Verification by typing

Charles Grellois
joint work with Paul-André Melliès

PPS & LIAFA

25 juin 2014
Model-checking

Usual approach in verification: model-checking (Clarke, Emerson). Interaction of a program and a property.

How do we model them?

Many possible answers depending on the kind of program and property. A general approach would be undecidable...

Need to find a balance between expressivity and complexity.
Model-checking

Usual approach in verification: model-checking (Clarke, Emerson). Interaction of a program and a property.

How do we model them?

Many possible answers depending on the kind of program and property. A general approach would be undecidable...

Need to find a balance between expressivity and complexity.
Model-checking

Usual approach in verification: model-checking (Clarke, Emerson). Interaction of a program and a property.

How do we model them?

Many possible answers depending on the kind of program and property. A general approach would be undecidable...

Need to find a balance between expressivity and complexity.
In this work we are concerned with higher-order programs: a function may take a function as input.

Example: \( \text{compose } \phi \ x = \phi(\phi(x)) \)

A model for such programs is higher-order recursion schemes (HORS).
Model-checking

In this work we are concerned with higher-order programs: a function may take a function as input.

Example: \( \text{compose } \phi \ x \ = \ \phi(\phi(x)) \)

A model for such programs is higher-order recursion schemes (HORS).
Model-checking

In this work we are concerned with higher-order programs: a function may take a function as input.

Example: \( \text{compose } \phi \ x = \phi(\phi(x)) \)

A model for such programs is higher-order recursion schemes (HORS).
Higher-order recursion schemes

Idea:

- LIAFA-style: it is a kind of grammar whose parameters may be functions and which generates trees.
- PPS-style: it is a formalism equivalent to $\lambda Y$ calculus with uninterpreted constants from a ranked alphabet.

(remember we are supposed to merge ;-) )
Higher-order recursion schemes

Idea:

- **LIAFA-style**: it is a kind of grammar whose parameters may be functions and which generates trees.
- **PPS-style**: it is a formalism equivalent to $\lambda Y$ calculus with uninterpreted constants from a ranked alphabet.

(remember we are supposed to merge ;-) )
A silly functional program

\[
\begin{align*}
\text{Main} & \quad = \quad \text{Listen Nil} \\
\text{Listen } x & \quad = \quad \text{if } \text{end} \ \text{then } x \ \text{else } \text{Listen (data } x)\end{align*}
\]

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 if. We shall see that the problem is already pretty complex without this kind of additional reduction rules...
A silly functional program

\[
\begin{align*}
\text{Main} & \quad = \quad \text{Listen Nil} \\
\text{Listen } x & \quad = \quad \text{if } end \text{ then } x \text{ else Listen (data } x)\end{align*}
\]

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 if. We shall see that the problem is already pretty complex without this kind of additional reduction rules...
A silly functional program

\[
\text{Main} \quad = \quad \text{Listen Nil}
\]
\[
\text{Listen } x \quad = \quad \text{if } \text{end} \text{ then } x \text{ else Listen (data } x\text{)}
\]

formulated as a recursion scheme:

\[
S \quad = \quad L \text{ Nil}
\]
\[
L \ x \quad = \quad \text{if } x \ (L \ (\text{data } x))
\]

or, in \(\lambda\)-calculus style :

\[
S \quad = \quad L \text{ Nil}
\]
\[
L \quad = \quad \lambda x. \text{if } x \ (L \ (\text{data } x))
\]
A silly functional program

Main = Listen Nil
Listen x = if end then x else Listen (data x)

formulated as a recursion scheme:

S = L Nil
L x = if x (L (data x))

or, in \(\lambda\)-calculus style:

S = L Nil
L = \(\lambda x.\) if x (L (data x))
A silly functional program

\[
\begin{align*}
\text{Main} & = \text{Listen Nil} \\
\text{Listen } x & = \text{if } end \text{ then } x \text{ else Listen (data } x) \\
\end{align*}
\]

formulated as a recursion scheme:

\[
\begin{align*}
S & = L \text{ Nil} \\
L \ x & = \text{if } x (L \text{ (data } x) \\
\end{align*}
\]
or, in λ-calculus style:

\[
\begin{align*}
S & = L \text{ Nil} \\
L & = \lambda x. \text{if } x (L \text{ (data } x) \\
\end{align*}
\]
Value tree of a recursion scheme

\[
\begin{align*}
S &= L \text{ Nil} \\
L \ x &= \text{if } x (L \ (\text{data } x)) \quad \text{generates:}
\end{align*}
\]
Value tree of a recursion scheme

\[
\begin{align*}
S &= L \text{ Nil} \\
L \times &= \text{if } x (L (\text{data } x)) \\
S &\rightarrow L \\
&\quad | \\
&\quad \text{Nil}
\end{align*}
\]
Value tree of a recursion scheme

\[ S = L \text{ Nil} \]
\[ L \ x = \text{if} \ x \ (L \ (\text{data} \ x)) \]

generates:

```
if
  L
    Nil
  data
    Nil
    L
```

Charles Grellois (PPS & LIAFA)
Value tree of a recursion scheme

\[ S = L \text{ Nil} \]
\[ L \: x = \text{if } x \: (L \: (\text{data } x)) \]

generates:

\[ \text{if} \]
\[ \text{Nil} \quad L \]
\[ \text{data} \]
\[ \text{Nil} \]

\[ \Rightarrow \]

\[ \text{if} \]
\[ \text{Nil} \quad \text{if} \]
\[ \text{data} \quad L \]
\[ \text{Nil} \quad \text{data} \]
\[ \text{data} \]
\[ \text{Nil} \]
Value tree of a recursion scheme

```
if
  Nil
  if
    data
    if
      Nil
      data
      : 
        data
        Nil

Silly program, but not regular tree…
```
Value tree of a recursion scheme

Silly program, but not regular tree...
Another recursion scheme

The previous recursion scheme was of order 1.

Indeed, the non-terminals are typed according to the ranked alphabet of constants.

We had $S : o$ and $L : o \rightarrow o$, of order 0 and 1 respectively. Their maximum is the order of the scheme.

We may understand it as a measure of the complexity of the rewriting process.
Another recursion scheme

The previous recursion scheme was of order 1.

Indeed, the non-terminals are typed according to the ranked alphabet of constants.

We had $S : o$ and $L : o \rightarrow o$, of order 0 and 1 respectively. Their maximum is the order of the scheme.

We may understand it as a measure of the complexity of the rewriting process.
Another recursion scheme

An order-2 example (from Serre et al.):

\[
\begin{align*}
S &= M \text{Nil} \\
M \times &= \text{if } (\text{commit } x) (A \times M) \\
A y \phi &= \text{if } (\phi (\text{error end})) (\phi (\text{cons } y))
\end{align*}
\]
Value tree of a recursion scheme

\[
\begin{align*}
S &= M \text{ Nil} \\
M \times &= \text{if}(\text{commit } x)(A \times M) \\
A y \phi &= \text{if}(\phi(\text{error end}))(\phi(\text{cons } y))
\end{align*}
\]

\[
\begin{array}{c}
S \\
\Rightarrow \\
M \\
\text{Nil}
\end{array}
\]
Value tree of a recursion scheme

\[
S = M \text{ Nil}
\]

\[
M \times = \text{if (commit } x \text{) ( } A \times M \text{)}
\]

\[
A y \phi = \text{if ( } \phi (\text{error end}) \text{) ( } \phi (\text{cons } y)\text{)}
\]
Value tree of a recursion scheme

\[
S = M \text{Nil}
\]
\[
M \times = \text{if } (\text{commit } x) (A \times M)
\]
\[
A \ y \phi = \text{if } (\phi (\text{error end}))(\phi (\text{cons } y))
\]
Value tree of a recursion scheme

\[ S = M \text{Nil} \]
\[ M \times = \text{if} \left( \text{commit x} \right) \left( A \times M \right) \]
\[ A y \phi = \text{if} \left( \phi \left( \text{error end} \right) \right) \left( \phi \left( \text{cons y} \right) \right) \]
Value tree of a recursion scheme

We would like to check that the program modelled by this scheme never commits an error.
Modal $\mu$-calculus

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

In this work we use modal $\mu$-calculus. It is equivalent to MSO over trees.

Grammar: $\phi, \psi ::= X \mid a \mid \phi \lor \psi \mid \phi \land \psi \mid \Box \phi \mid \Diamond i \phi \mid \mu X. \phi \mid \nu X. \phi$
Modal $\mu$-calculus

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

In this work we use modal $\mu$-calculus. It is equivalent to MSO over trees.

Grammar: $\phi, \psi ::= X \mid a \mid \phi \lor \psi \mid \phi \land \psi \mid \Box \phi \mid \Diamond_i \phi \mid \mu X. \phi \mid \nu X. \phi$
Modal $\mu$-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \lor \psi \mid \phi \land \psi \mid \Box \phi \mid \Diamond_i \phi \mid \mu X. \phi \mid \nu X. \phi$

$X$ is a variable

$a$ is a predicate corresponding to a symbol of $\Sigma$

$\Box \phi$ means that $\phi$ should hold on every successor of the current node

$\Diamond_i \phi$ means that $\phi$ should hold on one successor of the current node (in direction $i$)

We can also define (variant) $\Diamond = \bigvee_i \Diamond_i$. 
Modal $\mu$-calculus

Grammar: $\phi, \psi ::= X | a | \phi \lor \psi | \phi \land \psi | \Box \phi | \Diamond_i \phi | \mu X. \phi | \nu X. \phi$

$X$ is a variable

$a$ is a predicate corresponding to a symbol of $\Sigma$

$\Box \phi$ means that $\phi$ should hold on every successor of the current node

$\Diamond_i \phi$ means that $\phi$ should hold on one successor of the current node (in direction $i$)

We can also define (variant) $\Diamond = \bigvee_i \Diamond_i$. 
Modal $\mu$-calculus

Grammar:  
$\phi, \psi ::= X | a | \phi \lor \psi | \phi \land \psi | \Box \phi | \Diamond_i \phi | \mu X. \phi | \nu X. \phi$

$X$ is a variable

$a$ is a predicate corresponding to a symbol of $\Sigma$

$\Box \phi$ means that $\phi$ should hold on every successor of the current node

$\Diamond_i \phi$ means that $\phi$ should hold on one successor of the current node (in direction $i$)

We can also define (variant) $\Diamond = \bigvee_i \Diamond_i$. 
Modal $\mu$-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \lor \psi \mid \phi \land \psi \mid \Box \phi \mid \Diamond_i \phi \mid \mu X. \phi \mid \nu X. \phi$

$\mu X. \phi$ is the least fixpoint of $\phi(X)$. It is computed by expanding finitely the formula:

$$
\mu X. \phi(X) \quad \rightarrow \quad \phi(\mu X. \phi(X)) \quad \rightarrow \quad \phi(\phi(\mu X. \phi(X)))
$$
Modal $\mu$-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \lor \psi \mid \phi \land \psi \mid \Box \phi \mid \lozenge_i \phi \mid \mu X. \phi \mid \nu X. \phi$

$\nu X. \phi$ is the greatest fixpoint of $\phi(X)$. It is computed by expanding infinitely the formula:

$$\nu X. \phi(X) \rightarrow \phi(\nu X. \phi(X)) \rightarrow \phi(\phi(\nu X. \phi(X)))$$
Specifying a property in modal $\mu$-calculus

How do we specify that the second scheme does not commit an error? We want to forbid the existence of an instance of the symbol $error$ on a branch after $commit$ was seen.

There is an error on a branch $\iff \mu X. (\Diamond X \lor error)$

There is an error on a branch after a commit $\iff commit \land (\mu X. (\Diamond X \lor error))$
Specifying a property in modal $\mu$-calculus

How do we specify that the second scheme does not commit an error? We want to forbid the existence of an instance of the symbol `error` on a branch after `commit` was seen.

There is an error on a branch $\iff \mu X. (\Diamond X \vee \text{error} )$

There is an error on a branch after a commit $\iff \text{commit} \land (\mu X. (\Diamond X \vee \text{error} ) )$
Specifying a property in modal $\mu$-calculus

How do we specify that the second scheme does not commit an error? We want to forbid the existence of an instance of the symbol $error$ on a branch after $commit$ was seen.

There is an error on a branch $\iff \mu X. (\diamond X \lor error)$

There is an error on a branch after a commit $\iff commit \land (\mu X. (\diamond X \lor error))$
Specifying a property in modal $\mu$-calculus

There is an error on a branch after a commit
\[ \iff \text{commit} \land ( \mu X. ( \Diamond X \vee \text{error} ) ) \]

There is a branch with an error after a commit
\[ \iff \mu Y. ( \Diamond Y \lor ( \text{commit} \land ( \mu X. ( \Diamond X \lor \text{error} ) ) ) ) \]
Specifying a property in modal $\mu$-calculus

Over the first example:

- Every branch ends by $\text{Nil}$:
  \[
  \mu X. ( \text{Nil} \lor \Box X )
  \]
  but is it true? Take instead
  \[
  \nu X. ( \text{Nil} \lor \Box X )
  \]

- What does
  \[
  \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. ( \text{Nil} \lor \Box Y ) ) \land \Diamond_2 X )
  \]
  mean?
Specifying a property in modal $\mu$-calculus

Over the first example:

- Every branch ends by $\text{Nil}$:

$$\mu X. (\text{Nil} \lor \square X)$$

but is it true? Take instead

$$\nu X. (\text{Nil} \lor \square X)$$

- What does

$$\nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil} \lor \square Y)) \land \Diamond_2 X)$$

mean?
Specifying a property in modal $\mu$-calculus

Over the first example:

- Every branch ends by Nil:

$$\mu X. (\text{Nil} \lor \Box X)$$

but is it true? Take instead

$$\nu X. (\text{Nil} \lor \Box X)$$

- What does

$$\nu X. (\text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X)$$

mean?
Specifying a property in modal $\mu$-calculus

Over the first example:

- Every branch ends by $\text{Nil}$:

$$\mu X. (\text{Nil} \lor \Box X)$$

but is it true? Take instead

$$\nu X. (\text{Nil} \lor \Box X)$$

- What does

$$\nu X. (\text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X)$$

mean?
Over the first example:

What does

$$\nu X. ( \text{true} \land \Diamond_1 ( \mu Y. (\text{Nil} \lor \Box Y )) \land \Diamond_2 X )$$

mean?

- There is an infinite branch, the rightmost one, only labelled with \text{if}.
- Every other branch is finite and ends with a \text{Nil}.
Interaction with trees: a shift to automata theory

Logic is great!

...but how does it interact with a tree?

An usual approach, notably over words, is to find an equi-expressive automaton model.
Interaction with trees: a shift to automata theory

Logic is great!

...but how does it interact with a tree?

An usual approach, notably over words, is to find an equi-expressive automaton model.
Interaction with trees: a shift to automata theory

Logic is great!

...but how does it interact with a tree?

An usual approach, notably over words, is to find an equi-expressive automaton model.
Alternating parity tree automata

Idea: the formula "starts" on the root

\[ \phi = \nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil } \lor \Box Y )) \land \Diamond_2 X ) \]

where \( \phi \) = \nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil } \lor \Box Y )) \land \Diamond_2 X )
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{if } \text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 \phi
\]

where \( \phi = \nu X. (\text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{if } \Diamond_1 ( \mu Y. (\text{Nil } \lor \Box Y) ) \land \Diamond_2 \phi
\]

where \( \phi = \nu X. (\text{if } \land \Diamond_1 ( \mu Y. (\text{Nil } \lor \Box Y) ) \land \Diamond_2 X ) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{if } \Diamond_2 \phi
\]

\[
\text{Nil } \mu Y. (\text{Nil } \lor \Box Y)
\]

\[
\text{data if}
\]

\[
\text{Nil data :}
\]

\[
\text{data}
\]

\[
\text{Nil}
\]

where \( \phi = \nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil } \lor \Box Y)) \land \Diamond_2 X) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{if } \begin{cases} \text{Nil} & \mu Y. (\text{Nil} \lor \Box Y) \land \Diamond_2 X \\ \text{data} & \text{if } \begin{cases} \text{Nil} & \nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X) \\ \text{data} \end{cases} \\ \text{Nil} \end{cases}
\]

where \( \phi = \nu X. (\text{if } \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{where } \phi = \nu X. (\text{if } \land \Box_1 (\mu Y. (\text{Nil } \lor \Box Y)) \land \Box_2 X)
\]
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\begin{align*}
\text{if} & \quad \text{Nil} \quad \text{Nil} \quad \text{if} \quad \phi \\
\text{data} & \quad \text{if} \\
\text{Nil} & \quad \text{data} : \\
\text{data} & \\
\text{Nil} &
\end{align*}
\]

where \( \phi = \nu X. ( \text{if} \land \diamond_1 ( \mu Y. ( \text{Nil} \lor \Box Y )) \land \diamond_2 X ) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

where $\phi = \nu X. (\text{if } \land \diamond_1 (\mu Y. (\text{Nil } \lor \Box Y)) \land \diamond_2 X)$
Alternating parity tree automata

Idea: the formula "starts" on the root

\[
\text{if } \begin{array}{c}
\text{Nil} \\
\text{Nil}
\end{array} \\
\text{if } \begin{array}{c}
\text{data } \text{Nil} \\
\text{Nil}
\end{array} \\
\text{data } \text{Nil} \\
\text{Nil}
\end{array} \\
\text{if } \begin{array}{c}
\text{data } \text{Nil} \\
\text{Nil}
\end{array} \\
\text{data } \text{Nil}
\]

\[\phi = \nu X. (\text{if } \land \land_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \land_2 X)\]

where \(\phi\) = \(\nu X. (\text{if } \wedge \land_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \land_2 X)\)
Alternating parity tree automata

Idea: the formula "starts" on the root

where \( \phi = \nu X. (\text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X) \)
Alternating parity tree automata

Idea: the formula "starts" on the root

where $\phi = \nu X. (\text{if} \land \diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \diamond_2 X)$
Alternating parity tree automata

Idea: the formula "starts" on the root

$\phi = \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X )$

where $\phi = \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X )$
Alternating parity tree automata

Idea: the formula "starts" on the root

where $\phi = \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. (\text{Nil} \lor \Box Y )) ) \land \Diamond_2 X$
Alternating parity tree automata

Conversion to an automaton?

- Needs to play the formula over the tree, but always by reading a letter.
- Idea: iterate the formula several times until you find a letter.
- Needs non-determinism for $\lor$ and alternation for $\land$
- Needs a parity condition for distinguishing $\mu$ and $\nu$
Alternating parity tree automata

Conversion to an automaton?

- Needs to play the formula over the tree, but always by reading a letter.
- Idea: iterate the formula several times until you find a letter.
- Needs non-determinism for $\lor$ and alternation for $\land$.
- Needs a parity condition for distinguishing $\mu$ and $\nu$. 
Alternating parity tree automata

Conversion to an automaton?

- Needs to play the formula over the tree, but always by reading a letter.
- Idea: iterate the formula several times until you find a letter.
- Needs non-determinism for $\lor$ and alternation for $\land$
- Needs a parity condition for distinguishing $\mu$ and $\nu$
Alternating parity tree automata

\[ \phi = \nu X. (\text{if} \land \Diamond_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \Diamond_2 X) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. (\text{Nil} \lor \Box Y) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

\[ \phi = \nu X. \left( \text{if} \land \diamondsuit_1 (\mu Y. (\text{Nil} \lor \Box Y)) \land \diamondsuit_2 X \right) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. (\text{Nil} \lor \Box Y) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

\[ \phi = \nu X.(\text{if} \land \diamond 1(\mu Y. (\text{Nil} \lor \Box Y)) \land \diamond 2 X) \]

To translate \(\phi\) to an automaton, consider its set of states \(Q\) as the set of subformulas of \(\phi\). Its initial state \(q_0\) corresponds to \(\phi\), and \(q_1\) to \(\mu Y. (\text{Nil} \lor \Box Y)\).

Then:

- \(\delta(q_0, \text{Nil}) = \bot\)
- \(\delta(q_0, \text{data}) = \bot\)
- \(\delta(q_0, \text{if}) = (1, q_1) \land (2, q_0)\)
- \(\delta(q_1, \text{Nil}) = \top\)
- \(\delta(q_1, \text{data}) = (1, q_1)\)
- \(\delta(q_1, \text{if}) = (1, q_1) \land (2, q_1)\)

Inductive/coinductive behaviour limitations: you can only play \(q_1\) finitely, but there are no restrictions over \(q_0\).
Alternating parity tree automata

\[ \phi = \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. ( \text{Nil} \lor \Box Y )) \land \Diamond_2 X ) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. ( \text{Nil} \lor \Box Y ) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

\[ \phi = \nu X. (\text{if} \land \diamond_1 (\mu Y. (\text{Nil} \lor \square Y)) \land \diamond_2 X) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. (\text{Nil} \lor \square Y) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

\[ \phi = \nu X. ( \text{if} \land \Diamond_1 ( \mu Y. ( \text{Nil} \lor \Box Y )) \land \Diamond_2 X ) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. ( \text{Nil} \lor \Box Y ) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

\[ \phi = \nu X. ( \text{if} \land \diamond_1 ( \mu Y. ( \text{Nil} \lor \square Y ) ) \land \diamond_2 X ) \]

To translate \( \phi \) to an automaton, consider its set of states \( Q \) as the set of subformulas of \( \phi \). Its initial state \( q_0 \) corresponds to \( \phi \), and \( q_1 \) to \( \mu Y. ( \text{Nil} \lor \square Y ) \).

Then:

- \( \delta(q_0, \text{Nil}) = \bot \)
- \( \delta(q_0, \text{data}) = \bot \)
- \( \delta(q_0, \text{if}) = (1, q_1) \land (2, q_0) \)
- \( \delta(q_1, \text{Nil}) = \top \)
- \( \delta(q_1, \text{data}) = (1, q_1) \)
- \( \delta(q_1, \text{if}) = (1, q_1) \land (2, q_1) \)

Inductive/coinductive behaviour limitations: you can only play \( q_1 \) finitely, but there are no restrictions over \( q_0 \).
Alternating parity tree automata

In general, transitions may duplicate or drop a subtree.

Example: \( \delta(q_0, \text{if}) = (2, q_0) \land (2, q_1) \).
Alternating parity tree automata

\[ \delta(q_0, \text{if}) = (2, q_0) \wedge (2, q_1). \]
Alternating parity tree automata

$$\delta(q_0, \text{if}) = (2, q_0) \land (2, q_1).$$

and so on. This gives the notion of run-tree.
Alternating parity tree automata

\[ \delta(q_0, \text{if}) = (2, q_0) \land (2, q_1). \]

and so on. This gives the notion of run-tree.
Alternating parity tree automata

And for the inductive/coinductive behaviour?

→ parity conditions

Over a branch of a run-tree, say $q_0$ has colour 0 and $q_1$ has colour 1.

Now consider an infinite branch, and the maximal colour you see infinitely often on this branch.

If it is even, accept: it means you looped infinitely on $\nu$.

Else if it is odd the automaton rejects: it means $\mu$ was unfolded infinitely, and this is forbidden.
Alternating parity tree automata

And for the inductive/coinductive behaviour?

→ parity conditions

Over a branch of a run-tree, say $q_0$ has colour 0 and $q_1$ has colour 1.

Now consider an infinite branch, and the maximal colour you see infinitely often on this branch.

If it is even, accept: it means you looped infinitely on $\nu$.

Else if it is odd the automaton rejects: it means $\mu$ was unfolded infinitely, and this is forbidden.
Alternating parity tree automata

And for the inductive/coinductive behaviour?

→ parity conditions

Over a branch of a run-tree, say $q_0$ has colour 0 and $q_1$ has colour 1.

Now consider an infinite branch, and the maximal colour you see infinitely often on this branch.

If it is even, accept: it means you looped infinitely on $\nu$.

Else if it is odd the automaton rejects: it means $\mu$ was unfolded infinitely, and this is forbidden.
Parity condition on an example

would not be a winning run-tree: the automaton unfolded $\mu$ infinitely on the infinite branch (note: $\delta$ needs to be modified a little to produce this run-tree).
Alternating parity tree automata

In general, every state is given a colour, and a run-tree is accepting if and only if all its branches have an even maximal infinitely seen colour.

A tree is accepted iff it admits a winning run-tree. This is equivalent to satisfying the modal $\mu$-calculus property encoded by the automaton.
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

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

and what about colours ?

Consider $(\Box c_0 \ q_0 \wedge \Box c_1 \ q_1) \Rightarrow \Box c_2 \ q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)
Alternating parity tree automata and intersection types

A key remark (Kobayashi 2009): if $\delta(q, a) = (1, q_0) \land (1, q_1) \land (2, q_2) \ldots$
then we may consider that $a$ has a refined intersection type

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

and what about colours?

Consider $\Box_{c_0} q_0 \land \Box_{c_1} q_1 \Rightarrow \Box_{c_2} q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)
Alternating parity tree automata and intersection types

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

then we may consider that $a$ has a refined intersection type

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

and what about colours ?

Consider $(\square_{c_0} q_0 \land \square_{c_1} q_1) \Rightarrow \square_{c_2} q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)
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 silly program models can be not regular. But schemes always are finite — and most of the time rather small.
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 silly program models can be not regular. But schemes always are finite — and most of the time rather small.
A type-system for verification: without colours

Axiom

\[ x : \land_{\{i\}} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \delta \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{satisfies} \; \delta_A(q, a) \]

\[ \emptyset \vdash a : \land_{j=1}^{k_1} q_{1j} \to \ldots \to \land_{j=1}^{k_n} q_{nj} \to q :: \bot \to \cdots \to \bot \]

\[ \Delta \vdash t : (\theta_1 \land \cdots \land \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \Delta_1 + \ldots + \Delta_k \vdash t u : \theta :: \kappa' \]

\[ \text{fix} \quad 
\Gamma \vdash R(F) : \theta :: \kappa \\
F : \theta :: \kappa \vdash F : \theta :: \kappa' \]

\[ \Delta, x : \land_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]

\[ \Delta \vdash \lambda x . t : (\land_{j \in J} \theta_j) \to \theta :: \kappa \to \kappa' \]
A type-system for verification: without colours

Axiom

\[ x : \bigwedge_{i} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \delta \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} q_{1j} \to \ldots \to \bigwedge_{j=1}^{k_n} q_{nj} \to q :: \bot \to \cdots \to \bot \]

App

\[ \Delta \vdash t : (\theta_1 \land \cdots \land \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \Delta_1 + \ldots + \Delta_k \vdash t u : \theta :: \kappa' \]

fix

\[ \Gamma \vdash R(F) : \theta :: \kappa \]

\[ F : \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \lambda \]

\[ \Delta, x : \bigwedge_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]

\[ \Delta \vdash \lambda x . t : \left( \bigwedge_{j \in J} \theta_j \right) \to \theta :: \kappa \to \kappa' \]
A type-system for verification: without colours

Axiom

\[ x : \bigwedge_{\{i\}} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\( \delta \)

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} q_{1j} \to \cdots \to \bigwedge_{j=1}^{k_n} q_{nj} \to q :: \bot \to \cdots \to \bot \]

App

\[ \Delta \vdash t : (\theta_1 \land \cdots \land \theta_k) \to \theta :: \kappa \to \kappa' \]

\[ \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \Delta_1 + \cdots + \Delta_k \vdash t \ u : \theta :: \kappa' \]

fix

\[ \Gamma \vdash R(F) : \theta :: \kappa \]

\[ F : \theta :: \kappa \vdash F : \theta :: \kappa \]

\( \lambda \)

\[ \Delta, x : \bigwedge_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa' \]

\[ I \subseteq J \]

\[ \Delta \vdash \lambda x \cdot t : \left( \bigwedge_{j \in J} \theta_j \right) \to \theta :: \kappa \to \kappa' \]
A type-system for verification: without colours

Axiom
\[ x : \bigwedge_{\{i\}} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\( \delta \)
\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} q_{1j} \to \cdots \to \bigwedge_{j=1}^{k_n} q_{nj} \to q :: \bot \to \cdots \to \bot \]

App
\[ \Delta \vdash t : (\theta_1 \land \cdots \land \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]
\[ \Delta + \Delta_1 + \cdots + \Delta_k \vdash t u : \theta :: \kappa' \]

fix
\[ \Gamma \vdash \mathcal{R}(F) : \theta :: \kappa \]
\[ F : \theta :: \kappa \vdash F : \theta :: \kappa \]

\( \lambda \)
\[ \Delta, x : \bigwedge_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]
\[ \Delta \vdash \lambda x. t : \big(\bigwedge_{j \in J} \theta_j\big) \to \theta :: \kappa \to \kappa' \]
A type-system for verification: without colours

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

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

App
\[
\frac{\Delta \vdash t : (\theta_1 \land \cdots \land \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa}{\Delta + \Delta_1 + \cdots + \Delta_k \vdash t \ u : \theta :: \kappa'}
\]

fix
\[
\frac{\Gamma \vdash R(F) : \theta :: \kappa}{\Gamma \vdash \lambda x. t : \bigwedge_{i \in I} \theta_i :: \kappa \vdash F : \theta :: \kappa}
\]

\[\lambda\]
\[
\frac{\Delta, x : \bigwedge_{i \in I} \theta_i :: \kappa \vdash t : \theta :: \kappa'}{\Delta \vdash \lambda x. t : \bigwedge_{j \in J} \theta_j \rightarrow \theta :: \kappa \rightarrow \kappa'}
\]
A type-system for verification: example

\[ S = L \text{ Nil} \]
\[ L = \lambda x. \text{if } x (L \text{ (data } x \text{ )}) \]

and transitions:

\[ \delta(q_0, \text{Nil}) = \top \iff \text{Nil : } q_0 \]
\[ \delta(q_0, \text{data}) = (1, q_0) \iff \text{data : } q_0 \rightarrow q_0 \]
\[ \delta(q_0, \text{if}) = ((1, q_0) \land (2, q_0)) \lor (2, q_1) \iff \text{if : } (q_0 \rightarrow q_0 \rightarrow q_0) \land (\emptyset \rightarrow q_1 \rightarrow q_0) \]
\[ \delta(q_1, \text{if}) = (2, q_1) \iff \text{if : } \emptyset \rightarrow q_1 \rightarrow q_1 \]

( example on the board — mistakes to check attention only ;-) )
A type-system for verification (Grellois-Mellliès 2014)

Axiom

\[ x : \bigwedge_{\{i\}} \square_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} \square_{m_{1j}} q_{1j} \to \cdots \to \bigwedge_{j=1}^{k_n} \square_{m_{nj}} q_{nj} \to q :: \bot \to \cdots \to \bot \to \bot \]

App

\[ \Delta \vdash t : (\square_{m_1} \theta_1 \land \cdots \land \square_{m_k} \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \square_{m_1} \Delta_1 + \cdots + \square_{m_k} \Delta_k \vdash t u : \theta :: \kappa' \]

fix

\[ \Gamma \vdash \mathcal{R}(F) : \theta :: \kappa \]

\[ F : \square_{\Omega(\theta)} \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \Delta, x : \bigwedge_{i \in I} \square_{m_i} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]

\[ \Delta \vdash \lambda x . t : \left( \bigwedge_{j \in J} \square_{m_j} \theta_j \right) \to \theta :: \kappa \to \kappa' \]
A type-system for verification (Grellois-Melliès 2014)

Axiom

\[ x : \bigwedge_{\{i\}} \Box_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} \Box_{m_{1j}} q_{1j} \to \ldots \to \bigwedge_{j=1}^{k_n} \Box_{m_{nj}} q_{nj} \to q :: \bot \to \cdots \to \bot \to \bot \]

App

\[ \Delta \vdash t : (\Box_{m_1} \theta_1 \land \cdots \land \Box_{m_k} \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \Box_{m_1} \Delta_1 + \ldots + \Box_{m_k} \Delta_k \vdash t u : \theta :: \kappa' \]

fix

\[ \Gamma \vdash \mathcal{R}(F) : \theta :: \kappa \]

\[ F : \Box_{\Omega(\theta)} \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \Delta, x : \bigwedge_{i \in I} \Box_{m_i} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]

\[ \Delta \vdash \lambda x \cdot t : \left(\bigwedge_{j \in J} \Box_{m_j} \theta_j\right) \to \theta :: \kappa \to \kappa' \]
A type-system for verification (Grellois-Melliès 2014)

Axiom

\[ x : \bigwedge \{ i \} \square_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} \square_{m_{1j}} q_{1j} \rightarrow \cdots \rightarrow \bigwedge_{j=1}^{k_n} \square_{m_{nj}} q_{nj} \rightarrow q :: \bot \rightarrow \cdots \rightarrow \bot \rightarrow \bot \]

App

\[ \Delta \vdash t : (\bigwedge_{m_1} \theta_1 \wedge \cdots \wedge \bigwedge_{m_k} \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \square_{m_1} \Delta_1 + \cdots + \square_{m_k} \Delta_k \vdash t \ u : \theta :: \kappa' \]

fix

\[ \Gamma \vdash R(F) : \theta :: \kappa \]

\[ F : \square_{\Omega(\theta)} \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \Delta, x : \bigwedge_{i \in I} \square_{m_i} \theta_i :: \kappa \vdash t : \theta :: \kappa' \]

\[ I \subseteq J \]

\[ \Delta \vdash \lambda x . t : \left( \bigwedge_{j \in J} \square_{m_j} \theta_j \right) \rightarrow \theta :: \kappa \rightarrow \kappa' \]
A type-system for verification (Grellois-Melliès 2014)

\[ x : \bigwedge\{i\} \square_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} \square_{m_{1j}} q_{1j} \to \cdots \to \bigwedge_{j=1}^{k_n} \square_{m_{nj}} q_{nj} \to q :: \bot \to \cdots \to \bot \to \bot \]

\[ \Delta \vdash t : (\bigwedge_{m_1} \theta_1 \land \cdots \land \bigwedge_{m_k} \theta_k) \to \theta :: \kappa \to \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]

\[ \Delta + \square_{m_1} \Delta_1 + \cdots + \square_{m_k} \Delta_k \vdash t u : \theta :: \kappa' \]

\[ \Gamma \vdash R(F) : \theta :: \kappa \]

\[ F : \square_{\Omega(\theta)} \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \Delta, x : \bigwedge_{i \in I} \square_{m_i} \theta_i :: \kappa \vdash t : \theta :: \kappa' \quad I \subseteq J \]

\[ \Delta \vdash \lambda x . t : \left( \bigwedge_{j \in J} \square_{m_j} \theta_j \right) \to \theta :: \kappa \to \kappa' \]
A type-system for verification (Grellois-Melliès 2014)

Axiom
\[ x : \bigwedge_{\{i\}} \square_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa \]

\[ \{ (i, q_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i \} \text{ satisfies } \delta_A(q, a) \]

\[ \emptyset \vdash a : \bigwedge_{j=1}^{k_1} \square_{m_{1j}} q_{1j} \rightarrow \cdots \rightarrow \bigwedge_{j=1}^{k_n} \square_{m_{nj}} q_{nj} \rightarrow q :: \perp \rightarrow \cdots \rightarrow \perp \rightarrow \perp \]

App
\[ \Delta \vdash t : (\square_{m_1} \theta_1 \land \cdots \land \square_{m_k} \theta_k) \rightarrow \theta :: \kappa \rightarrow \kappa' \quad \Delta_i \vdash u : \theta_i :: \kappa \]
\[ \Delta + \square_{m_1} \Delta_1 + \cdots + \square_{m_k} \Delta_k \vdash tu : \theta :: \kappa' \]

fix
\[ \Gamma \vdash \mathcal{R}(F) : \theta :: \kappa \]
\[ F : \square_{\Omega(\theta)} \theta :: \kappa \vdash F : \theta :: \kappa \]

\[ \Delta, x : \bigwedge_{i \in I} \square_{m_i} \theta_i :: \kappa \vdash t : \theta :: \kappa' \]
\[ I \subseteq J \]
\[ \Delta \vdash \lambda x. t : \left(\bigwedge_{j \in J} \square_{m_j} \theta_j\right) \rightarrow \theta :: \kappa \rightarrow \kappa' \]
A type-system for verification (Grellois-Mellliès 2014)

This type system can have infinite-depth derivation.

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

**Theorem:** there is a winning run-tree over the tree produced by a scheme if and only if there exists a winning derivation of $\vdash S : q_0$ in the type system.

**Complexity (Ong):** rather huge... $n$-EXPTIME complete, for $n$ the order of the scheme. But actually not that awfull in practice.
A type-system for verification (Grellois-Melliès 2014)

This type system can have infinite-depth derivation.

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

**Theorem:** there is a winning run-tree over the tree produced by a scheme if and only if there exists a winning derivation of $\vdash S : q_0$ in the type system.

**Complexity (Ong):** rather huge... $n$-EXPTIME complete, for $n$ the order of the scheme. But actually not that awful in practice.
A type-system for verification (Grellois-Melliès 2014)

This type system can have infinite-depth derivation.

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

**Theorem:** there is a winning run-tree over the tree produced by a scheme if and only if there exists a winning derivation of $\vdash S : q_0$ in the type system.

**Complexity (Ong):** rather huge... $n$-EXPTIME complete, for $n$ the order of the scheme. But actually not that awfull in practice.
Consequences and remarks

- We can extend the theorem about the existence of a memoryless strategy for parity games in this setting and give a proof of **decidability** of model-checking in this way.

- We can work further on the type system and relax some colouring notions. This way we proved that the colouring operation is a modality (a comonad), and interpreted the verification problem in tensorial logic with colouring boxes.

- Many connections with models of linear logic: indexed logic, relational semantics (= run-tree), lattice semantics (= decidability)

- Also led to practical tools (C-Shore, TRECS)
Consequences and remarks

- We can extend the theorem about the existence of a memoryless strategy for parity games in this setting and give a proof of decidability of model-checking in this way.
- We can work further on the type system and relax some colouring notions. This way we proved that the colouring operation is a modality (a comonad), and interpreted the verification problem in tensorial logic with colouring boxes.
- Many connections with models of linear logic: indexed logic, relational semantics (= run-tree), lattice semantics (= decidability)
- Also led to practical tools (C-Shore, TRECS)
Consequences and remarks

- We can extend the theorem about the existence of a memoryless strategy for parity games in this setting and give a proof of **decidability** of model-checking in this way.

- We can work further on the type system and relax some colouring notions. This way we proved that the colouring operation is a modality (a comonad), and interpreted the verification problem in tensorial logic with colouring boxes.

- Many connections with models of linear logic: indexed logic, relational semantics (= run-tree), lattice semantics (= decidability)

- Also led to practical tools (C-Shore, TRECS)
Consequences and remarks

- We can extend the theorem about the existence of a memoryless strategy for parity games in this setting and give a proof of **decidability** of model-checking in this way.

- We can work further on the type system and relax some colouring notions. This way we proved that the colouring operation is a modality (a comonad), and interpreted the verification problem in tensorial logic with colouring boxes.

- Many connections with models of linear logic: indexed logic, relational semantics (= run-tree), lattice semantics (= decidability)

- Also led to practical tools (C-Shore, TRECS)