# 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

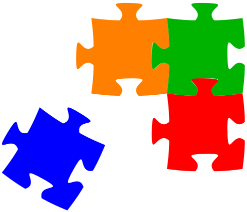
Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting

# The two Big Pictures

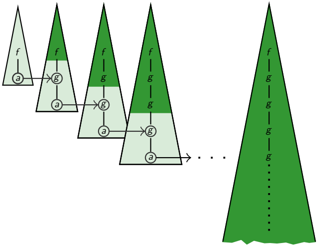
Modular Implementation of Programming Languages and a Partial Order Approach to Infinitary Rewriting

# The two Big Pictures

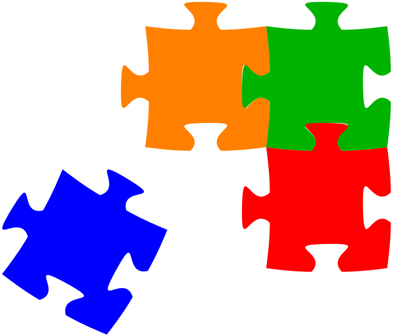
Modular Implementation of Programming Languages



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

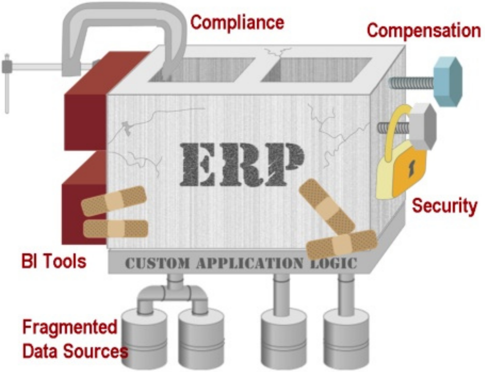
## Implementation of a DSL-Based ERP System

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?

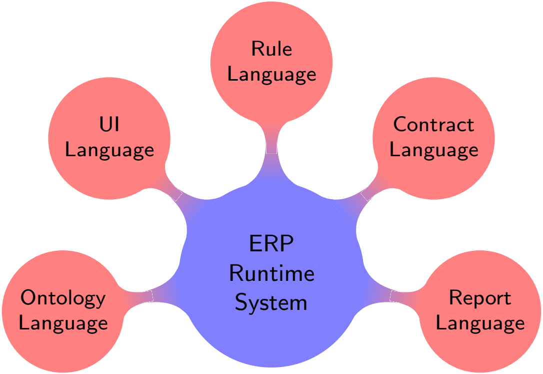


# An Alternative Approach

## POETS [Henglein et al. 2009]

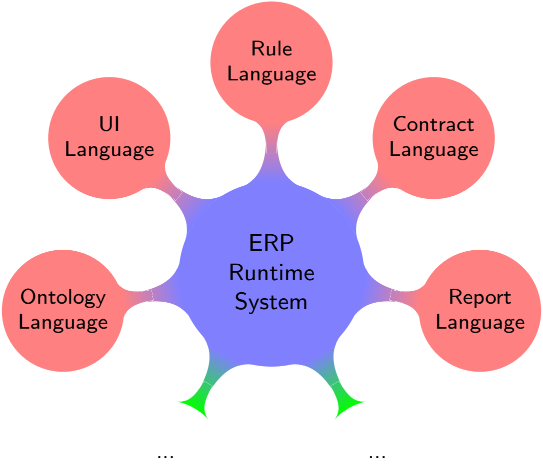
# An Alternative Approach

## POETS [Henglein et al. 2009]



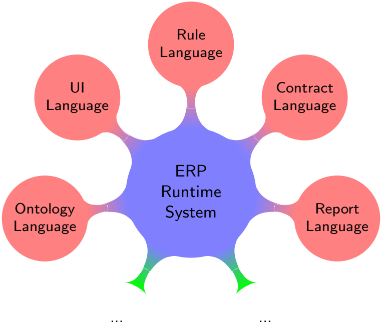
# An Alternative Approach

## POETS [Henglein et al. 2009]



# 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

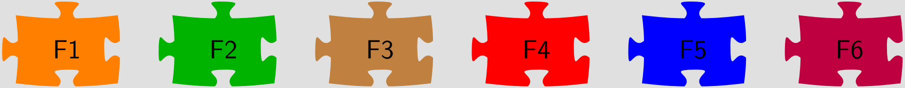
F4 contract clauses

F5 type definitions

F6 inference rules

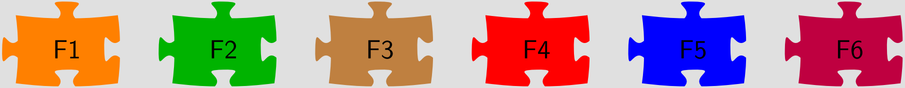
# Piecing Together DSLs – Syntax

## Library of language features



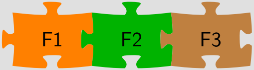
# Piecing Together DSLs – Syntax

## Library of language features



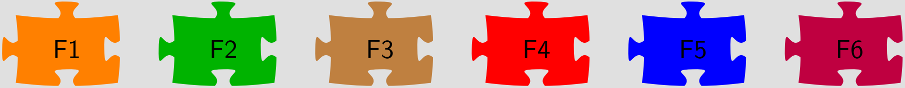

## Constructing the DSLs

Report Language =



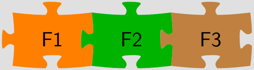
# Piecing Together DSLs – Syntax

## Library of language features

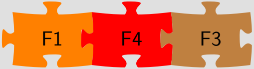



## Constructing the DSLs

Report Language =

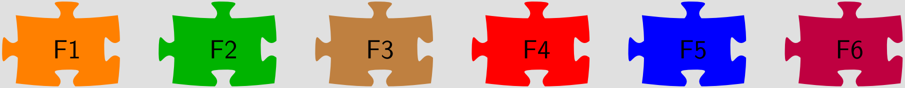



Contract Language =



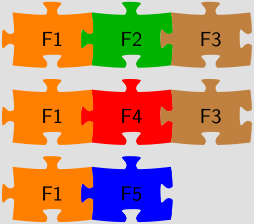
# Piecing Together DSLs – Syntax

## Library of language features



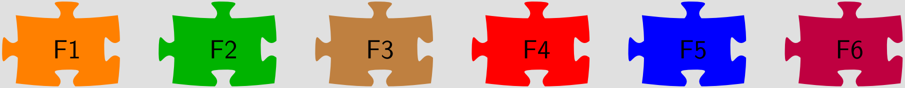

## Constructing the DSLs

Report Language =
Contract Language =
Ontology Language =



# Piecing Together DSLs – Syntax

## Library of language features



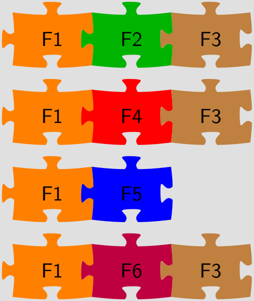

## Constructing the DSLs

Report Language =

Contract Language =

Ontology Language =

Rule Language =



# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $\mathit{Program}_L \longrightarrow \texttt{String}$ for each language $L$

# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $\mathit{Program}_L \longrightarrow \texttt{String}$ for each language $L$

**"functions" for each feature**

$pp_1 :$ F1 $\longrightarrow \texttt{String}$

$pp_2 :$ F2 $\longrightarrow \texttt{String}$

$pp_3 :$ F3 $\longrightarrow \texttt{String}$

$pp_4 :$ F4 $\longrightarrow \texttt{String}$

$pp_5 :$ F5 $\longrightarrow \texttt{String}$

$pp_6 :$ F6 $\longrightarrow \texttt{String}$

# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $Program_L \longrightarrow \texttt{String}$ for each language $L$

### "functions" for each feature

$pp_1$ : F1 $\longrightarrow$ String

$pp_2$ : F2 $\longrightarrow$ String

$pp_3$ : F3 $\longrightarrow$ String

$pp_4$ : F4 $\longrightarrow$ String

$pp_5$ : F5 $\longrightarrow$ String

$pp_6$ : F6 $\longrightarrow$ String

### Combine functions

$pp_1$
$+$
$pp_2$
$+$
$pp_3$

# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $Program_L \longrightarrow$ `String` for each language $L$

### "functions" for each feature

$pp_1$ : F1 $\longrightarrow$ `String`

$pp_2$ : F2 $\longrightarrow$ `String`

$pp_3$ : F3 $\longrightarrow$ `String`

$pp_4$ : F4 $\longrightarrow$ `String`

$pp_5$ : F5 $\longrightarrow$ `String`

$pp_6$ : F6 $\longrightarrow$ `String`

### Combine functions

$pp_1 + pp_2 + pp_3$ : F1 F2 F3 $\longrightarrow$ `String`

# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $Program_L \longrightarrow$ `String` e $L$

Report Language

### "functions" for each feature

$pp_1$ : F1 $\longrightarrow$ `String`

$pp_2$ : F2 $\longrightarrow$ `String`

$pp_3$ : F3 $\longrightarrow$ `String`

$pp_4$ : F4 $\longrightarrow$ `String`

$pp_5$ : F5 $\longrightarrow$ `String`

$pp_6$ : F6 $\longrightarrow$ `String`

### Combine functions

$pp_1 + pp_2 + pp_3$ : F1 F2 F3 $\longrightarrow$ `String`

# Piecing Together Functions

## Example: Pretty Printing

Goal: functions of type $\mathit{Program}_L \longrightarrow \texttt{String}$ for each language $L$

### "functions" for each feature

$pp_1$ : F1 $\longrightarrow$ `String`

$pp_2$ : F2 $\longrightarrow$ `String`

$pp_3$ : F3 $\longrightarrow$ `String`

$pp_4$ : F4 $\longrightarrow$ `String`

$pp_5$ : F5 $\longrightarrow$ `String`

$pp_6$ : F6 $\longrightarrow$ `String`

### Combine functions

$pp_1 + pp_2 + pp_3$ : F1 F2 F3 $\longrightarrow$ `String`

### Other combinations

$pp_1 + pp_5 + pp_6$ : F1 F5 F6 $\longrightarrow$ `String`

$\vdots$

# How does it work?

Based on: Wouter Swierstra. *Data types à la carte*

# How does it work?

$$\begin{array}{llll} \textbf{data}\ \mathit{Exp} = & \mathit{Lit} & \mathit{Int} \\ \mid & \mathit{Add} & \mathit{Exp}\ \mathit{Exp} \\ \mid & \mathit{Mult} & \mathit{Exp}\ \mathit{Exp} \end{array}$$

# How does it work?

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

recursion

decompose

signature

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

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

# How does it work?

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

recursion

signature

decompose

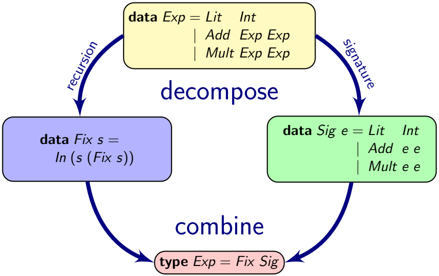
```
data Fix s =
  In (s (Fix s))
```

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

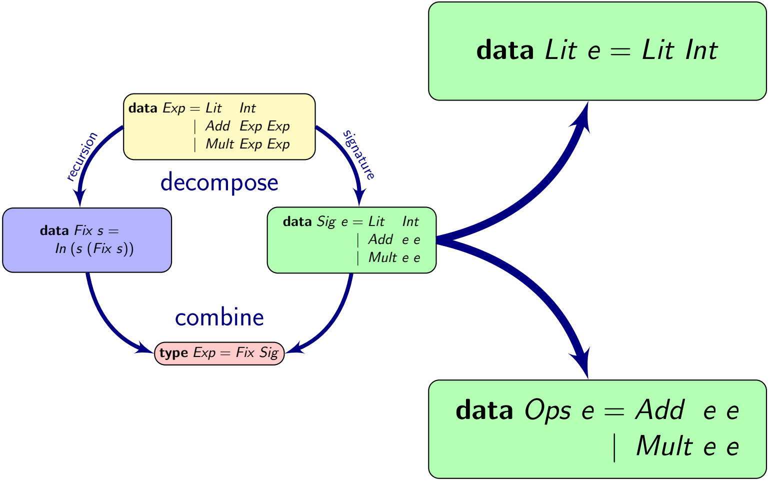
combine

```
type Exp = Fix Sig
```

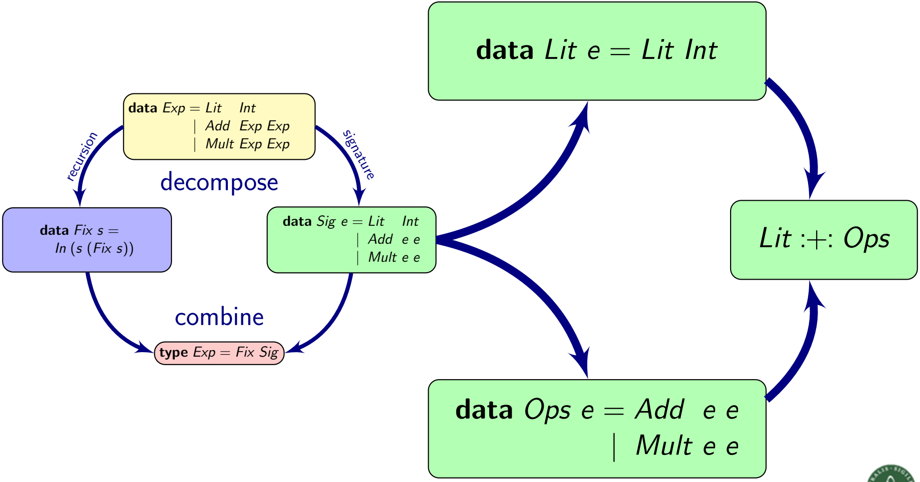
# How does it work?



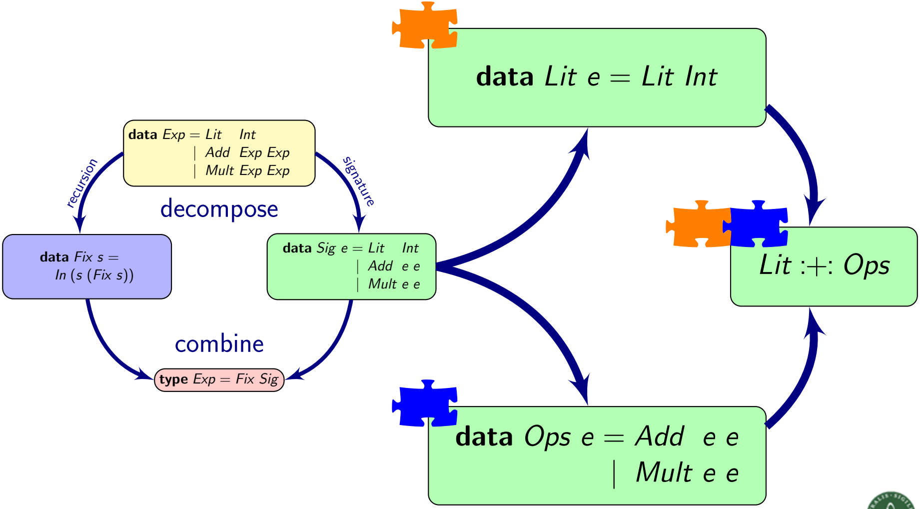
# How does it work?



# How does it work?



# How does it work?



# Combining Functions

## Explicit recursion

$$
\begin{aligned}
&pp :: Exp \rightarrow String \\
&pp\ (Lit\ i) = show\ i \\
&pp\ (Add\ e_1\ e_2) = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"} \\
&pp\ (Mult\ e_1\ e_2) = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"}
\end{aligned}
$$

# Combining Functions

## Explicit recursion

$$\begin{aligned}
&pp :: Exp \to String\\
&pp\ (Lit\ i) \quad\quad\ = show\ i\\
&pp\ (Add\ e_1\ e_2)\ = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"}\\
&pp\ (Mult\ e_1\ e_2) = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"}
\end{aligned}$$

## Non-recursive function

$$\begin{aligned}
&pp' :: Sig\ String \to String\\
&pp'\ (Lit\ i) \quad\quad\ = show\ i\\
&pp'\ (Add\ e_1\ e_2)\ = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ e_2 +\!\!+ \texttt{")"}\\
&pp'\ (Mult\ e_1\ e_2) = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ e_2 +\!\!+ \texttt{")"}
\end{aligned}$$

# Combining Functions

## Explicit recursion

$$
\begin{aligned}
&pp :: Exp \to String \\
&pp\ (Lit\ i) = show\ i \\
&pp\ (Add\ e_1\ e_2) = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"} \\
&pp\ (Mult\ e_1\ e_2) = \texttt{"("} +\!\!+\ pp\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ pp\ e_2 +\!\!+ \texttt{")"}
\end{aligned}
$$

## Non-recursive function

$$
\begin{aligned}
&pp_1 :: Lit\ String \to String \\
&pp_1\ (Lit\ i) = show\ i
\end{aligned}
$$

$$
\begin{aligned}
&pp_2 :: Ops\ String \to String \\
&pp_2\ (Add\ e_1\ e_2) = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ e_2 +\!\!+ \texttt{")"} \\
&pp_2\ (Mult\ e_1\ e_2) = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ e_2 +\!\!+ \texttt{")"}
\end{aligned}
$$

# Combining Functions

## Non-recursive function

$$pp_1 :: Lit\ String \rightarrow String$$
$$pp_1\ (Lit\ i) = show\ i$$

$$pp_2 :: Ops\ String \rightarrow String$$
$$pp_2\ (Add\ e_1\ e_2) = \texttt{"("} + e_1 + \texttt{" + "} + e_2 + \texttt{")"}$$
$$pp_2\ (Mult\ e_1\ e_2) = \texttt{"("} + e_1 + \texttt{" * "} + e_2 + \texttt{")"}$$

# Combining Functions

## Non-recursive function

$$pp_1 :: Lit\ String \rightarrow String$$
$$pp_1\ (Lit\ i) \qquad = show\ i$$

$$pp_2 :: Ops\ String \rightarrow String$$
$$pp_2\ (Add\ e_1\ e_2) \ = \texttt{"("} \mathbin{+\!\!+} e_1 \mathbin{+\!\!+} \texttt{" + "} \mathbin{+\!\!+} e_2 \mathbin{+\!\!+} \texttt{")"}$$
$$pp_2\ (Mult\ e_1\ e_2) = \texttt{"("} \mathbin{+\!\!+} e_1 \mathbin{+\!\!+} \texttt{" * "} \mathbin{+\!\!+} e_2 \mathbin{+\!\!+} \texttt{")"}$$

## Fold

$$fold :: Functor\ f \Rightarrow (f\ a \rightarrow a) \rightarrow Fix\ f \rightarrow a$$
$$fold\ f\ (In\ t) = f\ (fmap\ (fold\ f)\ t)$$

# Combining Functions

## Non-recursive function

$$pp_1 :: \mathit{Lit}\ \mathit{String} \to \mathit{String}$$
$$pp_1\ (\mathit{Lit}\ i) = \mathit{show}\ i$$

$$pp_2 :: \mathit{Ops}\ \mathit{String} \to \mathit{String}$$
$$pp_2\ (\mathit{Add}\ e_1\ e_2) = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" + "} +\!\!+\ e_2 +\!\!+ \texttt{")"}$$
$$pp_2\ (\mathit{Mult}\ e_1\ e_2) = \texttt{"("} +\!\!+\ e_1 +\!\!+ \texttt{" * "} +\!\!+\ e_2 +\!\!+ \texttt{")"}$$

## Fold

$$\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to \mathit{Fix}\ f \to a$$
$$\mathit{fold}\ f\ (\mathit{In}\ t) = f\ (\mathit{fmap}\ (\mathit{fold}\ f)\ t)$$

## Applying Fold

$$pp :: \mathit{Fix}\ (\mathit{Lit} :\!+\!: \mathit{Ops}) \to \mathit{String}$$
$$pp = \mathit{fold}\ (pp_1 :\!+\!: pp_2)$$

# Our Contributions

# 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

$[\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R$

$[\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R$

# Compositionality

We may compose tree automata along 3 different dimensions.

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

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{S}_1 + \mathcal{S}_2) \to R$$

# Compositionality

We may compose tree automata along 3 different dimensions.

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

$$[\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R \qquad [\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R \quad \Longrightarrow \quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{S}_1 + \mathcal{S}_2) \to R$$

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

$$\mu\mathcal{S}_1 \xrightarrow{[\![\mathcal{A}_1]\!]} \mu\mathcal{S}_2 \xrightarrow{[\![\mathcal{A}_2]\!]} \mu\mathcal{S}_3$$

# Compositionality

We may compose tree automata along 3 different dimensions.

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

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R \end{array} \quad\Longrightarrow\quad [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{S}_1 + \mathcal{S}_2) \to R$$

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

$$\begin{tikzcd} \mu\mathcal{S}_1 \arrow[r, "{[\![\mathcal{A}_1]\!]}"] \arrow[rr, bend right, "{[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]}"'] & \mu\mathcal{S}_2 \arrow[r, "{[\![\mathcal{A}_2]\!]}"] & \mu\mathcal{S}_3 \end{tikzcd}$$

# Compositionality

We may compose tree automata along 3 different dimensions.

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

$$\begin{array}{l}[\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R\end{array} \implies [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{S}_1 + \mathcal{S}_2) \to R$$

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

$$\begin{tikzcd} \mu\mathcal{S}_1 \arrow[r, "{[\![\mathcal{A}_1]\!]}"] \arrow[rr, bend right, "{[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]}"'] & \mu\mathcal{S}_2 \arrow[r, "{[\![\mathcal{A}_2]\!]}"] & \mu\mathcal{S}_3 \end{tikzcd}$$

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

$$[\![\mathcal{A}_1]\!] : \mu\mathcal{S} \to R_1$$

$$[\![\mathcal{A}_2]\!] : \mu\mathcal{S} \to R_2$$

# Compositionality

We may compose tree automata along 3 different dimensions.

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

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{S}_1 \to R \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S}_2 \to R \end{array} \implies [\![\mathcal{A}_1 + \mathcal{A}_2]\!] : \mu(\mathcal{S}_1 + \mathcal{S}_2) \to R$$

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

$$\begin{tikzcd} \mu\mathcal{S}_1 \arrow[r, "{[\![\mathcal{A}_1]\!]}"] \arrow[rr, bend right, "{[\![\mathcal{A}_1 \circ \mathcal{A}_2]\!]}"'] & \mu\mathcal{S}_2 \arrow[r, "{[\![\mathcal{A}_2]\!]}"] & \mu\mathcal{S}_3 \end{tikzcd}$$

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

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{S} \to R_1 \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S} \to R_2 \end{array} \implies [\![\mathcal{A}_1 \times \mathcal{A}_2]\!] : \mu\mathcal{F} \to R_1 \times R_2$$

# Contextuality

**tupling / product automaton construction**

$$\begin{array}{l} [\![\mathcal{A}_1]\!] : \mu\mathcal{S} \to R_1 \\ [\![\mathcal{A}_2]\!] : \mu\mathcal{S} \to R_2 \end{array} \quad \Longrightarrow \quad [\![\mathcal{A}_1 \times \mathcal{A}_2]\!] : \mu(\mathcal{S}) \to R_1 \times R_2$$

# Contextuality

**tupling / product automaton construction**

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: \mathcal{S} \to R_2 \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{S} \to R_1 \times R_2$$

# Contextuality

**tupling / product automaton construction**

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: \mathcal{S} \to R_2 \end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{S} \to R_1 \times R_2$$

**mutumorphisms / dependent product automata**

$$\begin{aligned} \mathcal{A}_1 &: \phantom{R_1 \Rightarrow {}} \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: R_1 \Rightarrow \mathcal{S} \to R_2 \end{aligned}$$

# Contextuality

**tupling / product automaton construction**

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: \mathcal{S} \to R_2 \end{aligned} \implies \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{S} \to R_1 \times R_2$$

**mutumorphisms / dependent product automata**

$$\begin{aligned} \mathcal{A}_1 &: \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: R_1 \Rightarrow \mathcal{S} \to R_2 \end{aligned} \implies \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{S} \to R_1 \times R_2$$

# Contextuality

**tupling / product automaton construction**

$$\begin{array}{lcl} \mathcal{A}_1 & : & \mathcal{S} \to R_1 \\ \mathcal{A}_2 & : & \mathcal{S} \to R_2 \end{array} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 \; : \; \mathcal{S} \to R_1 \times R_2$$

**mutumorphisms / dependent product automata**

$$\begin{array}{l} \mathcal{A}_1 : R_2 \Rightarrow \mathcal{S} \to R_1 \\ \mathcal{A}_2 : R_1 \Rightarrow \mathcal{S} \to R_2 \end{array} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 \; : \; \mathcal{S} \to R_1 \times R_2$$

# Contextuality

**tupling / product automaton construction**

$$\begin{aligned}\mathcal{A}_1 &: \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: \mathcal{S} \to R_2\end{aligned} \quad \Longrightarrow \quad \mathcal{A}_1 \times \mathcal{A}_2 : \mathcal{S} \to R_1 \times R_2$$

**mutumorphisms / dependent product automata**

$$\begin{aligned}\mathcal{A}_1 &: R_2 \Rightarrow \mathcal{S} \to R_1 \\ \mathcal{A}_2 &: R_1 \Rightarrow \mathcal{S} \to R_2\end{aligned} \quad \Longrightarrow \quad [\![\mathcal{A}_1 \times \mathcal{A}_2]\!] : \mu\mathcal{S} \to R_1 \times R_2$$

# Discussion

## Advantages

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

# 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

# Discussion

## Advantages

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

## Drawbacks

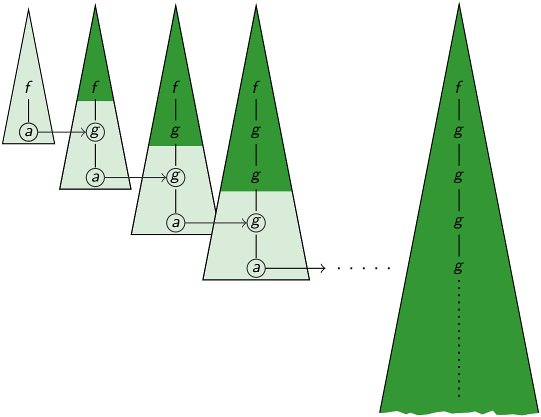
- it's just a Haskell library
- error messages are sometimes rather cryptic
- learning curve
- typical drawbacks of higher-order abstract syntax

## 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 $l \to r$
- semantics: any instance of a left-hand side may be replaced by the corresponding instance of the right-hand side

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$s(s(0)) * s(s(0))$

# 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)

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

$s(s(0)) * s(s(0))$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \;\to\; s(s(0)) + (s(s(0)) * s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to s(s(0)) + (s(s(0)) * s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^2 s(s(0)) + (s(s(0)) + (s(s(0)) * 0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^2 s(s(0)) + (s(s(0)) + (s(s(0)) * 0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^3 s(s(0)) + (s(s(0)) + 0)$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^3 s(s(0)) + (s(s(0)) + 0)$$

# Rewriting Systems

**What are (term) rewriting systems?**

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)**

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

$$s(s(0)) * s(s(0)) \to^4 s(s(0)) + s(s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^4 s(s(0)) + s(s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^5 s(s(s(0)) + s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^5 s(s(s(0)) + s(0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^6 s(s(s(s(0)) + 0))$$

# Rewriting Systems

**What are (term) rewriting systems?**

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)**

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

$$s(s(0)) * s(s(0)) \to^6 s(s(s(s(0)) + 0))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^7 s(s(s(s(0))))$$

# Rewriting Systems

## What are (term) rewriting systems?

- generalisation of (first-order) functional programs
- consist of directed symbolic equations of the form $l \to r$
- 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)

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

$$s(s(0)) * s(s(0)) \to^7 s(s(s(s(0))))$$

$\mathcal{R}_{+*}$ is terminating!

# Non-Terminating Rewriting Systems

Termination: repeated rewriting eventually reaches a normal form.

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$\mathit{from}(0)$

# 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)**

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \;\to\; 0 : \mathit{from}(1)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^2 0 : 1 : \mathit{from}(2)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^3 0 : 1 : 2 : \mathit{from}(3)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \;\to^4 0 : 1 : 2 : 3 : \mathit{from}(4)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^5 0 : 1 : 2 : 3 : 4 : \mathit{from}(5)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^6 0 : 1 : 2 : 3 : 4 : 5 : \mathit{from}(6)$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^{6} 0 : 1 : 2 : 3 : 4 : 5 : \mathit{from}(6) \to \ \ldots$$

# 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)

$$\mathcal{R}_{nats} = \left\{ \quad \mathit{from}(x) \to x : \mathit{from}(s(x)) \right.$$

$$\mathit{from}(0) \to^6 0 : 1 : 2 : 3 : 4 : 5 : \mathit{from}(6) \to \ldots$$

intuitively this converges to the infinite list $0 : 1 : 2 : 3 : 4 : 5 : \ldots$.

# Infinitary Term Rewriting – The Metric Approach

**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 approximates a uniquely determined term $t$ arbitrary 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.

$$t_0 \quad \to \quad t_1 \quad \to \quad t_2 \quad \to \quad \dots \quad t$$

# 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.

$$t_0 \quad \to \quad t_1 \quad \to \quad t_2 \quad \to \quad \dots \quad t$$

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

# 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?

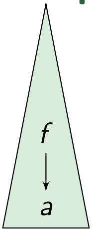
A converging rewrite sequence approximates a uniquely determined term $t$ arbitrary well.

$$t_0 \quad \to \quad t_1 \quad \to \quad t_2 \quad \to \quad \dots \quad t$$

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

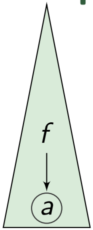
$$t_0 \quad \to \quad t_1 \quad \to \quad \dots \quad \to \quad \underbrace{t_n \quad \to \quad t_{n+1} \quad \to \quad \dots \quad t}_{\text{do not differ up to depth } d}$$

# Example: Convergence of a Reduction



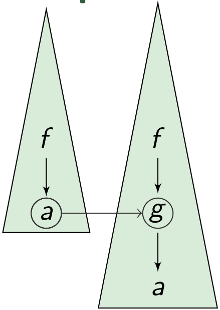

$\mathcal{R} = \{a \to g(a)\}$

# Example: Convergence of a Reduction



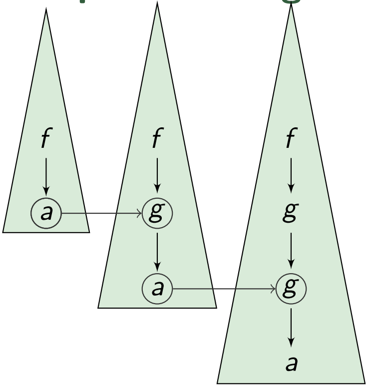

$\mathcal{R} = \{a \to g(a)\}$

# Example: Convergence of a Reduction



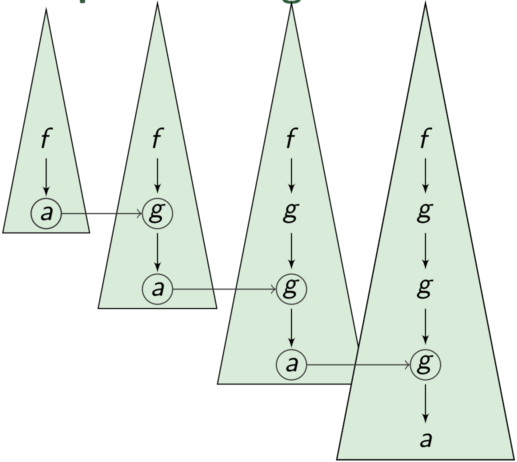

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Convergence of a Reduction



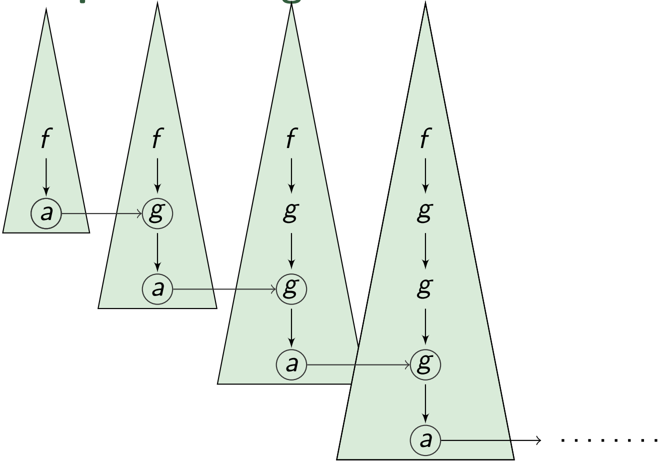

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Convergence of a Reduction



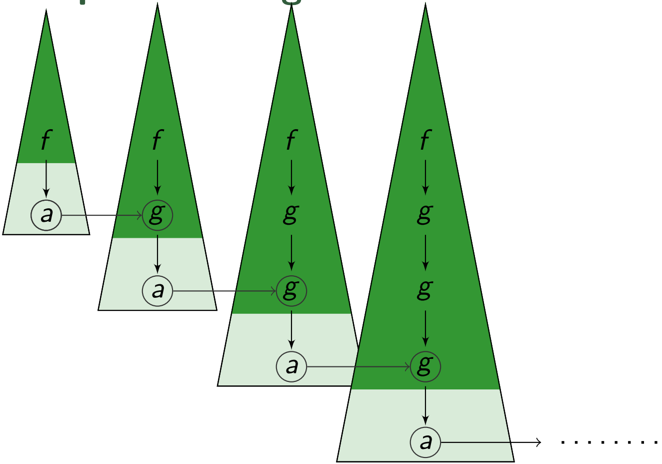

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Convergence of a Reduction



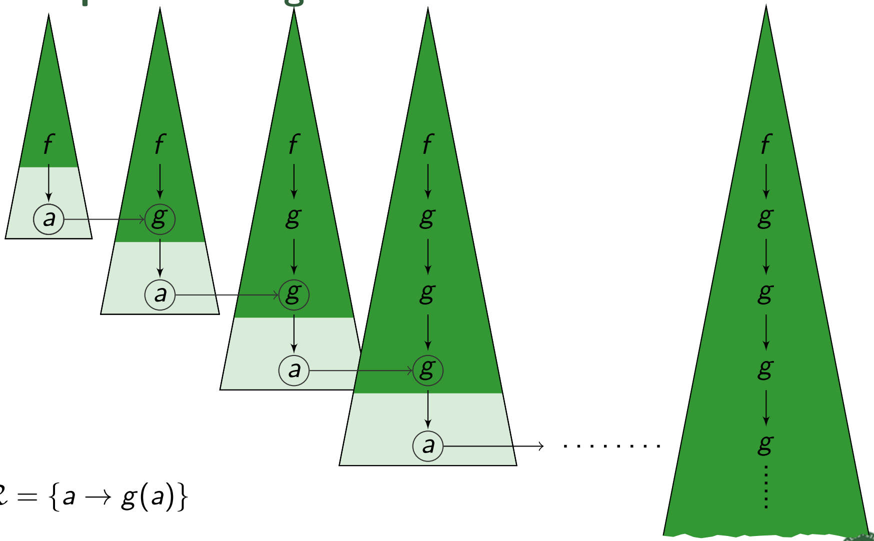

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Convergence of a Reduction



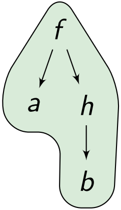

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Convergence of a Reduction



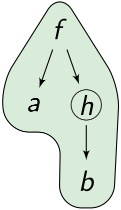

$$\mathcal{R} = \{a \to g(a)\}$$

# Example: Non-Convergence of a Reduction



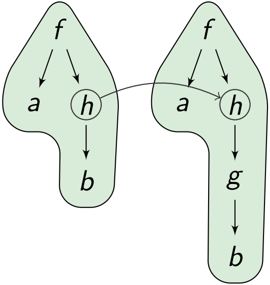

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



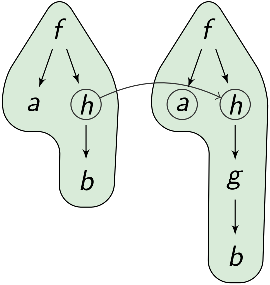

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



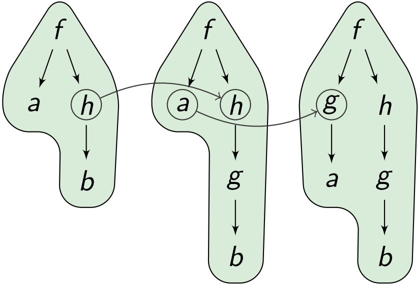

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



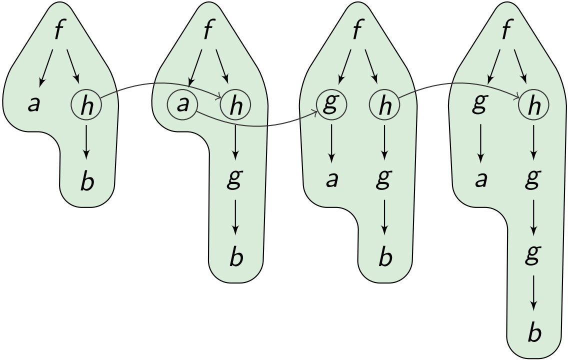

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



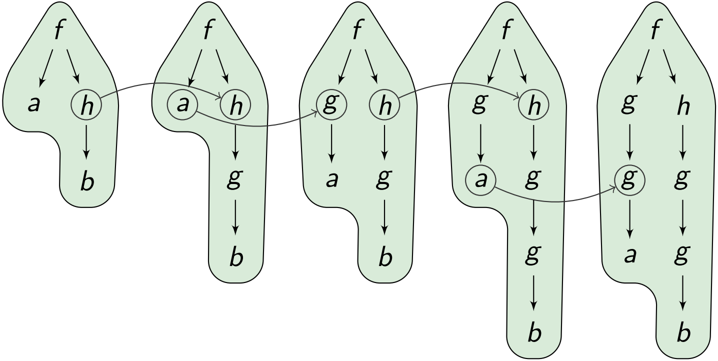

$$\mathcal{R} = \begin{cases} a \rightarrow g(a) \\ h(x) \rightarrow h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



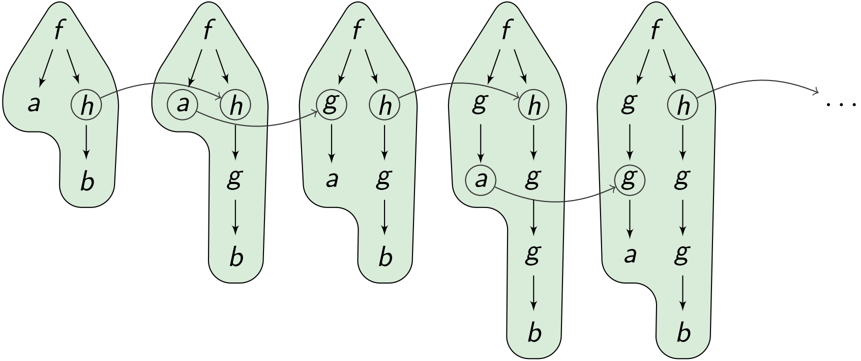

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction



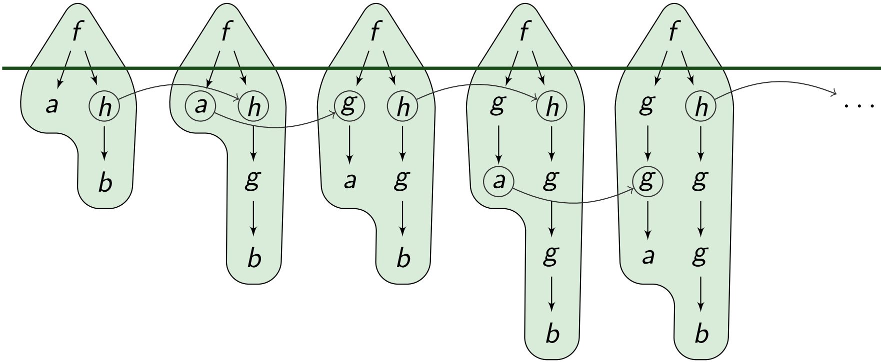

$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction




$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# Example: Non-Convergence of a Reduction




$$\mathcal{R} = \begin{cases} a \to g(a) \\ h(x) \to h(g(x)) \end{cases}$$

# 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

$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ t_1 & & t_2 \end{tikzcd}$$

For every $t, t_1, t_2 \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ with $t_1 \twoheadleftarrow t \twoheadrightarrow t_2$

# 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

$$\begin{tikzcd}
 & t \arrow[dl, two heads] \arrow[dr, two heads] & \\
t_1 \arrow[dr, dashed, two heads] & & t_2 \arrow[dl, dashed, two heads] \\
 & t' &
\end{tikzcd}$$

For every $t, t_1, t_2 \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$
with $t_1 \twoheadleftarrow t \twoheadrightarrow t_2$
there is a $t' \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$
with $t_1 \twoheadrightarrow t' \twoheadleftarrow t_2$

# Partial Order Approach to Infinitary Term Rewriting

## Partial order on terms

- partial terms: terms with additional constant $\perp$ (read as "undefined")
- partial order $\leq_\perp$ reads as: "is less defined than"
- $\leq_\perp$ is a complete semilattice ($=$ bounded complete cpo)

# 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:

$$\liminf_{\iota \to \alpha} t_\iota = \bigsqcup_{\beta < \alpha} \bigsqcap_{\beta \leq \iota < \alpha} t_\iota$$

# Partial Order Approach to Infinitary Term Rewriting

## Partial order on terms

- partial terms: terms with additional constant $\perp$ (read as "undefined")
- partial order $\leq_\perp$ reads as: "is less defined than"
- $\leq_\perp$ is a complete semilattice ($=$ bounded complete cpo)

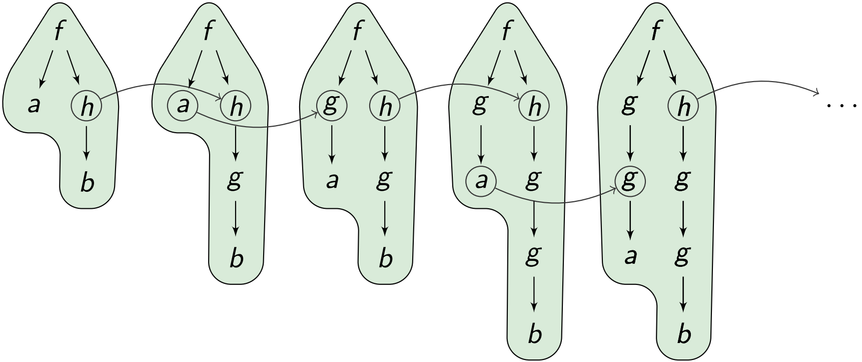
## Convergence

- formalised by the limit inferior:

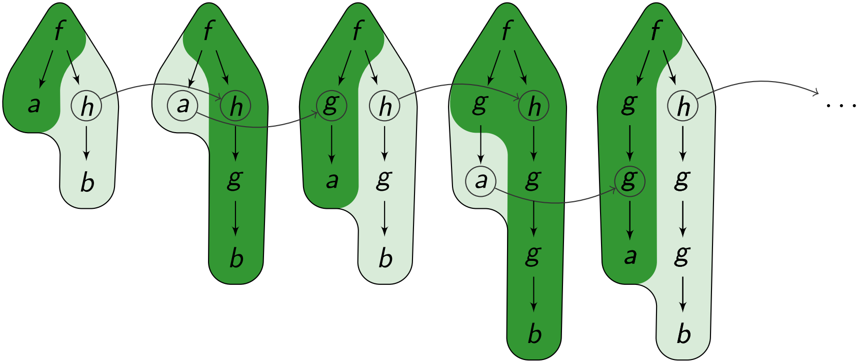
$$\liminf_{\iota \to \alpha} t_\iota = \bigsqcup_{\beta < \alpha} \bigsqcap_{\beta \leq \iota < \alpha} t_\iota$$

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

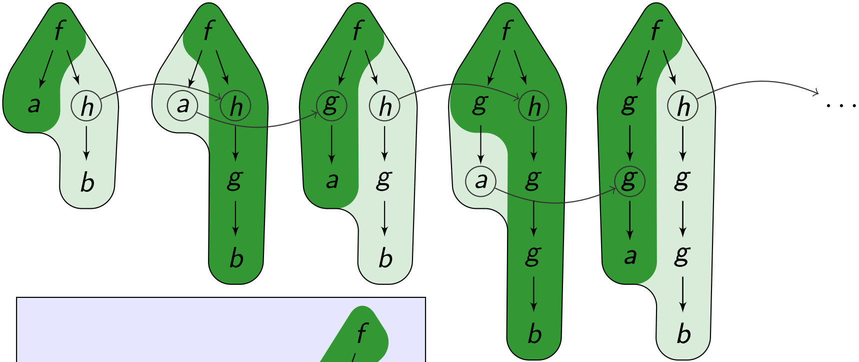
# An Example



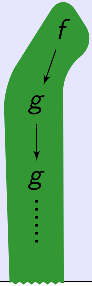
# An Example



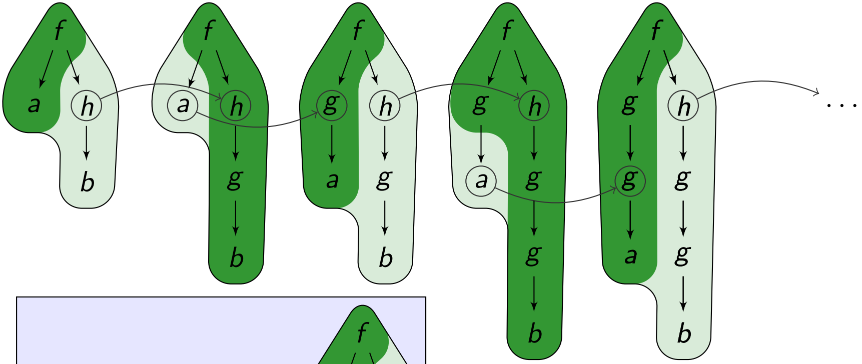
# An Example

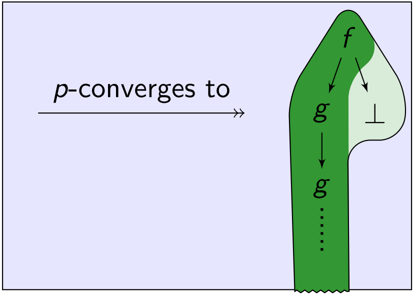


eventually stable:



# An Example





# 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

# 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*

$$S\colon\ s \xrightarrow{p}\!\!\!\!\twoheadrightarrow t \text{ is total} \iff S\colon\ s \xrightarrow{m}\!\!\!\!\twoheadrightarrow t.$$

# Properties of the Partial Order Approach

## Benefits

- reduction sequences always converge (but result may contain $\bot$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*

$$S\colon\ s \xrightarrow{p}\!\!\!\!\!\rightarrow t \text{ is total} \iff S\colon\ s \xrightarrow{m}\!\!\!\!\!\rightarrow t.$$

## 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.*

# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

Skip term graphs

Lazy evaluation consists of two things:

- non-strict evaluation
- sharing

# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

Skip term graphs

Lazy evaluation consists of two things:

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

# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

Skip term graphs

Lazy evaluation consists of two things:

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

## Example

$$\mathit{from}(x) \to x : \mathit{from}(s(x))$$

# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

Skip term graphs

Lazy evaluation consists of two things:

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

## Example

$$\mathit{from}(x) \to x : \mathit{from}(s(x))$$

# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

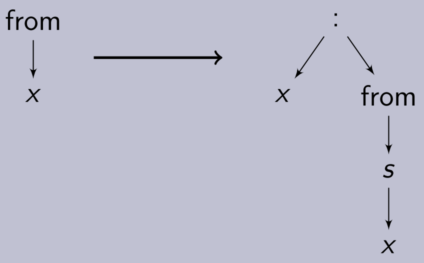
▸ Skip term graphs

Lazy evaluation consists of two things:

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

## Example

$$\mathit{from}(x) \to x : \mathit{from}(s(x))$$



# Sharing – From Terms to Term Graphs

## Lazy evaluation and infinitary rewriting

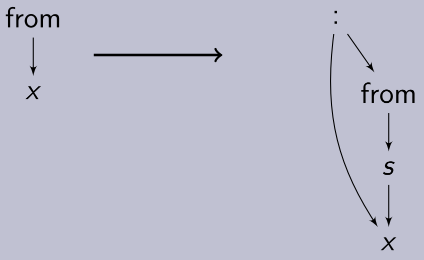
Skip term graphs

Lazy evaluation consists of two things:

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

## Example

$$\mathit{from}(x) \to x : \mathit{from}(s(x))$$



# Example

$$\text{from} \downarrow 0$$

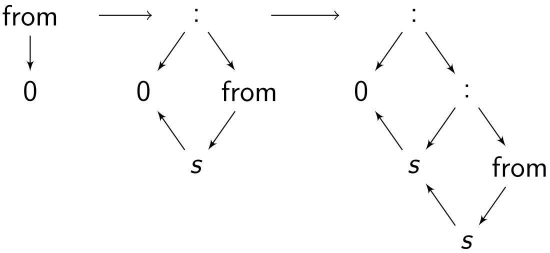
# Example

$$\begin{tikzcd}
\mathsf{from} \arrow[d] & \longrightarrow & & : \arrow[dl] \arrow[dr] & \\
0 & & 0 & & \mathsf{from} \arrow[d] \\
 & & & & s \arrow[d] \\
 & & & & 0
\end{tikzcd}$$

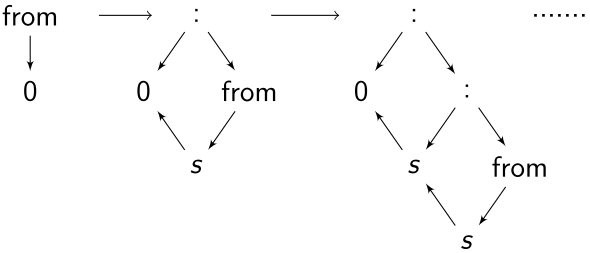
# Example

$$
\begin{tikzcd}
\mathrm{from} \arrow[d] \arrow[rr, "\longrightarrow" description, phantom] & & : \arrow[dl] \arrow[dr] & \\
0 & 0 & & \mathrm{from} \arrow[dl] \\
& & s \arrow[ul] &
\end{tikzcd}
$$

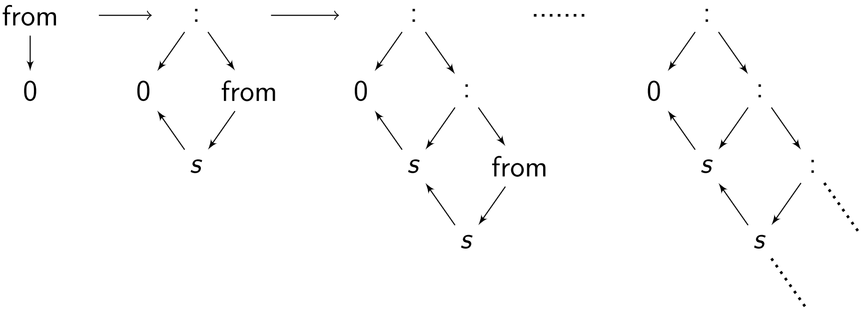
# Example



# Example



# Example



# Properties of Infinitary Term Graph Rewriting

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

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

$$S\colon\ g \overset{p}{\twoheadrightarrow} h \text{ is total} \quad\iff\quad S\colon\ g \overset{m}{\twoheadrightarrow} h.$$

# Properties of Infinitary Term Graph Rewriting

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

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

$$S\colon\ g \overset{p}{\twoheadrightarrow} h \text{ is total} \quad\Longleftrightarrow\quad S\colon\ g \overset{m}{\twoheadrightarrow} h.$$

**Theorem (soundness)**

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

$$\underline{\mathcal{R}} \qquad g \xrightarrow{\qquad\qquad p \qquad\qquad}\!\!\!\!\twoheadrightarrow h$$

# Properties of Infinitary Term Graph Rewriting

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

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

$$S\colon\ g \xrightarrow{p}\!\!\!\!\!\twoheadrightarrow h \text{ is total} \quad \Longleftrightarrow \quad S\colon\ g \xrightarrow{m}\!\!\!\!\!\twoheadrightarrow h.$$

**Theorem (soundness)**

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

$$\begin{tikzcd}
\underline{\mathcal{R}} & g \arrow[rr, twoheadrightarrow, "p"] \arrow[d, "\mathcal{U}(\cdot)"'] & & h \arrow[d, dashed, "\mathcal{U}(\cdot)"] \\
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, dashed, twoheadrightarrow, "p"] & & t
\end{tikzcd}$$

# Properties of Infinitary Term Graph Rewriting

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

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

$$S\colon\ g \xrightarrow{p}\!\!\!\!\rightarrow h \text{ is total} \quad\Longleftrightarrow\quad S\colon\ g \xrightarrow{m}\!\!\!\!\rightarrow h.$$

**Theorem (soundness)**

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

$$\begin{tikzcd}
\underline{\mathcal{R}} & g \arrow[rr, twoheadrightarrow, "m"] \arrow[d, "\mathcal{U}(\cdot)"'] & & h \arrow[d, dashed, "\mathcal{U}(\cdot)"] \\
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, dashed, twoheadrightarrow, "m"] & & t
\end{tikzcd}$$

# Completeness

## Theorem (Completeness)

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

$$
\begin{array}{ccc}
\underline{\mathcal{U}(\mathcal{R})} & s & \xrightarrow{\quad p \quad} \!\!\!\!\twoheadrightarrow & t \\
 & {\color{blue}\mathcal{U}(\cdot)} \big\uparrow & & \\
 & g & &
\end{array}
$$

# Completeness

**Theorem (Completeness)**

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

$$\begin{tikzcd}
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, twoheadrightarrow, "p"] & & t \arrow[r, dashed, twoheadrightarrow, "p"] & t' \\
\underline{\mathcal{R}} & g \arrow[u, blue, "\mathcal{U}(\cdot)"] \arrow[rrr, dashed, twoheadrightarrow, "p"] & & & h \arrow[u, dashed, blue, "\mathcal{U}(\cdot)"']
\end{tikzcd}$$

# Completeness

## Theorem (Completeness)

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

$$
\begin{tikzcd}
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, "p", ->>] & & t \arrow[r, dashed, "p", ->>] & t' \\
\underline{\mathcal{R}} & g \arrow[u, "\mathcal{U}(\cdot)"] \arrow[rrr, dashed, "p", ->>] & & & h \arrow[u, dashed, "\mathcal{U}(\cdot)"']
\end{tikzcd}
$$

Does not hold for metric convergence!

# Completeness

## Theorem (Completeness)

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

$$\begin{tikzcd}
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, twoheadrightarrow, "p"] & & t \arrow[rr, dashed, twoheadrightarrow, "p"] & & t' \\
\underline{\mathcal{R}} & g \arrow[u, "\mathcal{U}(\cdot)"] \arrow[rrrr, dashed, twoheadrightarrow, "p"] & & & & h \arrow[u, dashed, "\mathcal{U}(\cdot)"']
\end{tikzcd}$$

Does not hold for metric convergence!

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

$$\begin{tikzcd}
\underline{\mathcal{U}(\mathcal{R})} & s \arrow[rr, twoheadrightarrow, "m"] & & t \in \mathrm{NF} \\
\underline{\mathcal{R}} & g \arrow[u, "\mathcal{U}(\cdot)"] \arrow[rr, dashed, twoheadrightarrow, "m"] & & h \arrow[u, dashed, "\mathcal{U}(\cdot)"']
\end{tikzcd}$$

# 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

$$s \xrightarrow{p}\!\!\!\!\!\rightarrow_{\mathcal{R}} t \qquad \Longleftrightarrow \qquad s \xrightarrow{m}\!\!\!\!\!\rightarrow_{\mathcal{B}} t$$

$\mathcal{B}$ adds to $\mathcal{R}$ rules of the form $t \to \bot$ 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

$$s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t \qquad \Longleftrightarrow \qquad s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$$

$\mathcal{B}$ adds to $\mathcal{R}$ rules of the form $t \to \bot$ for each term $t$ with $t \overset{p}{\twoheadrightarrow} \bot$.

# Discussion

## Contributions

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

## Note: Böhm reduction for TRSs

$$s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t \qquad \Longleftrightarrow \qquad s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$$

$\mathcal{B}$ adds to $\mathcal{R}$ rules of the form $t \to \bot$ for each term $t$ with $t \overset{p}{\twoheadrightarrow} \bot$.

## Future work: Infinitary term graph rewriting

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

# Publications


[1] Patrick Bahr. *Modes of Convergence for Term Graph Rewriting*. Logical Methods in Computer Science 8(2), pp. 1-60, 2012.

[2] Patrick Bahr. *Modular Tree Automata*. Mathematics of Program Construction, pp. 263-299, 2012.

[3] Patrick Bahr. *Infinitary Term Graph Rewriting is Simple, Sound and Complete*. 23rd International Conference on Rewriting Techniques and Applications (RTA'12) , pp. 69-84, 2012.

[4] Patrick Bahr. *Modes of Convergence for Term Graph Rewriting*. 22nd International Conference on Rewriting Techniques and Applications (RTA'11), pp. 139-154, 2011.

[5] Patrick Bahr. *Partial Order Infinitary Term Rewriting and Böhm Trees*. Proceedings of the 21st International Conference on Rewriting Techniques and Applications, pp. 67-84, 2010.

[6] Patrick Bahr. *Abstract Models of Transfinite Reductions*. Proceedings of the 21st International Conference on Rewriting Techniques and Applications, pp. 49-66, 2010.

[7] Patrick Bahr, Tom Hvitved. *Parametric Compositional Data Types*. Proceedings Fourth Workshop on Mathematically Structured Functional Programming, pp. 3-24, 2012.

[8] Patrick Bahr, Tom Hvitved. *Compositional data types*. Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, pp. 83-94, 2011.

[9] Patrick Bahr. *Evaluation à la Carte: Non-Strict Evaluation via Compositional Data Types*. Proceedings of the 23rd Nordic Workshop on Programming Theory, pp. 38-40, 2011.

[10] Patrick Bahr. *A Functional Language for Specifying Business Reports*. Proceedings of the 23rd Nordic Workshop on Programming Theory, pp. 24-26, 2011.

[11] Patrick Bahr. *Convergence in Infinitary Term Graph Rewriting Systems is Simple*. Submitted to Math. Structures Comput. Sci.

[12] Patrick Bahr. *Partial Order Infinitary Term Rewriting and Böhm Trees*. Submitted to Log. Methods Comput. Sci.