteq J}{\Delta \ \vdash \ \lambda x . t : \left(\bigwedge_{j \in J} \ \theta_j \right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$$

fix $$\frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \theta :: \kappa \vdash F : \theta :: \kappa}$$

A type-system for verification

In this type system, there is a proof of

$$S \,:\, \bigwedge q_0 :: o \;\vdash\; S \,:\, q_0 :: o$$

if and only if the alternating automaton has a run-tree over $[\![\mathcal{G}]\!]$.

Note that these intersection types are idempotent:

$$q_0 \wedge q_0 \quad = \quad q_0$$

Intersection type systems have been studied a lot in semantics.

Moreover, for some appropriate intersection type systems, derivations may be understood via game semantics as the construction of denotations in associated models of linear logic.

A type-system for verification

In this type system, there is a proof of

$$S : \bigwedge q_0 :: o \vdash S : q_0 :: o$$

if and only if the alternating automaton has a run-tree over $[\![\mathcal{G}]\!]$.

Note that these intersection types are idempotent:

$$q_0 \wedge q_0 \quad = \quad q_0$$

Intersection type systems have been studied a lot in semantics.

Moreover, for some appropriate intersection type systems, derivations may be understood via game semantics as the construction of denotations in associated models of linear logic.

A type-system for verification

In this type system, there is a proof of

$$S : \bigwedge q_0 :: o \vdash S : q_0 :: o$$

if and only if the alternating automaton has a run-tree over $[\![\mathcal{G}]\!]$.

Note that these intersection types are idempotent:

$$q_0 \wedge q_0 \quad = \quad q_0$$

Intersection type systems have been studied a lot in semantics.

Moreover, for some appropriate intersection type systems, derivations may be understood via game semantics as the construction of denotations in associated models of linear logic.

Linear models of the λ-calculus

Linear decomposition of the intuitionnistic arrow

In linear logic, the intuitionnistic arrow $A \Rightarrow B$ factors as

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

In other terms, in a model of linear logic, a program of type $A \Rightarrow B$ is interpreted as a replication of its inputs, followed by a linear use of them.

Note that, given a categorical model of linear logic (with a suitable interpretation of !), considering only morphisms

$$!A \multimap B$$

automatically gives a model of λ-calculus (Kleisli construction).

Linear decomposition of the intuitionnistic arrow

In linear logic, the intuitionnistic arrow $A \Rightarrow B$ factors as

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

In other terms, in a model of linear logic, a program of type $A \Rightarrow B$ is interpreted as a replication of its inputs, followed by a linear use of them.

Note that, given a categorical model of linear logic (with a suitable interpretation of !), considering only morphisms

$$!\, A \multimap B$$

automatically gives a model of λ-calculus (Kleisli construction).

Linear decomposition of the intuitionnistic arrow

In linear logic, the intuitionnistic arrow $A \Rightarrow B$ factors as

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

In other terms, in a model of linear logic, a program of type $A \Rightarrow B$ is interpreted as a replication of its inputs, followed by a linear use of them.

Note that, given a categorical model of linear logic (with a suitable interpretation of !), considering only morphisms

$$!\, A \multimap B$$

automatically gives a model of λ-calculus (Kleisli construction).

Linear decomposition of the intuitionnistic arrow

With a “suitable interpretation” of ! comes an identity morphism

$$!\,A \multimap A$$

which uses an element of A once, and outputs it, and a comultiplication morphism used to define compositions:

$$!\,A \xrightarrow{\;comult\;} !\,!\,A \xrightarrow{\;!\,f\;} !\,B \xrightarrow{\;g\;} C$$

Models of linear logic

We would typically like to understand the refined intersection typing

$$a \ : \ (q_0 \wedge q_1) \Rightarrow q_2 \Rightarrow q \ :: \ o \rightarrow o \rightarrow o$$

as the fact that

$$(\{ q_0, q_1 \}, \{ q_2 \}, q) \quad \in \quad [\![a]\!]$$

However, set-based interpretations of the exponential lead to complicated models of linear logic.

Some additional ordering on sets is required, as well as a saturation property – roughly speaking, if a morphism can compute b out of X, it can also compute a worse output a out of a better input Y.

Models of linear logic

We would typically like to understand the refined intersection typing

$$a \;:\; (q_0 \wedge q_1) \Rightarrow q_2 \Rightarrow q \;::\; o \rightarrow o \rightarrow o$$

as the fact that

$$(\{\, q_0,\; q_1 \,\}, \{\, q_2 \,\},\; q) \quad \in \quad [\![a]\!]$$

However, set-based interpretations of the exponential lead to complicated models of linear logic.

Some additional ordering on sets is required, as well as a saturation property – roughly speaking, if a morphism can compute b out of X, it can also compute a worse output a out of a better input Y.

Models of linear logic

There are thus two main classes of denotational models of linear logic:

- qualitative models: the exponential modality enumerates the resources used by a program, but not their multiplicity,
- quantitative models, in which the number of occurences of a resource is precisely tracked.

The former use a set-based interpretation of the exponential, the latter a multiset-based one.

Models of linear logic

Typing in Kobayashi's system corresponds to interpretation in a qualitative model of linear logic — due to idempotency of types, multiplicities are not accounted for.

(only works for η-long forms. . .)

It is interesting to consider quantitative interpretations as well – they are bigger, yet simpler.

They correspond to non-idempotent intersection types.

Models of linear logic

Typing in Kobayashi's system corresponds to interpretation in a qualitative model of linear logic — due to idempotency of types, multiplicities are not accounted for.

(only works for η-long forms. . .)

It is interesting to consider quantitative interpretations as well – they are bigger, yet simpler.

They correspond to non-idempotent intersection types.

Relational model of linear logic

Consider the relational model, in which

- $[\![o]\!] \;=\; Q$
- $[\![A \multimap B]\!] \;=\; [\![A]\!] \times [\![B]\!]$
- $[\![!A]\!] \;=\; \mathcal{M}_{fin}([\![A]\!])$

where $\mathcal{M}_{fin}(A)$ is the set of finite multisets of elements of $[\![A]\!]$.

We have

$$[\![A \Rightarrow B]\!] \quad = \quad \mathcal{M}_{fin}([\![A]\!]) \times [\![B]\!]$$

It is some collection (with multiplicities) of elements of $[\![A]\!]$ producing an element of $[\![B]\!]$.

Relational model of linear logic

Consider the relational model, in which

- $[\![o]\!] \; = \; Q$
- $[\![A \multimap B]\!] \; = \; [\![A]\!] \times [\![B]\!]$
- $[\![!A]\!] \; = \; \mathcal{M}_{fin}([\![A]\!])$

where $\mathcal{M}_{fin}(A)$ is the set of finite multisets of elements of $[\![A]\!]$.

We have

$$[\![A \Rightarrow B]\!] \quad = \quad \mathcal{M}_{fin}([\![A]\!]) \times [\![B]\!]$$

It is some collection (with multiplicities) of elements of $[\![A]\!]$ producing an element of $[\![B]\!]$.

Relational model of linear logic

Consider the relational model, in which

- $[\![o]\!] \;=\; Q$
- $[\![A \multimap B]\!] \;=\; [\![A]\!] \times [\![B]\!]$
- $[\![!A]\!] \;=\; \mathcal{M}_{fin}([\![A]\!])$

where $\mathcal{M}_{fin}(A)$ is the set of finite multisets of elements of $[\![A]\!]$.

We have

$$[\![A \Rightarrow B]\!] \quad = \quad \mathcal{M}_{fin}([\![A]\!]) \times [\![B]\!]$$

It is some collection (with multiplicities) of elements of $[\![A]\!]$ producing an element of $[\![B]\!]$.

Intersection types and relational interpretations

Consider again the typing

$$a \,:\, (q_0 \wedge q_1) \rightarrow q_2 \rightarrow q \,::\, \perp \rightarrow \perp \rightarrow \perp$$

In the relational model:

$$[\![A]\!] \subseteq \mathcal{M}_{fin}(Q) \times \mathcal{M}_{fin}(Q) \times Q$$

and this example translates as

$$([q_0, q_1],\, [q_2], q) \in [\![a]\!]$$

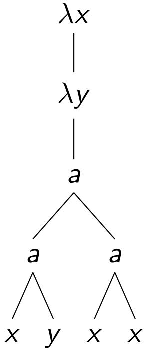
An example of interpretation

Terms are interpreted as subsets of the interpretation of their simple type.

Consider the rule

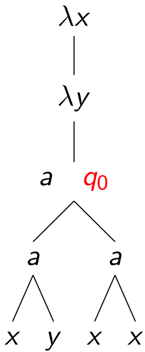
$$F\ x\ y\ =\ a\,(\,a\ x\ y\,)\,(\,a\ x\ x\,)$$

which corresponds to



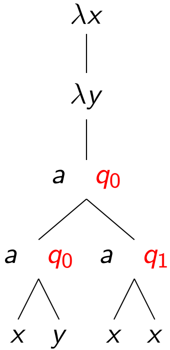
An example of interpretation

and suppose that $\mathcal{A}$ may run as follows on the tree:

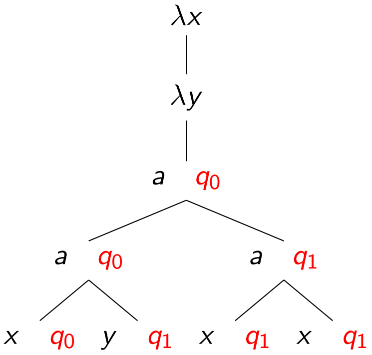


An example of interpretation

and suppose that $\mathcal{A}$ may run as follows on the tree:



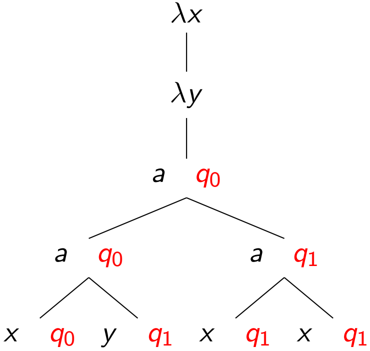
An example of interpretation



Then this rule will be interpreted in the model as

$$([q_0,\ q_1,\ q_1],\ [q_1],\ q_0)$$

An example of interpretation



Then this rule will be interpreted in the model as

$$([q_0, q_1, q_1], [q_1], q_0)$$

Relational interpretation and automata acceptance

Theorem (G.-Melliès 2014)

Consider an alternating tree automaton $\mathcal{A}$ *and a* λ*-term* t *reducing to a tree* T.

Then $\mathcal{A}$ *has a run-tree over* T *if and only if*

$$\{q_0\} \subseteq [\![t]\!]$$

where the interpretation is computed in the relational model.

Linear models of the λ-calculus with recursion

Recursion: the *fix* rule

This model lacks recursion, and can not interpret in general the rule

$$fix \quad \frac{\Gamma \vdash \mathcal{R}(F) \, : \, \theta \, :: \, \kappa}{F \, : \, \theta \, :: \, \kappa \, \vdash \, F \, : \theta \, :: \, \kappa}$$

Since schemes produce infinite trees, we need to shift to an infinitary variant of *Rel*.

We consider a new exponential modality $\lightning$:

$$[\![\lightning \, A]\!] \quad = \quad \mathcal{M}_{count}([\![A]\!])$$

(finite-or-countable multisets)

This exponential $\lightning$ satisfies the required axiom, and thus gives immediately an infinitary model of the λ-calculus.

Recursion: the *fix* rule

This model lacks recursion, and can not interpret in general the rule

$$fix \quad \frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \theta :: \kappa \vdash F : \theta :: \kappa}$$

Since schemes produce infinite trees, we need to shift to an infinitary variant of *Rel*.

We consider a new exponential modality $\lightning$:

$$[\![\lightning A]\!] \quad = \quad \mathcal{M}_{count}([\![A]\!])$$

(finite-or-countable multisets)

This exponential $\lightning$ satisfies the required axiom, and thus gives immediately an infinitary model of the λ-calculus.

Recursion: the *fix* rule

A function may now have a countable number of inputs.

This model has a coinductive fixpoint. The Theorem then extends:

Theorem (G.-Melliès 2014)

Consider an alternating tree automaton $\mathcal{A}$ and a λY-term t producing a tree T (or, equivalently, a higher-order recursion scheme).

Then $\mathcal{A}$ has a run-tree over T if and only if

$$\{q_0\} \subseteq [\![t]\!]$$

where the recursion operator of the lambda-calculus is computed using the coinductive fixed point operator of the infinitary relational model.

Recursion: the *fix* rule

A function may now have a countable number of inputs.

This model has a coinductive fixpoint. The Theorem then extends:

Theorem (G.-Melliès 2014)

Consider an alternating tree automaton $\mathcal{A}$ and a λY-term t producing a tree T (or, equivalently, a higher-order recursion scheme).

Then $\mathcal{A}$ has a run-tree over T if and only if

$$\{q_0\} \subseteq [\![t]\!]$$

where the recursion operator of the lambda-calculus is computed using the coinductive fixed point operator of the infinitary relational model.

Specifying inductive and coinductive behaviours: parity conditions

Alternating parity tree automata

Modal μ-calculus allows to discriminate inductive from coinductive behaviour.

This allows to express properties as

"a given operation is executed infinitely often in some execution"

or

"after a `read` operation, a `write` eventually occurs".

Alternating parity tree automata

In the APT, this inductive-coinductive policy is encoded using parity conditions. Every state receives a colour

$$\Omega(q) \in Col \subseteq \mathbb{N}$$

Say that an infinite branch of a run-tree is winning iff the maximal colour among the ones occuring infinitely often along it is even.

Say that a run-tree is winning iff all of its infinite branches are.

Then an APT has a winning run-tree over a tree T iff the root of T satisfies the corresponding MSO formula ϕ.

Alternating parity tree automata

In the APT, this inductive-coinductive policy is encoded using parity conditions. Every state receives a colour

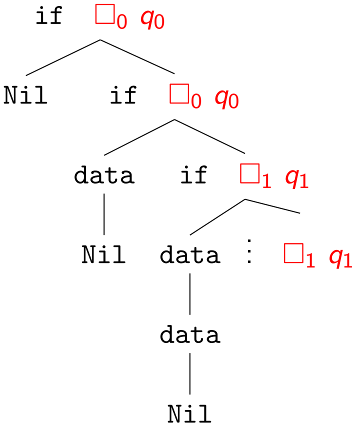
$$\Omega(q) \in Col \subseteq \mathbb{N}$$

Say that an infinite branch of a run-tree is winning iff the maximal colour among the ones occuring infinitely often along it is even.

Say that a run-tree is winning iff all of its infinite branches are.

Then an APT has a winning run-tree over a tree T iff the root of T satisfies the corresponding MSO formula ϕ.

Parity condition on an example



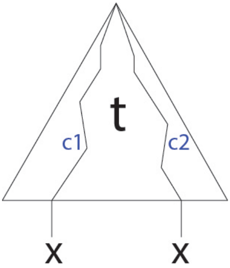
would not be a winning run-tree.

Parity conditions

Kobayashi and Ong extend the typings with a colouring operation:

$$a \;:\; (\emptyset \rightarrow \Box_{c_2}\ q_2 \rightarrow q_0) \wedge ((\Box_{c_1}\ q_1 \wedge \Box_{c_2}\ q_2) \rightarrow \Box_{c_0}\ q_0 \rightarrow q_0)$$

This operation lifts to higher-order.



In this setting, t will have some type $\Box_{c_1}\ \sigma_1 \wedge \Box_{c_2}\ \sigma_2 \rightarrow \tau$.

A type-system for verification (Grellois-Melliès 2014)

Axiom
$$\frac{}{x : \bigwedge_{\{i\}} \Box_{-1}\, \theta_i :: \kappa \ \vdash\ x : \theta_i :: \kappa}$$

δ
$$\frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\,\} \quad \text{satisfies} \quad \delta_A(q,a)}{\emptyset \vdash a : \bigwedge_{j=1}^{k_1} \Box_{m_{1j}}\, q_{1j} \rightarrow \ldots \rightarrow \bigwedge_{j=1}^{k_n} \Box_{m_{nj}}\, q_{nj} \rightarrow q :: \bot \rightarrow \cdots \rightarrow \bot \rightarrow \bot}$$

App
$$\frac{\Delta \vdash t : (\Box_{m_1}\, \theta_1 \wedge \cdots \wedge \Box_{m_k}\, \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \qquad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta + \Box_{m_1} \Delta_1 + \ldots + \Box_{m_k} \Delta_k \ \vdash\ t\, u : \theta :: \kappa'}$$

fix
$$\frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \Box_{-1}\, \theta :: \kappa \vdash F : \theta :: \kappa}$$

λ
$$\frac{\Delta, x : \bigwedge_{i \in I} \Box_{m_i}\, \theta_i :: \kappa \ \vdash\ t : \theta :: \kappa' \qquad I \subseteq J}{\Delta \ \vdash\ \lambda x . t : \left(\bigwedge_{j \in J} \Box_{m_j}\, \theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa'}$$

A type-system for verification (Grellois-Melliès 2014)

Axiom $$\frac{}{x : \bigwedge_{\{i\}} \Box_{-1}\, \theta_i :: \kappa \;\vdash\; x : \theta_i :: \kappa}$$

δ $$\frac{\{(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\} \quad \text{satisfies} \quad \delta_A(q,a)}{\emptyset \vdash a : \bigwedge_{j=1}^{k_1} \Box_{m_{1j}}\, q_{1j} \to \ldots \to \bigwedge_{j=1}^{k_n} \Box_{m_{nj}}\, q_{nj} \to q :: \bot \to \cdots \to \bot \to \bot}$$

App $$\frac{\Delta \vdash t : (\Box_{m_1}\, \theta_1 \wedge \cdots \wedge \Box_{m_k}\, \theta_k) \to \theta :: \kappa \to \kappa' \qquad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta + \Box_{m_1}\Delta_1 + \ldots + \Box_{m_k}\Delta_k \;\vdash\; t\, u : \theta :: \kappa'}$$

fix $$\frac{\Gamma \vdash \mathcal{R}(F) : \theta :: \kappa}{F : \Box_{-1}\, \theta :: \kappa \vdash F : \theta :: \kappa}$$

λ $$\frac{\Delta, x : \bigwedge_{i \in I} \Box_{m_i}\, \theta_i :: \kappa \;\vdash\; t : \theta :: \kappa' \qquad I \subseteq J}{\Delta \;\vdash\; \lambda x . t : \left(\bigwedge_{j \in J} \Box_{m_j}\, \theta_j\right) \to \theta :: \kappa \to \kappa'}$$

A type-system for verification (Grellois-Melliès 2014)

Axiom $\dfrac{}{x \,:\, \bigwedge_{\{i\}} \square_{-1}\, \theta_i \,::\, \kappa \;\vdash\; x \,:\, \theta_i \,::\, \kappa}$

$$\delta \quad \frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\,\} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a \,:\, \bigwedge_{j=1}^{k_1} \square_{m_{1j}}\, q_{1j} \to \ldots \to \bigwedge_{j=1}^{k_n} \square_{m_{nj}}\, q_{nj} \to q \,::\, \bot \to \cdots \to \bot \to \bot}$$

$$\text{App} \quad \frac{\Delta \vdash t : (\square_{m_1}\, \theta_1 \wedge \cdots \wedge \square_{m_k}\, \theta_k) \to \theta \,::\, \kappa \to \kappa' \qquad \Delta_i \vdash u \,:\, \theta_i \,::\, \kappa}{\Delta + \square_{m_1} \Delta_1 + \ldots + \square_{m_k} \Delta_k \;\vdash\; t\, u \,:\, \theta :: \kappa'}$$

$$fix \quad \frac{\Gamma \vdash \mathcal{R}(F) \,:\, \theta \,::\, \kappa}{F \,:\, \square_{-1}\, \theta \,::\, \kappa \vdash F \,:\, \theta \,::\, \kappa}$$

$$\lambda \quad \frac{\Delta, x \,:\, \bigwedge_{i \in I} \square_{m_i}\, \theta_i \,::\, \kappa \;\vdash\; t \,:\, \theta \,::\, \kappa' \qquad I \subseteq J}{\Delta \;\vdash\; \lambda x . t \,:\, \left(\bigwedge_{j \in J} \square_{m_j}\, \theta_j\right) \to \theta \,::\, \kappa \to \kappa'}$$

A type-system for verification (Grellois-Melliès 2014)

Axiom $$\frac{}{x \,:\, \bigwedge_{\{i\}} \square_{-1}\, \theta_i \,::\, \kappa \;\vdash\; x \,:\, \theta_i \,::\, \kappa}$$

δ $$\frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\,\} \quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a \,:\, \bigwedge_{j=1}^{k_1} \square_{m_{1j}}\, q_{1j} \to \ \ldots \ \to \bigwedge_{j=1}^{k_n} \square_{m_{nj}}\, q_{nj} \to q \,::\, \bot \to \cdots \to \bot \to \bot}$$

App $$\frac{\Delta \vdash t : (\square_{m_1}\, \theta_1 \ \wedge \cdots \wedge \square_{m_k}\, \theta_k) \to \theta \,::\, \kappa \to \kappa' \qquad \Delta_i \vdash u \,:\, \theta_i \,::\, \kappa}{\Delta + \square_{m_1} \Delta_1 + \ldots + \square_{m_k} \Delta_k \;\vdash\; t\, u \,:\, \theta :: \kappa'}$$

fix $$\frac{\Gamma \vdash \mathcal{R}(F) \,:\, \theta \,::\, \kappa}{F \,:\, \square_{-1}\, \theta \,::\, \kappa \vdash F \,:\, \theta \,::\, \kappa}$$

λ $$\frac{\Delta, x : \bigwedge_{i \in I} \square_{m_i}\, \theta_i \,::\, \kappa \;\vdash\; t \,:\, \theta :: \kappa' \qquad I \subseteq J}{\Delta \;\vdash\; \lambda x \,.\, t \,:\, \left(\bigwedge_{j \in J} \square_{m_j}\, \theta_j\right) \to \theta :: \kappa \to \kappa'}$$

A type-system for verification (Grellois-Melliès 2014)

Axiom $$\frac{}{x \,:\, \bigwedge_{\{i\}} \square_{-1}\, \theta_i \,::\, \kappa \;\vdash\; x \,:\, \theta_i \,::\, \kappa}$$

δ $$\frac{\{\,(i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\}\quad \text{satisfies} \quad \delta_A(q, a)}{\emptyset \vdash a \,:\, \bigwedge_{j=1}^{k_1} \square_{m_{1j}}\, q_{1j} \to \ \ldots \ \to \bigwedge_{j=1}^{k_n} \square_{m_{nj}}\, q_{nj} \to q \,::\, \bot \to \cdots \to \bot \to \bot}$$

App $$\frac{\Delta \vdash t : (\square_{m_1}\, \theta_1 \ \wedge \cdots \wedge \square_{m_k}\, \theta_k) \to \theta \,::\, \kappa \to \kappa' \qquad \Delta_i \vdash u \,:\, \theta_i \,::\, \kappa}{\Delta + \square_{m_1}\Delta_1 + \ldots + \square_{m_k}\Delta_k \;\vdash\; t\, u \,:\, \theta :: \kappa'}$$

fix $$\frac{\Gamma \vdash \mathcal{R}(F) \,:\, \theta \,::\, \kappa}{F \,:\, \square_{-1}\, \theta \,::\, \kappa \vdash F \,:\, \theta \,::\, \kappa}$$

λ $$\frac{\Delta\,, x \,:\, \bigwedge_{i \in I} \square_{m_i}\, \theta_i \,::\, \kappa \;\vdash\; t \,:\, \theta \,::\, \kappa' \qquad I \subseteq J}{\Delta \;\vdash\; \lambda\, x \,.\, t \,:\, \left(\bigwedge_{j \in J} \square_{m_j}\, \theta_j\right) \to \theta \,::\, \kappa \to \kappa'}$$

A type-system for verification (Grellois-Melliès 2014)

This type system can have infinite-depth derivations.

The parity condition over branches of run-trees may be reformulated as a condition over infinite branches of a derivation tree.

Theorem (G.-Melliès 2014, refomulated from Kobayashi-Ong 2009)

Consider an alternating parity tree automaton $\mathcal{A}$ and a scheme $\mathcal{G}$ producing a tree T.

Then $\mathcal{A}$ has a winning run-tree over T if and only if there exists a winning typing tree of

$$\Gamma \vdash t(\mathcal{G}) \,:\, q_0 :: \perp$$

where $t(\mathcal{G})$ is the λ-term corresponding to $\mathcal{G}$.

This reformulation comes from a game semantics perspective.

A type-system for verification (Grellois-Melliès 2014)

This type system can have infinite-depth derivations.

The parity condition over branches of run-trees may be reformulated as a condition over infinite branches of a derivation tree.

Theorem (G.-Melliès 2014, refomulated from Kobayashi-Ong 2009)

Consider an alternating parity tree automaton $\mathcal{A}$ and a scheme $\mathcal{G}$ producing a tree T.

Then $\mathcal{A}$ has a winning run-tree over T if and only if there exists a winning typing tree of

$$\Gamma \vdash t(\mathcal{G}) \, : \, q_0 \, :: \, \perp$$

where $t(\mathcal{G})$ is the λ-term corresponding to $\mathcal{G}$.

This reformulation comes from a game semantics perspective.

A type-system for verification (Grellois-Melliès 2014)

This type system can have infinite-depth derivations.

The parity condition over branches of run-trees may be reformulated as a condition over infinite branches of a derivation tree.

Theorem (G.-Melliès 2014, refomulated from Kobayashi-Ong 2009)

Consider an alternating parity tree automaton $\mathcal{A}$ and a scheme $\mathcal{G}$ producing a tree T.

Then $\mathcal{A}$ has a winning run-tree over T if and only if there exists a winning typing tree of

$$\Gamma \vdash t(\mathcal{G}) \; : \; q_0 \; :: \; \bot$$

where $t(\mathcal{G})$ is the λ-term corresponding to $\mathcal{G}$.

This reformulation comes from a game semantics perspective.

Parity conditions

As in the prologue, we can take advantage of this type-theoretic approach to design an associated model.

An investigation of the semantic nature of $\Box$ shows that it has good properties: it is a parameterized comonad, which distributes over the exponential $\lightning$.

Roughly speaking, the composite

$$\lightning = \lightning \Box$$

is an exponential, so that we automatically obtain a model of the λ-calculus associated to the coloured typings.

Parity conditions

As in the prologue, we can take advantage of this type-theoretic approach to design an associated model.

An investigation of the semantic nature of $\Box$ shows that it has good properties: it is a parameterized comonad, which distributes over the exponential $\lightning$.

Roughly speaking, the composite

$$\boldsymbol{\lightning} = \lightning \Box$$

is an exponential, so that we automatically obtain a model of the λ-calculus associated to the coloured typings.

Parity conditions

The modality is incorporated to the model by setting

$$[\![\Box A]\!] \quad = \quad Col \times [\![A]\!]$$

and there is a very natural coloured interpretation of types:

$$[\![A \Rightarrow B]\!] \ = \mathcal{M}_{count}(Col \times [\![A]\!]) \times [\![B]\!]$$

There is a correspondence between denotations and typings.

Parity conditions

The modality is incorporated to the model by setting

$$[\![\Box A]\!] \quad = \quad Col \times [\![A]\!]$$

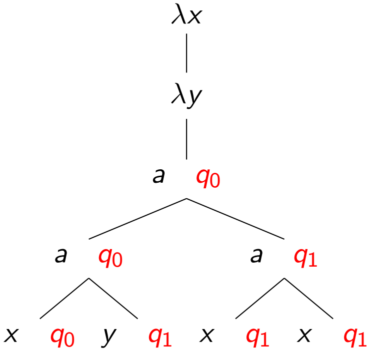
and there is a very natural coloured interpretation of types:

$$[\![A \Rightarrow B]\!] \ = \mathcal{M}_{count}(Col \times [\![A]\!]) \times [\![B]\!]$$

There is a correspondence between denotations and typings.

An example of coloured interpretation

Suppose $\Omega(q_0) = 0$ and $\Omega(q_1) = 1$.

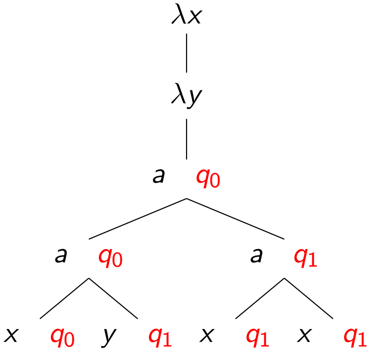


This rule will be interpreted in the model as

$$([(0, q_0), (1, q_1), (1, q_1)],\ [(1, q_1)],\ q_0)$$

An example of coloured interpretation

Suppose $\Omega(q_0) = 0$ and $\Omega(q_1) = 1$.



This rule will be interpreted in the model as

$$([(0, q_0), (1, q_1), (1, q_1)],\ [(1, q_1)],\ q_0)$$

Connection with the coloured relational model

To obtain the acceptance theorem for alternating parity automata, we need a fixpoint which reflects the parity condition.

This operator composes denotations infinitely, and only keeps the result if it comes from a winning composition tree.

Current work: define this fixpoint by combining the inductive and coinductive ones ?

Connection with the coloured relational model

To obtain the acceptance theorem for alternating parity automata, we need a fixpoint which reflects the parity condition.

This operator composes denotations infinitely, and only keeps the result if it comes from a winning composition tree.

Current work: define this fixpoint by combining the inductive and coinductive ones ?

Connection with the coloured relational model

Theorem (G.-Melliès 2015)

Consider an alternating parity tree automaton $\mathcal{A}$ and a higher-order recursion scheme $\mathcal{G}$ producing a tree T.

Then $\mathcal{A}$ has a winning run-tree over T if and only if

$$q_0 \in [\![\mathcal{G}]\!]$$

(where the fixed point is interpreted as previously sketched)

A finitary coloured model of the λY-calculus

Extensional collapses

In order to get a decidability proof and an estimation of complexity, we need to recast our work in a finitary setting.

If the exponential modality ! is interpreted with finite sets, we obtain the poset-based model of linear logic (a.k.a. its Scott model).

Ehrhard proved in 2012 that it is the extensional collapse of the relational model.

Extensional collapses

In order to get a decidability proof and an estimation of complexity, we need to recast our work in a finitary setting.

If the exponential modality ! is interpreted with finite sets, we obtain the poset-based model of linear logic (a.k.a. its Scott model).

Ehrhard proved in 2012 that it is the extensional collapse of the relational model.

Extensional collapses

Basically, the Scott model of linear logic is a qualitative model in which

$$[\![!\, A]\!] \quad = \quad \mathcal{P}_{fin}([\![A]\!])$$

But it requires to carry an ordering information.

It gives a model of the λ-calculus in which

1. Types are interpreted as preorders
2. Terms are interpreted as initial segments of the preorder: if $(X,\, a) \in [\![t]\!]$ then for every $Y \geq X$ and $b \leq a$ we have that $(Y,\, b) \in [\![t]\!]$.
   In other words, if a function can produce a out of X, it can also produce a worse output b out of a better input Y.

Extensional collapses

Basically, the Scott model of linear logic is a qualitative model in which

$$[\![!\, A]\!] \quad = \quad \mathcal{P}_{fin}([\![A]\!])$$

But it requires to carry an ordering information.

It gives a model of the λ-calculus in which

1. Types are interpreted as preorders
2. Terms are interpreted as initial segments of the preorder: if $(X,\, a) \in [\![t]\!]$ then for every $Y \geq X$ and $b \leq a$ we have that $(Y,\, b) \in [\![t]\!]$.
   In other words, if a function can produce a out of X, it can also produce a worse output b out of a better input Y.

Extensional collapses

Basically, the Scott model of linear logic is a qualitative model in which

$$[\![!\, A]\!] \quad = \quad \mathcal{P}_{fin}([\![A]\!])$$

But it requires to carry an ordering information.

It gives a model of the λ-calculus in which

1. Types are interpreted as preorders
2. Terms are interpreted as initial segments of the preorder: if $(X,\, a) \in [\![t]\!]$ then for every $Y \geq X$ and $b \leq a$ we have that $(Y,\, b) \in [\![t]\!]$.
   In other words, if a function can produce a out of X, it can also produce a worse output b out of a better input Y.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning A$ building multisets with finite-or-countable multiplicities.

Not needed in the finitary approach, we can use the finitary exponential !.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning A$ building multisets with finite-or-countable multiplicities.

Not needed in the finitary approach, we can use the finitary exponential !.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\oint A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\Box$, which propagates the colouring information of the APT in the denotations.

We can define it in the same way, we just need to take care of the saturation requirements.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\square$, which propagates the colouring information of the APT in the denotations.

We can define it in the same way, we just need to take care of the saturation requirements.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\Box$, which propagates the colouring information of the APT in the denotations,
3. There is a distributive law $\lambda : \lightning \Box \rightarrow \Box \lightning$, so that these two modalities can be composed to obtain a coloured exponential $\lightning$, giving by the Kleisli construction a coloured model of the λ-calculus.

Again, we define

$$\lambda : ! \Box \rightarrow \Box !$$

it in the same way, we just need to take care of the saturation requirements.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning\, A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\Box$, which propagates the colouring information of the APT in the denotations,
3. There is a distributive law $\lambda \,:\, \lightning\,\Box \to \Box\,\lightning$, so that these two modalities can be composed to obtain a coloured exponential $\boldsymbol{\lightning}$, giving by the Kleisli construction a coloured model of the λ-calculus.

Again, we define

$$\lambda \;: !\,\Box \to \Box\,!$$

it in the same way, we just need to take care of the saturation requirements.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\oint A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\Box$, which propagates the colouring information of the APT in the denotations,
3. There is a distributive law $\lambda : \oint \Box \to \Box \oint$, so that these two modalities can be composed to obtain a coloured exponential $\oint$, giving by the Kleisli construction a coloured model of the λ-calculus,
4. There is a coloured parameterized fixed point operator Y which extends this cartesian closed category to a model of the λY-calculus.

One more time: in the same way, we just need to take care of the saturation requirements.

Recipes to obtain a coloured model

1. The model is infinitary: there is an exponential $\lightning A$ building multisets with finite-or-countable multiplicities,
2. It features a parametric comonad $\Box$, which propagates the colouring information of the APT in the denotations,
3. There is a distributive law $\lambda : \lightning \Box \rightarrow \Box \lightning$, so that these two modalities can be composed to obtain a coloured exponential $\boldsymbol{\lightning}$, giving by the Kleisli construction a coloured model of the λ-calculus,
4. There is a coloured parameterized fixed point operator Y which extends this cartesian closed category to a model of the λY-calculus.

One more time: in the same way, we just need to take care of the saturation requirements.

Denotations, type-theoretically

Note that there is, again, a nice way to present the

computation of denotations

using the

typings of terms

in an associated type system.

In fact, the denotation of a closed term t is the set of elements $\alpha \in [\![type(t)]\!]$ such that

$$\emptyset \quad \vdash \quad t \, : \, \alpha \, :: \, type(t)$$

has a winning derivation tree in the associated type system.

Denotations, type-theoretically

Note that there is, again, a nice way to present the

computation of denotations

using the

typings of terms

in an associated type system.

In fact, the denotation of a closed term t is the set of elements $\alpha \in [\![type(t)]\!]$ such that

$$\emptyset \vdash t : \alpha :: type(t)$$

has a winning derivation tree in the associated type system.

Connection with higher-order model-checking

Theorem (G.-Melliès 2015)

Consider an alternating parity tree automaton $\mathcal{A}$ and a higher-order recursion scheme $\mathcal{G}$ producing a tree T.

Then $\mathcal{A}$ has a winning run-tree over T if and only if

$$q_0 \in [\![\mathcal{G}]\!]$$

where the interpretation is taken in this finitary, coloured model.

Decidability of higher-order model-checking

Note that the finiteness of the model implies, together with the memoryless decidability of parity games, that every element of the denotation of a term can be extracted from a finitary typing: a finite typing derivation with backtracking pointers, which unravels to the original one.

This implies:

Theorem

The higher-order model-checking problem is decidable.

Decidability of higher-order model-checking

The order of a scheme/of a term can be understood as a measure of its complexity.

It somehow characterizes the size of its set of refined intersection types/of its finitary denotation.

Proposition

If $\mathcal{G}$ has order n, then the complexity of the problem is $O(n\text{-}EXPTIME)$.

Decidability of selection

Given an APT $\mathcal{A}$ and a recursion scheme $\mathcal{G}$, the selection problem is to compute $\mathcal{G}'$ whose value tree is a winning run-tree of $\mathcal{A}$ over $[\![\mathcal{G}]\!]$.

From a finitary typing, we can build such a scheme, leading to

Theorem

The APT selection problem is decidable.

Conclusions and perspectives

- We studied linear models of the λY-calculus, designed to reflect the behaviour of alternating parity tree automata.
- In spite of its infinitary nature, the relational model is convenient for the theoretical study of the problem.
- From the recipes of the relational approach, we define a finitary model, which gives a new decidability proof of the problem.
- There is still a lot to do: study the coloured extensional collapse, axiomatize this extension of "recognition by monoid", define properly game semantics with parity, extend our approach to other models of automata ...

Conclusions and perspectives

- We studied linear models of the λY-calculus, designed to reflect the behaviour of alternating parity tree automata.
- In spite of its infinitary nature, the relational model is convenient for the theoretical study of the problem.
- From the recipes of the relational approach, we define a finitary model, which gives a new decidability proof of the problem.
- There is still a lot to do: study the coloured extensional collapse, axiomatize this extension of "recognition by monoid", define properly game semantics with parity, extend our approach to other models of automata ...

Conclusions and perspectives

- We studied linear models of the λY-calculus, designed to reflect the behaviour of alternating parity tree automata.
- In spite of its infinitary nature, the relational model is convenient for the theoretical study of the problem.
- From the recipes of the relational approach, we define a finitary model, which gives a new decidability proof of the problem.
- There is still a lot to do: study the coloured extensional collapse, axiomatize this extension of "recognition by monoid", define properly game semantics with parity, extend our approach to other models of automata ...

Conclusions and perspectives

- We studied linear models of the λY-calculus, designed to reflect the behaviour of alternating parity tree automata.
- In spite of its infinitary nature, the relational model is convenient for the theoretical study of the problem.
- From the recipes of the relational approach, we define a finitary model, which gives a new decidability proof of the problem.
- There is still a lot to do: study the coloured extensional collapse, axiomatize this extension of "recognition by monoid", define properly game semantics with parity, extend our approach to other models of automata ...