Polymorphic Rewrite Rules

Confluence, Type Inference, and Instance Validation

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

FLOPS 2018

Nagoya University, May, 2018

Motivation

- A calculus for programming languages is often given as simply-typed computation rules
- Require a schematic type notation
- Simply-typed $\boldsymbol{\lambda}$-calculus:

$$(\boldsymbol{\beta}) \qquad \boldsymbol{\Gamma} \vdash (\boldsymbol{\lambda} \boldsymbol{x}^{\boldsymbol{\sigma}}.\, \boldsymbol{M})\, \boldsymbol{N} \quad \Rightarrow \quad \boldsymbol{M}[\boldsymbol{x} := \boldsymbol{N}] \quad : \quad \boldsymbol{\tau}$$

  - Point: $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ are not fixed types, but schemata of types
  - Represents various **instances** of rules, such as

$$(\boldsymbol{\beta}_{\mathsf{bool,int}}) \qquad \boldsymbol{\Gamma} \vdash (\boldsymbol{\lambda} \boldsymbol{x}^{\mathsf{bool}}.\, \boldsymbol{M})\, \boldsymbol{N} \quad \Rightarrow \quad \boldsymbol{M}[\boldsymbol{x} := \boldsymbol{N}] \quad : \mathsf{int}$$

$$(\boldsymbol{\beta}_{\mathsf{int}\to\mathsf{int,bool}}) \qquad \boldsymbol{\Gamma} \vdash (\boldsymbol{\lambda} \boldsymbol{x}^{\mathsf{int}\to\mathsf{int}}.\, \boldsymbol{M})\, \boldsymbol{N} \quad \Rightarrow \quad \boldsymbol{M}[\boldsymbol{x} := \boldsymbol{N}] \quad : \mathsf{bool}$$

$$\ldots$$

Problems and This Work

- Not explored in the theory of rewriting
- How to ensure confluence of polymorphic rules generally
- A novel framework for polymorphic computation rules
- A computational refinement of second-order algebraic theories [Fiore, Hur CSL'10]
  - logic programming [Staton FoSSaCS'13]
  - algebraic effects [Staton LICS'13]
  - quantum computation [Staton POPL'15]

Example: Confluence of λ_c [Moggi'88]

- The computational λ-calculus

Call-by-value λ-calculus with `let` for sequential computation

```
(β-v)      lam(x.M[x]) @ V => M[V]    ; (η-v)    lam(x.V @ x) => V
(β-let-v) let(V, x.M[x])  => M[V]    ; (η-let) let(L, x.x)  => L
(let1-p)  P @ M           => let(P,x.x@M)
(let2-v)  V @ P           => let(P,y.V@y)
(assoc)   let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y]))
```

- Values and non-values:

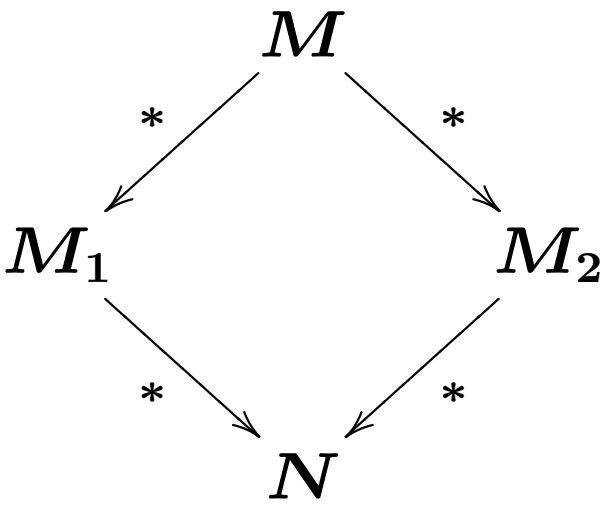
$$\text{Values} \quad \boldsymbol{V} ::= \boldsymbol{x} \mid \boldsymbol{\lambda x.M}$$

$$\text{Non-values} \quad \boldsymbol{P} ::= \boldsymbol{M@N} \mid \text{let } \boldsymbol{x} = \boldsymbol{M} \text{ in } \boldsymbol{N}$$

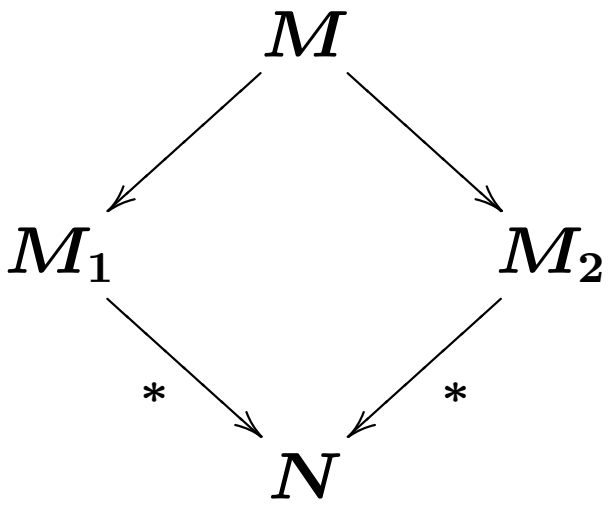
- Encoding: $\boldsymbol{\lambda x.M}$ as `lam(x.M)`, let $\boldsymbol{x} = \boldsymbol{M}$ in $\boldsymbol{N}$ as `let(M,x.N)`
- SN [Lindley, Stark TLCA'05]
- Confluent?

Two Rewriting Properties

Confluence

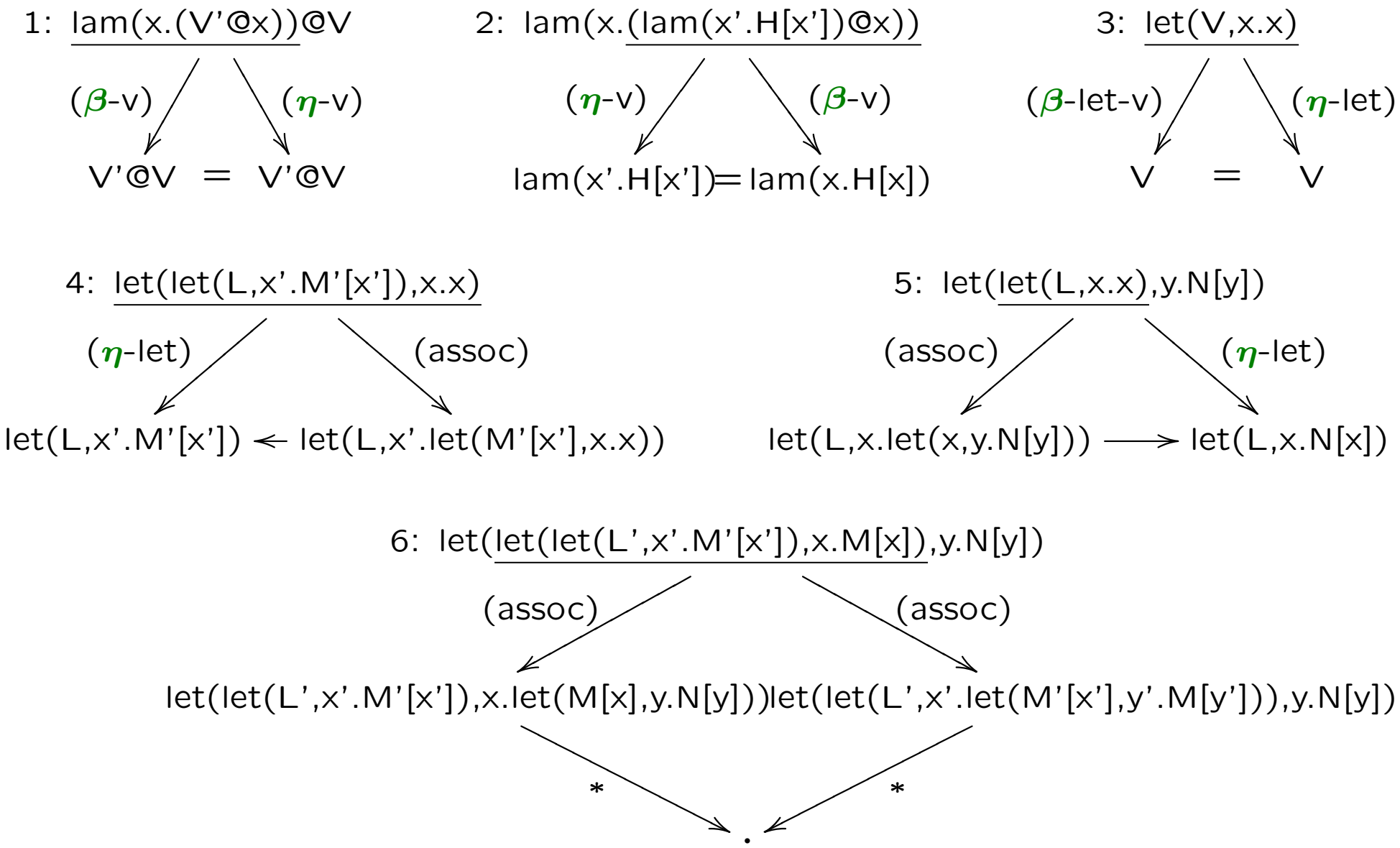


Local Confluence



Newman's lemma

If a rewrite system is locally confluent & terminating, then it is confluent.



The joinablity of Critical Pairs shows local confluence.

Applying Newman's lemma, we have confluence

- Knuth-Bendix critical pair checking

Critical Pair Checking by PolySOL (1)

PolySOL: Haskell-based tool for confluence and termination

Step 1. Write a Haskell script describing rewrite rules and
. signature using PolySOL's DSL (by Template Haskell)

```
siglamC = [signature|
 app : Arr(a,b),a -> b
 lam : (a -> b) -> Arr(a,b)
 let : a,(a -> b) -> b      |]
```

- `a,b` are type variables
- `Arr` is a type constructor

```
lamC = [rule|
 (β-v)      lam(x.M[x]) @ V => M[V]    ; (η-v)    lam(x.V @ x) => V
 (β-let-v) let(V, x.M[x])  => M[V]     ; (η-let) let(L, x.x)  => L
 (let1-p)  P @ M           => let(P,x.x@M)
 (let2-v)  V @ P           => let(P,y.V@y)
 (assoc)   let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y]))  |]
```

Critical Pair Checking by PolySOL (2)

Step 2. Invoke the command `cri` for critical pair checking

```
*PolSOL> cri lamC siglamC
1: Overlap (β-v)-(η-v)--- M|-> z1.(V'@z1)-----------------------------------
   L: lam(x.M[x])@V => M[V]
   R: lam(x'.(V'@x')) => V'
                                  (lam(x.(V'@x))@V)
                          (V'@V) <-(β-v)-∧-(η-v)-> (V'@V)
                            ---> (V'@V) =OK= (V'@V) <---
..
4: Overlap (η-let)-(assoc)--- M|-> let(L',x'.M'[x']), N'|-> z1.z1-----------------------
   L: let(M,x.x) => M
   R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
                               let(let(L',x'.M'[x']),x.x)
              let(L',x'.M'[x']) <-(eta-let)-∧-(assoc)-> let(L',xd13.let(M'[xd13],yd13.yd13))
                    ---> let(L',x'.M'[x']) =E= let(L',xd13.M'[xd13]) <---
..
#Joinable! (Total 6 CPs)
```

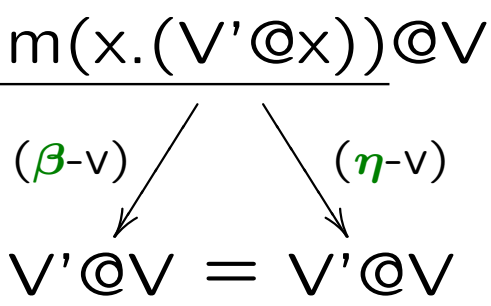
Critical Pair

```
(β-v)   lam(x.M[x]) @ V => M[V]
(η-v)   lam(x.V @ x)    => V
```

▷ Second-order unification

$$\texttt{lam(x.M[x])} \stackrel{?}{=} \texttt{lam(x.V @ x)}$$

a unifier $\boldsymbol{\theta} = \{\texttt{M} \mapsto \texttt{z.(V @ z)}\}$



▷ This is ordinary

Problems and Questions

1. The notion of unifier for an overlap is non-standard in the call-by-value case

```
(let1-p)  P @ M          => let(P,x.x@M)
(let2-v)  V @ P          => let(P,y.V@y)
```

- ▷ Not overlapped
- ▷ Values and non-values:

$$\text{Values} \quad \boldsymbol{V} \ ::= \ \boldsymbol{x} \mid \texttt{lam(x.M)}$$

$$\text{Non-values} \quad \boldsymbol{P} \ ::= \ \boldsymbol{M@N} \mid \texttt{let(M,x.N)}$$

Q1 ▶ What is a general definition of overlaps when there is a restriction on the term structures?

2. Different occurrences of the same function symbol may have different types

```
(assoc)   let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y]))
```

$$\boldsymbol{M : c \longrightarrow a,\ N : a \longrightarrow b,\ L : c \ \triangleright}$$

$$\boldsymbol{\Gamma} \vdash \mathsf{let}^{a,(a\to b)\to b}(\mathsf{let}^{c,(c\to a)\to a}(\boldsymbol{L}, \boldsymbol{x}^{c}.\boldsymbol{M}[\boldsymbol{x}]), \boldsymbol{y}^{a}.\boldsymbol{N}[\boldsymbol{y}])$$

$$\Rightarrow \mathsf{let}^{c,(c\to b)\to b}(\boldsymbol{L}, \boldsymbol{x}^{c}.\mathsf{let}^{a,(a\to b)\to b}(\boldsymbol{M}[\boldsymbol{x}], \boldsymbol{y}^{a}.\boldsymbol{N}[\boldsymbol{y}])) \quad : \ \boldsymbol{b}$$

Q2 ▶ What should be unification between polymorphic terms?

3. Specifying all the type annotations in the rule specification manually is tedious and prone to error

Q3 ▶ What is type inference algorithm for polymorphic computation rules?

This Work

A1▶ Introduce predicates called **instance validation** on substitutions

It is used for formulating computation steps having some restriction of term structures.
They affect to the notion of critical pairs.

A2▶ Formulate unification between polymorphic terms

A3▶ Give a type inference algorithm for polymorphic rules

Q1. What is a general definition of overlaps?

Q2. What should be unification between polymorphic terms?

Q3. What is type inference algorithm for polymorphic rules?

A1: Instance Validation

Def. An instance validation is an predicate valid
that takes a substitution $[\overline{M \mapsto \overline{x}.s}]$ for metavariables.

Any predicate is OK. e.g.

- ▷ **Any:** valid θ $\overset{\text{def}}{\Leftrightarrow}$ always true, i.e. any instantiation is valid.
- ▷ **Injectivity:** valid θ $\overset{\text{def}}{\Leftrightarrow}$ θ is an injective substitution
- ▷ **Values/non-values:** valid $[M_1 \mapsto \overline{x_1}.s_1, \ldots]$
  $\overset{\text{def}}{\Leftrightarrow}$ $((M_i \equiv V \Rightarrow s_i$ is a value$)$ & $(M_i \equiv P \Rightarrow s_i$ is a non-value$))$
  for all i.
  (values V and non-values P)

Instance validation valid is used in formulating

- ▷ one-step computation
- ▷ overlaps for critical pairs

A2: Unification between Polymorphic Terms

What should be a unifier between the following polymorphic terms?

$$\mathtt{lam}^{(\mathsf{bool}\to \mathrm{T})\to \mathsf{Arr}(\mathsf{bool},\mathrm{T})}(x^{\mathsf{bool}}.M[x]) \stackrel{?}{=} \mathtt{lam}^{(\mathrm{V}\to \mathsf{int})\to \mathsf{Arr}(\mathrm{V},\mathsf{int})}(x^{\mathrm{V}}.\mathrm{g}^{\mathrm{V}\to \mathsf{int}}(x))$$

Here T, V are type variables.

Answer: a type substitution $\xi : \mathrm{V} \mapsto \mathsf{bool}, \mathrm{T} \mapsto \mathsf{int}$
with a substitution of terms for variables $\theta : M \mapsto x^{\mathsf{bool}}.\mathrm{g}^{\mathsf{bool}\to \mathsf{int}}(x)$

Def.

A **unifier between polymorphic terms** s, t is a pair (θ, ξ) such that

$$s\,\xi\,\theta = t\,\xi\,\theta$$

Computation Step: $\Rightarrow_{\mathcal{C}}$

Given a set $\mathcal{C}$ of polymorphic computation rules

S is the set of all type variables in $\overline{\tau_i}, \overline{\sigma_i}, \tau$ type subst. $\xi : S \to \mathcal{T}$

$$\frac{\Theta \rhd \Gamma', \overline{x_i : \tau_i} \vdash s_i : \sigma_i\xi \quad (1 \le i \le k) \qquad \text{valid } [\overline{M \mapsto \overline{x}.s}] \qquad (M_1 : (\overline{\tau_1} \to \sigma_1), \ldots, M_k : (\overline{\tau_k} \to \sigma_k) \rhd \Gamma \vdash \ell \Rightarrow r : \tau) \in \mathcal{C}}{\Theta \rhd \Gamma, \Gamma' \vdash \ell\xi\,[\overline{M \mapsto \overline{x}.s}] \Rightarrow_{\mathcal{C}} r\xi\,[\overline{M \mapsto \overline{x}.s}] : \tau\xi}$$

$$\frac{S \rhd f : (\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau \in \Sigma \qquad \xi : S \to \mathcal{T} \qquad \Theta \rhd \Gamma, \overline{x_i : \sigma_i} \vdash t_i \Rightarrow_{\mathcal{C}} t_i' : \sigma_i\xi \quad (\text{some } i \text{ s.t. } 1 \le i \le m)}{\Theta \rhd \Gamma \vdash f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t_i, \ldots) \Rightarrow_{\mathcal{C}} f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t_i', \ldots) : \tau\xi}$$

A3: Type Inference

- Type inference function:

$$\text{infer}(\boldsymbol{\Sigma}, \boldsymbol{t})$$

  where $\boldsymbol{t}$ is a "plain" term, and $\boldsymbol{\Sigma}$ is a signature

- A modification Damas-Milner type inference algorithm W

- Types in our framework is basically ML polymorphism

Thm. (Soundness) If $\text{infer}(\boldsymbol{\Sigma}, \boldsymbol{t}) = (\boldsymbol{\Theta} \triangleright \boldsymbol{t'} : \boldsymbol{\tau})$, then there exists $\boldsymbol{\Gamma}$ such that $\boldsymbol{\Theta} \triangleright \boldsymbol{\Gamma} \vdash \boldsymbol{t'} : \boldsymbol{\tau}$.

Thm. (Completeness) If a given term is typable, then the algorithm infers a more general type.

Example: Confluence of λ_{NEED}

Maraist, Odersky, and Wadler's call-by-need λ-calculus λ_{NEED} [JFP'98]

```
sigNeed = [signature|
 lam : (a->b) -> Arr(a,b); app : Arr(a,b),a -> b; let : a,(a->b) -> b |]

lmdNeed = [rule|
 (rG)    let(M, x.N)                  => N
 (rI)    lam(x.M[x]) @ N              => let(N,x.M[x])
 (rV-v)  let(V, x.C[x])               => C[V]
 (rC-v)  let(V, x.M[x])@N             => let(V, x.M[x]@N)
 (rA)    let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y])) |]
```

- Demo: `http://www.cs.gunma-u.ac.jp/hamana/polysol/`

Summary of This Work

A1▶ Introduce instance validation predicates on substitutions for formulating computation steps and critical pairs when there are some restriction of the term structures

A2▶ Formulate unification between polymorphic terms

A3▶ Give a type inference algorithm for polymorphic rules

▶ PolySOL system

`http://www.cs.gunma-u.ac.jp/hamana/polysol/`

Q1. What is a general definition of overlaps?

Q2. What should be unification between polymorphic terms?

Q3. What is type inference algorithm for polymorphic rules?