Polymorphic Rewrite Rules: Confluence, Type Inference, and Instance Validation

Makoto Hamana

Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract. We present a new framework of polymorphic rewrite rules having predicates to restrict their instances. It is suitable for formulating and analysing fundamental calculi of programming languages. A type inference algorithm and a criterion to check local confluence property of polymorphic rules are also given, with demonstration of the effectiveness of our methodology by examinination of sample program calculi. It includes the call-by-need λ-calculus and Moggi's computational lambda-calculus.

1 Introduction

Fundamental calculi of programming languages are often formulated as simply-typed computation rules. Describing such a simply-typed system requires a schematic type notation that is best formulated in a *polymorphic* typed framework. To illustrate this situation, consider the simply-typed λ-calculus as a sample calculus:

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

An important point is that σ and τ are not fixed types, but schemata of types. Therefore, (β) actually describes a *family of actual computation rules*. Namely, it represents various instances of rules by varying σ and τ, such as the following.

$$\begin{array}{lllll} (\beta_{\mathsf{bool,int}}) & \Gamma \vdash (\lambda x^{\mathsf{bool}}.\, M)\, N & \Rightarrow & M[x := N] & : \ \mathsf{int} \\ (\beta_{\mathsf{int}\to\mathsf{int,bool}}) & \Gamma \vdash (\lambda x^{\mathsf{int}\to\mathsf{int}}.\, M)\, N & \Rightarrow & M[x := N] & : \ \mathsf{bool} \end{array}$$

From the viewpoint of meta-theory, as in a mechanised formalisation of mathematics, the (β)-rule should be formulated in a polymorphic typed framework, where types τ and σ vary over simple types. This viewpoint has not been well explored in the general theory of rewriting. For instance, no method has been established for checking the *confluence property* of a general kind of polymorphically typed computation rules automatically

We have already recognized this problem. In a previous paper [12], we investigated the decidability of various program calculi by confluence and termination checking. The type system used there was called molecular types, which was intended to mimic polymorphic types in a simple type setting. However, this mimic setting provided no satisfactory framework to address polymorphic typed rules. For example, molecular types did not have a way to vary types. Therefore, instantiation of (β) to ($\beta_{\mathsf{bool,int}}$)($\beta_{\mathsf{int}\to\mathsf{int,bool}}$) described above could not be obtained. For that reason, no confluence of instances of polymorphic computation rules is obtained automatically. Further manual meta-theoretic analysis is necessary.

$$\begin{tikzcd}
& \text{1: }\underline{\text{lam(x.(V'@x))}}\text{@V} \arrow[dl, "(\beta\text{-v})"'] \arrow[dr, "(\eta\text{-v})"] & \\
\text{V'@V} & = & \text{V'@V}
\end{tikzcd}$$

$$\begin{tikzcd}
& \text{2: lam(x.}\underline{\text{(lam(x'.H[x'])@x))}} \arrow[dl, "(\eta\text{-v})"'] \arrow[dr, "(\beta\text{-v})"] & \\
\text{lam(x'.H[x'])} & = & \text{lam(x.H[x])}
\end{tikzcd}$$

$$\begin{tikzcd}
& \text{3: }\underline{\text{let(V,x.x)}} \arrow[dl, "(\beta\text{-let-v})"'] \arrow[dr, "(\eta\text{-let})"] & \\
\text{V} & = & \text{V}
\end{tikzcd}$$

$$\begin{tikzcd}
& \text{4: }\underline{\text{let(let(L,x'.M'[x']),x.x)}} \arrow[dl, "(\eta\text{-let})"'] \arrow[dr, "(\text{assoc})"] & \\
\text{let(L,x'.M'[x'])} & & \text{let(L,x'.let(M'[x'],x.x))} \arrow[ll]
\end{tikzcd}$$

$$\begin{tikzcd}
& \text{5: let(}\underline{\text{let(L,x.x)}}\text{,y.N[y])} \arrow[dl, "(\text{assoc})"'] \arrow[dr, "(\eta\text{-let})"] & \\
\text{let(L,x.let(x,y.N[y]))} \arrow[rr] & & \text{let(L,x.N[x])}
\end{tikzcd}$$

$$\begin{tikzcd}
& \text{6: let(}\underline{\text{let(let(L',x'.M'[x']),x.M[x])}}\text{,y.N[y])} \arrow[dl, "(\text{assoc})"'] \arrow[dr, "(\text{assoc})"] & \\
\text{let(let(L',x'.M'[x']),x.let(M[x],y.N[y]))} \arrow[dr, "*"'] & & \text{let(let(L',x'.let(M'[x'],y'.M[y'])),y.N[y])} \arrow[dl, "*"] \\
& . &
\end{tikzcd}$$

Fig. 1. Critical pairs of the λ_C-calculus

In the present paper, we give an extended framework for polymorphic computation rules that solve these issues. The framework is polymorphic and a computational refinement of second-order algebraic theories by Fiore *et al.* [5, 6]. Second-order algebraic theories have been shown to be a useful framework that models various important notions of programming languages, such as logic programming [28] and algebraic effects [29], quantum computation [30]. The present polymorphic framework has also applications in these fields.

1.1 Example: confluence of the computational λ-calculus

We begin with examining a sample confluence problem to illustrate our framework and methodology. We consider Moggi's computational λ-calculus, the λ_C-calculus [21], which is a fundamental λ-calculus for effectful computation. It is an extension of the call-by-value λ-calculus enriched with `let`-construct to represent sequential computation. The λ_C has two classes of terms: values and non-values.

$$\text{Values } V \ ::= \ x \mid \lambda x.M \qquad\qquad \text{Non-values } P \ ::= \ M@N \mid \mathsf{let}\ x = M \ \mathsf{in}\ N \qquad (1)$$

We now express the expression $\lambda x.M$ as `lam(x.M)` and $\mathsf{let}\ x = M\ \mathsf{in}\ N$ as `let(M,x.N)`. Then the computation rules of λ_C are described as follows.

```
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])) |]
```

The descriptions "`lamC = [rule|`" and "`|]`" indicate the beginning and end of the rule specification in our confluence checker PolySOL. Here the metavariables `V` and `P` represent values and non-values, respectively. To the author's best knowledge, confluence of the λ_C-calculus has not been formally proved in the literature including the original [21] and subsequent works [26, 17, 24]. We now prove confluence of the simply typed λ_C-calculus.

$$\begin{tikzcd} & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\ M_1 \arrow[dr, "*"'] & \text{CR} & M_2 \arrow[dl, "*"] \\ & N & \end{tikzcd}$$

Confluence (CR) is a property of the reduction relation, stating that any two divergent computation paths finally commute (see the right figure). Hence, the proof requires to analyse all possible situations that admit two ways of reductions, and also to check convergence of them. In the case of λ_C, careful inspection of the rules reveals that it has, in all, 6 patterns of such situations as depicted in Fig. 1. Now we see that all of these patterns are convergent. Importantly, this finite number of checks is sufficient to conclude that all other infinite numbers of instances of the divergent situations are convergent. This property is called *local confluence*, meaning that every possible one-step divergence is joinable. By applying Newman's lemma [13, 1], stating "termination and local confluence imply confluence", we can conclude that λ_C is confluent since termination of λ_C has been proved [17]. This proof method is known as Knuth and Bendix's critical pair checking [16]. The divergent terms in Fig. 1 are called *critical pairs* because these show critical situations that may break confluence. The critical pair is obtained by an *overlap* between two rules, which is computed by *second-order unification* [20]. For example, there is an overlap between the rules (β-`let-v`),(η-`let`) because the left-hand sides of them

$$\texttt{let(V, x.M[x])} \overset{?}{=} \texttt{let(L, x.x)}$$

are unifiable by second-order unification with a unifier $\theta = \{\texttt{L} \mapsto \texttt{V},\ \texttt{M} \mapsto \texttt{x.x}\}$. The instantiated term `let(V, x.x)` by θ is the source of the divergence (3:) in Fig. 1, which admits reductions by the two rules (β-`let-v`),(η-`let`).

But there are problems. In computing the critical pairs of λ_C in Fig. 1, the classical critical pair method was not applicable. Actually we need a suitable extension of the method. In the following, we list the problems, related questions and the answers we will give in this paper.

Problem 1. *The notion of unifier for an overlap is non-standard* in the call-by-value case. For example, the left-hand sides of (`let1-p`) and (`let2-v`) look overlapped, but actually are not. A candidate unifier `P` $\mapsto$ `V` is not correct because `P` is a non-value while `V` is a value.

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

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

For example, in `(assoc)`, each `let` has actually a different type (highlighted one) as:

$$\begin{aligned}
&M: c \to a,\ N: a \to b,\ L: c \ \triangleright\\
&\Gamma \vdash \mathsf{let}^{a,(a\to b)\to b}(\mathsf{let}^{c,(c\to a)\to a}(L, x^c.M[x]), y^a.N[y])\\
&\qquad \Rightarrow\ \mathsf{let}^{c,(c\to b)\to b}(L, x^c.\mathsf{let}^{a,(a\to b)\to b}(M[x], y^a.N[y])) \quad : \ b
\end{aligned}$$

To compute an overlap between `let`-terms, we need also to adjust the types of `let` to equate them.

Q2. What should be the notion of unification between polymorphic second-order terms?

Moreover, specifying all the type annotations as above manually is tedious in practice. Ideally, we write a "plain" rule as `(assoc)`, and hope that some system automatically infers the type annotations.

Q3. What is the type inference algorithm for polymorphic second-order computation rules?

In this paper, we solve these questions.

A1▶ We introduce predicates called *instance validation* on substitutions (Def. 3). It is used for formulating computation steps having some restriction of term structures. Hence they affect to the notion of critical pairs.

A2▶ We formulate the notion of unifier for polymorphic terms (Def. 7).

A3▶ We give a type inference algorithm for polymorphic computation rules in Fig. 4.

1.2 Critical pair checking using the tool PolySOL

Based on the above answers, we have implemented these features in our tool PolySOL. PolySOL is a tool to check confluence and termination of polymorphic second-order computation systems. The system PolySOL consists of about 3000 line Haskell codes, and works on the interpreter of Glasgow Haskell Compiler (tested on GHCi, version 7.6.2). PolySOL uses the feature of quasi-quotation (i.e. `[signature|..]` and `[rule|..]` are quasi-quotations) of Template Haskell [27] with a custom parser generated by Alex (for lexer) and Happy (for paper), which realises readable notation for signature, terms and rules. It makes the language of our formal computation rules available within a Haskell script.

PolySOL first infers and checks the types of variables and terms in the computation rules using a given signature. To check confluence of the simply-typed λ_C-calculus, we declare the following signature in PolySOL:

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

where `a,b` are type variables, and `Arr(a,b)` encodes the arrow type of the target λ-calculus in PolySOL. The rule set `lamC` given in the beginning of this subsection is actually a part of rule specification written using PolySOL's language. Using these, we can command PolySOL to perform critical pair checking.

```
*SOL> 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) <---
2: Overlap (eta-v)-(β -v-x)--- V|-> lam(x'.H3[x']), M'|-> z1.z2.H3[z2], V'|-> z1.z1----
   L: lam(x. V@x ) => V
   R: (lam(x'.M'[x,x'])@V'[x]) => M'[x,V'[x]]
                             lam(x.(lam(x'.H3[x'])@x))
                lam(x'.H3[x']) <-(eta-v)-∧-(beta-v-x)-> lam(x.H3[x])
                    ---> lam(x'.H3[x']) =E= lam(x.H3[x]) <---
3: Overlap (beta-let-v)-(eta-let)--- M'|-> V, M|-> z1.z1---------------------------------
   L: let(V,x.M[x])  => M[V]
   R: let(M',x'.x') => M'
                                     let(V,x.x)
                        V <-(beta-let-v)-∧-(eta-let)-> V
                                ---> V =OK= V <---
4: Overlap (eta-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]) <---
5: Overlap (assoc)-(eta-let)--- M'|-> L, M|-> z1.z1--------------------------------------
   L: let( let(L,x.M[x]) ,y.N[y]) => let(L,x.let(M[x],y.N[y]))
   R: let(M',x'.x') => M'
                               let(let(L,x.x),y.N[y])
let(L,x19.let(x19,y19.N[y19])) <-(assoc)-∧-(eta-let)-> let(L,y.N[y])
                ---> let(L,x19.N[x19]) =E= let(L,y.N[y]) <---
6: Overlap (assoc)-(assoc)--- L|-> let(L',x'.M'[x']), N'|-> z1.M[z1]---------------------
   L: let( let(L,x.M[x]) ,y.N[y]) => let(L,x.let(M[x],y.N[y]))
   R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
                           let(let(let(L',x'.M'[x']),x.M[x]),y.N[y])
let(let(L',x'.M'[x']),x.let(M[x],y.N[y])) <-(assoc)-∧
                                    -(assoc)-> let(let(L',x.let(M'[x],y'.M[y'])),y.N[y])
-> let(L',x.let(M'[x],y'.let(M[y'],y.N[y])))
                                           =E= let(L',x.let(M'[x],x'.let(M[x'],y.N[y]))) <-
#Joinable! (Total 6 CPs)
```

The above PolySOL's output corresponds to the diagrams shown in Fig. 1. The labels `L:` and `R:` indicate the rules used in the left and right paths of a divergence, and the highlight in L-rule shows that the subterm is unifiable with the root of left-hand side of R-rule. For example, in the overlap 1, the subterm `lam(x.M[x])` in the L-rule is unifiable with the term `lam(x'.(V'@x'))` in the R-rule using the unifier `M|-> z1.(V'@z1)` described at the immediate above. Then using this information, PolySOL generates the underline term `lam(x.(V'@x))@V` which exactly corresponds to the source in the first divergent diagram (1:) in Fig. 1. The lines involving ∧ (indicating "divergence") mimics the divergence diagram and the joinablity test in text. The sign =OK= denotes syntactic equal, and =E= denotes the α-equivalence.

Organisation. The paper is organised as follows. We first introduce the framework of second-order algebraic theories and computation rules in §2. We next give a type inference algorithm for polymorphic computation rules in §3. We then establish a confluence criteria based on critical pair checking in §4. In §5, we prove confluece of Maraist, Odersky, and Wadler's call-by-need λ-calculus λ_{NEED} [18] using our framework. In §6, we summarise the paper and discuss related work.

$$\frac{y : \tau \in \Gamma}{\Theta \triangleright \Gamma \vdash\ y\ : \tau} \qquad \frac{\begin{array}{c}(M : \sigma_1, \cdots, \sigma_m \to \tau) \in \Theta \\ \Theta \triangleright \Gamma \vdash\ t_i\ : \sigma_i \quad (1 \le i \le m)\end{array}}{\Theta \triangleright \Gamma \vdash\ M[t_1, \ldots, t_m]\ : \tau}$$

$$\frac{\begin{array}{c}S \triangleright f : (\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau \in \Sigma \qquad \xi : S \to \mathcal{T} \\ \Theta \triangleright \Gamma, \overline{x_i : \sigma_i} \vdash\ t_i\ : \sigma_i\xi \quad (\text{some } i \text{ s.t. } 1 \le i \le m)\end{array}}{\Theta \triangleright \Gamma \vdash\ f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t_i, \ldots, \overline{x_m^{\sigma_m}}.t_m)\ : \tau\xi}$$

Here, $\sigma \triangleq ((\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau)\xi$.

Fig. 2. Typing rules of meta-terms

2 Polymorphic Computation Rules

In this section, we introduce the framework of polymorphic second-order computation rules. It gives a formal unified framework to provide syntax, types, and computation for various simply-typed computational structure. It is a simpified framework of general polymorphic framework [11, 4] of second-order abstract syntax with metavariables [7] and its rewriting system [8, 9, 10] with molecular types [12]. The present framework introduces type variables into types and the feature of instance validation for instantiation of axioms. The polymorphism in this framework is essentially ML polymorphism, i.e., predicative and only universally quantified at the outermost and has type constructors on types.

Notation 1. We use the notation $\overline{A}$ for a sequence $A_1, \cdots, A_n$, and $|\overline{A}|$ for its length. The notation $s[u]_p$ means replacing the position p of s at with u, and $s_{|p}$ means selecting a subterm of the position p. We use the abbreviations "lhs" and "rhs" to mean left-hand side and right-hand side, respectively.

Types. We assume that $\mathcal{A}$ is a set of *atomic types* (e.g. Bool, Nat, etc.), and a set $\mathcal{V}$ of *type variables* (written as $\mathrm{s}, \mathrm{t}, \cdots$). We also assume a set of *type constructors* together with arities $n \in \mathbb{N}$, $n \ge 1$. The sets of "0-order types" $\mathcal{T}_0$ and (at most first-order) ***types*** $\mathcal{T}$ are generated by the following rules:

$$\frac{b \in \mathcal{A}}{b \in \mathcal{T}_0} \quad \frac{\mathrm{s} \in \mathcal{V}}{\mathrm{s} \in \mathcal{T}_0} \quad \frac{\tau_1, \ldots, \tau_n \in \mathcal{T}_0 \quad T\ n\text{-ary type constructor}}{T(\tau_1, \ldots, \tau_n) \in \mathcal{T}_0} \quad \frac{\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}_0}{\sigma_1, \ldots, \sigma_n \to \tau \in \mathcal{T}}$$

We call $\overline{\sigma} \to \tau$ with $|\overline{\sigma}| > 0$ a *function type*. We usually write types as $\sigma, \tau, \ldots$. A sequence of types may be empty in the above definition. The empty sequence is denoted by (), which may be omitted, e.g., $() \to \tau$, or simply τ. For example, Bool is an atomic type, $\mathsf{List}^{(1)}$ is a type constructor, and Bool $\to$ List(Bool) is a type.

Terms. A ***signature*** Σ is a set of function symbols of the form

$$\mathrm{t}_1, \ldots, \mathrm{t}_n \triangleright f : (\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau$$

where $(\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m), \tau \in \mathcal{T}$ and type variables $\mathrm{t}_1, \ldots, \mathrm{t}_n$ may occur in these types. Any function symbol is of up to second-order type. A ***metavariable*** is a variable of (at most) first-order function type, declared as $M : \overline{\sigma} \to \tau$ (written as capital letters $M, N, K, \ldots$). A ***variable*** (of a 0-order type) is written usually $x, y, \ldots$, and sometimes written x^τ when it is of type τ. The raw syntax is given as follows.

- ***Terms*** have the form $t ::= x \mid x.t \mid f(t_1, \ldots, t_n)$.
- ***Meta-terms*** extend terms to $t ::= x \mid x.t \mid f(t_1, \ldots, t_n) \mid M[t_1, \ldots, t_n]$.

The last form $M[t_1, \ldots, t_n]$, called *meta-application*, means that when we instantiate $M : \overline{a} \to b$ with a meta-term s, free variables of s (which are of types $\overline{a}$) are replaced with meta-terms $t_1, \ldots, t_n$ (cf. Def. 2). We may write $x_1, \ldots, x_n.t$ for $x_1. \cdots .x_n.t$, and we assume ordinary α-equivalence for bound variables. An equational theory is a set of proved equations deduced from a set of axioms. A metavariable context Θ is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:type in $\mathcal{T}_1$)-pairs. A judgment is of the form $\Theta \triangleright \Gamma \vdash t : b$. A ***type substitution*** $\rho : S \to \mathcal{T}$ is a mapping that assigns a type $\sigma \in \mathcal{T}$ to each type variable s in S. We write $\tau\rho$ (resp. $t\rho$) to be the one obtained from a type τ (resp. a meta-term t) by replacing each type variable in τ (resp. t) with a type using the type substitution $\rho : S \to \mathcal{T}$. A meta-term t is *well-typed* by the typing rules Fig. 2. Note that in a well-typed function term, a function symbol is annotated by its type as

$$f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t_i, \ldots, \overline{x_m^{\sigma_m}}.t_m)$$

where f has the polymorphic type $\sigma \triangleq ((\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau)\xi$. The type annotation is important in confluence checking of polymorphic rules. The notation $t\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1, \ldots, s_n$.

Definition 2. (Substitution of meta-terms for metavariables [7, 5, 6])
Let $n_i = |\overline{\tau_i}|$ and $\overline{\tau_i} = \tau_i^1, \ldots, \tau_i^{n_i}$. Suppose

$$\Theta \triangleright \Gamma', x_i^1 : \tau_i^1, \ldots, x_i^{n_i} : \tau_i^{n_i} \vdash s_i : \sigma_i \qquad (1 \le i \le k),$$
$$\Theta, M_1 : \overline{\tau_1} \to \sigma_1, \ldots, M_k : \overline{\tau_k} \to \sigma_k \triangleright \Gamma \vdash e : \tau$$

Then the substituted meta-term $\Theta \triangleright \Gamma, \Gamma' \vdash e[\,\overline{M \mapsto \overline{x}.s}\,] : \tau$ is defined by

$$\begin{aligned}
x[\,\overline{M \mapsto \overline{x}.s}\,] &\triangleq x \\
M_i[t_1, \ldots, t_{n_i}][\,\overline{M \mapsto \overline{x}.s}\,] &\triangleq s_i\,\{x_i^1 \mapsto t_1[\,\overline{M \mapsto \overline{x}.s}\,], \ldots, x_i^{n_i} \mapsto t_{n_i}[\,\overline{M \mapsto \overline{x}.s}\,]\} \\
f^{\xi}(\overline{y_1}.t_1, \ldots, \overline{y_m}.t_m)[\,\overline{M \mapsto \overline{x}.s}\,] &\triangleq f^{\xi}(\overline{y_1}.t_1\,[\,\overline{M \mapsto \overline{x}.s}\,], \ldots, \overline{y_m}.t_m\,[\,\overline{M \mapsto \overline{x}.s}\,])
\end{aligned}$$

where $[\,\overline{M \mapsto \overline{x}.s}\,]$ denotes *a substitution for metavariables* $[M_1 \mapsto \overline{x_1}.s_1, \ldots], M_k \mapsto \overline{x_k}.s_k]$.

For meta-terms $\Theta \triangleright \Gamma \vdash \ell : \tau$ and $\Theta \triangleright \Gamma \vdash r : \tau$, a ***polymorphic second-order computation rule*** (or simply ***rule***) is of the form $\Theta \triangleright \Gamma \vdash \ell \Rightarrow r : \tau$ satisfying

(i) ℓ is a higher-order pattern [20], i.e., a meta-term in which every occurrence of meta-application in ℓ is of the form $M[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n$ are distinct bound variables.
(ii) All metavariables in r appear in ℓ.

$$\text{(RuleSub)}\;\frac{\begin{array}{c} S \text{ is the set of all type variables in } \overline{\tau_i}, \overline{\sigma_i}, \tau \qquad \xi : S \to \mathcal{T} \\ \Theta \triangleright \Gamma', \overline{x_i : \tau_i} \vdash s_i : \sigma_i\xi \quad (1 \leq i \leq k) \qquad \mathsf{valid}\ [\,\overline{M \mapsto \overline{x}.s}\,] \\ (M_1 : (\overline{\tau_1} \to \sigma_1), \ldots, M_k : (\overline{\tau_k} \to \sigma_k) \triangleright \Gamma \vdash \ell \Rightarrow r : \tau) \in C \end{array}}{\Theta \triangleright \Gamma, \Gamma' \vdash \ell\xi\,[\,\overline{M \mapsto \overline{x}.s}\,] \Rightarrow_C r\xi\,[\,\overline{M \mapsto \overline{x}.s}\,] : \tau\xi}$$

$$\text{(Fun)}\;\frac{\begin{array}{c} S \triangleright f : (\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau \in \Sigma \qquad \xi : S \to \mathcal{T} \\ \Theta \triangleright \Gamma, \overline{x_i : \sigma_i} \vdash t_i \Rightarrow_C t'_i : \sigma_i\xi \quad (\text{some } i \text{ s.t. } 1 \leq i \leq m) \end{array}}{\Theta \triangleright \Gamma \vdash f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t_i, \ldots, \overline{x_m^{\sigma_m}}.t_m) \Rightarrow_C f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1, \ldots, \overline{x_i^{\sigma_i}}.t'_i, \ldots, \overline{x_m^{\sigma_m}}.t_m) : \tau\xi}$$

Here, $\sigma \triangleq ((\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau)\xi$.

Fig. 3. Polymorphic second-order computation (one-step)

Definition 3. An ***instance validation*** is an arbitrary predicate valid that takes a substitution $[\,\overline{M \mapsto \overline{x}.s}\,]$ for metavariables and returns true or false. In this paper, we use the following various instance validations (but not limited to these).

- **Any:** $\mathsf{valid}\ \theta \overset{\text{def}}{\Leftrightarrow}$ always true, i.e. any substitution is valid.
- **Injectivity:** $\mathsf{valid}\ \theta \overset{\text{def}}{\Leftrightarrow}$ θ is an injective substitution (i.e. different metavariables must map to different meta-terms).
- **Values/non-values:** $\mathsf{valid}\ [M_1 \mapsto \overline{x_1}.s_1, \ldots, M_n \mapsto \overline{x_n}.s_n]$
  $\overset{\text{def}}{\Leftrightarrow}$ $((M_i \equiv V \Rightarrow s_i \text{ is a value})\ \&\ (M_i \equiv P \Rightarrow s_i \text{ is a non-value}))$ for all $i = 1, \ldots, n$. Here, the notation "$M_i \equiv V$" means that the metavariable M_i's letter is "V". The definitions of values and non-values should be given separately. For the case of λ_C-calculus, we take the definition (1) for values and non-values in §1.1.

A ***(polymorphic second-order) computation system*** is a triple $(\Sigma, C, \mathsf{valid})$ consisting of a signature Σ, a set C of rules, and an instance validation valid. We write $s \Rightarrow_C t$ to be one-step computation using $(\Sigma, C, \mathsf{valid})$ obtained by the inference system given in Fig. 3. The (RuleSub) instantiates a polymorphic computation rule $\ell \Rightarrow r$ in C by substitution $[\,\overline{M \mapsto \overline{x}.s}\,]$ of meta-terms for metavariables and substitution ξ on types, where the predicate valid checks whether $[\,\overline{M \mapsto \overline{x}.s}\,]$ is applicable. The (Fun) means that the computation step is closed under polymorphic function symbol contexts.

Example 4. (Decidable equality) The injectivity validation in Def. 3 is useful in formulating a system involving the notion of "names", such as π-calculus. We define the signature Σ by

$$\triangleright\, \mathsf{eq} : \mathsf{Name}, \mathsf{Name} \to \mathsf{Bool} \qquad \triangleright\, \mathsf{a,b,c} : \mathsf{Name} \qquad \triangleright\, \mathsf{false}, \mathsf{true} : \mathsf{Bool}$$

and the rules by

$$\begin{array}{lll} (\text{eq1}) & X : \mathsf{Name} & \triangleright \vdash \mathsf{eq}(X, X) \Rightarrow \mathsf{true} : \mathsf{Bool} \\ (\text{eq0}) & X, Y : \mathsf{Name} & \triangleright \vdash \mathsf{eq}(X, Y) \Rightarrow \mathsf{false} : \mathsf{Bool} \end{array}$$

We set the predicate valid to be **Injectivity**. Then we have expected computation, such as $\mathsf{eq(a,b)} \Rightarrow_C \mathsf{false}$, $\mathsf{eq(b,b)} \Rightarrow_C \mathsf{true}$. Without the instance validation, the rules

```
$\mathcal{W}(\Sigma, x)$              = if $x : \tau$ appears in $\Sigma$ then $([],\ \triangleright x^{\tau} : \tau)$ else $error$
$\mathcal{W}(\Sigma, x.t)$            = let $a$ = freshVar
                                          $(\theta',\ \Theta\ \triangleright\ t' : \tau') = \mathcal{W}(\{x : a\} \cup \Sigma, t)$
                                        in $(\theta',\ \Theta\ \triangleright\ x^{a}.t' : a \to \tau')$
$\mathcal{W}(\Sigma, f(\overline{t}))$          = if $f : \overline{d} \to c$ appears in $\Sigma$ then
                                        let $n$ = newNum in
                                          $(\overline{d'} \to c')$ = attach the index $n$ to all type vars in $(\overline{d} \to c)$
                                          $(\theta, \Theta,\ \overline{u}, \overline{a})$ = foldr $(\mathcal{W}_{\mathsf{iter}}\Sigma)$ $([], [], [], [])$ $\overline{t}$
                                          $b$ = freshVar
                                          $\theta'$ = unify$((\overline{a} \to b)\theta,\ \overline{d'} \to c')$
                                        in $(\theta' \circ \theta,\ \{f_n : (\overline{d'} \to c')\} \cup \Theta\ \triangleright\ f_n(\overline{u}) : b)$
                                       else $error$
$\mathcal{W}(\Sigma, M[\overline{t}])$         = let $(\theta, \Theta, \overline{u}, \overline{a})$ = foldr $(\mathcal{W}_{\mathsf{iter}}\Sigma)$ $([], [], [], [])$ $\overline{t}$
                                          $b$ = freshVar
                                        in $(\theta,\ \{M : \overline{a} \to b\} \cup \Theta\ \triangleright\ M[\overline{u}] : b)$

$\mathcal{W}_{\mathsf{iter}}\Sigma\,(t,\ (\theta_0, \Theta_0, \overline{u}, \overline{\tau}))$ = let $(\theta,\ \Theta\ \triangleright\ u : \tau) = \mathcal{W}(\Sigma, t)$
                                        in $(\theta \circ \theta_0,\ \Theta \cup \Theta_0, (u, \overline{u}), (\tau, \overline{\tau}))$

mkMatch$(\Theta)$            = $\{(\sigma, \tau) \mid (M : \sigma) \in \Theta,\ (M : \tau) \in \Theta, \sigma \neq \tau\}$

infer$(\Sigma, t)$             = let $(\theta,\ \Theta\ \triangleright\ u : \tau) = \mathcal{W}(\Sigma, t)$
                                          $\theta'$ = unify(mkMatch$(\Theta\theta)$) $\circ\ \theta$
                                        in $\Theta\theta'\ \triangleright\ u\theta' : \tau\theta'$
infer$(\Sigma,\ s \Rightarrow t)$        = infer$(\{$rule : s, s $\to$ T$\} \cup \Sigma$, rule$(s, t))$
```

- freshVar returns a new type variable
- newNum returns a new number (or by counting up the stored number)
- foldr is the usual "foldr" function for the sequence of terms (regarded as a list) to repeatedly apply the functionm $\mathcal{W}$ by the function $\mathcal{W}_{\mathsf{iter}}$
- unify returns the most general unifer of the pairs of types.
- "[]" denotes the empty sequence or substitution.

Fig. 4. Type inference algorithm

become meaningless because they admits an non-injective substitution $\{X \mapsto \mathsf{a}, Y \mapsto \mathsf{a}\}$, which entails eq(a,a) $\Rightarrow_C$ false. ■

3 Type Inference for Polymorphic Computation Rules

We have formulated that polymorphic computation rules were explicitly typed as Example 4. But when we give an implementation of confluence/termination checker, to insist that the user writes fully-annotated type and context information for computation rules is not a good system design. Hence we give a type inference algorithm. In the case of λ_C-calculus, the user only provides the signature `siglamC` and "plain" rules `lamC` in §1.1. The type inference algorithm infers the missing context and type annotations (highlights) as:

$$M\colon \mathrm{S} \to \mathrm{T},\ N\colon \mathrm{S} \ \triangleright\ \vdash \mathsf{app}^{\mathsf{Arr}(\mathrm{S},\mathrm{T}),\,\mathrm{S}\to\mathrm{T}}(\,\mathsf{lam}^{(\mathrm{S}\to\mathrm{T})\to\mathsf{Arr}(\mathrm{S},\mathrm{T})}(x^{\mathrm{S}}.\,M[x]),N\,) \Rightarrow M[N]\ :\ \mathrm{T}$$

These annotations are important for checking confluence of polymorphic rules in computing overlapping between rules.

Algorithm. Our algorithm is given in Fig. 4, which is a modification of Damas-Milner type inference algorithm W [3]. It has several modifications to cope with the language of meta-terms and to return enough type information for confluence checking. The algorithm takes a signature Σ and an un-annotated meta-term t. A sub-function $\mathcal{W}$ returns $(\theta,\ \Theta\ \triangleright u : \tau)$, which is a pair of type substitution θ and an inferred judgment. The types in it are still need to be unified. The context Θ may contain unifiable declarations, such as $M : \sigma$ and $M : \tau$ with $\sigma \neq \tau$, and these σ and τ should be unified. The main function $\mathsf{infer}(\Sigma, t)$ does it, and returns the form

$$\Theta\ \triangleright t' : \tau.$$

The meta-term t' is a renamed t, where every function symbol f in the original t now has a unique index as f_n, and Θ is the set of inferred type declarations for f_n's and all the metavariables occurring in t'. Similarly, for a given plain rule $s \Rightarrow t$, the function $\mathsf{infer}(\Sigma, s \Rightarrow t)$ returns $\Theta \triangleright s' \Rightarrow t' : \tau$, where Θ is an inferred context and corresponding renamed terms s', t' as the sole term case. This is realised as inferring types for a meta-term to implement a rule using the new binary function symbol rule (see the definition of $\mathsf{infer}(\Sigma,\ s \Rightarrow t)$).

We denote by $|\,t\,|$ a meta-term obtained from t by erasing all type annotations in the variables and the function symbols of t. We use the usual notion of "more general" relation on substitutions, denoted by $\tau' \geq \tau$, if there exists a substitution σ such that $\sigma \circ \tau' = \tau$.

Theorem 5. **(Soundness)** If $\mathsf{infer}(\Sigma, t) = (\Theta\ \triangleright t' : \tau)$, then there exists Γ such that $\Theta \triangleright \Gamma \vdash t' : \tau$.

Theorem 6. **(Completeness)** If $\Theta\ \triangleright \Gamma \vdash t : \tau$ holds and $\mathsf{infer}(\Sigma, |\,t\,|) = (\Theta' \triangleright t' : \tau')$, then $\tau' \geq \tau$ and

- If $M : \sigma \in \Theta$ then, there exists $M : \sigma' \in \Theta'$ such that $\sigma' \geq \sigma$
- If $f^{\overline{\sigma}\to\tau}$ occurs in t, then there exists $f_n : \overline{\sigma'} \to \tau' \in \Theta'$ such that f_n occurs in t' at the same position as t, and $\overline{\sigma}' \to \tau' \geq \overline{\sigma} \to \tau$.

The reason why our algorithm attaches an index n to each occurence of a function symbol f as "f_n" is to distinguish different occurrences of the same f in a term, and to correctly infer the type of each of them (see Prob. 2 in §1.1). If we have $\mathsf{infer}(\Sigma, t) = (\Theta\ \triangleright t' : \tau)$, then we can fully annotate types for the plain term t. We can pick the type of each function symbol in t by finding $f_n : \overline{\sigma'} \to \tau' \in \Theta$, which means that this f has the inferred type $\overline{\sigma'} \to \tau'$.

4 Confluence of Polymorphic Computation Systems

In this section, we establish a confluence criterion of polymorphic computation systems based on critical pair checking. For a computation system C, we regard " $\Rightarrow_C$ " defined by Fig. 3 as a binary relation on well-typed terms. Moreover, we write $\Rightarrow_C^*$ for the reflexive transitive closure, $\Rightarrow_C^+$ for the transitive closure, and $\Leftarrow_C$ for the converse of $\Rightarrow_C$, respectively. We say:

1. $a, b \in A$ are *joinable*, written $a \downarrow b$, if $\exists c \in A.\, a \Rightarrow_C^* c \;\;\&\;\; b \Rightarrow_C^* c$.
2. $\Rightarrow_C$ is *confluent* if $\forall a, b \in A.\; a \Rightarrow_C^* b \;\;\&\;\; a \Rightarrow_C^* c$ implies $b \downarrow c$.
3. $\Rightarrow_C$ is *locally confluent* if $\forall a, b \in A.\; a \Rightarrow_C b \;\;\&\;\; a \Rightarrow_C c$ implies $b \downarrow c$.
4. $\Rightarrow_C$ is *strongly normalising (SN)* if $\forall a \in A$, there is no infinite sequence $a \Rightarrow_C a_1 \Rightarrow_C a_2 \Rightarrow_C \cdots$.

We call a meta-term linear if no metavariable occurs more than once, and C is *left-linear* if for every $\ell \Rightarrow r$ in C, ℓ is linear.

Notion of unifier between two polymorphic meta-terms. To compute critical pairs, we need to compute overlapping between rules using second-order unification. Ordinary unifier between terms s and t is a substitution θ of terms for variables that makes $s\theta = t\theta$. In case of polymorphic second-order algebraic theory, we should also take into account of types. For example, what should be a unifier between the following terms?

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

Here T, V are type variables. These terms are unifiable by a substitution of meta-terms $\theta : M \mapsto x^{\mathsf{bool}}.\mathsf{g}^{\mathsf{bool}\to\mathsf{int}}(x)$ *together with a type substitution* $\xi : \mathrm{V} \mapsto \mathsf{bool}, \mathrm{T} \mapsto \mathsf{int}$. Therefore, we define:

Definition 7. A *unifier between meta-terms* s, t is a pair (θ, ξ) such that $s\xi\theta = t\xi\theta$, where ξ is a substitution of types for type variables, and θ is a substitution of metaterms for metavariables.

Critical pairs of polymorphic computation systems. We now formulate the notion of critical pairs for our polymorphic case. We first recall basic notions. A position p is a finite sequence of natural numbers. The empty sequence ε is the root position. Given a meta-term t, $t_{|p}$ denotes a subterm of t at a position p.Suppose a computation system C is given. We say two rules $l_1 \Rightarrow r_1, l_2 \Rightarrow r_2$ in C are *variant* if $l_1 \Rightarrow r_1$ is obtained by injectively renaming variables and metavariables of $l_2 \Rightarrow r_2$. We say that a position p in a term t is a *metavariable position* if $t_{|p}$ is a metavariable or meta-application.

Definition 8. An *overlap* between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ of a computation system $(\Sigma, C, \mathsf{valid})$ is a tuple $\langle l_1 \Rightarrow r_1, p, l_2 \Rightarrow r_2, \theta, \xi\rangle$ satisfying the following properties:

- $l_1 \Rightarrow r_1,\; l_2 \Rightarrow r_2$ are variants of rules in C without common (meta)variables.
- p is a non-metavariable position of l_1.
- If p is the root position, $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ are not variants.
- (θ, ξ) is a unifier between $l_{1|p}$ and l_2 such that **valid**(θ) **holds.**

The component θ in an overlap is intended to be the output of the pattern unification algorithm [20]. An overlap represents an overlapping situation of computation, meaning that the term t is rewritten to the two different ways. We define $\mathsf{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \triangleq \{$all possible overlaps between $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2\}$. We collect all the overlaps in C by $O \triangleq \bigcup\{\mathsf{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \mid l_1 \Rightarrow r_1,\, l_2 \Rightarrow r_2 \in C\}$.

Definition 9. The ***critical pair (CP)*** generated from an overlap $\langle \ell_1 \Rightarrow r_1,\, p,\, \ell_2 \Rightarrow r_2, \theta, \xi\rangle$ is a triple $\langle r_1',\ \ t,\ \ r_2'\rangle$ where $t = \ell_1\xi\theta$ and $t \ \Rightarrow_C \ r_1'$ which rewrites the root position of t using $\ell_1 \Rightarrow r_1$, and $t \ \Rightarrow_C \ r_2'$ which rewrites the position p of t using $\ell_2 \Rightarrow r_2$.

Then, we obtain the critical pairs of C by collecting all the critical pairs generated from overlaps in O.

Proposition 10. Let $(\Sigma, C, \mathsf{valid})$ be a polymorphic second-order computation system. Suppose C is left-linear. If for every critical pair $\langle t, u, t'\rangle$ of $(\Sigma, C, \mathsf{valid})$, we have $t \downarrow t'$, then $\Rightarrow_C$ is locally confluent.

Theorem 11. Let $(\Sigma, C, \mathsf{valid})$ be a polymorphic second-order algebraic theory. Assume that C is left-linear and strongly normalising. If for every critical pair $\langle t, u, t'\rangle$ of $(\Sigma, C, \mathsf{valid})$, we have $t \downarrow t'$, then $\Rightarrow_C$ is confluent.
Proof. By Prop. 10 and Newman's lemma.

5 Example: Confluence of the Call-by-Need λ-Calculus

We examine confluece of Maraist, Odersky, and Wadler's call-by-need λ-calculus λ_{NEED} [18]. We consider the simply-typed version of it. The signature is:

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

which consists of the function symbol `lam` for λ-abstraction, app (also written as an infix operator `@` below) for application, and `let`-construct. We represent the arrow types of the "object-level" λ_{NEED}-calculus by the binary type constructor `Arr`, and use the function types `a->b` of the "meta-level" polymorphic second-order computation system to represent binders, where `a,b` are type variables. The λ_{NEED} has five rules, which are straightforwardly defined in PolySOL as:

```
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])) |]
```

We also impose the distinction of values and non-values as in the λ_C-calculus. Here `V` is a metavariable for values, and `M,N,C,L` are metavariables for all terms. We choose the instance validation valid to be **Values/non-values** in Def. 3. We can tell it to PolySOL by writing the suffix "`-v`" in the labels `(rV-v),(rC-v)`. We command PolySOL to perform critical pair checking.

```
*PolySOL> cri lmdNeed sigNeed
1: Overlap (rG)-(rA)--- M|-> let(L',x'.M'[x']), N'|-> z1.N------------------------------
   L:  let(M,x.N)  => N
   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.N)
                                N <-(rG)-∧-(rA)-> let(L',xd2.let(M'[xd2],yd2.N))
                                 ---> N =OK= N <---
2: Overlap (rC-v)-(rG)--- M'|-> V, M|-> z1.N'-------------------------------------------
   L:  let(V,x.M[x])@N => let(V,x.(M[x]@N))
   R: let(M',x'.N') => N'
                                  (let(V,x.N')@N)
               let(V,x5.(N'@N)) <-(rC-v)-∧-(rG)-> (N'@N)
                           ---> (N'@N) =OK= (N'@N) <---
3: Overlap (rC-v)-(rV-v)--- V'|-> V, C'|-> z1.M[z1]-------------------------------------
   L:  let(V,x.M[x])@N => let(V,x.(M[x]@N))
   R: let(V',x'.C'[x']) => C'[V']
                                 (let(V,x.M[x])@N)
            let(V,x7.(M[x7]@N)) <-(rC-v)-∧-(rV-v)-> (M[V]@N)
                          ---> (M[V]@N) =OK= (M[V]@N) <---
4: Overlap (rA)-(rG)--- M'|-> L, M|-> z1.N'---------------------------------------------
   L: let( let(L,x.M[x]) ,y.N[y]) => let(L,x.let(M[x],y.N[y]))
   R: let(M',x'.N') => N'
                              let(let(L,x.N'),y.N[y])
    let(L,x10.let(N',y10.N[y10])) <-(rA)-∧-(rG)-> let(N',y.N[y])
             ---> let(N',y10.N[y10]) =E= let(N',y.N[y]) <---
5: Overlap (rA)-(rA)--- L|-> let(L',x'.M'[x']), N'|-> z1.M[z1]--------------------------
   L: let( let(L,x.M[x]) ,y.N[y]) => let(L,x.let(M[x],y.N[y]))
   R: let(let(L',x'.M'[x']),y'.N'[y']) => let(L',x'.let(M'[x'],y'.N'[y']))
                     let(let(let(L',x'.M'[x']),x.M[x]),y.N[y])
let(let(L',x'.M'[x']),x.let(M[x],y.N[y])) <-(rA)-∧
                                        -(rA)-> let(let(L',x'.let(M'[x'],y'.M[y'])),y.N[y])
---> let(L',x.let(M'[x],y.let(M[y],y.N[y])))
                                         =E= let(L',x'.let(M'[x'],x.let(M[x],y.N[y]))) <---
#Joinable! (Total 5 CPs)
```

PolySOL reports that there are 5 critical pairs, and all are successfully joinable. This shows local confluence. Strong normalisation of λ_{NEED} can be shown by a translation as the treatment of λ_{C} in [24]. Hence we conclude that λ_{NEED} is confluent. In [18], confluence of untyped λ_{NEED} has been established by a different proof method, i.e., analysis of developments steps in the λ-calculus [2]. This method is somewhat specific to the case of λ-calculus. In contrast to it, our approach is general rewriting theoretic, and not specific to variants of λ-calculus, i.e., based on critical pair checking of computation rules.

6 Summary and Related Work

6.1 Summary

We have presented a new framework of polymorphic computation rules having predicates to restrict their instances. It was shown to be suitable for formulating and analysing fundamental calculi of programming languages. We have given a type inference algorithm and a criteria to check confluence property of polymorphic rules. These have given a handy method to prove confluence of second-order computation rules. We have demonstrated effectiveness of our methodology by examining sample program calculi using our framework.

6.2 Related Work

Nipkow has studied critical pairs for confluence of higher-order rewrite systems [23],[19]. The rewrite rule format there was rules on simply-typed λ-terms modulo $\beta\eta$-equivalence, and there are *no* polymorphic types nor instance validation. Therefore, none of our examples (λ_C,λ_{NEED}, and decidable equality) and the confluence shown in the present paper can be directly formulated and checked in Nipkow's framework.

There are systems of automatic checking of confluence of Nipkow's higher-order rule format, such as ACPH [25] and CSIˆho [22]. Because these are based on Nipkow's format, these tool do not have the features of polymorphic types nor instance validation. Hence, all of our examples are beyond the scope of the existing confluence checking systems. In this respect, to the best of the author's knowledge, our system is the first automatic tool that can check confluence of the call-by-value variants of the λ-calculus directly, as shown in the paper.

A framework of polymorphic higher-order rewrite rules was given [14, 15], and our framework is similar but there are several differences in the foundations, e.g., our framework is based on (polymorphic) second-order algebraic theories [6, 4], while their is based on a polymorphic λ-calculus. The main purpose of [14, 15] was to establish termination criterion of higher-order rewrite rules. The issue of confluence of polymorphic rules has remain untouched. To the best of the author's knowledge, the present paper is probably the first study of confluence of general kind of polymorphic second-order rewrite rules (N.B. this is not about the polymorphic λ-calculus).

We gave a type inference algorithm of polymorphic computation rules, which has not been given in the context of rewriting theory (while it may be standard in the context of the theory of programming languages). Lack of a suitable type inference algorithm in the theory of rewriting has affected to the existing higher-order confluence tools such as CSIˆho and ACPH. These tools force the user to write more detailed type information in rewrite rule specifications than PolySOL. The user need to declear all free and bound variables with their types used in the rules. PolySOL's rule specification is simpler due to the type inference. Our algorithm may also be beneficial to other tools to improve this situation.

In [12], the present author developed a simply-typed framework of second-order equational logic and computation rules, and gave a tool SOL for checking methods of termination confluence. It lacked proper polymorphism and instance validation, hence the examples considered in the present paper could not be handled. Note that Plotkin's call-by-value λ-calculus could be formulated in [12] by explicitly modifying the rules by meta-programming like method to reflect value variables. But this approach works only for small calculi, such as the call-by-value λ-calculus, and for larger calculi, such as λ_C and λ_{NEED}, the number of overlaps explodes and we cannot hope to manage it.

In [4], we gave a general framework of multiversal polymorphic algebraic theories and their algebraic models. It admits multiple type universes and higher-kinded polymorphic types, hence it is richer than the present setting. The model theory encompasses the present simpler framework. But in [4], we did not develop neither polymorphic computation rules, confluence, nor instance validation.

Acknowledgments. I am grateful to Masahito Hasegawa for his question on how to check confluence of Moggi's computational λ-calculus by my previous tool SOL. It opened my eyes to the necessity of proper treatment of call-by-value calculi and led me to the notion of instance validation developped in this paper.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.
[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. POPL'82*, pages 207–212, 1982.
[4] M. Fiore and M. Hamana. Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, pages 520–529, 2013.
[5] M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.
[6] M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.
[7] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Proc. of APLAS'04*, LNCS 3302, pages 348–363, 2004.
[8] M. Hamana. Universal algebra for termination of higher-order rewriting. In *Proc. of RTA'05*, LNCS 3467, pages 135–149, 2005.
[9] M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of PPDP'07*, pages 97–108. ACM Press, 2007.
[10] M. Hamana. Semantic labelling for proving termination of combinatory reduction systems. In *Proc. WFLP'09*, LNCS 5979, pages 62–78, 2010.
[11] M. Hamana. Polymorphic abstract syntax via Grothendieck construction. In *FoSSaCS'11*, LNCS3467, pages 381–395, 2011.
[12] M. Hamana. How to prove your calculus is decidable: Practical applications of second-order algebraic theories and computation. *Proceedings of the ACM on Programming Languages*, 1(22):1–28, September 2017.
[13] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980.
[14] J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *Journal of ACM*, 54(1):2:1–2:48, 2007.
[15] J.-P. Jouannaud and A. Rubio. Normal higher-order termination. *ACM Trans. Comput. Log.*, 16(2):13:1–13:38, 2015.
[16] D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problem in abstract algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
[17] S. Lindley and I. Stark. Reducibility and $\top\top$-lifting for computation types. In *Proc. of TLCA'05*, pages 262–277, 2005.
[18] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.
[19] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 192(1):3–29, 1998.
[20] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
[21] E. Moggi. Computational lambda-calculus and monads. LFCS ECS-LFCS-88-66, University of Edinburgh, 1988.

[22] J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – a progress report. In *Proc. of CADE'17*, LNCS (LNAI) 10395, pages 385–397, 2017.

[23] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.

[24] Y. Ohta and M. Hasegawa. A terminating and confluent linear lambda calculus. In *Proc. of RTA'06*, pages 166–180, 2006.

[25] K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description. In *6th Confluence Competition (CoCo 2017)*, 2017.

[26] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

[27] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Proc. Haskell Workshop 2002*, 2002.

[28] S. Staton. An algebraic presentation of predicate logic. In *Proc. of FOSSACS 201*, pages 401–417, 2013.

[29] S. Staton. Instances of computational effects: An algebraic perspective. In *Proc. of LICS'13*, page 519, 2013.

[30] S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of POPL'15*, pages 395–406, 2015.