Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting

Patrick Bahr
paba@diku.dk

University of Copenhagen
Department of Computer Science

PhD Defence
30 November 2012
The Big Picture
The Big Picture

Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting
The Big Picture

Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting
Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting
two

The Big Pictures

Modular Implementation of Programming Languages

Partial Order Approach to Infinitary Rewriting
Modular Implementation of Programming Languages
Motivation

Implementation of a DSL-Based ERP System
Motivation
Implementation of a DSL-Based ERP System

Enterprise resource planning systems integrate several software components that are essential for managing a business.
Motivation
Implementation of a DSL-Based ERP System

Enterprise resource planning systems integrate several software components that are essential for managing a business.

**ERP systems integrate**

- Financial Management
- Supply Chain Management
- Manufacturing Resource Planning
- Human Resource Management
- Customer Relationship Management
- ...
Motivation
Implementation of a DSL-Based ERP System

Enterprise resource planning systems integrate several software components that are essential for managing a business.

ERP systems integrate
- Financial Management
- Supply Chain Management
- Manufacturing Resource Planning
- Human Resource Management
- Customer Relationship Management
- ...
What do ERP systems look like under the hood?
An Alternative Approach

POETS [Henglein et al. 2009]
An Alternative Approach
POETS [Henglein et al. 2009]

How do we implement this system without duplicating code?!
An Alternative Approach
POETS [Henglein et al. 2009]

ERP Runtime System

Rule Language

Contract Language

Report Language

Ontology Language

UI Language

...
An Alternative Approach
POETS [Henglein et al. 2009]

The abstract picture
- We have a number of domain-specific languages.
- Each pair of DSLs shares some common sublanguage.
- All of them share a common language of values.
- We have the same situation on the type level!
An Alternative Approach
POETS [Henglein et al. 2009]

The abstract picture
- We have a number of domain-specific languages.
- Each pair of DSLs shares some common sublanguage.
- All of them share a common language of values.
- We have the same situation on the type level!

How do we implement this system without duplicating code?!
Piecing Together DSLs – Syntax

Library of language features

F1 basic data structures
F2 reading and aggregating data from the database
F3 arithmetic operations
F4 contract clauses
F5 type definitions
F6 inference rules
Piecing Together DSLs – Syntax

Library of language features

F1  F2  F3  F4  F5  F6
Piecing Together DSLs – Syntax

Library of language features

- F1
- F2
- F3
- F4
- F5
- F6

Constructing the DSLs

Report Language = F1 F2 F3
Piecing Together DSLs – Syntax

Library of language features

Constructing the DSLs

Report Language = F1 F2 F3

Contract Language = F1 F4 F3
Piecing Together DSLs – Syntax

Library of language features

F1  F2  F3  F4  F5  F6

Constructing the DSLs

Report Language = F1 F2 F3

Contract Language = F1 F4 F3

Ontology Language = F1 F5
Piecing Together DSLs – Syntax

Library of language features

- F1
- F2
- F3
- F4
- F5
- F6

Constructing the DSLs

Report Language = F1 F2 F3

Contract Language = F1 F4 F3

Ontology Language = F1 F5

Rule Language = F1 F6 F3
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type $Program_L \rightarrow \text{String}$ for each language $L$
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type \( Program_L \rightarrow \text{String} \) for each language \( L \)

“functions” for each feature

\( pp_1 : F_1 \rightarrow \text{String} \)

\( pp_2 : F_2 \rightarrow \text{String} \)

\( pp_3 : F_3 \rightarrow \text{String} \)

\( pp_4 : F_4 \rightarrow \text{String} \)

\( pp_5 : F_5 \rightarrow \text{String} \)

\( pp_6 : F_6 \rightarrow \text{String} \)
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type $Program_L \rightarrow \text{String}$ for each language $L$

<table>
<thead>
<tr>
<th>“functions” for each feature</th>
<th>Combine functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pp_1 : F_1 \rightarrow \text{String}$</td>
<td>$pp_1$</td>
</tr>
<tr>
<td>$pp_2 : F_2 \rightarrow \text{String}$</td>
<td>$pp_2$</td>
</tr>
<tr>
<td>$pp_3 : F_3 \rightarrow \text{String}$</td>
<td>$pp_3$</td>
</tr>
<tr>
<td>$pp_4 : F_4 \rightarrow \text{String}$</td>
<td>$pp_1 + pp_2 + pp_3$</td>
</tr>
<tr>
<td>$pp_5 : F_5 \rightarrow \text{String}$</td>
<td>$pp_1 + pp_2 + pp_3$</td>
</tr>
<tr>
<td>$pp_6 : F_6 \rightarrow \text{String}$</td>
<td>$pp_1 + pp_2 + pp_3$</td>
</tr>
</tbody>
</table>
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type $\text{Program}_L \rightarrow \text{String}$ for each language $L$

<table>
<thead>
<tr>
<th>“functions” for each feature</th>
<th>Combine functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pp_1: F_1 \rightarrow \text{String}$</td>
<td>$pp_1 + pp_2 + pp_3 \rightarrow \text{String}$</td>
</tr>
<tr>
<td>$pp_2: F_2 \rightarrow \text{String}$</td>
<td></td>
</tr>
<tr>
<td>$pp_3: F_3 \rightarrow \text{String}$</td>
<td></td>
</tr>
<tr>
<td>$pp_4: F_4 \rightarrow \text{String}$</td>
<td></td>
</tr>
<tr>
<td>$pp_5: F_5 \rightarrow \text{String}$</td>
<td></td>
</tr>
<tr>
<td>$pp_6: F_6 \rightarrow \text{String}$</td>
<td></td>
</tr>
</tbody>
</table>
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type $Program_L \rightarrow \text{String}$ for each language $L$

"functions" for each feature

- $pp_1: F_1 \rightarrow \text{String}$
- $pp_2: F_2 \rightarrow \text{String}$
- $pp_3: F_3 \rightarrow \text{String}$
- $pp_4: F_4 \rightarrow \text{String}$
- $pp_5: F_5 \rightarrow \text{String}$
- $pp_6: F_6 \rightarrow \text{String}$

Combine functions

- $pp_1 + pp_2 + pp_3$: $F_1 + F_2 + F_3 \rightarrow \text{String}$
- $pp_1 + pp_5 + pp_6$: $F_1 + F_5 + F_6 \rightarrow \text{String}$

Report Language $L$
Piecing Together Functions

Example: Pretty Printing

Goal: functions of type $Program_L \rightarrow \text{String}$ for each language $L$

"functions" for each feature

- $pp_1 : F_1 \rightarrow \text{String}$
- $pp_2 : F_2 \rightarrow \text{String}$
- $pp_3 : F_3 \rightarrow \text{String}$
- $pp_4 : F_4 \rightarrow \text{String}$
- $pp_5 : F_5 \rightarrow \text{String}$
- $pp_6 : F_6 \rightarrow \text{String}$

Combine functions

- $pp_1 + pp_2 + pp_3 : F_1 F_2 F_3 \rightarrow \text{String}$

Other combinations

- $pp_1 + pp_5 + pp_6 : F_1 F_5 F_6 \rightarrow \text{String}$
- $\vdots$
How does it work?

Based on: Wouter Swierstra. *Data types à la carte*
How does it work?

```haskell
data Exp = Lit Int
           | Add Exp Exp
           | Mult Exp Exp
```

How does it work?

**Expression Data Type**

```
data Exp = Lit Int 
  | Add Exp Exp 
  | Mult Exp Exp
```

**Fixpoint Data Type**

```
data Fix s = 
  In (s (Fix s))
```

**Signature Data Type**

```
data Sig e = Lit Int 
  | Add e e 
  | Mult e e
```
How does it work?

**data** \( Exp = Lit \ Int \ |
\quad Add \ Exp \ Exp \ |
\quad Mult \ Exp \ Exp \)

**data** \( Fix \ s =
\quad In \ (s \ (Fix \ s)) \)

**data** \( Sig \ e = Lit \ Int \ |
\quad Add \ e \ e \ |
\quad Mult \ e \ e \)

**type** \( Exp = Fix \ Sig \)
How does it work?

```
data Exp = Lit Int
        | Add Exp Exp
        | Mult Exp Exp

data Fix s = In (s (Fix s))

data Sig e = Lit Int
           | Add e e
           | Mult e e

data Ops e = Add e e
            | Mult e e

type Exp = Fix Sig
```
How does it work?

```
data Exp = Lit | Add Exp Exp | Mult Exp Exp

data Fix s = In (s (Fix s))

data Sig e = Lit | Add e e | Mult e e

data Lit e = Lit Int

data Ops e = Add e e | Mult e e
```

**decompose**

**combine**

**type** $Exp = Fix Sig$
How does it work?

**Data**

\[ \text{data } \text{Exp} = \text{Lit } \text{Int} | \text{Add } \text{Exp} \text{Exp} | \text{Mult } \text{Exp} \text{Exp} \]

**Decompose**

\[ \text{data } \text{Fix } s = \text{In } (s (\text{Fix } s)) \]

**Signature**

\[ \text{data } \text{Sig } e = \text{Lit } \text{Int} | \text{Add } e e | \text{Mult } e e \]

**Type**

\[ \text{type } \text{Exp} = \text{Fix } \text{Sig} \]

**Combine**

\[ \text{data } \text{Ops } e = \text{Add } e e | \text{Mult } e e \]

\[ \text{data } \text{Lit } e = \text{Lit } \text{Int} \]

\[ \text{Lit } :+ : \text{Ops} \]
How does it work?

**Data**

- **Exp**
  - **Lit**
  - **Add Exp Exp**
  - **Mult Exp Exp**

- **Fix s**
  - **In (s (Fix s))**

- **Sig e**
  - **Lit Int**
  - **Add e e**
  - **Mult e e**

**Type**

- **Exp**
  - **Fix Sig**

- **Ops e**
  - **Add e e**
  - **Mult e e**

**Signature**

- **Lit :+: Ops**
Combining Functions

Explicit recursion

\[ pp :: \text{Exp} \rightarrow \text{String} \]
\[ pp \ (\text{Lit } i) \quad = \quad \text{show } i \]
\[ pp \ (\text{Add } e_1 \ e_2) \quad = \quad "\ (\ + \ pp \ e_1 \ + \ + \ pp \ e_2 \ + \ )" \]
\[ pp \ (\text{Mult } e_1 \ e_2) \quad = \quad "\ (\ + \ pp \ e_1 \ + \ * \ + \ pp \ e_2 \ + \ )" \]
Combining Functions

Explicit recursion

\[
pp :: \text{Exp} \rightarrow \text{String}
\]

\[
pp \ (\text{Lit } i) = \text{show } i
\]

\[
pp \ (\text{Add } e_1 \ e_2) = "(" + pp e_1 + " + " + pp e_2 + ")"
\]

\[
pp \ (\text{Mult } e_1 \ e_2) = "(" + pp e_1 + " * " + pp e_2 + ")"
\]

Non-recursive function

\[
pp' :: \text{Sig String} \rightarrow \text{String}
\]

\[
pp' \ (\text{Lit } i) = \text{show } i
\]

\[
pp' \ (\text{Add } e_1 \ e_2) = "(" + e_1 + " + " + e_2 + ")"
\]

\[
pp' \ (\text{Mult } e_1 \ e_2) = "(" + e_1 + " * " + e_2 + ")"
\]
Combining Functions

Explicit recursion

\[ pp :: \text{Exp} \rightarrow \text{String} \]
\[ pp (\text{Lit } i) = \text{show } i \]
\[ pp (\text{Add } e_1 e_2) = "(\text{ } + + pp e_1 + + " + + pp e_2 + + ")" \]
\[ pp (\text{Mult } e_1 e_2) = "(\text{ } + + pp e_1 + + " * + + pp e_2 + + ")" \]

Non-recursive function

\[ pp_1 :: \text{Lit String} \rightarrow \text{String} \]
\[ pp_1 (\text{Lit } i) = \text{show } i \]
\[ pp_2 :: \text{Ops String} \rightarrow \text{String} \]
\[ pp_2 (\text{Add } e_1 e_2) = "(\text{ } + + e_1 + + " + + e_2 + + ")" \]
\[ pp_2 (\text{Mult } e_1 e_2) = "(\text{ } + + e_1 + + " * + + e_2 + + ")" \]
Combining Functions

Non-recursive function

\[ pp_1 :: \text{Lit String} \rightarrow \text{String} \]
\[ pp_1 \ (\text{Lit } i) = \text{show } i \]

\[ pp_2 :: \text{Ops String} \rightarrow \text{String} \]
\[ pp_2 \ (\text{Add } e_1 \ e_2) = "( \ + \ e_1 \ + \ " + \ + \ e_2 \ + \ " )" \]
\[ pp_2 \ (\text{Mult } e_1 \ e_2) = "( \ + \ e_1 \ + \ " * \ + \ e_2 \ + \ " )" \]
Combining Functions

Non-recursive function

\[ pp_1 :: \text{Lit String} \rightarrow \text{String} \]
\[ pp_1 (\text{Lit } i) = \text{show } i \]

\[ pp_2 :: \text{Ops String} \rightarrow \text{String} \]
\[ pp_2 (\text{Add } e_1 e_2) = "( \text{++ } e_1 \text{ ++ } " + " \text{ ++ } e_2 \text{ ++ } \text{")"} \]
\[ pp_2 (\text{Mult } e_1 e_2) = "( \text{++ } e_1 \text{ ++ } " * " \text{ ++ } e_2 \text{ ++ } \text{")"} \]

Fold

\[ fold :: \text{Functor } f \Rightarrow (f \text{ a } \rightarrow \text{ a}) \rightarrow \text{Fix } f \rightarrow \text{ a} \]
\[ fold f (\text{In } t) = f (\text{fmap (fold f) } t) \]
Combining Functions

Non-recursive function

\[ pp_1 :: \text{Lit String} \rightarrow \text{String} \]
\[ pp_1 (\text{Lit } i) = \text{show } i \]

\[ pp_2 :: \text{Ops String} \rightarrow \text{String} \]
\[ pp_2 (\text{Add } e_1 e_2) = "(" ++ e_1 ++ " + " ++ e_2 ++ ")" \]
\[ pp_2 (\text{Mult } e_1 e_2) = "(" ++ e_1 ++ " * " ++ e_2 ++ ")" \]

Fold

\[ \text{fold} :: \text{Functor } f \Rightarrow (f \ a \rightarrow a) \rightarrow \text{Fix } f \rightarrow a \]
\[ \text{fold } f (\text{In } t) = f (\text{fmap } (\text{fold } f) \ t) \]

Applying Fold

\[ pp :: \text{Fix } (\text{Lit :+: Ops}) \rightarrow \text{String} \]
\[ pp = \text{fold } (pp_1 :+: pp_2) \]
Our Contributions

- Make compositional data types more useful in practice.
- Extend the class of definable types
- Mutually recursive types, GADTs
- Abstract syntax trees with variable binders
- "Algebras with more structure"
- Algebras with effects
- Tree homomorphisms, tree automata, tree transducers
- ▶ sequential composition
  ⇝ program optimisation (deforestation)
- ▶ tupling
  ⇝ additional modularity

Skip details
Our Contributions

Make compositional data types more useful in practise.
Our Contributions

Make compositional data types more useful in practise.

Extend the class of definable types
- mutually recursive types, GADTs
- abstract syntax trees with variable binders
Our Contributions

Make compositional data types more useful in practise.

Extend the class of definable types
- mutually recursive types, GADTs
- abstract syntax trees with variable binders

“Algebras with more structure”
- algebras with effects
- tree homomorphisms, tree automata, tree transducers
  - sequential composition $\rightsquigarrow$ program optimisation (deforestation)
  - tupling $\rightsquigarrow$ additional modularity

Skip details
Compositionality
We may compose tree automata along 3 different dimensions.
Compositionality

We may compose tree automata along 3 different dimensions.

**input signature:** the type of the AST

\[
[A_1]: \mu S_1 \to R \\
[A_2]: \mu S_2 \to R
\]
Compositionality

We may compose tree automata along 3 different dimensions.

**input signature**: the type of the AST

\[
\begin{align*}
[A_1] &: \mu S_1 \rightarrow R \\
[A_2] &: \mu S_2 \rightarrow R \\
\implies \quad [A_1 + A_2] &: \mu (S_1 + S_2) \rightarrow R
\end{align*}
\]
Compositionality

We may compose tree automata along 3 different dimensions.

**Input signature:** the type of the AST

\[
\begin{align*}
[A_1] & : \mu S_1 \rightarrow R \\
[A_2] & : \mu S_2 \rightarrow R
\end{align*}
\implies
\begin{align*}
[A_1 + A_2] & : \mu (S_1 + S_2) \rightarrow R
\end{align*}
\]

**Sequential composition:** a.k.a. deforestation

\[
\begin{array}{c}
\mu S_1 \\
\hline
[A_1]
\hline
\mu S_2 \\
\hline
[A_2]
\hline
\mu S_3
\end{array}
\]
Compositionality

We may compose tree automata along 3 different dimensions.

**input signature**: the type of the AST

\[
[A_1] : \mu S_1 \rightarrow R \\
[A_2] : \mu S_2 \rightarrow R \\
\Rightarrow \\
[A_1 + A_2] : \mu (S_1 + S_2) \rightarrow R
\]

**sequential composition**: a.k.a. deforestation

\[
\begin{align*}
\mu S_1 & \rightarrow [A_1] \\
[A_1] & \rightarrow \mu S_2 \\
\mu S_2 & \rightarrow [A_2] \\
[A_2] & \rightarrow \mu S_3 \\
\end{align*}
\]

\[
[A_1 \circ A_2]
\]
Compositionality
We may compose tree automata along 3 different dimensions.

**input signature:** the type of the AST

\[
[\mathcal{A}_1] : \mu S_1 \rightarrow R \\
[\mathcal{A}_2] : \mu S_2 \rightarrow R
\]

\[\Rightarrow\]

\[
[\mathcal{A}_1 + \mathcal{A}_2] : \mu (S_1 + S_2) \rightarrow R
\]

**sequential composition:** a.k.a. deforestation

\[
\begin{array}{c}
\mu S_1 \\
\mathcal{A}_1 \\
\mu S_2 \\
\mathcal{A}_2 \\
\mu S_3 \\
\end{array}
\]

\[
[\mathcal{A}_1 \circ \mathcal{A}_2]
\]

**output type:** tupling / product automaton construction

\[
[\mathcal{A}_1] : \mu S \rightarrow R_1 \\
[\mathcal{A}_2] : \mu S \rightarrow R_2
\]
Compositionality

We may compose tree automata along 3 different dimensions.

**input signature:** the type of the AST

\[
[A_1] : \mu S_1 \rightarrow R \\
[A_2] : \mu S_2 \rightarrow R
\]

\[\Rightarrow\]

\[
[A_1 + A_2] : \mu (S_1 + S_2) \rightarrow R
\]

**sequential composition:** a.k.a. deforestation

\[
\mu S_1 \quad [A_1] \quad \mu S_2 \quad [A_2] \quad \mu S_3
\]

\[
[A_1 \circ A_2]
\]

**output type:** tupling / product automaton construction

\[
[A_1] : \mu S \rightarrow R_1 \\
[A_2] : \mu S \rightarrow R_2
\]

\[\Rightarrow\]

\[
[A_1 \times A_2] : \mu F \rightarrow R_1 \times R_2
\]
Contextuality

tupling / product automaton construction

\[[A_1] : \mu S \rightarrow R_1\]
\[[A_2] : \mu S \rightarrow R_2\]  \implies  \[[A_1 \times A_2] : \mu(S) \rightarrow R_1 \times R_2\]
Contextuality

tupling / product automaton construction

$A_1: S \rightarrow R_1$  $A_2: S \rightarrow R_2$  $\Rightarrow$  $A_1 \times A_2: S \rightarrow R_1 \times R_2$
Contextuality

tupling / product automaton construction

\[ A_1 : S \to R_1 \quad \quad \quad A_2 : S \to R_2 \quad \quad \quad \implies \quad \quad \quad A_1 \times A_2 : S \to R_1 \times R_2 \]

mutumorphisms / dependent product automata

\[ A_1 : S \to R_1 \quad \quad A_2 : R_1 \Rightarrow S \to R_2 \]
Contextuality

**tupling / product automaton construction**

\[ \mathcal{A}_1 : S \to R_1 \]
\[ \mathcal{A}_2 : S \to R_2 \]
\[ \Rightarrow \]
\[ \mathcal{A}_1 \times \mathcal{A}_2 : S \to R_1 \times R_2 \]

**mutumorphisms / dependent product automata**

\[ \mathcal{A}_1 : S \to R_1 \]
\[ \mathcal{A}_2 : R_1 \Rightarrow S \to R_2 \]
\[ \Rightarrow \]
\[ \mathcal{A}_1 \times \mathcal{A}_2 : S \to R_1 \times R_2 \]
## Contextuality

### Tupling / Product Automaton Construction

<table>
<thead>
<tr>
<th>$\mathcal{A}_1$</th>
<th>$S \rightarrow R_1$</th>
<th>$\Rightarrow$</th>
<th>$\mathcal{A}_1 \times \mathcal{A}_2 : S \rightarrow R_1 \times R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{A}_2$</td>
<td>$S \rightarrow R_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Mutumorphisms / Dependent Product Automata

<table>
<thead>
<tr>
<th>$\mathcal{A}_1$</th>
<th>$R_2 \Rightarrow S \rightarrow R_1$</th>
<th>$\Rightarrow$</th>
<th>$\mathcal{A}_1 \times \mathcal{A}_2 : S \rightarrow R_1 \times R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{A}_2$</td>
<td>$R_1 \Rightarrow S \rightarrow R_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Contextuality

**tupling / product automaton construction**

\[
A_1 : S \rightarrow R_1 \\
A_2 : S \rightarrow R_2 \\
\Rightarrow \\
A_1 \times A_2 : S \rightarrow R_1 \times R_2
\]

**mutumorphisms / dependent product automata**

\[
A_1 : R_2 \Rightarrow S \rightarrow R_1 \\
A_2 : R_1 \Rightarrow S \rightarrow R_2 \\
\Rightarrow \\
[A_1 \times A_2] : \mu S \rightarrow R_1 \times R_2
\]
Discussion

Advantages

- it’s just a Haskell library
- uses well-known concepts (algebras, tree automata, functors etc.)
- high degree of modularity
- facilitates reuse

Drawbacks

- error messages are sometimes rather cryptic
- learning curve
- typical drawbacks of higher-order abstract syntax

Future work

- reasoning about modular implementations (Meta-Theory ` a la Carte [Delaware et al. 2013])
- describing interactions between modules
- how well does modularity scale?
## Discussion

### Advantages
- it’s just a Haskell library
- uses well-known concepts (algebras, tree automata, functors etc.)
- high degree of modularity
- facilitates reuse

### Drawbacks
- it’s just a Haskell library
- error messages are sometimes rather cryptic
- learning curve
- typical drawbacks of higher-order abstract syntax

### Future work
- reasoning about modular implementations (Meta-Theory `a la Carte [Delaware et al. 2013])
- describing interactions between modules
- how well does modularity scale?
## Discussion

### Advantages
- It’s just a Haskell library
- Uses well-known concepts (algebras, tree automata, functors etc.)
- High degree of modularity
- Facilitates reuse

### Drawbacks
- It's just a Haskell library
- Error messages are sometimes rather cryptic
- Learning curve
- Typical drawbacks of higher-order abstract syntax

### Future work
- **Reasoning** about modular implementations
  
  *(Meta-Theory à la Carte* [Delaware et al. 2013])
- Describing **interactions** between modules
- How well does modularity **scale**?
And now it’s time for something completely different.
Partial Order Approach to Infinitary Rewriting
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
R_{+\ast} = \begin{align*}
x + 0 & \rightarrow x \\
x \ast 0 & \rightarrow 0 \\
x + s(y) & \rightarrow s(x + y) \\
x \ast s(y) & \rightarrow x + (x \ast y) \\
s(s(0)) \ast s(s(0)) & \rightarrow R_{+\ast}
\end{align*}
\]

\( R_{+\ast} \) is terminating!
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$$
\mathcal{R}_{+*} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x * 0 & \rightarrow 0 \\
  x * s(y) & \rightarrow x + (x * y)
\end{cases}
$$
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[ \mathcal{R}_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \cdot 0 & \rightarrow 0 \\
  x \cdot s(y) & \rightarrow x + (x \cdot y) \\
  s(s(0)) \cdot s(s(0)) 
\end{cases} \]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$$
\mathcal{R}_{++} = \begin{cases} 
 x + 0 & \rightarrow x \\
 x + s(y) & \rightarrow s(x + y) \\
 x \ast 0 & \rightarrow 0 \\
 x \ast s(y) & \rightarrow x + (x \ast y) \\
 s(s(0)) \ast s(s(0)) & 
\end{cases}
$$
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
R_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x * 0 & \rightarrow 0 \\
  x * s(y) & \rightarrow x + (x * y) 
\end{cases}
\]

\[
s(s(0)) * s(s(0)) \rightarrow s(s(0)) + (s(s(0)) * s(0))
\]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[ \mathcal{R}_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \ast 0 & \rightarrow 0 \\
  x \ast s(y) & \rightarrow x + (x \ast y) 
\end{cases} \]

\[ s(s(0)) \ast s(s(0)) \rightarrow s(s(0)) + (s(s(0)) \ast s(0)) \]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{++} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x \cdot 0 & \rightarrow 0 \\
    x \cdot s(y) & \rightarrow x + (x \cdot y)
\end{cases}
\]

\[
s(s(0)) \cdot s(s(0)) \rightarrow^2 s(s(0)) + (s(s(0)) + (s(s(0)) \cdot 0))
\]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$$\mathcal{R}_{++} = \begin{cases} 
 x + 0 & \rightarrow x \\
 x + s(y) & \rightarrow s(x + y) \\
 x \times 0 & \rightarrow 0 \\
 x \times s(y) & \rightarrow x + (x \times y) \\
s(s(0)) \times s(s(0)) & \rightarrow^2 s(s(0)) + (s(s(0)) + (s(s(0)) \times 0))
\end{cases}$$
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{++} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x * 0 & \rightarrow 0 \\
    x * s(y) & \rightarrow x + (x * y) \\
\end{cases}
\]

\[
s(s(0)) * s(s(0)) \rightarrow^3 s(s(0)) + (s(s(0)) + 0)
\]
Rewriting Systems

What are (term) rewriting systems?
- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{++} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x \ast s(y) & \rightarrow x + (x \ast y) \\
    s(s(0)) \ast s(s(0)) & \rightarrow^3 s(s(0)) + (s(s(0)) + 0)
\end{cases}
\]

\( \mathcal{R}_{++} \) is terminating!
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[ R_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \times 0 & \rightarrow 0 \\
  x \times s(y) & \rightarrow x + (x \times y) \\
  s(s(0)) \times s(s(0)) & \rightarrow^4 s(s(0)) + s(s(0)) 
\]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \ast 0 & \rightarrow 0 \\
  x \ast s(y) & \rightarrow x + (x \ast y) \\
\end{cases}
\]

\[
s(s(0)) \ast s(s(0)) \rightarrow^4 s(s(0)) + s(s(0))
\]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$\mathcal{R}_{+*} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x + (s(0)) \ast s(s(0)) & \rightarrow^5 s(s(s(0)) + s(0)) \\
    x \ast 0 & \rightarrow 0 \\
    x \ast s(y) & \rightarrow x + (x \ast y) 
\end{cases}$
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[ R_{++} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \cdot 0 & \rightarrow 0 \\
  x \cdot s(y) & \rightarrow x + (x \cdot y) \\
  s(s(0)) \cdot s(s(0)) & \rightarrow^5 s(s(s(0)) + s(0)) 
\]
Rewriting Systems

What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{+*} = \begin{cases} 
  x + 0 & \rightarrow x \\
  x + s(y) & \rightarrow s(x + y) \\
  x \times 0 & \rightarrow 0 \\
  x \times s(y) & \rightarrow x + (x \times y)
\end{cases}
\]

\[
s(s(0)) \times s(s(0)) \rightarrow^6 s(s(s(s(0)) + 0))
\]
What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form \( l \rightarrow r \)
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

\[
\mathcal{R}_{+*} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x \ast 0 & \rightarrow 0 \\
    x \ast s(y) & \rightarrow x + (x \ast y)
\end{cases}
\]

\[s(s(0)) \ast s(s(0)) \rightarrow^6 s(s(s(s(0)) + 0))\]
What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$$\mathcal{R}_{++} = \begin{cases} 
  x + 0 & \rightarrow \ x \\
  x + s(y) & \rightarrow \ s(x + y) \\
  x \times 0 & \rightarrow \ 0 \\
  x \times s(y) & \rightarrow \ x + (x \times y)
\end{cases}$$

$$s(s(0)) \times s(s(0)) \rightarrow^7 s(s(s(s(0)))))$$
Rewriting Systems

What are (term) rewriting systems?
- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \rightarrow r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

Example (Term rewriting system defining addition and multiplication)

$$R_{++} = \begin{cases} 
    x + 0 & \rightarrow x \\
    x + s(y) & \rightarrow s(x + y) \\
    x \times 0 & \rightarrow 0 \\
    x \times s(y) & \rightarrow x + (x \times y) \\
\end{cases}$$

$$s(s(0)) \times s(s(0)) \rightarrow^7 s(s(s(s(0))))$$

$R_{++}$ is terminating!
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Example (Infinite lists)

\[ \mathbb{N} = \{ \text{from}(x) \to x : \text{from}(s(x)) \to \text{from}(0) \} \]

Intuitively this converges to the infinite list 0 : 1 : 2 : 3 : 4 : 5 : ...
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling **reactive systems**, e.g. by process calculi
- **approximation algorithms** which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of **infinite data structures**, e.g. streams
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$\mathcal{R}_{nats} = \begin{cases} 
\text{from}(x) \rightarrow x : \text{from}(s(x)) \\
\text{from}(0) 
\end{cases}$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$R_{nats} = \begin{cases} 
from(x) \rightarrow x : from(s(x)) 
\end{cases}$$

$$from(0) \rightarrow 0 : from(1)$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

\[ R_{nats} = \left\{ \begin{array}{l} 
   \text{from}(x) \rightarrow x : \text{from}(s(x)) \\
   \text{from}(0) \rightarrow \underbrace{0 : 1 : \text{from}(2)}_{2}
\end{array} \right\} \]
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$\mathcal{R}_{nats} = \begin{cases} 
from(x) \rightarrow x : from(s(x)) 
\end{cases}$$

$$from(0) \Rightarrow^3 0 : 1 : 2 : from(3)$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$\mathcal{R}_{\text{nats}} = \left\{ \text{from}(x) \rightarrow x : \text{from}(s(x)) \right\}$$

$$\text{from}(0) \rightarrow^4 0 : 1 : 2 : 3 : \text{from}(4)$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$R_{nats} = \left\{ from(x) \rightarrow x : from(s(x)) \right\}$$

$$from(0) \rightarrow^5 0 : 1 : 2 : 3 : 4 : from(5)$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing \( \pi \)
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

\[
\mathcal{R}_{\text{nats}} = \left\{ \begin{array}{l}
\text{from}(x) \rightarrow x : \text{from}(s(x)) \\
\text{from}(0) \rightarrow^6 0 : 1 : 2 : 3 : 4 : 5 : \text{from}(6)
\end{array} \right. 
\]
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing $\pi$
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

$$\mathcal{R}_{nats} = \left\{ \text{from}(x) \rightarrow x : \text{from}(s(x)) \right\}$$

$$\text{from}(0) \rightarrow^6 0 : 1 : 2 : 3 : 4 : 5 : \text{from}(6) \rightarrow \ldots$$
Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

Non-terminating systems can be meaningful

- modelling reactive systems, e.g. by process calculi
- approximation algorithms which enhance the accuracy of the approximation with each iteration, e.g. computing \( \pi \)
- specification of infinite data structures, e.g. streams

Example (Infinite lists)

\[
R_{\text{nats}} = \left\{ \text{from}(x) \rightarrow x : \text{from}(s(x)) \right\}
\]

\[
\text{from}(0) \rightarrow^6 0 : 1 : 2 : 3 : 4 : 5 : \text{from}(6) \rightarrow \ldots
\]

intuitively this converges to the infinite list 0 : 1 : 2 : 3 : 4 : 5 : \ldots
When does a rewrite sequence converge?

Rewrite rules are applied at increasingly deeply nested subterms.
Infinitary Term Rewriting – The Metric Approach

When does a rewrite sequence converge?
Rewrite rules are applied at increasingly deeply nested subterms.

What is the result of a converging rewrite sequence?
A converging rewrite sequence \textit{approximates} a uniquely determined term $t$ arbitrarily well.
Infinitary Term Rewriting – The Metric Approach

When does a rewrite sequence converge?
Rewrite rules are applied at increasingly deeply nested subterms.

What is the result of a converging rewrite sequence?
A converging rewrite sequence approximates a uniquely determined term $t$ arbitrary well.

$t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t$
When does a rewrite sequence converge?
Rewrite rules are applied at increasingly deeply nested subterms.

What is the result of a converging rewrite sequence?
A converging rewrite sequence approximates a uniquely determined term \( t \) arbitrary well.

\[
t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t
\]

For each depth \( d \in \mathbb{N} \) there is some \( n \in \mathbb{N} \), such that
When does a rewrite sequence converge?
Rewrite rules are applied at increasingly deeply nested subterms.

What is the result of a converging rewrite sequence?
A converging rewrite sequence approximates a uniquely determined term \( t \) arbitrary well.

\[
\begin{align*}
t_0 & \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t \\
\end{align*}
\]

For each depth \( d \in \mathbb{N} \) there is some \( n \in \mathbb{N} \), such that

\[
\begin{align*}
t_0 & \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_{n+1} \rightarrow \ldots \rightarrow t \\
\end{align*}
\]

\( \text{do not differ up to depth } d \)
Example: Convergence of a Reduction

\[ R = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ R = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ \mathcal{R} = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ \mathcal{R} = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ \mathcal{R} = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ R = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ R = \{ a \rightarrow g(a) \} \]
Example: Convergence of a Reduction

\[ R = \{ a \rightarrow g(a) \} \]
Example: Non-Convergence of a Reduction

\[ R = \begin{cases} 
    a \rightarrow g(a) \\
    h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \begin{cases} 
  a \rightarrow g(a) \\
  h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \begin{cases} 
  a \rightarrow g(a) \\
  h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ R = \begin{cases} 
  a \rightarrow g(a) \\
  h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \begin{cases} 
    a \rightarrow g(a) \\
    h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \left\{ \begin{array}{c} a \rightarrow g(a) \\ h(x) \rightarrow h(g(x)) \end{array} \right\} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \begin{cases} 
    a \rightarrow g(a) \\
    h(x) \rightarrow h(g(x)) 
\end{cases} \]
Example: Non-Convergence of a Reduction

\[ R = \left\{ \begin{array}{l} a \rightarrow g(a) \\ h(x) \rightarrow h(g(x)) \end{array} \right\} \]
Example: Non-Convergence of a Reduction

\[ \mathcal{R} = \begin{cases} 
    a \rightarrow g(a) \\
    h(x) \rightarrow h(g(x)) 
\end{cases} \]
Issues of the Metric Approach

- Notion of convergence is too restrictive (no notion of local convergence)
- May still not reach a normal form
- Orthogonal TRSs are not infinitarily confluent
Issues of the Metric Approach

- Notion of convergence is too restrictive (no notion of local convergence)
- May still not reach a normal form
- Orthogonal TRSs are not infinitarily confluent

Infinitary confluence

For every $t, t_1, t_2 \in T^\infty(\Sigma, \mathcal{V})$ with $t_1 \leftarrow t \rightarrow t_2$
Issues of the Metric Approach

- Notion of convergence is too restrictive
  (no notion of local convergence)
- May still not reach a normal form
- Orthogonal TRSs are not infinitarily confluent

Infinitary confluence

For every \( t, t_1, t_2 \in T^\infty(\Sigma, \mathcal{V}) \)
with \( t_1 \leftarrow t \rightarrow t_2 \)
there is a \( t' \in T^\infty(\Sigma, \mathcal{V}) \)
with \( t_1 \rightarrow t' \leftarrow t_2 \)
Partial Order Approach to Infinitary Term Rewriting

Partial order on terms

- **partial terms**: terms with additional constant \( \bot \) (read as “undefined”)
- partial order \( \leq \bot \) reads as: “is less defined than”
- \( \leq \bot \) is a **complete semilattice** (\( = \) bounded complete cpo)

Convergence formalised by the limit inferior:

\[
\liminf_{\iota} \alpha_t \iota = \bigwedge_{\beta < \alpha} l_\beta \leq \iota < \alpha t \iota
\]

Intuition: eventual persistence of nodes of the terms

Convergence: limit inferior of the contexts of the reduction
Partial Order Approach to Infinitary Term Rewriting

Partial order on terms

- **Partial terms**: terms with additional constant ⊥ (read as “undefined”)
- Partial order \( \leq_{\bot} \) reads as: “is less defined than”
- \( \leq_{\bot} \) is a complete semilattice (= bounded complete cpo)

Convergence

- formalised by the limit inferior:

\[
\liminf_{t} t = \bigcup_{\beta < \alpha} \bigcap_{\beta \leq t < \alpha} t
\]
Partial Order Approach to Infinitary Term Rewriting

Partial order on terms

- **partial terms**: terms with additional constant ⊥ (read as “undefined”)
- partial order \( \leq \perp \) reads as: “is less defined than”
- \( \leq \perp \) is a complete semilattice (= bounded complete cpo)

Convergence

- formalised by the limit inferior:

\[
\liminf_{i \to \alpha} t_i = \bigcup_{\beta < \alpha} \bigcap_{\beta \leq i < \alpha} t_i
\]

- intuition: eventual persistence of nodes of the terms
- convergence: limit inferior of the contexts of the reduction
An Example

Eventually stable: \( \bot \)
An Example

Eventually stable: $\bot$
An Example

eventually stable:
Eventually stable:

$p$-converges to

\[ \bot \]
Properties of the Partial Order Approach

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>- reduction sequences always converge (but result may contain ⊥s)</td>
</tr>
<tr>
<td>- more fine-grained than the metric approach</td>
</tr>
<tr>
<td>- subsumes metric approach, i.e. both approaches agree on total reductions</td>
</tr>
</tbody>
</table>

Theorem (total \( p \)-convergence = \( m \)-convergence)

For every reduction \( S \) in a TRS, we have

\[ S : s \rightarrow^p t \text{ is total } \quad \iff \quad S : s \rightarrow^m t. \]

Theorem (confluence, normalisation)

Every orthogonal TRS is normalising and confluent w.r.t. \( p \)-convergent reductions, i.e. every term has a unique normal form.
Properties of the Partial Order Approach

Benefits
- reduction sequences always converge (but result may contain ⊥s)
- more fine-grained than the metric approach
- subsumes metric approach, i.e. both approaches agree on total reductions

Theorem (total $p$-convergence $\equiv m$-convergence)

For every reduction $S$ in a TRS, we have

$S: s \xrightarrow{p} t$ is total $\iff S: s \xrightarrow{m} t$. 


Properties of the Partial Order Approach

**Benefits**
- reduction sequences always converge (but result may contain \(\perp\)s)
- more fine-grained than the metric approach
- subsumes metric approach, i.e. both approaches agree on total reductions

**Theorem (total \(p\)-convergence = \(m\)-convergence)**

For every reduction \(S\) in a TRS, we have

\[ S: s \xrightarrow[p]{} t \text{ is total} \iff S: s \xrightarrow[m]{} t. \]

**Theorem (confluence, normalisation)**

Every orthogonal TRS is normalising and confluent w.r.t. \(p\)-convergent reductions, i.e. every term has a unique normal form.
Lazy evaluation consists of two things:

- non-strict evaluation
- sharing
Sharing – From Terms to Term Graphs

Lazy evaluation and infinitary rewriting

Lazy evaluation consists of two things:

- non-strict evaluation
- sharing \( \sim \) avoids duplication
Lazy evaluation consists of two things:

- non-strict evaluation
- sharing $\sim$ avoids duplication

Example:

$\text{from}(x) \rightarrow x : \text{from}(s(x))$
Lazy evaluation and infinitary rewriting

Lazy evaluation consists of two things:

- non-strict evaluation
- sharing \( \rightsquigarrow \) avoids duplication

Example

\[ \text{from}(x) \rightarrow x : \text{from}(s(x)) \]
Lazy evaluation and infinitary rewriting

Lazy evaluation consists of two things:
- non-strict evaluation
- sharing \( \rightsquigarrow \) avoids duplication

Example

\[
\text{from}(x) \rightarrow x : \text{from}(s(x))
\]
Lazy evaluation and infinitary rewriting

Lazy evaluation consists of two things:

- non-strict evaluation
- sharing $\sim$ avoids duplication

Example

\[ \text{from}(x) \rightarrow x : \text{from}(s(x)) \]
Example

from

\downarrow

0
Example
Example

```
from 0 : 0 from
   ↓      ↓      ↓
  0      0      s
```
Example

```
from 0 0
\downarrow \downarrow
s
```

```
0 0 from 0
\downarrow \downarrow
s
```

```
s from
\downarrow \downarrow
s
```
Example

\[
\begin{array}{c}
\text{from} & \rightarrow & : & : & \cdots \\
0 & & 0 & & 0 \\
\ \ & & s & & s \\
\downarrow & & \uparrow & & \uparrow \\
0 & & \text{from} & & 0 \\
\end{array}
\]
Example
Properties of Infinitary Term Graph Rewriting

Theorem (total $p$-convergence = $m$-convergence)

For every reduction $S$ in a GRS, we have

\[ S: g \xrightarrow{p} h \text{ is total} \iff S: g \xrightarrow{m} h. \]
Theorem (total $p$-convergence = $m$-convergence)

For every reduction $S$ in a GRS, we have

\[ S : g \xrightarrow{p} h \text{ is total} \iff S : g \xrightarrow{m} h. \]

Theorem (soundness)

For every left-linear, left-finite GRS $R$ we have

\[ R \quad g \quad \overrightarrow{p} \quad h \]
Properties of Infinitary Term Graph Rewriting

**Theorem (total \( p \)-convergence = \( m \)-convergence)**

For every reduction \( S \) in a GRS, we have

\[
S : g \xrightarrow{p} h \text{ is total } \iff \ S : g \xrightarrow{m} h.
\]

**Theorem (soundness)**

For every left-linear, left-finite GRS \( R \) we have

\[
\begin{array}{cccccc}
\mathcal{U}(\cdot) & g & \xrightarrow{p} & \mathcal{U}(\cdot) & h \\
\mathcal{U}(R) & s & \xrightarrow{p} & \mathcal{U}(R) & t
\end{array}
\]
Properties of Infinitary Term Graph Rewriting

Theorem (total $p$-convergence $=$ $m$-convergence)

For every reduction $S$ in a GRS, we have

\[ S : g \xrightarrow{p} h \text{ is total} \iff S : g \xrightarrow{m} h. \]

Theorem (soundness)

For every left-linear, left-finite GRS $\mathcal{R}$ we have

\[ \mathcal{R}, \quad g \quad m \quad h \quad \underline{U} (\cdot) \quad u (\cdot) \quad s \quad m \quad t \]
Completeness

Theorem (Completeness)

*p*-convergence in an orthogonal, left-finite GRS \( \mathcal{R} \) is complete:

\[
\begin{array}{ccl}
\mathcal{U}(\mathcal{R}) & \xrightarrow{s} & \mathcal{U}(\cdot) \\
\mathcal{U}(\cdot) & \xrightarrow{p} & t \\
\end{array}
\]
Completeness

Theorem (Completeness)

\( p \)-convergence in an orthogonal, left-finite GRS \( \mathcal{R} \) is complete:

\[
\begin{align*}
\mathcal{U}(\mathcal{R}) & \xrightarrow{\mathcal{U}(\cdot)} t \\
g & \xrightarrow{p} h
\end{align*}
\]
Completeness

Theorem (Completeness)

\textit{p-convergence in an orthogonal, left-finite GRS }\mathcal{R}\textit{ is complete:}

\[
\begin{align*}
\mathcal{U}(\mathcal{R}) & \xrightarrow{p} t & \xrightarrow{p} t' \\
\mathcal{U}(\cdot) & & \\
\mathcal{R} & \xrightarrow{p} h
\end{align*}
\]

Does not hold for metric convergence!
Completeness

**Theorem (Completeness)**

*p*-convergence in an orthogonal, left-finite GRS $\mathcal{R}$ is complete:

\[
\begin{array}{c}
\mathcal{U}(\mathcal{R}) \\
\mathcal{U}(\cdot) \\
\mathcal{R}
\end{array}
\xrightarrow{s} 
\xrightarrow{p} 
\xrightarrow{t} 
\xrightarrow{p} 
\xrightarrow{t'} 
\xrightarrow{\mathcal{U}(\cdot)} 
\xrightarrow{h}
\]

Does not hold for metric convergence!

**Completeness of $m$-convergence for normalising reductions**

\[
\begin{array}{c}
\mathcal{U}(\mathcal{R}) \\
\mathcal{U}(\cdot) \\
\mathcal{R}
\end{array}
\xrightarrow{s} 
\xrightarrow{m} 
\xrightarrow{t \in \text{NF}} 
\xrightarrow{\mathcal{U}(\cdot)} 
\xrightarrow{h}
\]
Discussion

Contributions

- novel approach to infinitary term rewriting
- first formalisation of infinitary term graph rewriting
Discussion

Contributions

- novel approach to infinitary term rewriting
- first formalisation of infinitary term graph rewriting

Note: Böhm reduction for TRSs

\[ s \xrightarrow[p]{R} t \iff s \xrightarrow[m]{B} t \]

\( B \) adds to \( R \) rules of the form \( t \rightarrow \bot \) for each root-active term \( t \).
Discussion

Contributions

- novel approach to infinitary term rewriting
- first formalisation of infinitary term graph rewriting

Note: Böhm reduction for TRSs

\[
s \xrightarrow{p} \mathcal{R} t \iff s \xrightarrow{m} \mathcal{B} t
\]

\(\mathcal{B}\) adds to \(\mathcal{R}\) rules of the form \(t \rightarrow \bot\) for each term \(t\) with \(t \xrightarrow{p} \bot\).
Discussion

Contributions

- novel approach to infinitary term rewriting
- first formalisation of infinitary term graph rewriting

Note: Böhm reduction for TRSs

\[ s \xrightarrow{\mathcal{P}}_\mathcal{R} t \iff s \xrightarrow{m}_\mathcal{B} t \]

\( \mathcal{B} \) adds to \( \mathcal{R} \) rules of the form \( t \rightarrow \bot \) for each term \( t \) with \( t \xrightarrow{\mathcal{P}} \bot \).

Future work: Infinitary term graph rewriting

- Are orthogonal systems infinitarily confluent?
- higher-order systems (e.g. lambda calculus with letrec)
Publications


