universität
innsbruck

Patterns

Vincent van Oostrom & Luigi Liquori

http://cl-informatik.uibk.ac.at

1. Motivation

2. Design space/decisions

3. Some λ-calculi with patterns

4. Patterns in Haskell

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

$$(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$$

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

  $$(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$$

- term rewriting: pattern matching rules $\ell \to r$ (pattern ℓ) such as $0 + x \to x$

  $$0 + 3 \to 3$$

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

  $(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$
- term rewriting: pattern matching rules $\ell \to r$ (pattern ℓ) such as $0 + x \to x$

  $0 + 3 \to 3$
- λ-calculi with patterns combine parameter passing with pattern matching

  $(\lambda[x, y].x)\, [1, 2, 3, 4, 5] \to 1 \quad \text{and} \quad (\lambda[x, y].y)\, [1, 2, 3, 4, 5] \to [2, 3, 4, 5]$

  Haskell-like list $[1, 2, 3, 4, 5]$ abbreviates $[1, [2, [3, [4, [5]]]]]$; repeated cons

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

  $$(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$$

- term rewriting: pattern matching rules $\ell \to r$ (pattern ℓ) such as $0 + x \to x$

  $$0 + 3 \to 3$$

- λ-calculi with patterns combine parameter passing with pattern matching

  $$(\lambda[x, y].x)\,[1, 2, 3, 4, 5] \to 1 \quad \text{and} \quad (\lambda[x, y].y)\,[1, 2, 3, 4, 5] \to [2, 3, 4, 5]$$

  Haskell-like list $[1, 2, 3, 4, 5]$ abbreviates $[1, [2, [3, [4, [5]]]]]$; repeated cons convenient for writing projection functions like head and tail

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

  $$(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$$

- term rewriting: pattern matching rules $\ell \to r$ (pattern ℓ) such as $0 + x \to x$

  $$0 + 3 \to 3$$

- λ-calculi with patterns combine parameter passing with pattern matching

  $$(\lambda[x, y].x)\,[1, 2, 3, 4, 5] \to 1 \quad \text{and} \quad (\lambda[x, y].y)\,[1, 2, 3, 4, 5] \to [2, 3, 4, 5]$$

  Haskell-like list $[1, 2, 3, 4, 5]$ abbreviates $[1, [2, [3, [4, [5]]]]]$; repeated cons convenient for writing projection functions like head and tail

in math common practice to write $(x, y) \mapsto x + y$ for addition function on pairs
a pair (x, y) is already a rudimentary form of a pattern

Motivation

Conceptual

- λ-calculus: parameter passing rule $\beta : (\lambda x.M)N \to M[x := N]$

  $$(\lambda x\, y.x + y)\, 0\, 3 \to (\lambda y.0 + y)\, 3 \to 0 + 3$$

- term rewriting: pattern matching rules $\ell \to r$ (pattern ℓ) such as $0 + x \to x$

  $$0 + 3 \to 3$$

- λ-calculi with patterns combine parameter passing with pattern matching

  $$(\lambda[x, y].x)\,[1, 2, 3, 4, 5] \to 1 \quad \text{and} \quad (\lambda[x, y].y)\,[1, 2, 3, 4, 5] \to [2, 3, 4, 5]$$

  Haskell-like list $[1, 2, 3, 4, 5]$ abbreviates $[1, [2, [3, [4, [5]]]]]$; repeated cons convenient for writing projection functions like head and tail

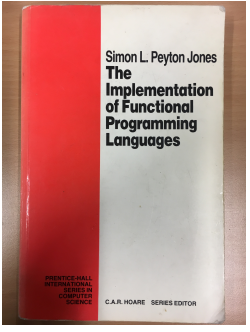
This lecture

Trying to make sense of this idea, of λ-calculi with patterns

Motivation

Personal (1990)

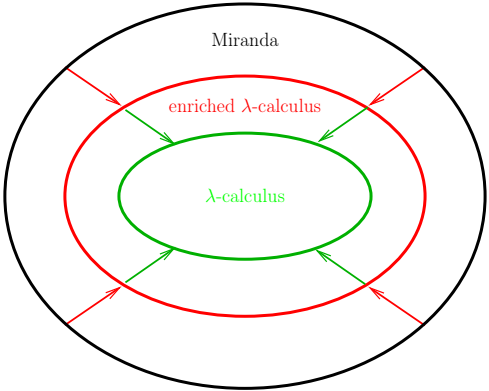
Klop (PhD supervisor) to Vincent (PhD student): the enriched λ-calculus in the recent book [SPJ] sounds interesting, could you study its rewrite properties?



How to implement functional programming languages?

Idea of [SPJ]: transformational

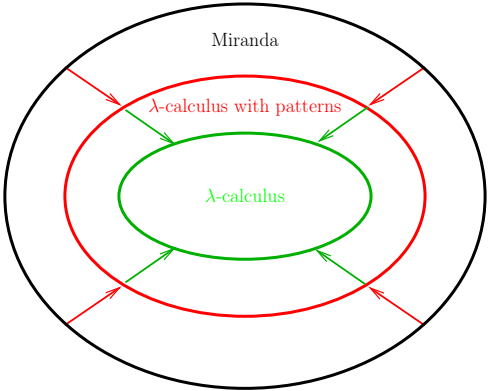
FPL $\Longrightarrow$ enriched λ-calculus (Chapter 4) $\Longrightarrow$ λ-calculus (with []; Chapter 6)



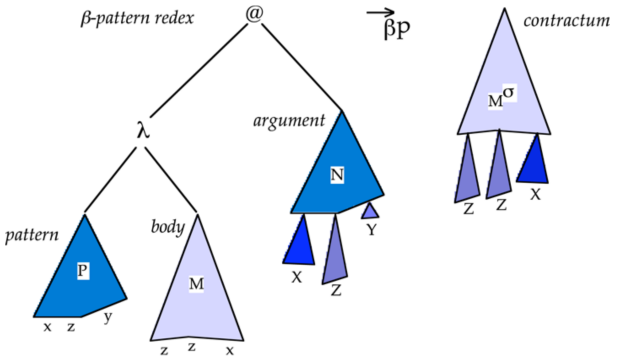
How to implement functional programming languages?

Idea of [SPJ]: transformational

FPL $\Longrightarrow$ λ-calculus with patterns (Chapter 4) $\Longrightarrow$ λ-calculus (with []; Chapter 6)



Pattern reduction in a picture [KvOdV]

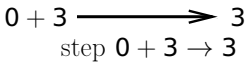


Idea

if argument is a substitution σ ($x \mapsto X$, $z \mapsto Z$, $y \mapsto Y$) instance N^σ of the pattern ($P = N$), then contractum M^σ is substitution instance of body M

Design space

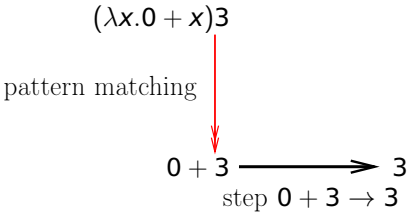
The three stages of reduction (for rule $0 + x \to x$)



Design space

The three stages of reduction (for rule $0 + x \to x$)

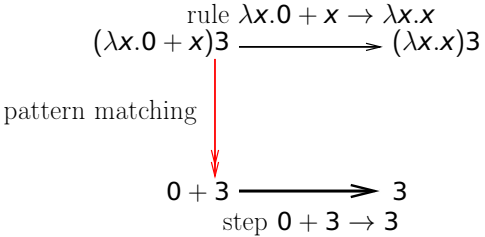
1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$



Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

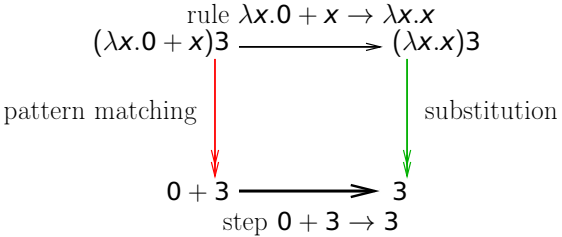
1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r (closure: $\forall \vec{x}.(\ell \to r) \implies (\lambda \vec{x}.\ell) \to (\lambda \vec{x}.r)$; Frege)



Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

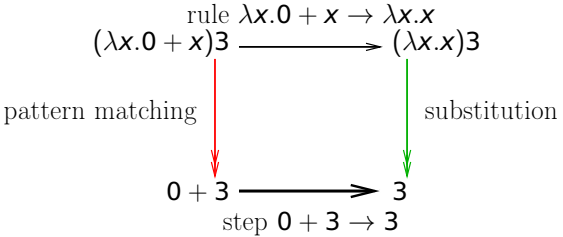
1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r



Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

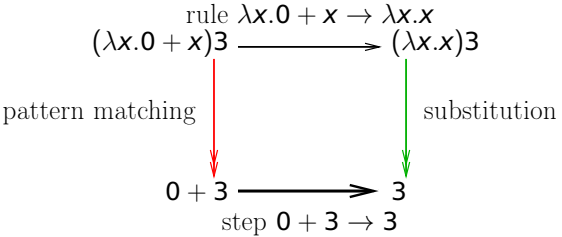


analogy = generalisation + specialisation (Pólya's △)

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r



substitution calculus ($\lambda\alpha\beta\eta^{\to}$) = matching + substitution (van Raamsdonk, vO)

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

- represent partial functions (λ) or anything goes (TRS)?

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

- represent partial functions (λ) or anything goes (TRS)?
- matching and/or substitution explicit (e.g. nominal, explicit α) or implicit

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

- represent partial functions (λ) or anything goes (TRS)?
- matching and/or substitution explicit (e.g. nominal, explicit α) or implicit
- matching and/or substitution syntactic (unitary) or modulo theory (e.g. AC)

Design space

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

- represent partial functions (λ) or anything goes (TRS)?
- matching and/or substitution explicit (e.g. nominal, explicit α) or implicit
- matching and/or substitution syntactic (unitary) or modulo theory (e.g. AC)
- how to handle match failures? (wait? error if stably fails?)

Design decisions

The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

λ-calculi with patterns design space

- represent partial functions (λ) or anything goes (TRS)?
- matching and/or substitution explicit (e.g. nominal, explicit α) or implicit
- matching and/or substitution syntactic (unitary) or modulo theory (e.g. AC)
- how to handle match failures? (wait? error if stably fails?)
- untyped or typed, fixed or arbitrary signature, ...

Design

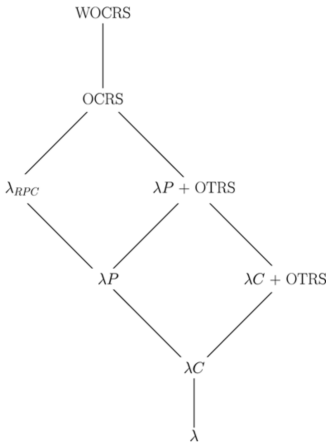
The three stages of reduction (for rule $\lambda x.0 + x \to \lambda x.x$)

1. matching: matching and abstracting lhs ℓ of rule $\ell \to r$
2. replacement: replace ℓ by r
3. substitution: substituting the parameters into rhs r

This lecture/hour

- represent (partial) functions
- implicit, syntactic matching and substitution (possibly higher-order)
- first just wait then with errors
- untyped and both arbitrary (λC and CRS/HRS) and fixed signatures (λ_{RPC})

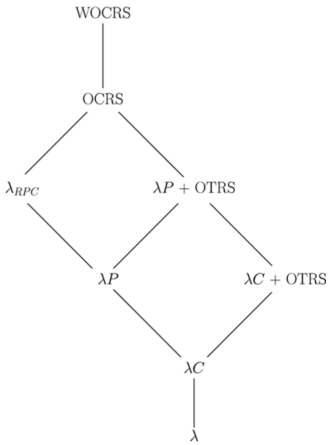
λ-calculi with patterns of [KvOdV]



Lattice of λ-calculi with patterns

WOCRS: weakly ortho 2nd order TRS
OCRS: orthogonal 2nd/higher order TRS
λ_{RPC} : λP with rigid pattern condition
$\lambda P +$ OTRS: $\lambda P +$ orthogonal TRS
λP: $\lambda +$ pure patterns
$\lambda C +$ OTRS: $\lambda C +$ orthogonal TRS
λC: $\lambda +$ constructor patterns
λ: ordinary $\lambda\beta$-calculus

λ-calculi with patterns of [KvOdV]



Lattice of λ-calculi with patterns

WOCRS: weakly ortho 2nd order TRS
OCRS: orthogonal 2nd/higher order TRS
λ_{RPC} : λP with rigid pattern condition
$\lambda P +$ OTRS: $\lambda P +$ orthogonal TRS
λP: $\lambda +$ pure patterns
$\lambda C +$ OTRS: $\lambda C +$ orthogonal TRS
λC: $\lambda +$ constructor patterns
λ: ordinary $\lambda\beta$-calculus

λC, λ-calculus with constructor patterns (wait)

Idea

terms over an arbitrary first-order signature of constructor symbols as patterns

λC, λ-calculus with constructor patterns (wait)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P.M)P^\sigma \to M^\sigma \qquad \text{vars in } P \text{ bound in } M \text{ in } \lambda P.M$$

λC, λ-calculus with constructor patterns (wait)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$
$$M ::= P \mid \lambda P.M \mid M\,M \mid c(\vec{M}^n)$$
$$(\lambda P.M)P^\sigma \to M^\sigma \qquad \text{vars in } P \text{ bound in } M \text{ in } \lambda P.M$$

Example (Simple list projection functions for $C = \{\mathrm{CONS}/2, A/0\}$)

$$\lambda \mathrm{CONS}(x,y).x$$
$$\lambda \mathrm{CONS}(x,y).y$$
$$(\lambda \mathrm{CONS}(x,y).y)\,\mathrm{CONS}(A,z) \to z$$

λC, λ-calculus with constructor patterns (wait)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P.M)P^\sigma \to M^\sigma \qquad \text{vars in } P \text{ bound in } M \text{ in } \lambda P.M$$

(Dis)advantages

- compared to λ-calculus: express (free) data types without coding

λC, λ-calculus with constructor patterns (wait)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P.M)P^\sigma \to M^\sigma \qquad \text{vars in } P \text{ bound in } M \text{ in } \lambda P.M$$

(Dis)advantages

- compared to λ-calculus: express (free) data types without coding
- compared to TRS: anonymous functions on data-types (see example)

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

Θ is Turing's fixed point combinator: $\Theta\, M \twoheadrightarrow M\, (\Theta\, M)$ for any term M

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

Remark (Imitating Turing (Klop))

How to build your own fixed-point combinator in three easy steps:

- *assume your fpc has repetitive shape $A\,A$ for some A*
  (idea: left A is the actor, right A is the passor)

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

Remark (Imitating Turing (Klop))

How to build your own fixed-point combinator in three easy steps:

- *assume your fpc has repetitive shape $A\,A$ for some A*
  (idea: left A is the actor, right A is the passor)
- *assume fp-behaviour is exhibited by expansion $A\,A\,M \twoheadrightarrow M\,(A\,A\,M)$*

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

Remark (Imitating Turing (Klop))

How to build your own fixed-point combinator in three easy steps:

- *assume your fpc has repetitive shape $A\,A$ for some A*
  (idea: left A is the actor, right A is the passor)
- *assume fp-behaviour is exhibited by expansion $A\,A\,M \twoheadrightarrow M\,(A\,A\,M)$*
- *conclude may set $A = \lambda am.m\,(a\,a\,m)$. Indeed $\Theta = AA$.*

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

$$\begin{array}{ccccccccc} A & \twoheadrightarrow & CA & \twoheadrightarrow & DP(A,CA) & \twoheadrightarrow & DP(CA,CA) & \to & E \\ \downarrow & & & & & & & & \\ CA & \twoheadrightarrow & C(CA) & \twoheadrightarrow & C(DP(A,CA)) & \twoheadrightarrow & C(DP(CA,CA)) & \to & CE \end{array}$$

Non-linearity breaks representing partial functions

Example (Klop's non-overlapping but non-confluent TRS $\{P/2, E/0\}$)

$$D = \lambda P(x,x).E$$
$$C = \Theta\, \lambda cx.D\, P(x, c\, x)$$
$$A = \Theta\, C$$

$$\begin{array}{ccccccccc} A & \twoheadrightarrow & CA & \twoheadrightarrow & DP(A,CA) & \twoheadrightarrow & DP(CA,CA) & \to & E \\ \downarrow & & & & & & & & \\ CA & \twoheadrightarrow & C(CA) & \twoheadrightarrow & C(DP(A,CA)) & \twoheadrightarrow & C(DP(CA,CA)) & \to & CE \end{array}$$

Comment on non-joinability of E and $C\,E$

if $C\,E$ and E were joinable $\implies C\,E \twoheadrightarrow E \implies C\,E \twoheadrightarrow D\,P(E, C\,E) \twoheadrightarrow E$

Non-linear non-overlapping TRSs not representable

Example (Huet's non-overlapping but non-confluent TRS)

$$\infty \to S(\infty)$$
$$E(x,x) \to \mathit{true}$$
$$E(x,S(x)) \to \mathit{false}$$

Non-linear non-overlapping TRSs not representable

Example (Huet's non-overlapping but non-confluent TRS)

$$\infty \to S(\infty)$$
$$E(x,x) \to \mathit{true}$$
$$E(x,S(x)) \to \mathit{false}$$

$$E(\infty,\infty) \to \mathit{true}$$
$$\downarrow$$
$$E(\infty,S(\infty)) \to \mathit{false}$$

Comment on non-definability in λC

true and *false* are distinct normal forms, but λC has the unique normal form property (also if non-linear patterns are allowed).

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Proof.

boring/wasteful: adapting the Tait–Martin-Löf proof □

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Proof.

better/reuse: linear λC embeds into orthogonal HRS (use Nipkow) □

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Theorem

λC has the unique normal form property
(normal forms N_1, N_2 are convertible $N_1 \leftrightarrow^* N_2 \implies N_1 = N_2$ the same)

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Theorem

λC has the unique normal form property

Proof.

boring/wasteful: adapting de Vrijer's proof □

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Theorem

λC has the unique normal form property

Proof.

better/reuse: λC embeds in strongly non-overlapping HRS (Mano,Ogawa) idea:
N_1, N_2 minimal distinct normal forms with $N_1 \leftrightarrow^* N_2 \implies$
conditional linearizations (cl; de Vrijer) N_1^*, N_2^* are cl-convertible $\implies$
these are cl-joinable $\implies$
one of N_1^*, N_2^* is cl-reducible $\implies$
one of them has distinct convertible normal forms as subterms. contradiction $\square$

Some λC rewriting meta-theory

Theorem (Linear λC represents partial functions)

λC is confluent if patterns are required to be linear

Theorem

λC has the unique normal form property

Example (of waiting and its solution [SPJ])

for NIL/0 a nullary constant, the term $(\lambda \mathrm{CONS}(x, y).y)\,\mathrm{NIL}$ is stuck; it waits until the NIL-argument matches the CONS-pattern which will never happen. this can be observed in λC because constructors are distinct $\Longrightarrow$ adjoin case construct

λC, λ-calculus with constructor patterns (case)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \;[]\; \ldots \;[]\; P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P_1.M_1 \;[]\; \ldots \;[]\; P_n.M_n)P_i^\sigma \to M_i^\sigma$$

λC, λ-calculus with constructor patterns (case)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \,[]\, \ldots \,[]\, P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P_1.M_1 \,[]\, \ldots \,[]\, P_n.M_n)P_i^\sigma \to M_i^\sigma$$

Example (Simple list projection functions for $C = \{\mathrm{CONS}/2, A/0, \mathrm{NIL}/0\}$)

$$(\lambda \mathrm{CONS}(x,y).y \,[]\, \mathrm{NIL}.A)\,\mathrm{NIL} \to A$$

λC, λ-calculus with constructor patterns (case)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \,[]\, \ldots \,[]\, P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P_1.M_1 \,[]\, \ldots \,[]\, P_n.M_n)P_i^\sigma \to M_i^\sigma$$

Example (Non-ortho patterns do not represent partial functions)

$$\infty = \Theta\, \lambda i.S(i)$$

$$E = \lambda P(x,x).true \,[]\, P(x,S(x)).false$$

$$N = (\lambda A.false \,[]\, A.true)\,A$$

∞ and E represent Huet's non-linear example; N has overlapping patterns

λC, λ-calculus with constructor patterns (case)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \;[]\; \ldots \;[]\; P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P_1.M_1 \;[]\; \ldots \;[]\; P_n.M_n)P_i^\sigma \to M_i^\sigma$$

Theorem (Representing partial functions in λC with cases)

λC is confluent if patterns are required to be linear and pairwise non-unifiable

λC, λ-calculus with constructor patterns (case)

Definition (λC for C arbitrary first-order signature)

$$P ::= x \mid c(\vec{P}^n) \qquad x \text{ a variable, } c \in C \text{ and } n\text{-ary}$$

$$M ::= P \mid \lambda P.M \,[]\, \ldots \,[]\, P.M \mid M\,M \mid c(\vec{M}^n)$$

$$(\lambda P_1.M_1 \,[]\, \ldots \,[]\, P_n.M_n)P_i^\sigma \to M_i^\sigma$$

Theorem (Representing partial functions in λC with cases)

λC is confluent if patterns are required to be linear and pairwise non-unifiable

Remark

can be extended with error rule in case argument is instance of (linear) pattern not unifiable with case-patterns. [SPJ]/Haskell use [] for sequential matching.

Gustave's TRS

Exercise

TRS with signature $\{G/3, \top/0, \bot/0, 1/0, 2/0, 3/0\}$ *and three rules*

$$G(\top, \bot, x) \to 1 \qquad G(\bot, x, \top) \to 2 \qquad G(x, \top, \bot) \to 3$$

and let $\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$ *as usual*

- *is the TRS orthogonal (left-linear and non-overlapping/lhss pairwise non-unifiable)?*
- *give a* λC*-expression* D*, where all symbols in the above signature are now constructors, that exhibits the behaviour as specified, i.e.* $D\,G(\top, \bot, \Omega) \twoheadrightarrow 1$*,* $D\,G(\bot, \Omega, \top) \twoheadrightarrow 2$*, and* $D\,G(\Omega, \top, \bot) \twoheadrightarrow 3$
- *can you express this in Haskell (thinking of* $\top, \bot$ *as booleans, say) (for 2nd part but you can think about it already)*

Idea

have λ-terms themselves as patterns
no extension of signature; minimal calculus in spirit of beginning of 20th century

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

λ_{RPC}

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Example (Simple list projection function in λ_{RPC})

$$(\lambda(\lambda z.z\,x\,y).x)\,\lambda z.z\,M\,N \to M$$
$$(\lambda(\lambda z.z\,x\,y).y)\,\lambda z.z\,M\,N \to N$$

$\lambda z.z\,M\,N$ is standard representation of a pair (M, N) in untyped λ-calculus

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Example (Problematic unstability)

- if variable is erased variables may be freed: $\lambda(\lambda x.y)z.z \to \lambda y.z$??

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Example (Problematic unstability)

- if variable is erased variables may be freed: $\lambda(\lambda x.y)z.z \to \lambda y.z$??
- applicative patterns give rise to inconsistency: $(\lambda(x\,y).x)\,I\,(K\,z)$ reduces in one step to I and in two steps to K?? for $I = \lambda x.x$ and $K = \lambda x.\lambda y.x$ as usual

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Definition (Restricting patterns P)

λ_{RPC} restricts patterns in abstractions to RPC comprising ordinary λ-terms that

- are linear; (free) variables occur at most once

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Definition (Restricting patterns P)

λ_{RPC} restricts patterns in abstractions to RPC comprising ordinary λ-terms that

- are linear; (free) variables occur at most once
- are in normal form; contain no β-redex

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Definition (Restricting patterns P)

λ_{RPC} restricts patterns in abstractions to RPC comprising ordinary λ-terms that

- are linear; (free) variables occur at most once
- are in normal form; contain no β-redex
- have no active variables; no subterms of the form $x\,M$ with x free

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Definition (Restricting patterns P)

λ_{RPC} restricts patterns in abstractions to RPC comprising ordinary λ-terms that

- are linear; (free) variables occur at most once
- are in normal form; contain no β-redex
- have no active variables; no subterms of the form $x\,M$ with x free

simple list projection functions in λ_{RPC}

λ_{RPC}

Definition (Provisional minimal syntax (λP))

$$M, P ::= x \mid \lambda P.M \mid M\,M \qquad x \text{ a variable; free vars of } P \text{ bound in } M \text{ in } \lambda P.M$$

$$(\lambda P.M)P^{\sigma} \to M^{\sigma}$$

Definition (Restricting patterns P)

λ_{RPC} restricts patterns in abstractions to RPC comprising ordinary λ-terms that

- are linear; (free) variables occur at most once
- are in normal form; contain no β-redex
- have no active variables; no subterms of the form $x\,M$ with x free

Theorem

λ_{RPC} is confluent

The case for higher-order term rewriting; CRSs/HRSs

Solution for ?

combinatory Logic (Schönfinkel) : first-order term rewrite systems

$=$

λ-calculus (Church) : ?

The case for higher-order term rewriting; CRSs/HRSs

Solution for ?

combinatory Logic (Schönfinkel) : first-order term rewrite systems

$=$

λ-calculus (Church) : ?

Desiderata

- closed under change of alphabet (CL,λ-calculi never are)

The case for higher-order term rewriting; CRSs/HRSs

Solution for ?

combinatory Logic (Schönfinkel) : first-order term rewrite systems

$=$

λ-calculus (Church) : ?

Desiderata

- closed under change of alphabet (CL,λ-calculi never are)
- rules having object variables (CL,λ-calculi typically rule schemata)

The case for higher-order term rewriting; CRSs/HRSs

Solution for ?

combinatory Logic (Schönfinkel) : first-order term rewrite systems

$=$

λ-calculus (Church) : ?

Desiderata

- closed under change of alphabet (CL,λ-calculi never are)
- rules having object variables (CL,λ-calculi typically rule schemata)
- object level substitution,matching and critical pairs (CL,λ-calculi never have)

The case for higher-order term rewriting; CRSs/HRSs

Universal algebra approach to CL,λ-calculus

combinatory Logic (Schönfinkel) : first-order term rewrite systems

$=$

λ-calculus (Church) : higher-order term rewrite systems

Desiderata

- closed under change of alphabet (CL,λ-calculi never are)
- rules having object variables (CL,λ-calculi typically rule schemata)
- object level substitution,matching and critical pairs (CL,λ-calculi never have)

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ $(\ell \to r)$ is just a pair of terms $\Longrightarrow$ introduce a binary symbol for pairing into alphabet

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ $(\ell \to r)$ is just a pair of terms $\Longrightarrow$ introduce a binary symbol for pairing into alphabet
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$ explicitly abstract free variables in ℓ

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ $(\ell \to r)$ is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ
3. application $(\lambda\ell.r)$ to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ
3. application $(\lambda\ell.r)$ to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet

Example

$(\lambda\,[x,y].x)\,[1,2,3,4,5] \Longrightarrow \lambda\mathsf{RULE}([x,y],x)\,[1,2,3,4,5]$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application $(\lambda\ell.r)$ to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet

Example

$(\lambda\,[x,y].x)\,[1,2,3,4,5] \Longrightarrow \lambda x\,y.\mathrm{RULE}([x,y],x)\,[1,2,3,4,5]$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application $(\lambda\ell.r)$ to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet: @

Example

$(\lambda\,[x,y].x)\,[1,2,3,4,5] \Longrightarrow @(\lambda x\,y.\mathrm{RULE}([x,y],x),[1,2,3,4,5])$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ $(\ell \to r)$ is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application $(\lambda\ell.r)$ to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet: @

Example

$@(\lambda x\, y.\mathrm{RULE}([x, y], x), [1, 2, 3, 4, 5])$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application ($\lambda\ell.r$) to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet: @
4. one rewrite rule per ℓ pattern: $@(\lambda\vec{x}.\mathrm{RULE}(\ell, F(\vec{x})), \ell(\vec{X})) \to F(\vec{X})$

Example

$@(\lambda x\, y.\mathrm{RULE}([x, y], x), [1, 2, 3, 4, 5])$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application ($\lambda\ell.r$) to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet: @
4. one rewrite rule per ℓ pattern: $@(\lambda\vec{x}.\mathrm{RULE}(\ell, F(\vec{x})), \ell(\vec{X})) \to F(\vec{X})$

Example

$@(\lambda x\, y.\mathrm{RULE}([x, y], x), [1, 2, 3, 4, 5])$
$@(\lambda x\, y.\mathrm{RULE}([x, y], F(x, y)), [X, Y]) \to F(X, Y)$

Embedding λ-calculi with patterns into HRSs

Idea

1. term rewrite rule $\lambda\ell.r$ ($\ell \to r$) is just a pair of terms $\Longrightarrow$
   introduce a binary symbol for pairing into alphabet: RULE
2. variables in $\lambda\ell.r$ implicitly universally quantified $\Longrightarrow$
   explicitly abstract free variables in ℓ: use HRS binder λ
3. application ($\lambda\ell.r$) to some argument t $\Longrightarrow$
   introduce a binary symbol for application into alphabet: @
4. one rewrite rule per ℓ pattern: $@(\lambda\vec{x}.\text{RULE}(\ell, F(\vec{x})), \ell(\vec{X})) \to F(\vec{X})$

Example

$@(\lambda x\, y.\text{RULE}([x, y], x), [1, 2, 3, 4, 5]) \to 1$ (with $F = \lambda ab.a$, $X = 1$, $Y = [2, 3, 4, 5]$)
$@(\lambda x\, y.\text{RULE}([x, y], F(x, y)), [X, Y]) \to F(X, Y)$

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

adapt? meta-theory of (weakly) orthogonal higher/2nd order TRSs:

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate! meta-theory of (weakly) orthogonal higher/2nd order TRSs:

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*
- *matching/unification (Nipkow)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*
- *matching/unification (Nipkow)*
- *semantics/universal algebra (Hamana,Fiore)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*
- *matching/unification (Nipkow)*
- *semantics/universal algebra (Hamana,Fiore)*
- *termination methods (Jouannaud, Rubio, Kop, van Raamsdonk, Blanqui)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*
- *matching/unification (Nipkow)*
- *semantics/universal algebra (Hamana,Fiore)*
- *termination methods (Jouannaud, Rubio, Kop, van Raamsdonk, Blanqui)*
- *standardisation, developments, residuation (Klop,Bruggink,vO)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Corollary

instantiate meta-theory of (weakly) orthogonal higher/2nd order TRSs:

- *confluence (Nipkow,Klop,van Raamsdonk,vO)*
- *matching/unification (Nipkow)*
- *semantics/universal algebra (Hamana,Fiore)*
- *termination methods (Jouannaud, Rubio, Kop, van Raamsdonk, Blanqui)*
- *standardisation, developments, residuation (Klop,Bruggink,vO) . . .*

HRSs may capture stages of transformation from FPL into assembly (CRSX, Rose)

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

- *typed patterns $\Longrightarrow$ sub-HRS induced by inference system (see slides LL)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

- *typed patterns $\Longrightarrow$ sub-HRS induced by inference system (see slides LL)*
- *ρ-calculus (Kirchner et al.) $\Longrightarrow$ explicit manipulation of rules themselves*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

- *typed patterns $\Longrightarrow$ sub-HRS induced by inference system (see slides LL)*
- *ρ-calculus (Kirchner et al.) $\Longrightarrow$ explicit manipulation of rules themselves*
- *pure patterns (Jay,Kesner) $\Longrightarrow$ handling names (with F van Raamsdonk)*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

- *typed patterns $\Longrightarrow$ sub-HRS induced by inference system (see slides LL)*
- *ρ-calculus (Kirchner et al.) $\Longrightarrow$ explicit manipulation of rules themselves*
- *pure patterns (Jay,Kesner) $\Longrightarrow$ handling names (with F van Raamsdonk)*
- *. . .*

Embedding yields orthogonal HRS

Theorem

translation yields a (potentially infinite,weakly) orthogonal HRS
(if patterns in λC or λ_{RPC} are restricted to linear ones)

Remark (Personal)

embedding reason to focus on higher-order TRSs after [IR-228] (1990)
perspective: how do novel features of λ-calculi with patterns embed?

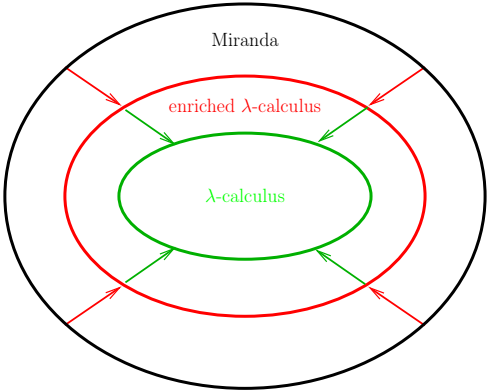
- *typed patterns $\Longrightarrow$ sub-HRS induced by inference system (see slides LL)*
- *ρ-calculus (Kirchner et al.) $\Longrightarrow$ explicit manipulation of rules themselves*
- *pure patterns (Jay,Kesner) $\Longrightarrow$ handling names (with F van Raamsdonk)*
- *. . .*

but always from a higher-order perspective

Recall implementing functional languages in [SPJ]

Idea of [SPJ]: transformational

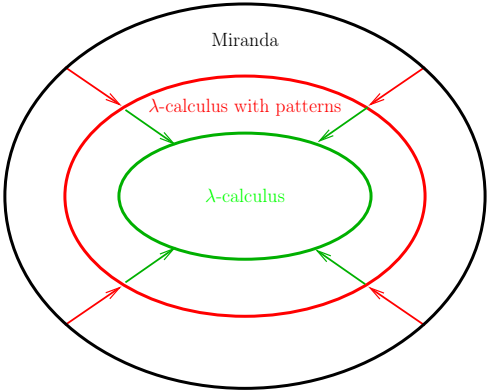
FPL $\Longrightarrow$ enriched λ-calculus (Chapter 4) $\Longrightarrow$ λ-calculus (with []; Chapter 6)



Recall implementing functional languages in [SPJ]

Idea of [SPJ]: transformational

FPL $\Longrightarrow$ λ-calculus with patterns (Chapter 4) $\Longrightarrow$ λ-calculus (with []; Chapter 6)



Compiling pattern matching into λ-calculus for Haskell

Idea

sequentialise matching symbol by symbol [SPJ] (Chapter 5, Wadler)

Compiling pattern matching into λ-calculus for Haskell

Idea

sequentialise matching symbol by symbol [SPJ] (Chapter 5, Wadler)

Example (Remove repetitions of elements by pattern matching)

```
unrep [] = []
unrep [x] = [x]
unrep (x:y:xs) = if x == y then unreptail else x:unreptail where
  unreptail = unrep (y:xs)
```

patterns match top–bottom (left–right); may have nested patterns like `x:y:xs` compile away using case

Compiling pattern matching into λ-calculus for Haskell

Example (Remove repetitions of elements by pattern matching)

```
unrep [] = []
unrep [x] = [x]
unrep (x:y:xs) = if x == y then unreptail else x:unreptail where
  unreptail = unrep (y:xs)
```

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Compiling pattern matching into λ-calculus for Haskell

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Compiling pattern matching into λ-calculus for Haskell

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Discussion

Compiling pattern matching into λ-calculus for Haskell

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Discussion

- sequentialisation due to top–bottom and left–right order (Haskell)

Compiling pattern matching into λ-calculus for Haskell

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Discussion

- sequentialisation due to top–bottom and left–right order (Haskell)
- proper case split due to non-ambiguity and linearity of patterns (Haskell)

Compiling pattern matching into λ-calculus for Haskell

Example (Sequentialise using case expressions)

```
unrepc l = case l of
  [] -> []
  (x:t) -> case t of
    []      -> l
    (y:t') -> if x == y then unrepc t else x:unrepc t
```

Discussion

- sequentialisation due to top–bottom and left–right order (Haskell)
- proper case split due to non-ambiguity and linearity of patterns (Haskell)
- case implementable in λ-calculus by discriminating constructors (Böhm)

Lazy patterns in Haskell

Motivation

sometimes we know, say due to typing or due to knowing that we have an infinite stream, that only one pattern is possible, say a pair, but we want to match lazily, e.g. we are not interested in the arguments, or only want to match when the arguments/projections are needed

Lazy patterns in Haskell

Motivation

sometimes we know, say due to typing or due to knowing that we have an infinite stream, that only one pattern is possible, say a pair, but we want to match lazily, e.g. we are not interested in the arguments, or only want to match when the arguments/projections are needed

Haskell has lazy patterns; indicated by ~ in Haskell or pattern bindings

Lazy patterns in Haskell

Motivation

sometimes we know, say due to typing or due to knowing that we have an infinite stream, that only one pattern is possible, say a pair, but we want to match lazily, e.g. we are not interested in the arguments, or only want to match when the arguments/projections are needed
Haskell has lazy patterns; indicated by ~ in Haskell or pattern bindings

Example of pattern binding

```
fib@(1:tfib)     = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

where `@` makes `fib` an alias of the pattern binding `1:tfib`

Lazy patterns in Haskell

Motivation

sometimes we know, say due to typing or due to knowing that we have an infinite stream, that only one pattern is possible, say a pair, but we want to match lazily, e.g. we are not interested in the arguments, or only want to match when the arguments/projections are needed
Haskell has lazy patterns; indicated by ~ in Haskell or pattern bindings

Example of pattern binding

```
fib@(1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

where `@` makes `fib` an alias of the pattern binding `1:tfib`

lazy patterns akin to speculative execution (bad things may happen)

Lazy patterns in Haskell translation exercise

Exercise on translating (lazy) patterns into cases

how does `ispair mypair` for the code below evaluate and why? same question for `ispairl` and `ispairw`. can your translate these definitions into case expressions/projections? what about when changing each `True` into `x+y`?

```
ispair (x,y) = True
ispairl ~(x,y) = True
ispairw p = True where
  x = fst p
  y = snd p
mypair = mypair
```

Note `(,)` is the only constructor of the pair data type

Lazy patterns in Haskell exercise

Exercise on lazy patterns

consider the following core client–server program; from Haskell [tutorial]:

```
reqs  = client initmsg resps
resps = server reqs
client msg (r:rs) = msg : client (next r) rs
server (r:rs) = process r : server rs
initmsg = 0
next r = r
process r = r+1
```

for this program `take 10 reqs` does not generate any output, but yields `[0,1,2,3,4,5,6,7,8,9]` when making `client` lazy by means of `~(r:rs)`.

can you explain this, based on sequentialising patterns as presented?