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.

Model-checking

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

Example: `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: `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: `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 λY 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 λY calculus with uninterpreted constants from a ranked alphabet.

(remember we are supposed to merge ;-))

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

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

A silly 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:

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

or, in λ-calculus style :

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

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 λ-calculus style :

```
S   =   L Nil
L   =   λx. if x (L (data x ))
```

A silly 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 :

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

Value tree of a recursion scheme

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

generates:

S

Value tree of a recursion scheme

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

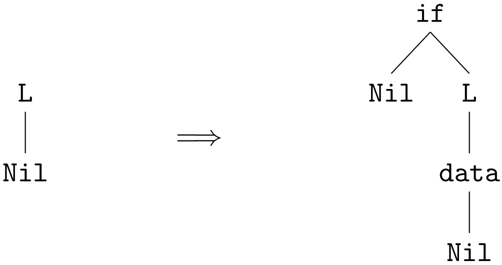
generates:

$\mathtt{S} \Longrightarrow$

Value tree of a recursion scheme

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

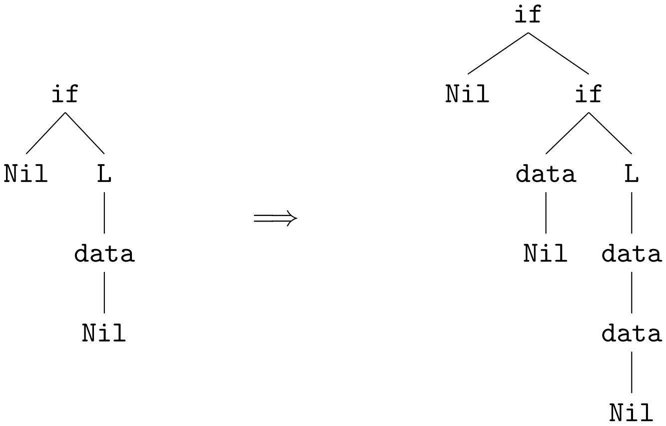
generates:



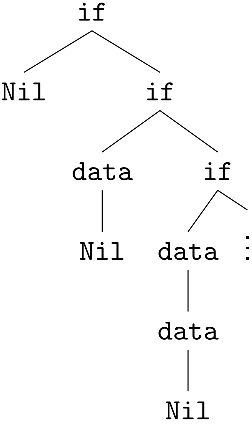
Value tree of a recursion scheme

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

generates:

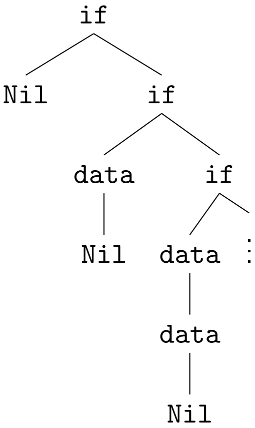


Value tree of a recursion scheme



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 $L : o \to o$, 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 $L : o \to o$, 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.):

$$\begin{array}{rcl} S & = & M\ \texttt{Nil} \\ M\ x & = & \texttt{if}\ (\ \mathit{commit}\ x\ )\ (\ A\ x\ M\ ) \\ A\ y\ \phi & = & \texttt{if}\ (\ \phi\ (\ \mathit{error}\ \mathit{end}\ )\ )\ (\ \phi\ (\ \mathit{cons}\ y\ )\ ) \end{array}$$

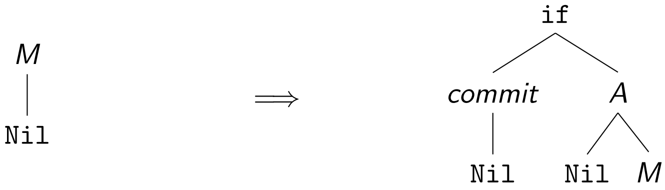
Value tree of a recursion scheme

$$\begin{array}{rcl} S & = & M\ \texttt{Nil} \\ M\ x & = & \texttt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\ A\ y\ \phi & = & \texttt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ ) \end{array}$$

$$S \qquad \Longrightarrow$$

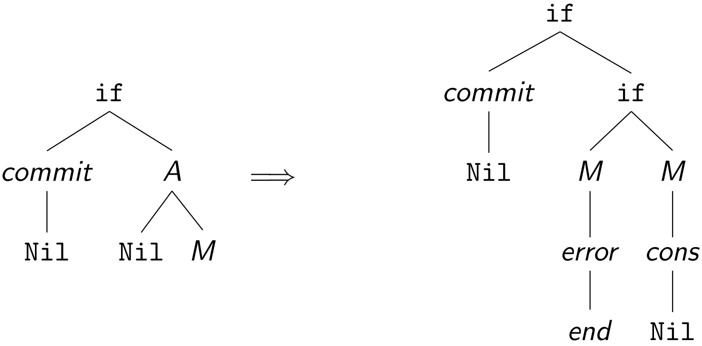
Value tree of a recursion scheme

$$\begin{array}{rcl} S & = & M\ \mathtt{Nil} \\ M\ x & = & \mathtt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\ A\ y\ \phi & = & \mathtt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ ) \end{array}$$



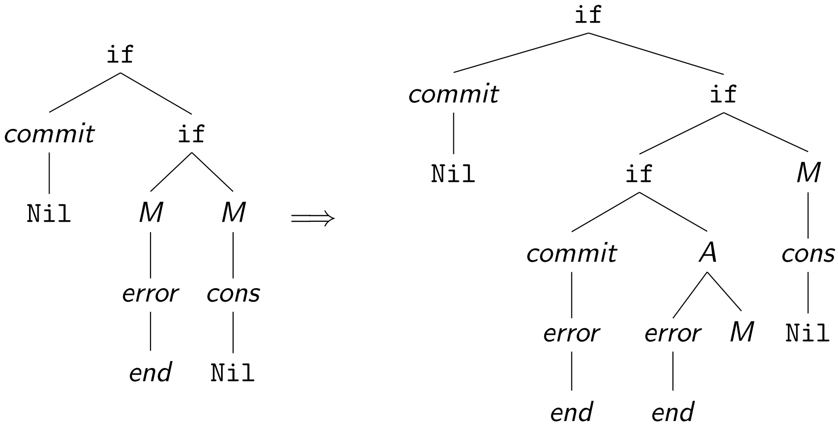
Value tree of a recursion scheme

$$\begin{array}{rcl} S & = & M\ \texttt{Nil} \\ M\ x & = & \texttt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\ A\ y\ \phi & = & \texttt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ ) \end{array}$$

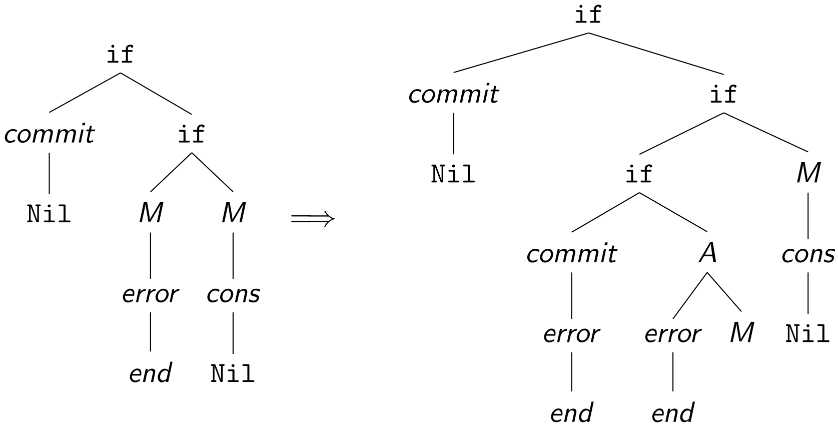


Value tree of a recursion scheme

$$\begin{array}{rcl} S & = & M\ \mathtt{Nil} \\ M\ x & = & \mathtt{if}\ (\ commit\ x\ )\ (\ A\ x\ M\ ) \\ A\ y\ \phi & = & \mathtt{if}\ (\ \phi\ (\ error\ end\ )\ )\ (\ \phi\ (\ cons\ y\ )\ ) \end{array}$$



Value tree of a recursion scheme



We would like to check that the program modelled by this scheme never commits an error.

Modal μ-calculus

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

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

Grammar: $\phi, \psi ::= X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\, \phi \mid \diamond_i\, \phi \mid \mu X\,.\,\phi \mid \nu X\,.\,\phi$

Modal μ-calculus

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

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

Grammar: $\phi, \psi \ ::= \ X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\,\phi \mid \ \diamond_i\,\phi \mid \mu X.\,\phi \mid \nu X.\,\phi$

Modal μ-calculus

Grammar: $\phi, \psi \ ::= \ X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\, \phi \mid \ \diamond_i\, \phi \mid \mu X.\, \phi \mid \nu X.\, \phi$

X is a variable

a is a predicate corresponding to a symbol of Σ

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

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

We can also define (variant) $\diamond \ = \ \bigvee_i \ \diamond_i$.

Modal μ-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\, \phi \mid \diamond_i\, \phi \mid \mu X.\, \phi \mid \nu X.\, \phi$

X is a variable

a is a predicate corresponding to a symbol of Σ

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

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

We can also define (variant) $\diamond = \bigvee_i \diamond_i$.

Modal μ-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\,\phi \mid \diamond_i\,\phi \mid \mu X.\,\phi \mid \nu X.\,\phi$

X is a variable

a is a predicate corresponding to a symbol of Σ

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

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

We can also define (variant) $\diamond = \bigvee_i \diamond_i$.

Modal μ-calculus

Grammar: $\phi, \psi ::= X \mid a \mid \phi \vee \psi \mid \phi \wedge \psi \mid \Box\, \phi \mid \diamond_i\, \phi \mid \mu X.\, \phi \mid \nu X.\, \phi$

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

$$\mu X.\, \phi(X) \quad \longrightarrow \quad \phi(\mu X.\, \phi(X)) \quad \longrightarrow \quad \phi(\phi(\mu X.\, \phi(X)))$$

Modal μ-calculus

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

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

$$\nu X.\phi(X) \quad \longrightarrow \quad \phi(\nu X.\phi(X)) \quad \longrightarrow \quad \phi(\phi(\nu X.\phi(X)))$$

Specifying a property in modal μ-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 $\Longleftrightarrow \mu X.(\ \diamond X \vee \mathit{error}\ )$

There is an error on a branch after a commit
$\Longleftrightarrow \mathit{commit} \wedge (\ \mu X.(\ \diamond X \vee \mathit{error}\ )\ )$

Specifying a property in modal μ-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 $\Longleftrightarrow \mu X.\,(\ \diamond X \vee error\ )$

There is an error on a branch after a commit
$\Longleftrightarrow commit \wedge (\ \mu X.\,(\ \diamond X \vee error\ )\ )$

Specifying a property in modal μ-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 $\iff \mu X.\,(\ \diamond X \vee \mathit{error}\ )$

There is an error on a branch after a commit
$\iff \mathit{commit} \wedge (\ \mu X.\,(\ \diamond X \vee \mathit{error}\ )\ )$

Specifying a property in modal μ-calculus

There is an error on a branch after a commit
$\Longleftrightarrow$ $\mathit{commit} \wedge (\ \mu X.\,(\ \diamond X \vee \mathit{error}\ )\ )$

There is a branch with an error after a commit
$\Longleftrightarrow$ $\mu Y.\ (\ \diamond Y \vee (\ \mathit{commit} \wedge (\ \mu X.\,(\ \diamond X \vee \mathit{error}\ )\ )\ )\ )$

Specifying a property in modal μ-calculus

Over the first example:

- Every branch ends by `Nil`:

  $$\mu X. \,(\, \texttt{Nil} \vee \Box X \,)$$

  but is it true ? Take instead

  $$\nu X. \,(\, \texttt{Nil} \vee \Box X \,)$$

- What does

  $$\nu X. \,(\, \texttt{if} \wedge \diamond_1 \,(\, \mu Y. \,(\, \texttt{Nil} \vee \Box Y \,)\,) \wedge \diamond_2 \, X \,)$$

  mean ?

Specifying a property in modal μ-calculus

Over the first example:

- Every branch ends by `Nil`:

  $$\mu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

  but is it true ? Take instead

  $$\nu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

- What does

  $$\nu X.\ (\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$$

  mean ?

Specifying a property in modal μ-calculus

Over the first example:

- Every branch ends by `Nil`:

  $$\mu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

  but is it true ? Take instead

  $$\nu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

- What does

  $$\nu X.\ (\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$$

  mean ?

Specifying a property in modal μ-calculus

Over the first example:

- Every branch ends by `Nil`:

  $$\mu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

  but is it true ? Take instead

  $$\nu X.\ (\ \texttt{Nil} \vee \Box X\ )$$

- What does

  $$\nu X.\ (\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$$

  mean ?

Specifying a property in modal μ-calculus

Over the first example:
What does

$$\nu X.\,(\,\texttt{if} \wedge \diamond_1\,(\,\mu Y.\,(\,\texttt{Nil} \vee \Box Y\,)\,) \wedge \diamond_2\,X\,)$$

mean ?

- There is an infinite branch, the rightmost one, only labelled with `if`.
- Every other branch is finite and ends with a `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

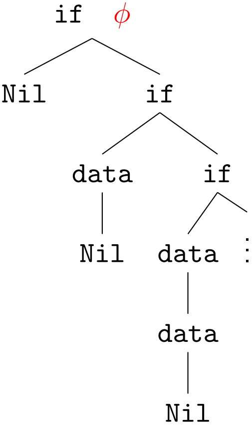
Logic is great !

. . . 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

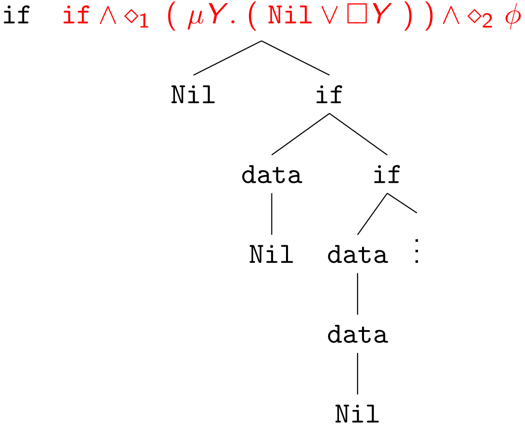
Idea: the formula "starts" on the root



where $\phi \ = \ \nu X.\ (\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2 X\ )$

Alternating parity tree automata

Idea: the formula "starts" on the root

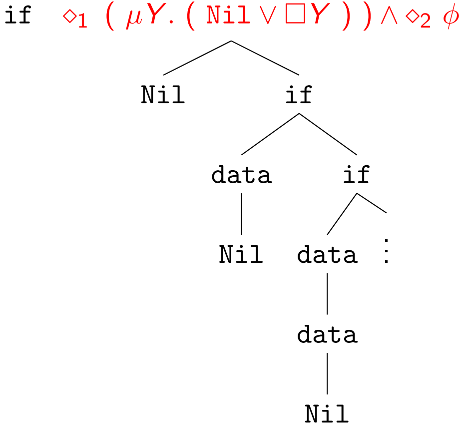


$$\texttt{if} \wedge \diamond_1 \; (\; \mu Y. \, (\; \texttt{Nil} \vee \Box Y \;) \;) \wedge \diamond_2 \; \phi$$

where $\phi \; = \; \nu X. \, (\; \texttt{if} \wedge \diamond_1 \; (\; \mu Y. \, (\; \texttt{Nil} \vee \Box Y \;) \;) \wedge \diamond_2 \; X \;)$

Alternating parity tree automata

Idea: the formula "starts" on the root

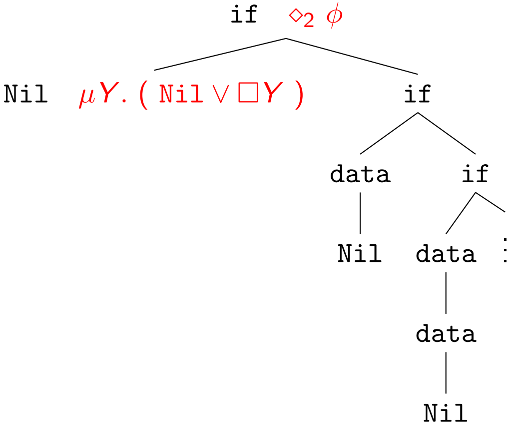


$\diamond_1 \; (\; \mu Y.\; (\; \mathtt{Nil} \vee \Box Y \;)\;) \wedge \diamond_2 \; \phi$

where $\phi \; = \; \nu X.\; (\; \mathtt{if} \wedge \diamond_1 \; (\; \mu Y.\; (\; \mathtt{Nil} \vee \Box Y \;)\;) \wedge \diamond_2 \, X \;)$

Alternating parity tree automata

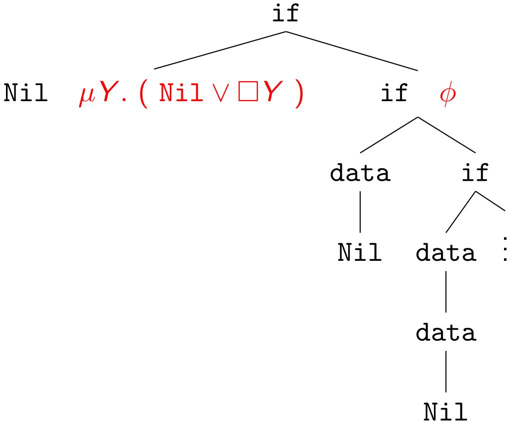
Idea: the formula "starts" on the root



where $\phi\ =\ \nu X.\ (\ \mathtt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \mathtt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$

Alternating parity tree automata

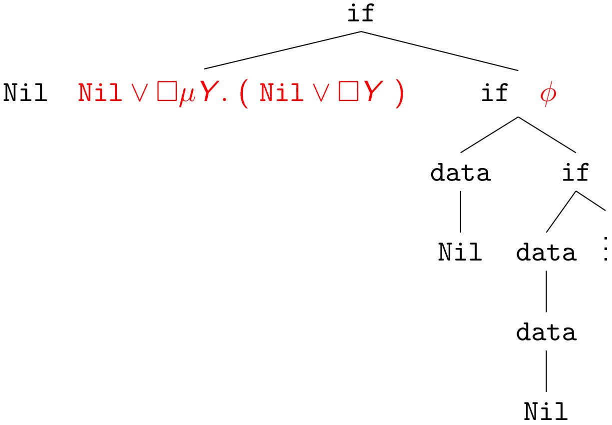
Idea: the formula "starts" on the root



where $\phi \;=\; \nu X.\,(\,\texttt{if} \wedge \diamond_1\,(\,\mu Y.\,(\,\texttt{Nil} \vee \Box Y\,)\,) \wedge \diamond_2\, X\,)$

Alternating parity tree automata

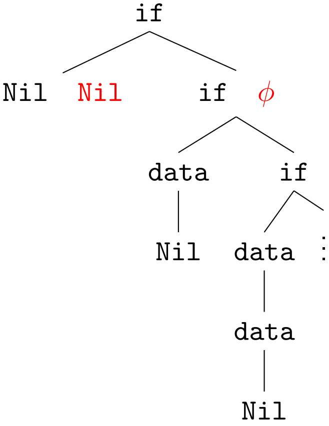
Idea: the formula "starts" on the root



where $\phi \;=\; \nu X.\,(\,\texttt{if} \wedge \diamond_1\,(\,\mu Y.\,(\,\texttt{Nil} \vee \Box Y\,)\,) \wedge \diamond_2\, X\,)$

Alternating parity tree automata

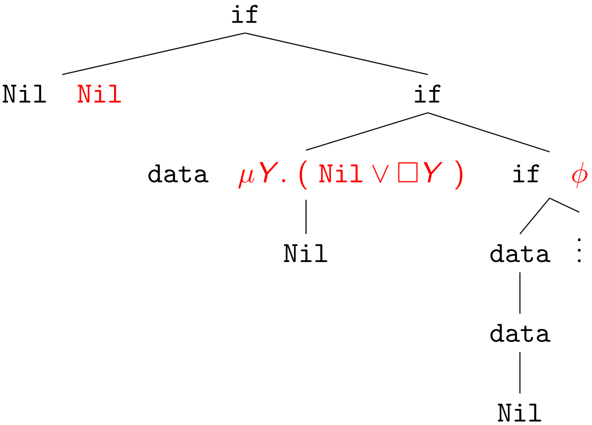
Idea: the formula "starts" on the root



where $\phi \;=\; \nu X.\;(\;\texttt{if} \wedge \diamond_1\;(\;\mu Y.\;(\;\texttt{Nil} \vee \Box Y\;)\;) \wedge \diamond_2\; X\;)$

Alternating parity tree automata

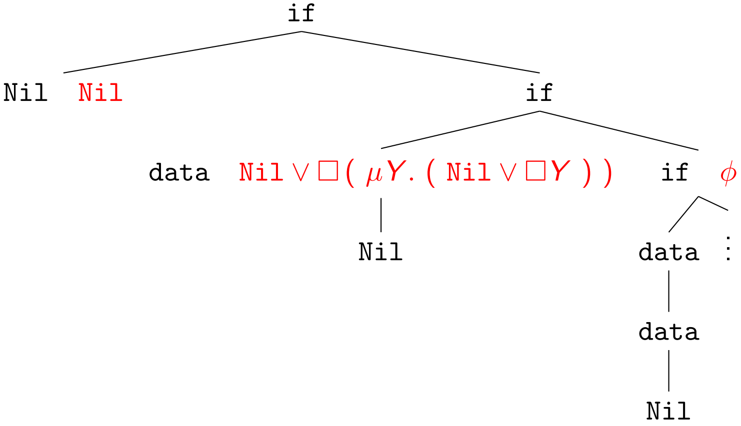
Idea: the formula "starts" on the root



where $\phi \; = \; \nu X. \, (\, \texttt{if} \wedge \diamond_1 \, (\, \mu Y. \, (\, \texttt{Nil} \vee \Box Y \,) \,) \wedge \diamond_2 X \,)$

Alternating parity tree automata

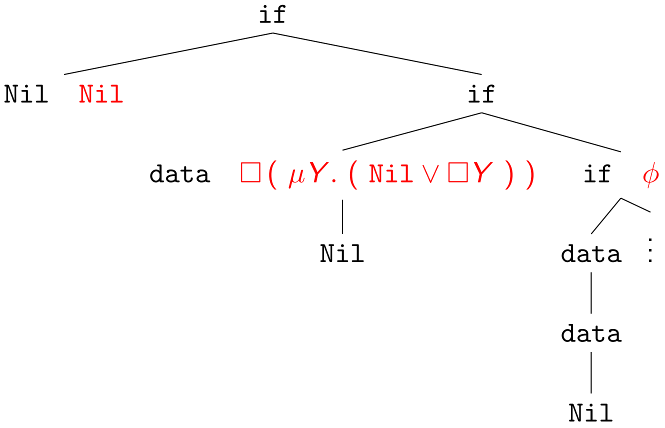
Idea: the formula "starts" on the root



where $\phi = \nu X. (\,\mathtt{if} \wedge \diamond_1 (\mu Y. (\mathtt{Nil} \vee \Box Y)) \wedge \diamond_2 X\,)$

Alternating parity tree automata

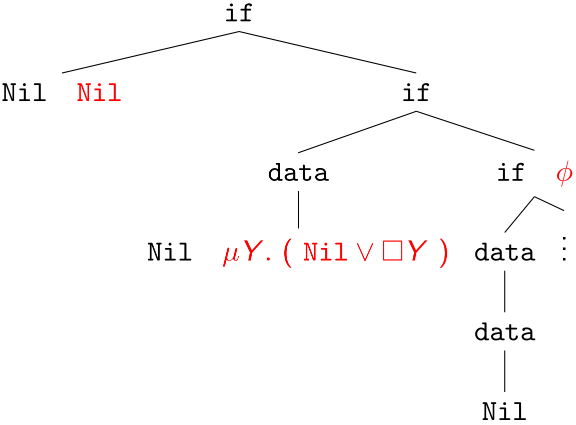
Idea: the formula "starts" on the root



where $\phi \; = \; \nu X. \, (\, \texttt{if} \wedge \diamond_1 \, (\, \mu Y. \, (\, \texttt{Nil} \vee \Box Y \,) \,) \wedge \diamond_2 \, X \,)$

Alternating parity tree automata

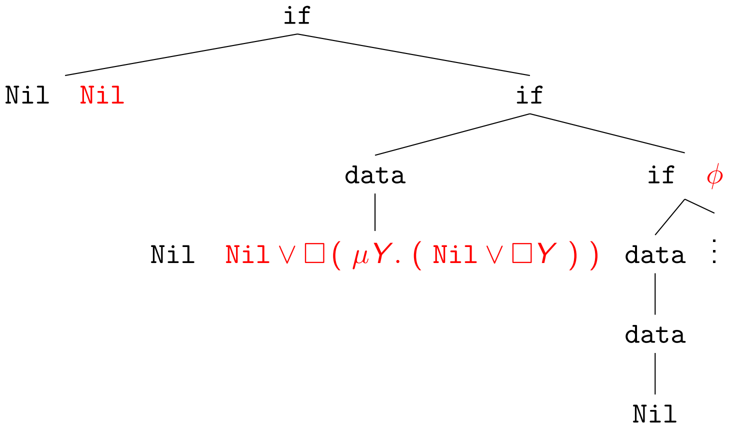
Idea: the formula "starts" on the root



where $\phi = \nu X.\,(\,\mathtt{if} \wedge \diamond_1\,(\,\mu Y.\,(\,\mathtt{Nil} \vee \Box Y\,)\,) \wedge \diamond_2 X\,)$

Alternating parity tree automata

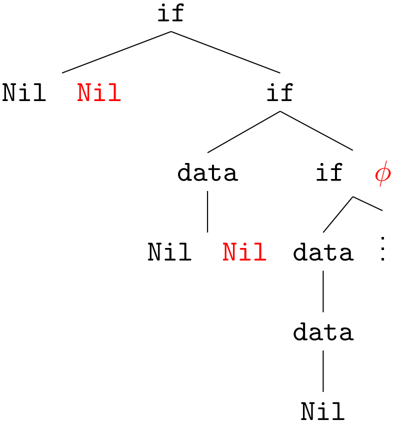
Idea: the formula "starts" on the root



where $\phi = \nu X. (\mathtt{if} \wedge \diamond_1 (\mu Y. (\mathtt{Nil} \vee \Box Y)) \wedge \diamond_2 X)$

Alternating parity tree automata

Idea: the formula "starts" on the root



where $\phi = \nu X. \, (\, \texttt{if} \wedge \Diamond_1 \, (\, \mu Y. \, (\, \texttt{Nil} \vee \Box Y \,) \,) \wedge \Diamond_2 \, X \,)$

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 $\vee$ and alternation for $\wedge$
- Needs a parity condition for distinguishing μ and ν

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 $\vee$ and alternation for $\wedge$
- Needs a parity condition for distinguishing μ and ν

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 $\vee$ and alternation for $\wedge$
- Needs a parity condition for distinguishing μ and ν

Alternating parity tree automata

$$\phi \;=\; \nu X.\,(\;\texttt{if} \wedge \diamond_1\;(\;\mu Y.\,(\;\texttt{Nil} \vee \Box Y\;)\;) \wedge \diamond_2\;X\;)$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \bot$
- $\delta(q_0, \texttt{data}) \;=\; \bot$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \;=\; \nu X.\,(\,\texttt{if} \wedge \diamond_1\,(\,\mu Y.\,(\,\texttt{Nil} \vee \Box Y\,)\,) \wedge \diamond_2\,X\,)$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \perp$
- $\delta(q_0, \texttt{data}) \;=\; \perp$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \ = \ \nu X.\ (\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\ (\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \ = \ \bot$
- $\delta(q_0, \texttt{data}) \ = \ \bot$
- $\delta(q_0, \texttt{if}) \ = \ (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \ = \ \top$
- $\delta(q_1, \texttt{data}) \ = \ (1, q_1)$
- $\delta(q_1, \texttt{if}) \ = \ (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \;=\; \nu X.\;(\;\texttt{if} \wedge \diamond_1\;(\;\mu Y.\;(\;\texttt{Nil} \vee \Box Y\;)\;) \wedge \diamond_2\;X\;)$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \perp$
- $\delta(q_0, \texttt{data}) \;=\; \perp$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \;=\; \nu X.\,(\ \texttt{if} \wedge \diamond_1\ (\ \mu Y.\,(\ \texttt{Nil} \vee \Box Y\ )\ ) \wedge \diamond_2\ X\ )$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \perp$
- $\delta(q_0, \texttt{data}) \;=\; \perp$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \;=\; \nu X.\,(\;\texttt{if} \wedge \diamond_1\;(\;\mu Y.\,(\;\texttt{Nil} \vee \Box Y\;)\;) \wedge \diamond_2\;X\;)$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \bot$
- $\delta(q_0, \texttt{data}) \;=\; \bot$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

Alternating parity tree automata

$$\phi \;=\; \nu X.\,(\;\texttt{if} \wedge \diamond_1\;(\;\mu Y.\,(\;\texttt{Nil} \vee \Box Y\;)\;) \wedge \diamond_2\; X\;)$$

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

Then:

- $\delta(q_0, \texttt{Nil}) \;=\; \perp$
- $\delta(q_0, \texttt{data}) \;=\; \perp$
- $\delta(q_0, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_0)$
- $\delta(q_1, \texttt{Nil}) \;=\; \top$
- $\delta(q_1, \texttt{data}) \;=\; (1, q_1)$
- $\delta(q_1, \texttt{if}) \;=\; (1, q_1) \wedge (2, q_1)$

Inductive/coinductive behaviour limitations: you can only play q_1 finitely, but there are no restrictions over q_0.

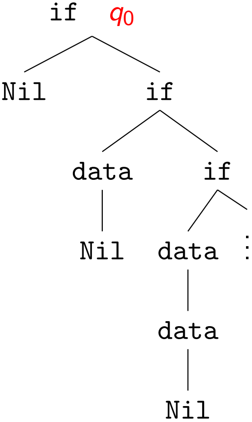
Alternating parity tree automata

In general, transitions may duplicate or drop a subtree.

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

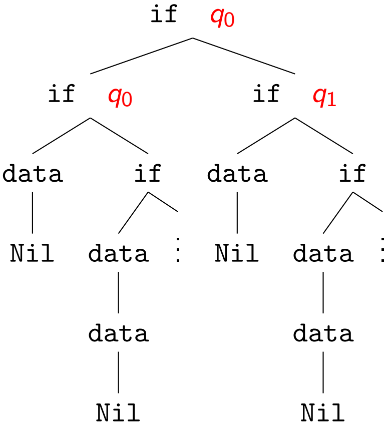
Alternating parity tree automata

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



Alternating parity tree automata

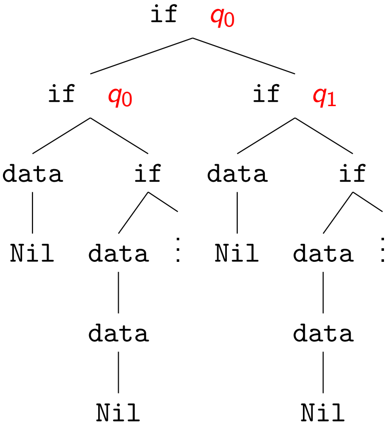
$\delta(q_0, \texttt{if}) \ = \ (2, q_0) \wedge (2, q_1).$



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

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.

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 ν.

Else if it is odd the automaton rejects: it means μ was unfolded infinitely, and this is forbidden.

Alternating parity tree automata

And for the inductive/coinductive behaviour ?

$\rightarrow$ 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 ν.

Else if it is odd the automaton rejects: it means μ was unfolded infinitely, and this is forbidden.

Alternating parity tree automata

And for the inductive/coinductive behaviour ?

$\rightarrow$ parity conditions

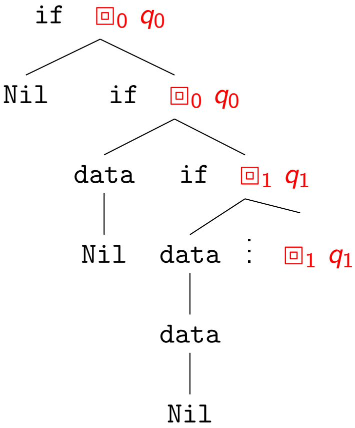
Over a branch of a run-tree, say q_0 has colour 0 and q_1 has colour 1.

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 ν.

Else if it is odd the automaton rejects: it means μ was unfolded infinitely, and this is forbidden.

Parity condition on an example



would not be a winning run-tree: the automaton unfolded μ infinitely on the infinite branch (note: δ 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 μ-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 $(\boxdot_{c_0}\; q_0 \wedge \boxdot_{c_1}\; q_1) \Rightarrow \boxdot_{c_2}\; q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)

Alternating parity tree automata and intersection types

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

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 $(\boxbox_{c_0}\ q_0 \wedge \boxbox_{c_1}\ q_1) \Rightarrow \boxbox_{c_2}\ q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)

Alternating parity tree automata and intersection types

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

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 $(\boxdot_{c_0}\ q_0 \wedge \boxdot_{c_1}\ q_1) \Rightarrow \boxdot_{c_2}\ q_2 \Rightarrow q$

(Kobayashi-Ong 2009, Grellois-Melliès 2014)

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 silly program models can be not regular. But schemes always are finite — and most of the time rather small.

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 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

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

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

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

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

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

A type-system for verification: without colours

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

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

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

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

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

A type-system for verification: without colours

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

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

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

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

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

A type-system for verification: without colours

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

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

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

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

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

A type-system for verification: without colours

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

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

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

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

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

A type-system for verification: example

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

and transitions:

$$\begin{array}{rcl} \delta(q_0, \texttt{Nil}) = \top & \Leftrightarrow & \texttt{Nil} : q_0 \\ \delta(q_0, \texttt{data}) = (1, q_0) & \Leftrightarrow & \texttt{data} : q_0 \rightarrow q_0 \\ \delta(q_0, \texttt{if}) = (\ (1, q_0) \wedge (2, q_0)\ ) \vee (2, q_1) & \Leftrightarrow & \texttt{if} : (q_0 \rightarrow q_0 \rightarrow q_0) \\ & & \quad \wedge(\emptyset \rightarrow q_1 \rightarrow q_0) \\ \delta(q_1, \texttt{if}) = (2, q_1) & \Leftrightarrow & \texttt{if} : \emptyset \rightarrow q_1 \rightarrow q_1 \end{array}$$

(example on the board — mistakes to check attention only ;-))

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

Axiom
$$\frac{}{x : \bigwedge_{\{i\}} \boxdot_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa}$$

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

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

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

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

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

Axiom $$\frac{}{x\ :\ \bigwedge_{\{i\}}\ \boxbox_{\Omega(\theta_i)}\ \theta_i\ ::\ \kappa\ \vdash\ x\ :\ \theta_i\ ::\ \kappa}$$

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

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

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

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

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

$$\text{Axiom} \qquad \frac{}{x : \bigwedge_{\{i\}} \boxdot_{\Omega(\theta_i)} \theta_i :: \kappa \ \vdash \ x : \theta_i :: \kappa}$$

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

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

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

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

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

$$\text{Axiom}\qquad \frac{}{x : \bigwedge_{\{i\}} \boxdot_{\Omega(\theta_i)} \theta_i :: \kappa \vdash x : \theta_i :: \kappa}$$

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

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

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

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

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

$$\text{Axiom}\qquad \frac{}{x \,:\, \bigwedge_{\{i\}} \boxdot_{\Omega(\theta_i)}\ \theta_i \ :: \ \kappa \ \vdash \ x \,:\, \theta_i \ :: \ \kappa}$$

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

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

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

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

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

This type system can have infinite-depth 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 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 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)