Science of Computer Programming 187 (2020) 102322

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Polymorphic computation systems: Theory and practice of confluence with call-by-value

Makoto Hamana [a,*], Tatsuya Abe [b], Kentaro Kikuchi [c]

[a] Department of Computer Science, Gunma University, Japan
[b] STAIR Lab, Chiba Institute of Technology, Japan
[c] Research Institute of Electrical Communication, Tohoku University, Japan

ARTICLE INFO

Article history:
Received 21 November 2018
Received in revised form 7 September 2019
Accepted 19 September 2019
Available online 17 October 2019

Keywords:
Polymorphism
λ-calculus
Type inference
Second-order algebraic theory
Confluence

ABSTRACT

We present a new framework of polymorphic computation rules that can accommodate a distinction between values and non-values. It is suitable for analysing fundamental calculi of programming languages. We develop a type inference algorithm and new criteria to check the confluence property. These techniques are then implemented in our automated confluence checking tool PolySOL. Its effectiveness is demonstrated through examination of various calculi, including the call-by-need lambda-calculus, Moggi's computational lambda-calculus, and skew-monoidal categories.

1. Introduction

1.1. How to formalise a concrete calculus

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 *polymorphically* 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 neither σ nor τ is a fixed type, but each is a schema of types. Therefore, (β) actually describes a *family of actual computation rules*. It represents various instances of rules by varying σ and τ, such as the following.

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

From the viewpoint of meta-theory, as in a mechanised formalisation of mathematics, the (β)-rule should be formulated in a polymorphically 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.

* Corresponding author.
E-mail addresses: hamana@cs.gunma-u.ac.jp (M. Hamana), abet@stair.center (T. Abe), kentaro.kikuchi@riec.tohoku.ac.jp (K. Kikuchi).

https://doi.org/10.1016/j.scico.2019.102322
0167-6423/

We were aware of this difficulty. Our earlier study [1] 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 means of instantiating types by replacing type variables with other types. Therefore, instantiation of (β) to $(\beta_{\mathsf{bool},\mathsf{int}})(\beta_{\mathsf{int}\to\mathsf{int},\mathsf{bool}})$ described above was unobtainable. For that reason, no confluence of instances of polymorphic computation rules is obtained automatically. Further manual meta-theoretic analysis is necessary.

1.2. A new framework of polymorphic computation rules with values and non-values distinction

To resolve these issues, we present an extended framework for polymorphic computation rules herein. To make the framework applicable to describing and analysing fundamental calculi of programming languages, we also introduce a distinction between values and non-values in the framework.

This framework is polymorphic and a computational refinement of second-order algebraic theories by Fiore et al. [2, 3]. Second-order algebraic theories have been shown to be a useful framework that models various important notions of programming languages such as logic programming [4], algebraic effects [5], and quantum computation [6]. The present polymorphic framework also has applications in these fields.

1.3. Differences from ordinary untyped rewriting systems

This new framework is not merely an instance of ordinary untyped rewriting systems [7–9]. One cannot simply apply existing theory to the new typed framework. The framework introduces new features: polymorphic types and values/non-values distinctions in the rule format. We provide suitable adaptation and extension of the theory and techniques of rewriting.

1.3.1. Polymorphic typed rules and their confluence

The following examples underscore that the confluence property in the present setting differs from that in the untyped setting. Let $\mathsf{ck} : a \to \mathsf{bool}$ be a polymorphic function symbol, where a is a type variable. Moreover, atomic types $\mathsf{int}, \mathsf{char}, \mathsf{bool}$ and constants $\mathsf{true}, \mathsf{false}$ are assumed. The following rules use different instances of the polymorphic function symbol ck.

$$\begin{array}{lllllll} (1) & x : \mathsf{int} & \vdash \mathsf{ck}^{\mathsf{int}\to\mathsf{bool}}(x) & \Rightarrow & \mathsf{true} & : & \mathsf{bool} \\ (2) & x : \mathsf{char} & \vdash \mathsf{ck}^{\mathsf{char}\to\mathsf{bool}}(x) & \Rightarrow & \mathsf{false} & : & \mathsf{bool} \end{array}$$

These formulate the function ck as a checking function of whether the type of the argument is of int or not. These are written in our formal framework we introduce into §2. The system is confluent because there is no overlapping and it is terminating.

However, the corresponding *untyped rules* are non-confluent. Forgetting types, we have the following rules as

$$\begin{array}{lllll} (1) & \mathsf{ck}(x) & \Rightarrow & \mathsf{true} \\ (2) & \mathsf{ck}(x) & \Rightarrow & \mathsf{false} \end{array}$$

where ck is a unary untyped function symbol. As one might expect, the untyped system is not confluent because $\mathsf{ck}(x)$ is rewritten to two normal forms: true and false. This result shows that the types are important and that they affect the notion of overlap between rules, which is a core notion of critical pair checking.

1.3.2. Values and non-values

The notion of values is important in programming languages. Not only call-by-value programming languages such as ML, but functional programming languages having distinction of pure functions and effectful computations, such as Haskell, have the notion of (pure) values.

Plotkin's call-by-value λ-calculus [10] is the most fundamental λ-calculus having the notion of values. This λ-calculus is defined by a version of β-reduction law as

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

where V denotes a value that is either a variable or an abstraction

$$\text{Values} \qquad V ::= y \mid \lambda w.M$$

Therefore, the metavariable V is only instantiated by a term in the particular subclass of terms, called values. The ordinary theory of term rewriting does not allow such syntactic restrictions on the form of terms. Because the restriction of terms reflects the notion of substitutions of terms for variables, we cannot apply the ordinary theory directly to investigate confluence of call-by-value calculi. We will introduce the following new notions for it:

1. Overlaps for rules having metavariables for values and non-values (Definition 4.3)
2. Call-by-value joinability (Definition 4.5)
3. Meta-confluence and object confluence (§4.5)

The first two notions are necessary to establish confluence in the call-by-value setting. The third notion is important for establishing confluence of object λ-calculi. Meta-confluence is a confluence property at the meta-level, i.e., confluence on "terms with metavariables", which are wider than the terms with no metavariables. In §6, it is explained that the polymorphic computation system of Moggi's computational λ-calculus, the λ_{C}-calculus, is not meta-confluent. Nevertheless, its object confluence can be proved using the notion of call-by-value joinability.

1.4. Example 1: confluence of the call-by-need λ-calculus

Examination of a sample confluence problem is useful to illustrate our framework and methodology. Here we consider Maraist, Odersky, and Wadler's call-by-need λ-calculus λ_{NEED} [11]. Its simply-typed version is explained below. The λ_{NEED} has two classes of terms: values and non-values.

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

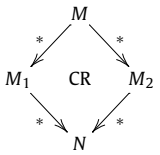
We now express the expression $\lambda x.M$ as `lam(x.M)` and $\mathsf{let}\, x = M \,\mathsf{in}\, N$ as `let(M,x.N)`. Then the computation rules of λ_{NEED} are described as shown below:

```
lmdNeed = [rule|
 (G)    let(M, x.N)                => N
 (I)    lam(x.M[x]) @ N            => let(N,x.M[x])
 (V-v)  let(V, x.C[x])             => C[V]
 (C-v)  let(V, x.M[x])@N           => let(V, x.M[x]@N)
 (A)    let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y])) |]
```

Descriptions "`lmdNeed = [rule|`" and "`|]`" present the beginning and end of the rule specification in our confluence checker PolySOL. Here `V` is a metavariable for values. Also, `M`,`N`,`C`,`L` are metavariables for all terms.

Remark 1.1. The value metavariable `V` cannot be replaced with non-values. That is, `V` can only be replaced with variables or abstractions. This fundamental assumption is important for a formal description of a calculus having the notion of values/non-values, such as the call-by-need λ-calculus and the computational λ-calculus (see §6). □

Next, confluence of the simply-typed λ_{NEED}-calculus is proved. Confluence (CR) is a property of the reduction relation, stating that any two divergent computation paths are finally joinable, as shown in the panel:



The proof requires analysis of all possible situations that admit two ways of reductions, and also to check their convergence. In the case of λ_{NEED}, careful inspection of the rules reveals that it has, in all, seven patterns of such situations, as depicted in Fig. 1. Next it is apparent that all of these patterns are convergent. Importantly, this finite number of checks is sufficient to infer 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 [12,7], which states that "termination and local confluence imply confluence", we infer that λ_{NEED} is confluent because termination (i.e. strong normalisation) of λ_{NEED} can be shown by a translation into a terminating λ-calculus, as Ohta et al. have described [13]. Therefore, we conclude that λ_{NEED} is confluent.

This proof method is known as Knuth and Bendix's critical pair checking [14]. The divergent terms in Fig. 1 are designated as *critical pairs* because these show critical situations that may break confluence. A critical pair is obtained by an *overlap* between two rules, which is computed using *second-order unification* [15]. For instance, an overlap exists between the rules `(V-v)`, `(C-v)` because their left-hand sides

$$\texttt{let(V, x.C[x])} \stackrel{?}{=} \texttt{let(V', x.M[x])}$$

are unifiable by second-order unification with a unifier $\theta = \{\texttt{V} \mapsto \texttt{V'},\ \texttt{C} \mapsto \texttt{x.M[x]}\}$. The instantiated term `let(V', x.M[x])@N` by θ is the source of the divergence (4:) in Fig. 1, which admits reductions by the two rules `(C-v)`, `(V-v)`.

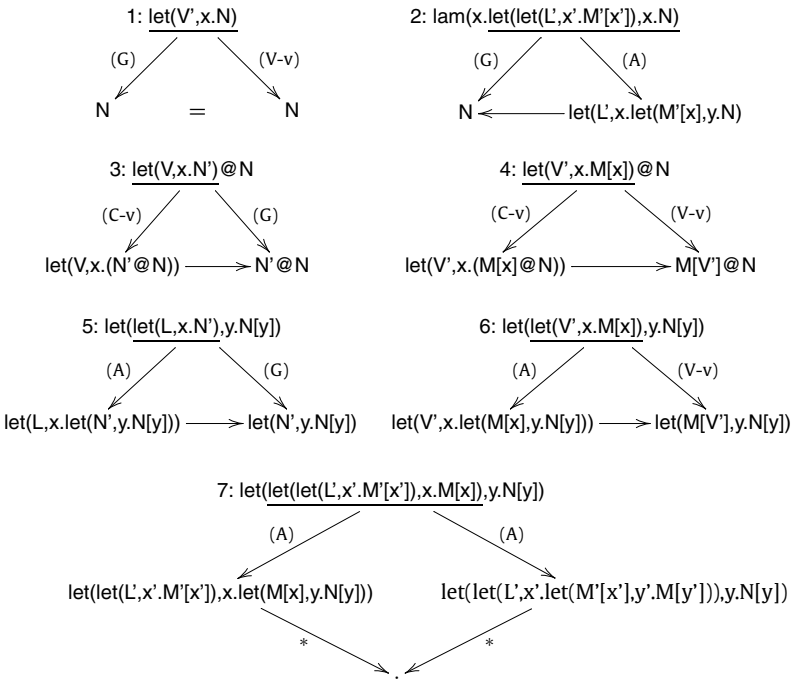


Fig. 1. Critical pairs of λ_{NEED}-calculus.

Nevertheless, some difficulties arise. In computing the critical pairs of λ_{NEED} in Fig. 1, the classical critical pair method is not applicable. Actually, suitable extensions of the method are required. In the following herein, we list the problems, related questions, and the answers we obtained.

Problem 1. *The notion of a unifier for an overlap is non-standard* in the call-by-value case. For example, the left-hand sides of `(V-v)` and `(A)` appear to be overlapped, but actually they are not. A candidate unifier $\mathtt{V} \mapsto \mathtt{let(L,x.M[x])}, \mathtt{C} \mapsto \mathtt{y.N[y]}$ is not correct because `V` is a value, whereas `let(L,x.M[x])` is a non-value.

Q1. What is a general definition of overlaps in the call-by-value setting?

Problem 2. *Different occurrences of the same function symbol might have different types.*
For example, in `(A)`, each `let` has actually a different type (highlighted one) as

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

Computing an overlap between `let`-terms demands adjustment of 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 described above manually is tedious in practice. Ideally, we write a "plain" rule as `(A)`, and expect that some system infers the type annotations automatically.

Q3. What is the type inference algorithm for polymorphic second-order computation rules?
As described in this paper, we solve these questions.

A1. We introduce a *validation* predicate on substitutions (Definition 2.5) to check value/non-value restrictions of the term structures. It is used for formulating computation steps in call-by-value and overlaps between rules.
Moreover, we reformulate the notion of overlaps to compute critical pairs in call-by-value (Definition 4.3), and introduce the notion of *call-by-value joinability* (Definition 4.5) of critical pairs.
A2. We formulate the notion of a unifier for call-by-value polymorphic terms (Definition 4.1).
A3. We give a type inference algorithm for polymorphic computation rules in Fig. 4.

1.5. 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 works on top of the interpreter of Glasgow Haskell Compiler. PolySOL uses the feature of quasi-quotation (i.e. `[signature|..]` and `[rule|..]` are quasi-quotations) of Template Haskell [16], with a custom parser which provides a 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_{NEED}-calculus, we declare the following signature in PolySOL:

```
sigNeed = [signature|
 lam : (a -> b) -> Arr(a,b)  ;  app : Arr(a,b),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 `lmdNeed` given in the beginning of this section is actually a part of rule specification written using PolySOL's language. Using these, we can command PolySOL to perform critical pair checking.

```
*SOL> criCBV lmdNeed sigNeed
1: Overlap (G)-(V-v)--- M|-> V', C'|-> z1.N ----------------------------------
   (G) let(M,x.N) => N
   (V-v) let(V',x'.C'[x']) => C'[V']
                              let(V',x.N)
                         N <-(G)-^-(V-v)-> N
                          ---> N =OK= N <---
2: Overlap (G)-(A)--- M|-> let(L',x'.M'[x']), N'|-> z1.N --------------------
   (G) let(M,x.N) => N
   (A) 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 <-(G)-^-(A)-> let(L',xd3.let(M'[xd3],yd3.N))
                           ---> N =OK= N <---
3: Overlap (C-v)-(G)--- M'|-> V, M|-> z1.N' ----------------------------------
   (C-v) let(V,x.M[x])@N => let(V,x.(M[x]@N))
   (G) let(M',x'.N') => N'
                             let(V,x.N')@N
        let(V,x8.(N'@N)) <-(C-v)-^-(G)-> N'@N
                         ---> N'@N =OK= N'@N <---
4: Overlap (C-v)-(V-v)--- V|-> V', C'|-> z1.M[z1] ----------------------------
   (C-v) let(V,x.M[x])@N => let(V,x.(M[x]@N))
   (V-v) let(V',x'.C'[x']) => C'[V']
                            let(V',x.M[x])@N
  let(V',x11.(M[x11]@N)) <-(C-v)-^-(V-v)-> M[V']@N
                       ---> M[V']@N =OK= M[V']@N <---
5: Overlap (A)-(G)--- M'|-> L, M|-> z1.N' ------------------------------------
   (A) let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
   (G) let(M',x'.N') => N'
                         let(let(L,x.N'),y.N[y])
let(L,x15.let(N',y15.N[y15])) <-(A)-^-(G)-> let(N',y.N[y])
         ---> let(N',y15.N[y15]) =E= let(N',y.N[y]) <---
6: Overlap (A)-(V-v)--- L|-> V', C'|-> z1.M[z1] ------------------------------
   (A) let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
   (V-v) let(V',x'.C'[x']) => C'[V']
                         let(let(V',x.M[x]),y.N[y])
let(V',x27.let(M[x27],y27.N[y27])) <-(A)-^-(V-v)-> let(M[V'],y.N[y])
      ---> let(M[V'],y27.N[y27]) =E= let(M[V'],y.N[y]) <---
7: Overlap (A)-(A)--- L|-> let(L',x'.M'[x']), N'|-> z1.M[z1] -----------------
   (A) let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
   (A) 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])) <-(A)-^-(A)-> let(let(L',xd.let(M'[xd],yd.M[yd])),y.N[y])
---> let(L',x.let(M'[x],y4.let(M[y4],y.N[y]))) =E= let(L',x.let(M'[x],x5.let(M[x5],y.N[y]))) <---
#Joinable! (Total 7 CPs)
```

The above PolySOL's output corresponds to the diagrams shown in Fig. 1. Each item shows which two rules are overlapped, a substitution for metavariables for the overlap, and the two rules. The highlight in the first rule shows that the subterm is unifiable with the root of left-hand side of the second rule.

For example, in the overlap `3:`, the subterm `let(V,x.M[x])` in the rule `(C-v)` is unifiable with the term `let(M',x'.N')` in the rule `(G)` using the unifier `M'|-> V, M|-> z1.N'` described immediately above `(C-v)`. Then using this information, PolySOL generates the underlined term `let(V,x.N')@N` which exactly corresponds to the source

in the first divergent diagram (3:) in Fig. 1. The lines involving $\wedge$ (indicating "divergence") mimics the divergence diagram and the joinability test in text. The sign =OK= denotes syntactic equality, and =E= denotes α-equivalence.

Maraist et al. has shown that confluence of untyped λ_{NEED} has been established by a different proof method [11], i.e., analysis of developments steps in the λ-calculus [17]. 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.

1.6. Contributions

This paper is the fully reworked and extended version of the conference paper [18]. Besides proofs of all results, the present paper establishes a theory of confluence of polymorphic computation systems with value/non-value metavariables. In addition, we provide a new example solving the coherence problem of skew-monoidal categories (§7). More precisely, the contributions of this paper are summarised as follows.

1. We develop a new framework of polymorphic second-order computation that has predicates of "instance validations" to reflect value/non-value distinctions (§2).
2. We present a type reconstruction algorithm for polymorphic second-order rules (§3).
3. We develop a theory of confluence in the call-by-value setting (§4) by introducing the notion of "call-by-value joinability" of critical pairs (§4.5). To do so, we formulate meta-confluence and object confluence (§4.2).
4. We develop confluence criteria without termination:
   - confluence by strong closedness (§5.1)
   - confluence by orthogonality (§5.2)
   - modular confluence checking (§5.3)

   The criterion for the first is new, and the criteria and proofs for the second and the third are adapted from the case of (simply-typed) higher-order rewrite systems [9] to our polymorphic setting.
5. Using PolySOL, we give confluence proofs of various calculi, including the call-by-need λ-calculus, Moggi's computational λ-calculus (§6), and skew-monoidal categories (§7).

1.7. 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 criterion based on critical pair checking in the call-by-value setting in §4. We further establish confluence criteria without requiring termination in §5. In §6, we prove confluence of Moggi's computational λ-calculus using PolySOL. In §7, we consider a further example, the coherence problem of skew-monoidal categories using PolySOL. In §8, we describe the implementation of PolySOL. In §9, we report the results in International Confluence Competition 2018, which showed the effectiveness of PolySOL's checking method and the new confluence criteria. In §10, we summarise the paper and discuss related work.

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 polymorphic extension of second-order rewriting systems [19–21] based on second-order abstract syntax with metavariables [2,22] with molecular types [1]. The present framework introduces type variables into types and the feature of instance validation for instantiation of axioms for value/non-value distinction. The polymorphism in this framework is essentially ML polymorphism, i.e., predicative and universally quantified at the outermost only and has type constructors on types.

Notation 2.1. We use the notation $\overline{A}$ for a sequence $A_1, \cdots, A_n$, and $|\overline{A}|$ for its length. We use the abbreviations "lhs" and "rhs" to mean left-hand side and right-hand side, respectively.

2.1. 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 $\mathrm{s}, \mathrm{t}, a, b, \cdots$). We also assume a set of *type constructors* together with arities $n \in \mathbb{N}$, $n \geq 1$. The sets of "0-order types" $\mathcal{T}_0$ and (at most first-order) ***types*** $\mathcal{T}$ are generated by the following rules:

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

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

$$\frac{S \rhd f:(\overline{\sigma_1}\to\tau_1),\ldots,(\overline{\sigma_m}\to\tau_m)\to\tau\in\Sigma \qquad \xi:S\to\mathcal{T} \quad \Theta \rhd \Gamma,\overline{x_i:\sigma_i}\vdash\ t_i\ :\tau_i\xi\quad (1\le i\le m)}{\Theta \rhd \Gamma\vdash\ f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1,\ldots,\overline{x_i^{\sigma_i}}.t_i,\ldots,\overline{x_m^{\sigma_m}}.t_m)\ :\tau\xi}$$

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.

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, List is a unary type constructor, and Bool → List(Bool) is a type.

2.2. Terms and meta-terms

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

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

where $(\overline{\sigma_1}\to\tau_1),\ldots,(\overline{\sigma_m}\to\tau_m)\in\mathcal{T}$, $\tau\in\mathcal{T}_0$, and type variables $\mathrm{T}_1,\ldots,\mathrm{T}_n$ may occur in these types. As a result function symbols have at most second-order function types. 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 $t ::= x \mid x^{\sigma}.t \mid f(t_1,\ldots,t_n)$.
- ***Meta-terms*** extend terms to $t ::= x \mid x^{\sigma}.t \mid f(t_1,\ldots,t_n) \mid M[t_1,\ldots,t_n]$.

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. Definition 2.2). 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 $\mathcal{T}_0$)-pairs. A judgment is of the form $\Theta \rhd \Gamma\vdash\ t\ :\tau$. A ***type substitution*** $\xi:S\to\mathcal{T}$ is a mapping that assigns a type $\sigma\in\mathcal{T}$ to each type variable s in S. We write $\tau\,\xi$ (resp. $t\,\xi$) to be the type (resp. meta-term) 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 $\xi:S\to\mathcal{T}$. A meta-term t is *well-typed* by the typing rules in 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 variables $x_1,\ldots,x_n$ with terms $s_1,\ldots,s_n$.

Definition 2.2 *(Substitution of meta-terms for metavariables [22,2,3]).* Let $n_i=|\overline{\tau_i}|$ and $\overline{\tau_i}=\tau_i^1,\ldots,\tau_i^{n_i}$. Suppose

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

Then the substituted meta-term $\Theta \rhd \Gamma,\Gamma'\vdash\ e\,[\,\overline{M\mapsto\overline{x}.s}\,]\ :\tau$ 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* $[M_1\mapsto\overline{x_1}.s_1,\ldots,M_k\mapsto\overline{x_k}.s_k]$.

We typically denote by θ a substitution, write $e\theta$ for $e\,[\,\overline{M\mapsto\overline{x}.s}\,]$, and call $t\theta$ *an instance* of t.

2.3. Notions of values and non-values

Next we introduce the syntactic notions for value/non-values in our framework. In the case of λ_{NEED}-calculus, we have used the following two classes of values and non-values

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

This defines two special names V, P of metavariables for values and non-values. To give a general definition of this kind of distinction, we need a syntactic equality on variable names. We write "$X \equiv V$" for it, which means that the metavariable X's letter is "V".

Definition 2.3. A metavariable X is called a ***value metavariable*** if $X \equiv V$ or $X \equiv V_i$. A metavariable is called a ***non-value metavariable*** if $X \equiv P$ or $X \equiv P_i$. Indices i are natural numbers or some syntactic objects (e.g. metavariables, such as P_M). A metavariable which is neither value nor non-value metavariable is often called a ***general metavariable***.

Definition 2.4. A *pattern meta-term* is a meta-term which is not a metavariable, nor a meta-application. *Value patterns* ***Val*** are a finite set of pattern meta-terms used for specifying values. Likewise, *non-value patterns* ***NonVal*** are a finite set of pattern meta-terms used for specifying non-values. These two sets must specify disjoint term sets, i.e., $\{t\theta \mid t \in \mathit{Val},\ \text{subst. } \theta\} \cap \{t\theta \mid t \in \mathit{NonVal},\ \text{subst. } \theta\} = \emptyset$.

In the case of λ_{NEED}-calculus, we take

$$\mathit{Val} \triangleq \{x,\ \lambda x.M\} \qquad \mathit{NonVal} \triangleq \{M@N,\ \mathsf{let}\, x = M \,\mathsf{in}\, N\}$$

Definition 2.5 *(Valid predicate for substitution).* A substitution θ of meta-terms for metavariables is ***valid*** if for every $X \mapsto \overline{x}.s$ in θ, the following holds:

(i) If X is a value metavariable, then s is an instance of a meta-term in Val.
(ii) If X is a non-value metavariable, then s is an instance of a meta-term in NonVal.
(iii) If X is a general metavariable, then there is no restriction on s.

We write $\mathsf{valid}\,\theta$ if a substitution θ is valid. This is an instance of a valid predicate given in our previous work [18].

Remark 2.6. An arbitrary predicate to classify terms, rather than the value/non-value predicate, was allowed in our previous work [23]. To keep the theory simple and to investigate more detailed properties in this paper, we focus only on value/non-value distinction of terms, and develop a general theory of confluence for call-by-value calculi. □

2.4. Polymorphic second-order computation system

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 not a metavariable nor meta-application, i.e., of the form $f(\overline{\overline{x}.t})$.
(ii) ℓ is a higher-order pattern [15], 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.
(iii) All metavariables in r appear in ℓ.

A ***polymorphic second-order computation system with values/non-values*** is formally given by a tuple $(\Sigma, \mathcal{C}, \mathit{Val}, \mathit{NonVal})$ consisting of a signature Σ, a set $\mathcal{C}$ of rules, and value and non-value patterns. We may call it simply a *computation system*.

Given a computation system $(\Sigma, \mathcal{C}, \mathit{Val}, \mathit{NonVal})$, one-step computation

$$\Theta \triangleright \Gamma \vdash s \Rightarrow_{\mathcal{C}} t : \tau$$

is obtained by the inference system given in Fig. 3. The (RuleSub) instantiates a polymorphic computation rule $\ell \Rightarrow r$ in $\mathcal{C}$ by substitution $[\overline{M \mapsto \overline{x}.s}]$ of meta-terms for metavariables and substitution ξ on types. The (Fun) means that the computation step is closed under polymorphic function symbol contexts.

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

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

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 $\Rightarrow_{\mathcal{C}}$ on meta-terms.

If one-step computation happens under the empty metavariable context obtained as $\cdot \rhd \Gamma \vdash\ s \Rightarrow_{\mathcal{C}} t\ : \tau$ (i.e. computation on terms, not on meta-terms), we write it as

$$\Gamma \vdash s \to_{\mathcal{C}} t\ : \tau.$$

We also usually omit contexts and types, and simply write $s \Rightarrow_{\mathcal{C}} t$ or $s \to_{\mathcal{C}} t$. We regard $\Rightarrow_{\mathcal{C}}$ and $\to_{\mathcal{C}}$ to be binary relations on meta-terms and terms, respectively.

Example 2.7. The simply-typed λ_{NEED}-calculus is formulated as a polymorphic second-order computation system with values/non-values

$$(\texttt{sigNeed}, \texttt{lmdNeed}, \mathit{Val}, \mathit{NonVal})$$

where the signature and rules are given as

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

lmdNeed = [rule|
 (G)    let(M, x.N)                 => N
 (I)    lam(x.M[x]) @ N             => let(N,x.M[x])
 (V-v)  let(V, x.C[x])              => C[V]
 (C-v)  let(V, x.M[x])@N            => let(V, x.M[x]@N)
 (A)    let(let(L,x.M[x]), y.N[y]) => let(L,x.let(M[x],y.N[y])) |]
```

and value and non-value patterns are defined by

$$\mathit{Val} \triangleq \{x,\ \lambda x.M\} \qquad \mathit{NonVal} \triangleq \{M@N,\ \mathsf{let}\ x = M\ \mathsf{in}\ N\}.$$

Note that in the signature, the type `Arr(a,b)` encodes a function type `a` → `b` of any order in the λ_{NEED}-calculus. The limitation to second-order in our framework is irrelevant to the type structure of the object language. As this example shows, the second-order computation system can express a λ-calculus of *any order* (not only second-order) using type constructors. □

3. Type inference for polymorphic computation rules

We have formulated that polymorphic computation rules were explicitly typed. 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_{NEED}-calculus, the user only provides the signature `sigNeed` and "plain" rules `lmdNeed` in §1.4. The type inference algorithm infers the missing context and type annotations (highlights) as:

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

These annotations are important for computing overlaps between rules when checking confluence of polymorphic rules.

3.1. Algorithm

Our algorithm is given in Fig. 4, which is a variant of Damas–Milner type inference algorithm $\mathcal{W}$ [24]. It has several modifications to cope with the language of meta-terms and to return enough type information for confluence checking.

$$
\begin{array}{lll}
\mathcal{W}(\Sigma, x) & = & \text{if } x:\tau \text{ appears in } \Sigma \text{ then } ([],\ \triangleright x^{\tau}:\tau) \text{ else } \mathit{error} \\
\mathcal{W}(\Sigma, x.t) & = & \mathsf{let}\ a = \mathsf{freshVar} \\
 & & \quad (\theta',\ \Theta \triangleright t':\tau') = \mathcal{W}(\{x:a\} \cup \Sigma, t) \\
 & & \mathsf{in}\ (\theta',\ \Theta \triangleright x^{a}.t' : a \to \tau') \\
\mathcal{W}(\Sigma, f(\overline{t})) & = & \text{if } f:\overline{d} \to c \text{ appears in } \Sigma \text{ then} \\
 & & \quad \mathsf{let}\ n = \mathsf{newNum}\ \mathsf{in} \\
 & & \quad (\overline{d'} \to c') = \text{attach the index } n \text{ to all type vars in } (\overline{d} \to c) \\
 & & \quad (\theta, \Theta,\ \overline{u}, \overline{a}) = \mathsf{foldr}\ (\mathcal{W}_{\mathsf{iter}}\Sigma)\ ([],[],[],[])\ \overline{t} \\
 & & \quad b = \mathsf{freshVar} \\
 & & \quad \theta' = \mathsf{unify}((\overline{a} \to b)\theta,\ \overline{d'} \to c') \\
 & & \mathsf{in}\ (\theta' \circ \theta,\ \{f_n : (\overline{d'} \to c')\theta'\} \cup \Theta\theta' \triangleright f_n(\overline{u}) : b\theta') \\
 & & \mathsf{else}\ \mathit{error} \\
\mathcal{W}(\Sigma, M[\overline{t}]) & = & \mathsf{let}\ (\theta, \Theta, \overline{u}, \overline{a}) = \mathsf{foldr}\ (\mathcal{W}_{\mathsf{iter}}\Sigma)\ ([],[],[],[])\ \overline{t} \\
 & & \quad b = \mathsf{freshVar} \\
 & & \mathsf{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})) & = & \mathsf{let}\ (\theta,\ \Theta \triangleright u:\tau) = \mathcal{W}(\Sigma, t) \\
 & & \mathsf{in}\ (\theta \circ \theta_0,\ \Theta \cup \Theta_0, (u, \overline{u}), (\tau, \overline{\tau})) \\
\mathsf{mkMatch}(\Theta) & = & \{(\sigma, \tau) \mid (M:\sigma) \in \Theta,\ (M:\tau) \in \Theta, \sigma \neq \tau\} \\
\mathsf{infer}(\Sigma, t) & = & \mathsf{let}\ (\theta,\ \Theta \triangleright u:\tau) = \mathcal{W}(\Sigma, t) \\
 & & \quad \theta' = \mathsf{unify}(\mathsf{mkMatch}(\Theta\theta))\ \circ\ \theta \\
 & & \mathsf{in}\ \Theta\theta' \triangleright u\,\theta' : \tau\,\theta' \\
\mathsf{infer}(\Sigma,\ s \Rightarrow t) & = & \mathsf{infer}(\{\mathsf{rule} : \mathrm{S}, \mathrm{S} \to \mathrm{T}\} \cup \Sigma,\ \mathsf{rule}(s, t))
\end{array}
$$

- 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 function $\mathcal{W}$ by the function $\mathcal{W}_{\mathsf{iter}}$.
- unify returns the most general unifier of the pairs of types.
- "[]" denotes the empty sequence or substitution.

Fig. 4. Type inference algorithm.

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 needed 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 $\sigma \circ \tau' = \tau$. It is a preorder.

Theorem 3.1 *(Soundness). If infer*$(\Sigma, t) = (\Theta \triangleright t' : \tau)$*, then there exists* Γ *such that* $\Theta \triangleright \Gamma \vdash t' : \tau$.

Proof. By induction on the structure of t.

Theorem 3.2 *(Completeness). If* $\Theta \triangleright \Gamma \vdash t : \tau$ *holds under a signature* Σ *and infer*$(\Sigma, |t|) = (\Theta' \triangleright t' : \tau')$*, then there exists a substitution* θ *such that* $\tau'\theta = \tau$ *and*

- *If* $M : \sigma \in \Theta$ *then, there exists* $M : \sigma' \in \Theta'$ *such that* $\sigma'\theta = \sigma$.
- *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* $(\overline{\sigma}' \to \tau')\theta = \overline{\sigma} \to \tau$.

Proof. By induction on the typing derivation of $\Theta \triangleright \Gamma \vdash t : \tau$.

The reason why our algorithm attaches an index n to each occurrence of a function symbol f as "f_n" is to distinguish different occurrences of the same f in a meta-term, and to correctly infer the type of each of them (see Prob. 2 in §1.4). 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 call-by-value

In this section, we establish a confluence criterion of polymorphic computation systems based on critical pair checking.

4.1. Abstract rewriting

We review classical results on abstract rewriting [12,7]. Abstract rewriting is a general framework for analysing properties of rewriting without touching the structure of "terms", only focusing the rewrite relation between elements (in this sense "abstract").

An *abstract rewriting system (ARS)* is a pair $(A, \to)$ of a set A and a binary relation $\to$ on A. We write $\to^*$ for the reflexive transitive closure, $\to^+$ for the transitive closure, and $\leftarrow$ for the converse of $\to$. We define $\leftrightarrow \triangleq \to \cup \leftarrow$. We say:

1. $a, b \in A$ are *joinable*, written $a \downarrow b$, if $\exists c.\, a \to^* c \ \&\ b \to^* c$.
2. $\to$ is *confluent* if $\forall a, b, c \in A.\, a \to^* b \ \&\ a \to^* c$ implies $b \downarrow c$.
3. $\to$ is *Church-Rosser (CR)* if $\forall a, b \in A.\, a \leftrightarrow^* b$ implies $a \downarrow b$.
4. $\to$ is *locally confluent (WCR)* if $\forall a, b, c \in A.\, a \to b \ \&\ a \to c$ implies $b \downarrow c$.
5. $\to$ is *strongly normalising (SN)* if $\forall a \in A$, there is no infinite sequence $a \to a_1 \to a_2 \to \cdots$.
6. a is a *normal form* if there is no $b \in A$ such that $a \to b$.

We identify an ARS A with its relation $\to$ (e.g. we say A is CR to mean $\to$ is CR). It is well-known that confluence and Church-Rosser properties are equivalent, hence we have used the word CR to also mean confluence.

4.2. Two confluence properties

Suppose a polymorphic computation system $\mathcal{C} = (\Sigma, \mathcal{C}, \mathit{Val}, \mathit{NonVal})$ is given. First we note an important fact that the pair (the set of all meta-terms, $\Rightarrow_{\mathcal{C}}$) forms an ARS. Hence any notion and result on ARS are applicable to second-order computation. Henceforth, we may regard a computation system $\mathcal{C}$ as the ARS. We say:

(i) $\mathcal{C}$ is ***meta-confluent*** if the ARS (the set of all meta-terms, $\Rightarrow_{\mathcal{C}}$) is confluent.
(ii) $\mathcal{C}$ is ***object confluent*** if the ARS (the set of all terms, $\to_{\mathcal{C}}$) is confluent.

The difference between terms and meta-terms was in §2.2.

Since $\Rightarrow_{\mathcal{C}} \supseteq \to_{\mathcal{C}}$ (see §2.4) and rewrite steps do not introduce new metavariables, if $\mathcal{C}$ is meta-confluent, then it is also object confluent, but not vice versa. What we want to establish for concrete calculi (such as the λ_{NEED}-calculus) is object confluence.

Meta-confluence means the confluence property at the meta-level, i.e., confluence on the meta-terms ***with metavariables***, which are wider than the terms (with no metavariables). Practically, meta-confluence is usually easier to establish, therefore one often firstly establishes meta-confluence and next concludes object confluence as a derived property. The proof of λ_{NEED}-calculus in §1.4 actually chose this strategy: firstly established the meta-confluence of the polymorphic computation system, and then concluded the object confluence. But this strategy does not always work. In §6, we will see that the polymorphic computation system of $\lambda_{\mathcal{C}}$-calculus is not meta-confluent; nevertheless, we can prove its object confluence of it using the notion of call-by-value joinability (Definition 4.5).

4.3. Notion of unifier between two polymorphic meta-terms in call-by-value

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 the case of polymorphic second-order algebraic theory, we should also take types into account. For example, what should be a unifier between the following terms?

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

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* $\xi : \mathrm{V} \mapsto \mathsf{bool}, \mathrm{T} \mapsto \mathsf{int}$.

Moreover, a unifier for value/non-value metavariables should be valid. For example, consider a unification problem:

$$M \mathbin{@} \lambda^{(\mathrm{A}\to\mathrm{B})\to\mathsf{Arr}(\mathrm{A},\mathrm{B})}(x.N[x]) \stackrel{?}{=} P \mathbin{@} V$$

This is unifiable by a ***valid*** substitution $\theta = \{M \mapsto P,\ V \mapsto \lambda^{(A\to B)\to \mathsf{Arr}(A,B)}(x.N[x])\}$, because $\lambda^{(A\to B)\to \mathsf{Arr}(A,B)}(x.N[x])$ is a value, and a general metavariable M can be replaced with a non-value P (but $P \mapsto M$ is invalid). A non-solvable problem is

$$V \mathbin{@} \lambda^{(A\to B)\to \mathsf{Arr}(A,B)}(x.N[x]) \stackrel{?}{=} P_1 \mathbin{@} P_2$$

Neither $V \mapsto P_1$ nor $P_1 \mapsto V$ is valid. Moreover, $P_2 \mapsto \lambda^{(A\to B)\to \mathsf{Arr}(A,B)}(x.N[x])$ is invalid, because P_2 is a non-value metavariable while λ-abstraction is a value.

This leads us to the following definition.

Definition 4.1. A *unifier between meta-terms* s and t is a tuple (ξ, ϑ) such that $s\,\xi\,\vartheta = t\,\xi\,\vartheta$, where

(i) ξ is a substitution of types for type variables, and
(ii) ϑ is a ***valid*** substitution of meta-terms for metavariables.

The *most general unifier* is a maximal unifier with respect to the preorder on substitutions of type variables and of meta-terms.

4.4. Critical pairs of polymorphic computation systems with values/non-values

We now formulate the notion of critical pairs for our polymorphic case with values/non-values distinction. We first recall basic notions.

A *position* p is a finite sequence of natural numbers. The empty sequence ε is the root position, and the concatenation of positions is denoted by pq or $p.q$. The order on positions is defined by $p < q$ if there exists a non-empty p' such that $p.p' = q$. The set $\mathcal{P}os(t)$ of the positions of a meta-term t is defined by

$$\begin{aligned}
\mathcal{P}os(x) &= \{\varepsilon\}\\
\mathcal{P}os(x.t) &= \{\varepsilon\} \cup \{1.p \mid p \in \mathcal{P}os(t)\}\\
\mathcal{P}os(f(t_1,\ldots,t_n)) &= \{\varepsilon\} \cup \{i.p \mid 1 \le i \le n,\ p \in \mathcal{P}os(t_i)\}\\
\mathcal{P}os(M[t_1,\ldots,t_n]) &= \{\varepsilon\} \cup \{i.p \mid 1 \le i \le n,\ p \in \mathcal{P}os(t_i)\}
\end{aligned}$$

The notation $s[u]_p$ means replacing the subterm at the position p of s with u, and $s_{|p}$ means selecting the subterm of s at the position p.

Suppose a computation system $\mathcal{C}$ is given. We say two rules $l_1 \Rightarrow r_1, l_2 \Rightarrow r_2$ in $\mathcal{C}$ are *variant* if $l_1 \Rightarrow r_1$ is obtained by injectively renaming variables and metavariables of $l_2 \Rightarrow r_2$.

Definition 4.2. We say that a position p in a meta-term t is a *metavariable position* if $t_{|p}$ is a metavariable or meta-application, i.e., $t_{|p} = M[c_1,\ldots,c_n]$. This description includes the case $t_{|p} = M$ by the case $n = 0$, for which we identify $M[\,]$ with just a metavariable M.

An overlap represents an overlapping of the two rules, which admits the situation that a term can be rewritten by the two different rules.

Definition 4.3. An ***overlap*** between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ of a polymorphic computation system $(\Sigma, \mathcal{C}, \mathit{Val}, \mathit{NonVal})$ is a tuple $\langle l_1 \Rightarrow r_1,\ p,\ l_2 \Rightarrow r_2,\ \xi,\ \vartheta\rangle$ satisfying the following properties:

(i) $l_1 \Rightarrow r_1,\ l_2 \Rightarrow r_2$ are variants of rules in $\mathcal{C}$ without common (meta)variables.
(ii) (ξ, ϑ) is a most general unifier between $l_{1|p}$ and l_2.
(iii) p is a non-metavariable position of l_1, or **if p is a metavariable position of l_1, i.e., $l_{1|p} = M[c_1,\ldots,c_n]$, then M is a value or non-value metavariable.**
(iv) If p is the root position, $l_2 \Rightarrow r_2$ is not a variant of $l_1 \Rightarrow r_1$.

The ordinary definition of overlap requires that the position p of l_1 be a non-metavariable position because $l_{1|p}$ matches anything if it is a metavariable, which means that it should not be considered as overlapping. But in the call-by-value case, the situation differs, because a value metavariable V (resp. a non-value metavariable P) represents value patterns (resp. non-value patterns) in Val. If $l_{1|p}$ is a value metavariable, then we need to check overlapping between value patterns and a meta-term. We note that the notion of values is not stable under reduction (unlike the notion of types). For example, a β-reduction step

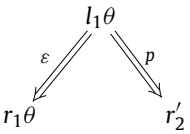
$$\lambda(x.x)@y \Rightarrow_{\mathcal{C}} y$$

transforms a non-value (an application) to a value (a variable). Therefore, a non-value metavariable position in a meta-term might become a value position during computation. This means that we need to analyse the cases when a meta-term involves a value/non-value metavariable. The bold face sentence in the item (iii) describes it.

Definition 4.4. The ***critical pair (CP)*** generated by an overlap $\langle l_1 \Rightarrow r_1,\ p,\ l_2 \Rightarrow r_2, \xi, \vartheta \rangle$ is a triple $\langle r_1\theta,\ l_1\theta,\ r_2' \rangle$ where

- $\theta \triangleq \vartheta \circ \xi$,
- $l_1\theta \Rightarrow_{\mathcal{C}} r_1\theta$ which rewrites the root position ε of $l_1\theta$ using $l_1 \Rightarrow r_1$,
- $l_1\theta \Rightarrow_{\mathcal{C}} r_2'$ which rewrites the position p of $l_1\theta$ using $l_2 \Rightarrow r_2$.

This is depicted as



This is a critical situation that admits two ways of reduction, hence called a critical pair. Ordinary Knuth-Bendix critical pairs lack the middle $l_1\theta$, hence "pairs". But including "the source of divergence" designates a situation more clearly (especially in the implementation), hence our notion of critical pair consists of three terms. We define

$$\mathsf{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \triangleq \{\text{all possible overlaps between } l_1 \Rightarrow r_1 \text{ and } l_2 \Rightarrow r_2\}.$$

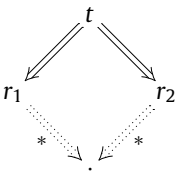
Algorithmically, this function scans all subterms of l_1 and tries to unify each of them with l_2 to produce an overlap using second-order unification. We then collect all overlaps in $\mathcal{C}$ by

$$\mathcal{O} \triangleq \bigcup \{\mathsf{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \mid l_1 \Rightarrow r_1,\ l_2 \Rightarrow r_2 \in \mathcal{C}\}.$$

Finally, we obtain all critical pairs of $\mathcal{C}$ by generating the critical pair of each overlap in $\mathcal{O}$.

4.5. Joinability of critical pairs

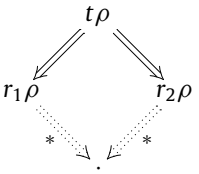
We assume that for each metavariable M, there exist a value metavariable V_M and a non-value metavariable P_M. We say that *a critical pair* $\langle r_1, t, r_2 \rangle$ *is joinable* if $r_1 \Downarrow_{\mathcal{C}} r_2$, depicted as



Definition 4.5. We say that a critical pair $\langle r_1, t, r_2 \rangle$ is ***call-by-value joinable*** if for every substitution ρ of metavariables for metavariables in t satisfying

$$\rho : M \mapsto V_M \text{ or } M \mapsto P_M \quad \text{for each metavariable } M \text{ occuring in } t,$$

then the joinability $r_1\rho \Downarrow_{\mathcal{C}} r_2\rho$ holds. This is depicted as



(2)

This amounts to checking joinability in the call-by-value setting. In the call-by-value setting, M denotes either a value or non-value. Hence we check for each M in t, two cases of the joinability: when M is a value metavariable and when M is a non-value metavariable. This is done by applying all possible renamings ρ that change M in t to either a value metavariable V_M or a non-value metavariable P_M.

Lemma 4.6.

1. *If a critical pair is joinable, then it is also call-by-value joinable, but not vice versa.*
2. *If a critical pair $\langle r_1, t, r_2\rangle$ is call-by-value joinable, then $r_1\theta \downarrow_{\mathcal{C}} r_2\theta$ for any substitution θ of terms (without metavariables) for metavariables.*

Proof.

(i) By definition.
(ii) Instantiating the joinability (2) of call-by-value critical pairs to terms, we have the result.

We first prove an easier case, which does not have value/non-value distinction.

Proposition 4.7. *Let $(\Sigma, \mathcal{C})$ be a polymorphic second-order computation system without value/non-value metavariables. If every critical pair of $(\Sigma, \mathcal{C})$ is joinable, then $\Rightarrow_{\mathcal{C}}$ is locally confluent.*

Proof. We show "if $u \Leftarrow_{\mathcal{C}} w \Rightarrow_{\mathcal{C}} s$ then $u \Downarrow_{\mathcal{C}} s$" by induction on the proof of $u \Leftarrow_{\mathcal{C}} w$, using the inference rules in Fig. 3.

- (RuleSub) Let $l \Rightarrow r \in \mathcal{C}$ and consider the situation

$$r\theta \xleftarrow{\varepsilon} l\theta \xrightarrow{p'} s$$

for a valid substitution θ for metavariables and type variables.
(i) If the rewrite position p' is not "a metavariable position of l or below it", then it is an instance of a critical pair, hence $r\theta \Downarrow_{\mathcal{C}} s$.
(ii) Case $p' = p \cdot q$ is a metavariable position in l or below it, i.e., there exists a metavariable $M : \sigma_1, \ldots, \sigma_n \to \tau$ such that

$$l_{|p} = M[\overline{c}]$$

where $c_1, \ldots, c_n$ are distinct bound variables, since l is a higher-order pattern. Suppose θ has the following assignment:

$$\theta : M \mapsto x_1, \ldots, x_n.\, t$$

Let t_p be a one-step reduct of t by contracting the position q as

$$t\{\overline{x} \mapsto \overline{c}\} \Rightarrow_{\mathcal{C}} t_p$$

We define

$$l_p^{M_p} \triangleq l[\,M_p\,]_p\,; \qquad \theta_p \triangleq [M_p \mapsto t_p]$$

i.e., $l_p^{M_p}$ as a modified l, where $M[\overline{c}]$ at the position p of l is replaced with a new metavariable $M_p : \tau$.
(Note that the arity changed from M to M_p, and since M is never a value/non-value metavariable, θ_p is always valid. If value/non-value metavariables are involved, the validity of θ_p is not ensured.)
Define θ' to be $M \mapsto \overline{c}.t_p$, $N \mapsto \theta(N)$ for $N \neq M$, which is valid. Since l is a higher-order pattern, we have

$$\begin{array}{ccc} r\theta & \xleftarrow{\varepsilon} & l\theta \xrightarrow{p'} l_p^{M_p}\theta_p\,\theta = s \\ {\scriptstyle *}\Big\Downarrow & & \Big\Downarrow{\scriptstyle *} \\ r\theta' & \xleftarrow{\qquad\qquad} & l\theta' \end{array}$$

- (Fun): Consider the situation $f^{\xi}(\overline{\overline{x}.s}) \xleftarrow{p'} f^{\xi}(\overline{\overline{x}.u}) \longrightarrow t$, where p' is not the root position because (Fun) is applied. Hereafter, we omit the superscript of f.
(i) Case that the right rewrite happens at the root, i.e., there exists $l \Rightarrow r \in \mathcal{C}$ and a valid θ such that

$$s \xleftarrow{p'} f(\overline{\overline{x}.u}) = l\theta \xrightarrow{\varepsilon} r\theta$$

Flipping the left and right rewrites, $r\theta \Downarrow_{\mathcal{C}} s$ is proved as the case for (RuleSub).

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

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

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

Fig. 5. Polymorphic second-order computation $\to_{\mathcal{C}}$ on terms.

(ii) Case the i, j-th arguments of f are rewritten. Without loss of generality, we assume $i < j \in \mathbb{N}$.

$$\begin{array}{ccccc} f(\cdots, \overline{x_i}.u_i', \cdots, \overline{x_j}.u_j, \cdots) & \overset{ip}{\Longleftarrow} & f(\overline{\overline{x}.u}) & \overset{jq}{\Longrightarrow} & f(\cdots, \overline{x_i}.u_i, \cdots, \overline{x_j}.u_j'', \cdots) \\ {\scriptstyle jq}\Big\Downarrow & & & & \Big\Downarrow{\scriptstyle ip} \\ f(\cdots, \overline{x_i}.u_i', \cdots, \overline{x_j}.u_j'', \cdots) & & \Longrightarrow & & f(\cdots, \overline{x_i}.u_i', \cdots, \overline{x_j}.u_j'', \cdots) \end{array}$$

(iii) Case the i-th argument of f is rewritten by two ways as:

$$\begin{array}{ccccc} f(\cdots, \overline{x_i}.u_i', \cdots) & \overset{ip}{\Longleftarrow} & f(\overline{\overline{x}.u}) & \overset{iq}{\Longrightarrow} & f(\cdots, \overline{x_i}.u_i'', \cdots) \\ {\scriptstyle *}\Big\Downarrow & & & & \Big\Downarrow{\scriptstyle *} \\ f(\cdots, \overline{x_i}.s, \cdots) & & \Longrightarrow & & f(\cdots, \overline{x_i}.s, \cdots) \end{array}$$

The above diagram commutes by the induction hypothesis:
$u_i' \Rightarrow_{\mathcal{C}}^{*} s \Leftarrow_{\mathcal{C}}^{*} u_i''$ and closedness of reduction by contexts.

The next is a key proposition, which takes value/non-values into account.

Proposition 4.8. *Let $(\Sigma, \mathcal{C}, Val, NonVal)$ be a polymorphic second-order computation system. If every critical pair is call-by-value joinable, then $\to_{\mathcal{C}}$ is locally confluent.*

Proof. We show "if $w \leftarrow_{\mathcal{C}} u \to_{\mathcal{C}} s$ then $w \downarrow_{\mathcal{C}} s$" by induction on the proof of $w \leftarrow_{\mathcal{C}} u$, using the inference rules in Fig. 5.

- (RuleSub) Let $l \Rightarrow r \in \mathcal{C}$ and consider the situation

$$r\theta \ \overset{\varepsilon}{\longleftarrow}\ l\theta \ \overset{p'}{\longrightarrow}\ s$$

for a valid substitution θ for metavariables and type variables.
  - (i) If the rewrite position p' is not "a metavariable position of l or below it", then $\langle r\theta, l\theta, s\rangle$ is an instance of a critical pair, hence by Lemma 4.6, it is joinable.
  - (ii) Case $p' = p \cdot q$ is a metavariable position in l or below it, i.e., there exists a metavariable $M : \sigma_1, \ldots, \sigma_n \to \tau$ such that $l_{|p} = M[\overline{c}]$. Suppose θ has the assignment

$$\theta : M \mapsto x_1, \ldots, x_n.t.$$

    - (a) If M is a value or non-value metavariable, then $\langle r\theta, l\theta, s\rangle$ is an instance of a critical pair, hence by Lemma 4.6, it is joinable.
    - (b) If M is a general metavariable, then it is proved by the same way as the proof of Proposition 4.7, (RuleSub) (ii).
- (Fun): This case is proved by the same way as the proof of Proposition 4.7 (Fun).

Theorem 4.9. *Let $(\Sigma, \mathcal{C}, Val, NonVal)$ be a polymorphic second-order computation system. Assume that $\mathcal{C}$ is strongly normalising. If every critical pair is call-by-value joinable, then $\mathcal{C}$ is object confluent, i.e., $\to_{\mathcal{C}}$ is confluent.*

Proof. By Proposition 4.8 and Newman's lemma.

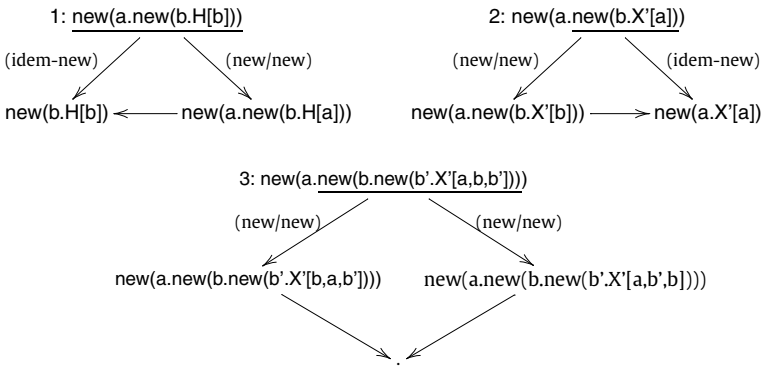


Fig. 6. Critical pairs of a calculus of the new name generation.

5. Confluence criteria without termination

A given calculus is often not terminating or proving its termination is difficult. Nevertheless we want to prove confluence. This section presents three additional confluence criteria: two techniques not requiring termination, called strong closedness (§5.1) and orthogonality (§5.2), and one technique by modularity (§5.3). These criteria have been implemented in the PolySOL system.

5.1. Confluence by strong closedness

As an example, we consider the laws of the new name generation operator (or, restriction operator) [5, Sec. V]. They include a function symbol `new` for new name generation and two rules. The idea is that `new(a.X[a])` first generates a name `a` and then continues as `X` using `a`, as in the `new` operator of the π-calculus [25].

```
sigrest = [signature|
  new : (N -> A) -> A |]

rest = [rule|
 (idem-new) new(a.X)              => X
 (new/new)  new(a.new(b.X[a,b])) => new(a.new(b.X[b,a])) |]
```

The law `(idem-new)` states that generating a new name `a` and continuing `X` without using `a` is equivalent to just doing `X`. The law `(new/new)` states that generating new names `a` and `b` and continuing `X` using `a,b` is equivalent to generating new names `a` and `b`, and continuing `X` using `b,a` (N.B. the application order is changed).

By invoking the command "`cri`" for **cri**tical pair checking, PolySOL checks their critical pairs. It reports three CPs, all of which are joinable (see Fig. 6). Importantly, the system is not strongly normalising because the rhs of the rule `(new/new)` matches its lhs. Therefore, we cannot infer confluence based on Newman's lemma. Other criteria must be sought.

Examining the CPs carefully as shown in Fig. 6, one can find a particular phenomenon: all the CPs are joinable by at-most one-step (from left, right, or both sides). This property, which is known as *strong closedness* of CPs, is useful to provide a sufficient condition deriving *strong confluence* [12] of a computation system, which implies confluence without termination.

We will formulate this method formally. For an ARS $(A, \to)$, we define $\to^{=} \triangleq \to \cup \mathrm{id}_A$ (i.e., the reflexive closure of $\to$). We say that $\to$ is ***strongly confluent*** if

$$\forall a, b, c \in A.\ a \to b\ \&\ a \to c \text{ implies } \exists d.\ b \to^{*} d\ \&\ c \to^{=} d.$$

Proposition 5.1 *(Strong confluence [12]). Every strongly confluent ARS is confluent.*

Definition 5.2. A computation system $(\Sigma, \mathcal{C})$ is ***strongly closed*** if for every critical pair $\langle r_1, t, r_2 \rangle$ of $(\Sigma, \mathcal{C})$, there exist u_1 and u_2 such that $r_1 \Rightarrow^{*}_{\mathcal{C}} u_1 \Leftarrow^{=}_{\mathcal{C}} r_2$ and $r_1 \Rightarrow^{=}_{\mathcal{C}} u_2 \Leftarrow^{*}_{\mathcal{C}} r_2$.

A meta-term is called *linear* if no metavariable occurs more than once. A computation system is called *linear* if both sides of rules are linear, and *left-linear* if every left-hand side of rules is linear. We now prove a theorem deriving strong confluence by checking the strong closedness of CPs.

Theorem 5.3. *Let $(\Sigma, \mathcal{C})$ be a linear polymorphic second-order computation system. If $(\Sigma, \mathcal{C})$ is strongly closed and both sides of each rule are higher-order patterns, then $\Rightarrow_{\mathcal{C}}$ is strongly confluent, hence confluent.*

Proof. We show "if $u \Leftarrow_{\mathcal{C}} w \Rightarrow_{\mathcal{C}} s$ then $u \Rightarrow_{\mathcal{C}}^{*} \circ \Leftarrow_{\mathcal{C}}^{=} s$" by induction on the proof of $u \Leftarrow_{\mathcal{C}} w$, using the inference rules in Fig. 3.

- (RuleSub) Let $l \Rightarrow r \in \mathcal{C}$ and consider the situation

$$r\theta \overset{\varepsilon}{\Longleftarrow} l\theta \overset{p}{\Longrightarrow} s$$

for a valid substitution θ for metavariables and type variables.

(i) If the rewrite position p is a non-metavariable position of l, then it is an instance of a critical pair, hence $r\theta \Rightarrow_{\mathcal{C}}^{*} \circ \Leftarrow_{\mathcal{C}}^{=} s$ by strong closedness.

(ii) Case p is a metavariable position in l or below it, i.e., there exists a metavariable $M : \sigma_1, \ldots, \sigma_n \to \tau$ such that $l_{|p} = M[\overline{c}]$. Since l is a higher-order pattern, $c_1, \ldots, c_n$ are distinct bound variables. Suppose $\theta : M \mapsto x_1, \ldots, x_n.t$ and $t\{\overline{x} \mapsto \overline{c}\} \Rightarrow_{\mathcal{C}} t_p$. We define $l_p^{M_p} \triangleq l[M_p]_p$, $\theta_p \triangleq [M_p \mapsto t_p]$, and $\theta' : M \mapsto \overline{c}.t_p,\ N \mapsto \theta(N)$ for $N \neq M$, which is valid. Since l and r are linear and higher-order patterns, we have

$$\begin{array}{ccc} r\theta & \overset{\varepsilon}{\Longleftarrow} & l\theta \overset{p}{\Longrightarrow} l_p^{M_p}\theta_p\theta = s \\ {\scriptstyle =}\Big\| & & \Big\| \\ r\theta' & \Longleftarrow & l\theta' \end{array}$$

- (Fun): This case is proved in the same way as the proof of Proposition 4.7 (Fun).

Example 5.4. The computation system `rest` given in the beginning of this subsection is strongly confluent because every CP is strongly closed, hence confluent. □

The method for proving confluence of linear rewriting systems through strong closedness of CPs has been used in Suzuki et al.'s paper [26] for nominal rewriting systems. We gave a sufficient condition for strong confluence for polymorphic computation systems. Our method is also straightforwardly applicable to the case of Mayr and Nipkow's higher-order rewrite systems [9].

5.2. Confluence by orthogonality

Another confluence criterion not requiring termination is "orthogonality".

Definition 5.5. A computation system $(\Sigma, \mathcal{C})$ is called ***orthogonal*** if it is left-linear and for any rules $l_1 \Rightarrow r_1$, $l_2 \Rightarrow r_2$ in $\mathcal{C}$, there exists no overlap between $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$.

We show that every orthogonal computation system is confluent. We first define a notion of parallel reduction $\Rightarrow_{\mathcal{C}}^{\mathrm{par}}$ using the inference system in Fig. 7.

Lemma 5.6.

(i) $s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} s$.
(ii) *If* $s \Rightarrow_{\mathcal{C}} t$ *then* $s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t$.
(iii) *If* $s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t$ *then* $s \Rightarrow_{\mathcal{C}}^{*} t$.

Proof.

(i) By induction on s.
(ii) By induction on the proof of $s \Rightarrow_{\mathcal{C}} t$, using (i).
(iii) By induction on the proof of $s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t$.

Thus, to obtain confluence of $\Rightarrow_{\mathcal{C}}$, it suffices to show that $\Rightarrow_{\mathcal{C}}^{\mathrm{par}}$ has the ***diamond property***: if $u \Leftarrow_{\mathcal{C}}^{\mathrm{par}} s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t$ then $u \Rightarrow_{\mathcal{C}}^{\mathrm{par}} \circ \Leftarrow_{\mathcal{C}}^{\mathrm{par}} t$.

Notation 5.7. We write $\theta \Rightarrow_{\mathcal{C}}^{\mathrm{par}} \delta$ to mean that for any metavariable M (with arity n) with $\theta : M \mapsto \overline{x}.s$, $\theta : M \mapsto \overline{x}.t$, $s \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t$ holds. We also write $\overline{t} \Rightarrow_{\mathcal{C}}^{\mathrm{par}} \overline{t'}$ as an abbreviation for $t_i \Rightarrow_{\mathcal{C}}^{\mathrm{par}} t_i'$ for all $i = 1, \ldots, |\overline{t}|$.

The following lemma is used in the proof of Theorem 5.11.

$$(\mathrm{Var}^{\mathrm{par}})\frac{y:\tau\in\Gamma}{\Theta\ \triangleright\ \Gamma\vdash\ y\Rightarrow^{\mathrm{par}}_{\mathcal{C}} y\ :\tau}$$

$$(\mathrm{Metavar}^{\mathrm{par}})\frac{\begin{array}{c}(M:\sigma_1,\cdots,\sigma_m\to\tau)\in\Theta\\ \Theta\ \triangleright\ \Gamma\vdash\ t_i\Rightarrow^{\mathrm{par}}_{\mathcal{C}} t'_i\ :\sigma_i\quad(1\le i\le m)\end{array}}{\Theta\ \triangleright\ \Gamma\vdash\ M[t_1,\ldots,t_m]\Rightarrow^{\mathrm{par}}_{\mathcal{C}} M[t'_1,\ldots,t'_m]\ :\tau}$$

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

$$(\mathrm{Fun}^{\mathrm{par}})\frac{\begin{array}{c}S\ \triangleright\ f:(\overline{\sigma_1}\to\tau_1),\ldots,(\overline{\sigma_m}\to\tau_m)\to\tau\in\Sigma\qquad\xi:S\to\mathcal{T}\\ \Theta\ \triangleright\ \Gamma,\overline{x_i:\sigma_i}\vdash\ t_i\Rightarrow^{\mathrm{par}}_{\mathcal{C}} t'_i\ :\tau_i\xi\quad(1\le i\le m)\end{array}}{\Theta\ \triangleright\ \Gamma\vdash\ f^{\sigma}(\overline{x_1^{\sigma_1}}.t_1,\ldots,\overline{x_m^{\sigma_m}}.t_m)\Rightarrow^{\mathrm{par}}_{\mathcal{C}} f^{\sigma}(\overline{x_1^{\sigma_1}}.t'_1,\ldots,\overline{x_m^{\sigma_m}}.t'_m)\ :\tau\xi}$$

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

Fig. 7. Parallel reduction $\Rightarrow^{\mathrm{par}}_{\mathcal{C}}$ on meta-terms.

Lemma 5.8. *If $\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta$ then $s\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}} s\delta$.*

The next lemma expresses the following idea: in a parallel-reduction step $l\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}} s$, where l is a linear higher-order pattern and does not overlap with any rule, the l-part cannot change, i.e., all reductions must take place inside the meta-terms introduced via θ.

Lemma 5.9. *Let $(\Sigma,\mathcal{C})$ be a polymorphic second-order computation system, and let l be a linear higher-order pattern that does not overlap with any left-hand side of $\mathcal{C}$. Then, if $l\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}} s$ then there exists a valid substitution δ such that $l\delta=s$ and $\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta$.*

Corollary 5.10. *Let $(\Sigma,\mathcal{C})$ be an orthogonal polymorphic second-order computation system, and let $l\Rightarrow r\in\mathcal{C}$. Then, if $l\theta=f(\overline{\overline{x}.t})$ and $\overline{t}\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\overline{t'}$ then there exists a valid substitution δ such that $l\delta=f(\overline{\overline{x}.t'})$ and $\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta$.*

We now prove a theorem deriving the diamond property of $\Rightarrow^{\mathrm{par}}_{\mathcal{C}}$.

Proposition 5.11. *Let $(\Sigma,\mathcal{C})$ be a polymorphic second-order computation system. If $(\Sigma,\mathcal{C})$ is orthogonal, then $\Rightarrow^{\mathrm{par}}_{\mathcal{C}}$ has the diamond property.*

Proof. We show "if $u\Leftarrow^{\mathrm{par}}_{\mathcal{C}} w\Rightarrow^{\mathrm{par}}_{\mathcal{C}} s$ then $u\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\circ\Leftarrow^{\mathrm{par}}_{\mathcal{C}} s$" by induction on the sum of the lengths of the proofs of $u\Leftarrow^{\mathrm{par}}_{\mathcal{C}} w$ and $w\Rightarrow^{\mathrm{par}}_{\mathcal{C}} s$, with a case analysis according to the inference rules in Fig. 7.

Here we consider the most difficult case where the last inference rule used in one of the two proofs is $(\mathrm{RuleSub}^{\mathrm{par}})$ and the other is $(\mathrm{Fun}^{\mathrm{par}})$. The other cases are shown by using the induction hypothesis.

- Let $l\Rightarrow r\in\mathcal{C}$, $\overline{t}\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\overline{t'}$ and consider the situation

$$r\theta'\xleftarrow{(\mathrm{RuleSub}^{\mathrm{par}})} l\theta=f(\overline{\overline{x}.t})\xrightarrow{(\mathrm{Fun}^{\mathrm{par}})} f(\overline{\overline{x}.t'})$$

  for valid substitutions θ,θ' for metavariables and type variables such that $\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\theta'$. Then by Corollary 5.10, there exists a valid substitution δ such that $l\delta=f(\overline{\overline{x}.t'})$ and $\theta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta$. Hence by the induction hypothesis, there exists a valid substitution δ' such that $\theta'\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta'$ and $\delta\Rightarrow^{\mathrm{par}}_{\mathcal{C}}\delta'$. By applying $(\mathrm{RuleSub}^{\mathrm{par}})$, we have $f(\overline{\overline{x}.t'})=l\delta\Rightarrow^{\mathrm{par}}_{\mathcal{C}} r\delta'$. On the other hand, by Lemma 5.8, we have $r\theta'\Rightarrow^{\mathrm{par}}_{\mathcal{C}} r\delta'$, thus closing the diamond.

Combining this with Lemma 5.6, we have:

Theorem 5.12. *Every orthogonal computation system is meta-confluent.*

Mayr and Nipkow have shown that every orthogonal higher-order pattern rewrite system *without polymorphism* is confluent [9] using a different notion of parallel reduction called *superdevelopments* originally used in Aczel's paper [27]. We have shown that every orthogonal polymorphism computation system is confluent using the parallel reduction "$\Rightarrow_{\mathcal{C}}^{\mathrm{par}}$" in the style of *developments* as in Tait and Martin-Löf's proof for λ-calculus [17]. In the next subsection, we also apply this technique to show commutation of mutually orthogonal computation systems.

5.3. Modular confluence checking

It is sometimes useful to split the system into two subsystems for making its confluence proof easier; in particular, into the higher-order part and the first-order part to which we can apply a powerful confluence prover for first-order term rewriting systems. In this subsection, we present a criterion for modular confluence checking.

As an example, consider a λ-calculus with a union operator, called the λG-calculus [28].

```
siglamU = [signature|
 app   : Arr(G,G),G -> G ; lam : (G -> G) -> Arr(G,G)
 union : G,G -> G        ; 0   : G                    |]

lamU = [rule|
 (beta)     lam(x.M[x]) @ N      => M[N]
 (eta)      lam(x.M@x)           => M
 (u-unitr)  union(X,0)           => X
 (u-unitl)  union(0,Y)           => Y
 (u-assoc)  union(union(X,Y),Z)  => union(X,union(Y,Z))
 (u-com)    union(X,Y)           => union(Y,X)          |]
```

The first two rules are for the λ-calculus, the `(u-*)` rules formalise the union operator (which may be considered as non-deterministic choice). This system is not terminating because of `(u-com)`. Moreover, the union rules have many overlaps, hence is not orthogonal, nor strong confluent. Hence, we cannot apply the previous methods to prove confluence of `lamU`.

Now we see that this system can be clearly split into the higher-order part (β and η) and the first-order part (the rules for union). There exists no overlap between these two parts. The situation is formally stated as follows.

Definition 5.13. Two computation systems $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ are *mutually orthogonal* if they are left-linear and for any rules $l_1 \Rightarrow r_1 \in \mathcal{C}_1$ and $l_2 \Rightarrow r_2 \in \mathcal{C}_2$, there exists no overlap between $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$.

We need abstract properties. Let $(A, \to_1)$ and $(A, \to_2)$ be two ARSs. We say that $\to_1$ and $\to_2$ ***commute*** if

$$\forall a, b, c \in A.\ a \to_1^* b \ \&\ a \to_2^* c \text{ implies } \exists d.\ b \to_2^* d \ \&\ c \to_1^* d$$

and that $\to_1$ and $\to_2$ have the ***commuting diamond property*** if

$$\forall a, b, c \in A.\ a \to_1 b \ \&\ a \to_2 c \text{ implies } \exists d.\ b \to_2 d \ \&\ c \to_1 d$$

The following lemmas on commutation hold [7, pp. 31–32].

Lemma 5.14. *If $\to_1$ and $\to_2$ are confluent and commute then $\to_1 \cup \to_2$ is also confluent.*

Lemma 5.15. *If $\to_1$ and $\to_2$ have the commuting diamond property then they commute.*

Since $\Rightarrow_{\mathcal{C}}^* = (\Rightarrow_{\mathcal{C}}^{\mathrm{par}})^*$ by Lemma 5.6, commutation of $\Rightarrow_{\mathcal{C}_1}$ and $\Rightarrow_{\mathcal{C}_2}$ equals commutation of $\Rightarrow_{\mathcal{C}_1}^{\mathrm{par}}$ and $\Rightarrow_{\mathcal{C}_2}^{\mathrm{par}}$. Hence, to obtain confluence of a computation system $(\Sigma_1 \cup \Sigma_2, \mathcal{C}_1 \cup \mathcal{C}_2)$ from two confluent computation systems $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$, it suffices to show that $\Rightarrow_{\mathcal{C}_1}^{\mathrm{par}}$ and $\Rightarrow_{\mathcal{C}_2}^{\mathrm{par}}$ have the commuting diamond property.

Proposition 5.16. *Let $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ be two polymorphic second-order computation systems. If $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ are mutually orthogonal, then $\Rightarrow_{\mathcal{C}_1}^{\mathrm{par}}$ and $\Rightarrow_{\mathcal{C}_2}^{\mathrm{par}}$ have the commuting diamond property.*

Proof. We show "if $u \Leftarrow_{\mathcal{C}_1}^{\mathrm{par}} w \Rightarrow_{\mathcal{C}_2}^{\mathrm{par}} s$ then $u \Rightarrow_{\mathcal{C}_2}^{\mathrm{par}} \circ \Leftarrow_{\mathcal{C}_1}^{\mathrm{par}} s$" by induction on the sum of the lengths of the proofs of $u \Leftarrow_{\mathcal{C}_1}^{\mathrm{par}} w$ and $w \Rightarrow_{\mathcal{C}_2}^{\mathrm{par}} s$, with a case analysis according to the inference rules in Fig. 7.

Here we consider the most difficult case where the last inference rule used in one of the two proofs is (RuleSub$^{\mathrm{par}}$) and the other is (Fun$^{\mathrm{par}}$). The other cases are shown by using the induction hypothesis.

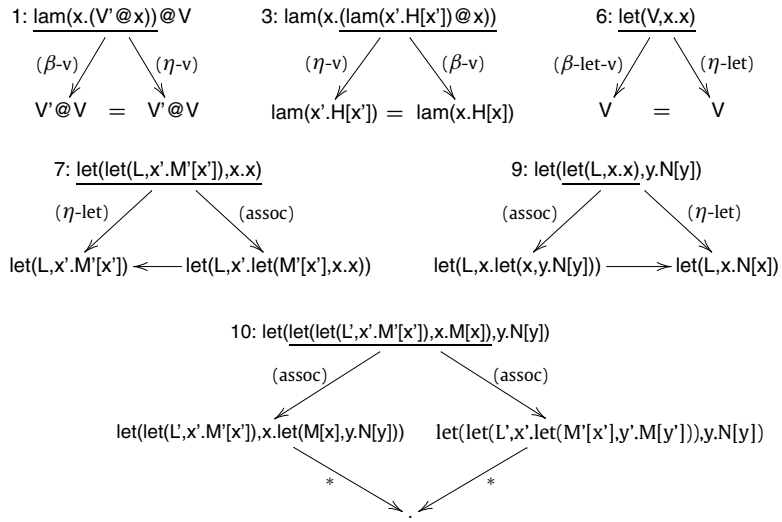


Fig. 8. Critical pairs of the λ_C-calculus (excerpt).

- Let $l \Rightarrow r \in \mathcal{C}_1$, $\overline{t} \Rightarrow^{\mathrm{par}}_{\mathcal{C}_2} \overline{t'}$ and consider the situation

$$r\theta' \xleftarrow{(\mathrm{RuleSub}^{\mathrm{par}})} l\theta = f(\overline{\overline{x}.t}) \xrightarrow{(\mathrm{Fun}^{\mathrm{par}})} f(\overline{\overline{x}.t'})$$

  for valid substitutions θ, θ' for metavariables and type variables such that $\theta \Rightarrow^{\mathrm{par}}_{\mathcal{C}_1} \theta'$. Then by Lemma 5.9, there exists a valid substitution δ such that $l\delta = f(\overline{\overline{x}.t'})$ and $\theta \Rightarrow^{\mathrm{par}}_{\mathcal{C}_2} \delta$. Hence by the induction hypothesis, there exists a valid substitution δ' such that $\theta' \Rightarrow^{\mathrm{par}}_{\mathcal{C}_1} \delta'$ and $\delta \Rightarrow^{\mathrm{par}}_{\mathcal{C}_2} \delta'$. By applying (RuleSub$^{\mathrm{par}}$), we have $f(\overline{\overline{x}.t'}) = l\delta \Rightarrow^{\mathrm{par}}_{\mathcal{C}_1} r\delta'$. On the other hand, by Lemma 5.8, we have $r\theta' \Rightarrow^{\mathrm{par}}_{\mathcal{C}_2} r\delta'$, thus closing the diamond.

Clearly, if the left-linear computation systems $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ have two disjoint signatures (i.e., $\Sigma_1 \cap \Sigma_2 = \emptyset$), they are mutually orthogonal. Thus, we have:

Theorem 5.17. *Let $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ be polymorphic second-order computation systems over two disjoint signatures. If $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ are left-linear and confluent then the computation system $(\Sigma_1 \cup \Sigma_2, \mathcal{C}_1 \cup \mathcal{C}_2)$ is meta-confluent.*

In particular, we can split the whole system into the higher-order and the first-order parts, provided that they have disjoint signatures.

Corollary 5.18. *Let $(\Sigma_1, \mathcal{C}_1)$ be a polymorphic second-order computation system and let $(\Sigma_2, \mathcal{C}_2)$ be a first-order computation system, where Σ_1 and Σ_2 are disjoint signatures. If $(\Sigma_1, \mathcal{C}_1)$ and $(\Sigma_2, \mathcal{C}_2)$ are left-linear and meta-confluent then the computation system $(\Sigma_1 \cup \Sigma_2, \mathcal{C}_1 \cup \mathcal{C}_2)$ is meta-confluent.*

Example 5.19. The computation system `lamU` given in the beginning of this subsection is left-linear. Splitting it into the higher-order rules (β and η rules) and the first-order rules for union, both of which are meta-confluent, we see that the computation system `lamU` is meta-confluent, and also object confluent. □

6. Example 2: confluence of the computational λ-calculus

In this section, we present a proof of confluence of Moggi's computational λ-calculus, the λ_C-calculus [29] using PolySOL.

The λ_C-calculus is a fundamental calculus for effectful computation, which is an extension of the call-by-value λ-calculus enriched with `let`-construct to represent sequential computation.

The λ_C-calculus has seven rules, which are straightforwardly defined in PolySOL as follows:

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

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])) |]
```

It consists of the function symbol `lam` for λ-abstraction, `app` (also written as the infix operator `@`) for application, and `let`-construct. We represent the arrow types of the "object-level" λ_C-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.

To the best of our knowledge, confluence of the λ_C-calculus has not been formally proved in the literature including the original [29] and subsequent works [30,31,13]. Next, we use PolySOL to check the confluence of the simply-typed λ_C-calculus.

We impose the distinction of values and non-values as in the λ_{NEED}-calculus. Here `V` is a metavariable for values, `P` is a metavariable for non-values, and `M`,`N`,`L` are general metavariables for all terms. We tell PolySOL the value/non-value metavariable distinction by writing the suffixes "`-v`" and "`-p`" in the labels. Although the rule set is not so large, the whole proof is fairly large because it has a number of non-trivial overlaps as a result of the value/non-value distinction.

By invoking the command "`criCBV`" for **cri**tical pair checking in **c**all-**b**y-**v**alue, PolySOL checks their critical pairs using value/non-value metavariables. It reports 23 CPs. All are successfully call-by-value joinable. Because the output is long, we present only some of them, and explain some details. The reader can obtain the full output easily using the web interface (see 8.3).

```
*SOL> criCBV lamC siglamC
1: Overlap (beta-v)-(eta-v)--- M|-> z1.(z1@z1), V'|-> x ------------------------
   (beta-v) lam(x.M[x])@V => M[V]
   (eta-v) lam(x'.(V'@x')) => V'
                           lam(x2.(x2@x2))@V
                  V@V <-(beta-v)-^-(eta-v)-> lam(x2.(x2@x2))@V
                       ---> V@V =OK= V@V <---
2: Overlap (beta-v)-(eta-v)--- V|-> lam(x'.(V'@x')) ----------------------------
   (beta-v) lam(x.M[x])@V => M[V]
   (eta-v) lam(x'.(V'@x')) => V'
                     lam(x.M[x])@lam(x'.(V'@x'))
    M[lam(x'.(V'@x'))] <-(beta-v)-^-(eta-v)-> lam(x.M[x])@V'
                      ---> M[V'] =OK= M[V'] <---
..
```

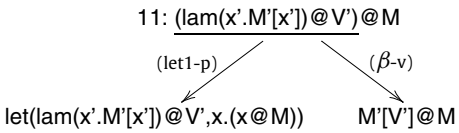
The CP `(2:)` results from the overlap between `(beta-v)` and `(eta-v)`, where the metavariable `V` matches with the root of the lhs of `(eta-v)`. Because ordinary critical pair checking does not check a metavariable position as an overlap, this overlap is particular to the call-by-value setting. The reason it should be regarded as an overlap is that `V` is either a variable or an abstraction because it represents a value. In the case of λ_C, the rule `(eta-v)` is the only rule having a value as lhs, which makes it matchable. The CPs `(4:)` and `(5:)` in the following have the same reason.

```
..
4: Overlap (eta-v)-(eta-v_x)--- V|-> lam(x'.(V'25@x')), V'|-> z1.V'25 ---------
   (eta-v) lam(x.(V@x)) => V
   (eta-v_x) lam(x'.(V'[x]@x')) => V'[x]
                     lam(x.(lam(x'.(V'25@x'))@x))
      lam(x'.(V'25@x')) <-(eta-v)-^-(eta-v_x)-> lam(x.(V'25@x))
                       ---> V'25 =OK= V'25 <---
5: Overlap (beta-let-v)-(eta-v)--- V|-> lam(x'.(V'@x')) ------------------------
   (beta-let-v) let(V,x.M[x]) => M[V]
   (eta-v) lam(x'.(V'@x')) => V'
                     let(lam(x'.(V'@x')),x.M[x])
M[lam(x'.(V'@x'))] <-(beta-let-v)-^-(eta-v)-> let(V',x.M[x])
                      ---> M[V'] =OK= M[V'] <---
..
```

We depict a graphical representation of some of them in Fig. 8. The following check illuminates the call-by-value CP checking.

```
..
11: Overlap (let1-p)-(beta-v)--- P|-> lam(x'.M'[x'])@V' ------------------------
 when M'|-> x95.VM'[x95], M|-> VM
 (let1-p) P@M => let(P,x.(x@M))
 (beta-v) lam(x'.M'[x'])@V' => M'[V']
                       (lam(x'.VM'[x'])@V')@VM
let(lam(x'.VM'[x'])@V',x92.(x92@VM)) <-(let1-p)-∧-(beta-v)-> VM'[V']@VM
                ---> VM'[V']@VM =OK= VM'[V']@VM <---
11: Overlap (let1-p)-(beta-v)--- P|-> lam(x'.M'[x'])@V' ------------------------
 when M'|-> x95.VM'[x95], M|-> PM
 (let1-p) P@M => let(P,x.(x@M))
 (beta-v) lam(x'.M'[x'])@V' => M'[V']
                       (lam(x'.VM'[x'])@V')@PM
let(lam(x'.VM'[x'])@V',x92.(x92@PM)) <-(let1-p)-∧-(beta-v)-> VM'[V']@PM
    ---> let(PM,y1.(VM'[V']@y1)) =E= let(PM,y4.(VM'[V']@y4)) <---
11: Overlap (let1-p)-(beta-v)--- P|-> lam(x'.M'[x'])@V' ------------------------
 when M'|-> x6.PM'[x6], M|-> VM
..
11: Overlap (let1-p)-(beta-v)--- P|-> lam(x'.M'[x'])@V' ------------------------
 when M'|-> x6.PM'[x6], M|-> PM
..
#Joinable! (Total 23 CPs)
```

The four CPs above are caused by a single CP `(11:)` generated by an overlap between `(let1-p)` and `(beta-v)` as



This CP results from the fact that the non-value metavariable `P` in `(let1-p)` can match with the lhs `lam(x'.M'[x'])@V'` of `(beta-v)`, which is the underlined part of the above CP. An important fact is that this is ***not joinable*** at the level of rewriting on meta-terms. The right M'[V']@M is a normal form, whereas the left is reduced to a *different* normal form by `(beta-v)` as

$$\mathsf{let}(\mathsf{lam}(x'.M'[x'])@V', x.(x@M)) \Rightarrow_{\mathcal{C}} \mathsf{let}(M'[V'], x.(x@M))$$

However, this is ***call-by-value joinable*** (Definition 4.5), meaning that it is joinable by case analysis varying M,M' over value/non-value metavariables. In the output presented above, the part like "`when M'|-> x95.VM'[x95], M|-> PM`" denotes checking the case in which `M'` is a value metavariable and `M` is a non-value metavariable. We have adopted a naming convention by which a metavariable starting with `V` is a value metavariable and a metavariable starting with `P` is a non-value metavariable (Definition 2.3). All possible instantiations of M,M' as value/non-value metavariables are the four cases described above. In each case, the left meta-term in a CP is reduced to the right meta-term.

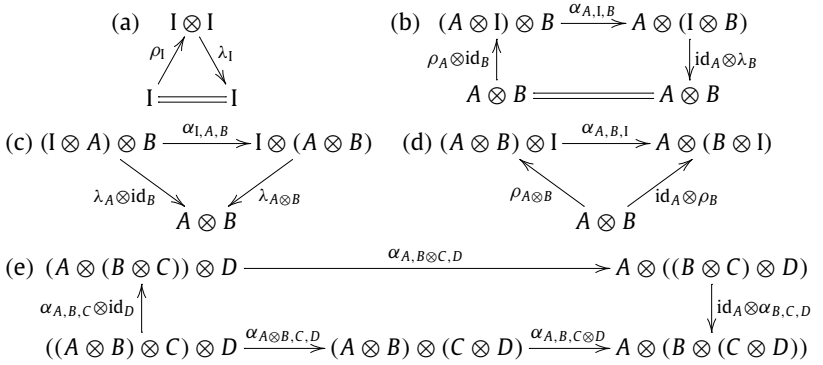
Termination of λ_C has been proved [31]. Hence, by Theorem 4.9, we have object confluence of `lamC`. This concludes that the λ_C-calculus is confluent.

7. Example 3: coherence of skew-monoidal categories

We show a further example, which is different from the λ-calculus. A *skew-monoidal category* [32,33] is a category $\mathbb{C}$ together with a distinguished object I, a functor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ (monoidal product) and natural transformations

$$\lambda_A : \mathrm{I} \otimes A \to A \qquad \rho_A : A \to A \otimes \mathrm{I} \qquad \alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$

subject to the axioms:



The idea of this categorical structure is as follows. A monoidal category [34] is a category equipped with a "monoidal" product $\otimes$ and three natural isomorphisms λ, ρ, α, considered as the unit and associative "laws" for the monoidal product. A skew-monoidal category is a variation of it, in which λ, ρ, α are just one-directional natural transformations, rather than isomorphisms. Since skew-monoidal categories are related to *relative monads* [35], these are related to computer science.

Uustalu [33] proved a remarkable property called *coherence*, which states that every formal arrow having a certain form of codomain has a unique normal form under the axioms of skew-monoidal category. His proof employed normalisation by evaluation technique. Here we show the unique normal form property by a different technique, namely, ***our critical pair checking method for polymorphic computation rules using*** **PolySOL**.

7.1. Laws

Following Uustalu's paper [33], the axioms of a skew-monoidal category $\mathbb{C}$ are equationally expressed as the following laws on arrows of $\mathbb{C}$, which we call **SkewMon**.

Laws of the composition
(idL) $\mathrm{id} \circ f = f$ (idR) $f = f \circ \mathrm{id}$
(assoc) $(f \circ g) \circ h = f \circ (g \circ h)$

Bi-functoriality of $\otimes$
($\otimes$-id) $\mathrm{id}_A \otimes \mathrm{id}_B = \mathrm{id}_{A \otimes B}$
($\otimes$-comp) $(h \otimes k) \circ (f \otimes g) = (h \circ f) \otimes (k \circ g)$

Laws of natural transformations λ, ρ, α
(λ-nat) $\lambda \circ (\mathrm{id} \otimes f) = f \circ \lambda$ (ρ-nat) $(f \otimes \mathrm{id}) \circ \rho = \rho \circ f$
(α-nat) $\alpha \circ (f \otimes g) \otimes h = f \otimes (g \otimes h) \circ \alpha$

Laws of skew monoidal category
(a) $\lambda \circ \rho = \mathrm{id}$ (b) $(\mathrm{id} \otimes \lambda) \circ \alpha \circ (\rho \otimes \mathrm{id}) = \mathrm{id}$
(c) $\lambda \circ \alpha = \lambda \otimes \mathrm{id}$ (d) $\alpha \circ \rho = \mathrm{id} \otimes \rho$
(e) $(\mathrm{id} \otimes \alpha) \circ \alpha \circ (\alpha \otimes \mathrm{id}) = \alpha \circ \alpha$

These laws can be regarded as textual representation of basic commutative diagrams. Namely, these specify which two arrows having the same source and target are equal (i.e. commute). For example, the law (e) is formally of the following form:

$$\vdash (\mathrm{id}_A \otimes \alpha_{B,C,D}) \circ \alpha_{A,B\otimes C,D} \circ (\alpha_{A\otimes B,C,D} \otimes \mathrm{id}_D) = \alpha_{A,B,C\otimes D} \circ \alpha_{A\otimes B,C,D} : \mathrm{Hom}((A \otimes B) \otimes C) \otimes D, A \otimes (B \otimes (C \otimes D)))$$

This specifies the equality of two arrows from $(A \otimes B) \otimes C) \otimes D$ to $A \otimes (B \otimes (C \otimes D))$, equivalently saying that the diagram (e) commutes.

In the description of **SkewMon**, we omitted the subscripts of $\mathrm{id}, \lambda, \rho, \alpha$ and types, but the laws officially have suitable subscripts, contexts and types, as the above highlights. Note that different occurrences of "α" has different subscripts. This is actually the point we have mentioned in Problem 2 in §1.4. Describing correct subscripts manually for all the rules is tedious, but necessary to check overlaps between rules.

Since confluence implies the unique normal form property [12,7], we show confluence of the computation rules obtained from **SkewMon** by orienting the laws as left-to-right computation rules.

7.2. Proof with PolySOL

We formulate a skew-monoidal category by using a signature

```
sigskew = [signature|
 times : Hom(a,c),Hom(b,d) -> Hom(Pr(a,b),Pr(c,d))
 app   : Hom(b,c),Hom(a,b) -> Hom(a,c)      ;  id  : Hom(a,a)
 lmd   : Hom(Pr(I,a),a)                     ;  rho : Hom(a,Pr(a,I))
 alpha : Hom(Pr(Pr(a,b),c),Pr(a,Pr(b,c)))                          |]
```

where we assume the binary type constructors `Hom` and `Pr` for the hom-sets and the monoidal product $\otimes$, and the function symbol `times` (which is written as the infix "$*$" below) is for arrow's $\otimes$, `app` (which is written as the infix "`@`" below) for the composition $\circ$ and the rest are for $\mathrm{id}, \lambda, \rho, \alpha$. We formulate the laws as computation rules in PolySOL, where capital letters such as `F`, `G` are metavariables for arrows.

```
skew = [rule|
 (idL)     id @ F => F    ;   (idR) F @ id => F
 (ox-id)   id * id                => id
```

```
(ox-comp) (H * K) @ (F * G)        => (H @ F) * (K @ G)
;
(alpha-nat) alpha @ ((F * G) * H) => (F * (G * H)) @ alpha
(lmd-nat)   lmd @ (id * F)         => F @ lmd
(rho-nat)   (F * id) @ rho         => rho @ F
;
(a)  lmd @ rho    => id
(b)   (id * lmd) @ (alpha @ (rho * id)) => id
(b') ((id * lmd) @ alpha) @ (rho * id)  => id
(c)  lmd @  alpha => lmd * id
(d)  alpha @ rho  => id * rho
(e)   (id * alpha) @ (alpha @ (alpha * id)) => alpha @ alpha
(e') ((id * alpha) @ alpha) @ (alpha * id)  => alpha @ alpha
|]
```

The rules are straightforward implementation of the original laws, except for the following point:

- We did not include the associative law of the composition:

```
(assoc) (F @ G) @ H => F @ (G @ H)
```

- Instead, we added the laws `(b')` and `(e')`, which are variants of `(b)` and `(e)`, where we applied `(assoc)` to their lhss.

The reason of this modification is adding `(assoc)` as a rule greatly increases critical pairs. Since the lhss of almost all rules of `skew` have the composition `@`, they overlap with the subterm `(F @ G)` of lhs of `(assoc)`. Many CPs generated by this way are not joinable. Therefore, we decided `(assoc)` should not be considered as a computation rule. Instead, we regard `@` as a flatten composition operator like a string concatenation in string rewriting system.

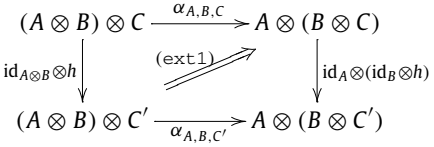
By invoking the command "`crity`" for **cri**tical pair checking with **ty**pes, PolySOL checks their critical pairs using the inferred well-typed rules. It reports one non-joinable critical pair out of 6.

```
*SOL> crity skew sigskew
1: Overlap (idL)-(idR)--- F'|-> id, F|-> id ------------------------------------
   (idL) (id@F) => F
   (idR) F'@id => F'
                                  id@id
                        id <-(idL)-^-(idR)-> id
                          ---> id =OK= id <---
2: Overlap (ox-comp)-(ox-id)--- H|-> id, K|-> id ------------------------------
   (ox-comp) (H * K)@(F * G) => (H@F) * (K@G)
   (ox-id) id * id => id
                           (id * id)@(F * G)
      (id@F) * (id@G) <-(ox-comp)-^-(ox-id)-> id@(F * G)
                      ---> F * G =OK= F * G <---
3: Overlap (ox-comp)-(ox-id)--- F|-> id, G|-> id ------------------------------
   (ox-comp) (H * K)@(F * G) => (H@F) * (K@G)
   (ox-id) id * id => id
                           (H * K)@(id * id)
      (H@id) * (K@id) <-(ox-comp)-^-(ox-id)-> (H * K)@id
                      ---> H * K =OK= H * K <---
4: Overlap (lmd-nat)-(ox-id)--- F|-> id ----------------------------------------
   (lmd-nat) lmd@(id * F) => F@lmd
   (ox-id) id * id => id
                             lmd@(id * id)
                  id@lmd <-(lmd-nat)-^-(ox-id)-> lmd@id
                        ---> lmd =OK= lmd <---
5: Overlap (rho-nat)-(ox-id)--- F|-> id ----------------------------------------
   (rho-nat) (F * id)@rho => rho@F
   (ox-id) id * id => id
                             (id * id)@rho
                  rho@id <-(rho-nat)-^-(ox-id)-> id@rho
                        ---> rho =OK= rho <---
6: Overlap (alpha-nat)-(ox-id)--- F|-> id, G|-> id -----------------------------
   (alpha-nat) alpha@((F * G) * H) => (F * (G * H))@alpha
   (ox-id) id * id => id
                            alpha@((id * id) * H)
 (id * (id * H))@alpha <-(alpha-nat)-^-(ox-id)-> alpha@(id * H)
      ---> (id * (id * H))@alpha =#= alpha@(id * H) <---
#NON 1 joinable... (Total 6 CPs)
```

To make the non-joinable CP `(6:)` joinable, we add a new computation rule

```
skewext = [rule|
  (ext1)   alpha@(id * H) => (id * (id * H))@alpha |]
```

This is a sound rule, because it is nothing but the naturality of α.



Note that each `alpha` and `id` in the rule `(ext1)` have officially different subscripts as the above diagram shows. They are automatically supplied in PolySOL by the type inference algorithm.

The diagram shows also a "2-cell" `(ext1)` transforming the down-lower-right arrow to the upper-right-down arrow, which is considered as a "proof" why the diagram commutes. Likewise, every rule in `skew` is considered as such a "proof", therefore if two arrows (represented as meta-terms) with the same source and target has the unique normal form using `skew`, then it has a proof of commutativity.

We check again `skew` with the extended rule `skewext`.

```
*SOL> crity (skewext++skew) sigskew
..
4: Overlap (alpha-nat)-(ox-id)--- F|-> id, G|-> id -----------------------------
   (alpha-nat) alpha@((F * G) * H) => (F * (G * H))@alpha
   (ox-id) id * id => id
                       alpha@((id * id) * H)
(id * (id * H))@alpha <-(alpha-nat)-^-(ox-id)-> alpha@(id * H)
      ---> (id * (id * H))@alpha =OK= (id * (id * H))@alpha <---
..
7: Overlap (ext1)-(ox-id)--- H|-> id ----------------------------------------
   (ext1) alpha@(id * H) => (id * (id * H))@alpha
   (ox-id) id * id => id
                          alpha@(id * id)
  (id * (id * id))@alpha <-(ext1)-^-(ox-id)-> alpha@id
                     ---> alpha =OK= alpha <---
#Joinable! (Total 7 CPs)
```

PolySOL reports 7 critical pairs, which are all successfully joinable. This shows local confluence of the computation system `(skewext++skew)`. We next consider critical pairs generated by the flatten composition. This is by computing overlaps between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$, where the prefix (w.r.t. the composition) of l_1 is unifiable with the suffix (w.r.t. the composition) of l_2, as critical pairs of string rewriting. There are 7 overlaps: (b)-(e), (λ-nat)-(e), (λ-nat)-(ρ-nat), (ρ-nat)-(b), (ρ-nat)-(e), (α-nat)-(c), (c)-(d) and all of these are joinable.

Finally, we check strong normalisation of `skew` using PolySOL's command `solsn`.

```
*SOL> solsn skew sigskew
..
This is a first-order system.

******** FO SN check ********
Check SN using NaTT (Nagoya Termination Tool)
Input TRS:
    1: app(id(),F) -> F
    2: app(F,id()) -> F
    3: times(id(),id()) -> id()
..
Direct POLO(bPol) ... removes: 4 8 1 3 5 10 7 14 12 11 9 13 6 2
      times     w: 2 * x1 + 2 * x2 + 2
      lmd       w: 1
      id        w: 1
      rho       w: 1
      alpha     w: 4
      app       w: x1 + 2 * x2 + 1
Number of strict rules: 0
>>YES
```

Since the types of all the function symbols in `sigskew` are up to first-order (N.B. `Hom` is just an algebraic data type constructor, not the arrow type constructor), PolySOL tried to call an external first-order termination checker NaTT (Nagoya Termination Tool) [36], and NaTT succeeded in proving termination. The technique used there was to assign a polynomial to each function symbol as

$$[\![\otimes]\!](x_1, x_2) = 2x_1 + 2x_2 + 2, \quad [\![\circ]\!](x_1, x_2) = x_1 + 2x_2 + 1, \quad [\![\lambda]\!] = [\![\mathrm{id}]\!] = [\![\rho]\!] = 1, \quad [\![\alpha]\!] = 4$$

Then all rules are strictly decreasing using the interpretation. Hence we conclude that it is confluent, which implies that **SkewMon** has the unique normal form property.

Moreover, we have an additional result: the theory of **Skew** is decidable, meaning that any equality on the arrows is decidable and checked by rewriting to normal forms.

8. Implementation of PolySOL

In this section, we describe some details of the PolySOL system. The PolySOL system consists of about 8000 line Haskell codes, and works on GHCi, the interpreter of Glasgow Haskell Compiler (tested on version 7.6.2 and 8.0.2). PolySOL uses Template Haskell [16] with a custom parser generated by Alex (for lexer) and Happy (for parser) to provide a readable notation for signatures and rules. There is a command line interface using GHCi (§8.2). There is also a web interface (§8.3).

8.1. Syntax

PolySOL's language is realised as an embedded domain specific language in Haskell by using the quasi-quotation feature of Template Haskell. The syntax `[signature|..]` and `[rule|..]` are quasi-quotations, and the keywords `signature` and `rule` are implemented as "quasi-quoters" of Template Haskell, which parse the contents of strings and return the algebraic data of parse trees. PolySOL's syntax has a simple layout rule. Newline is regarded as a separator of declarations in signature and rules. For example,

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

If one wants to write the declarations sequentially, one can also use the semicolon "`;`" for the separator, e.g.

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

The label for a rule is a string starting with a lower case letter, enclosed by round brackets, such as "`(beta)`", which is placed before a rule.

PolySOL has the following naming rule for identifiers. The start symbol of an identifier should be

- function symbol: lower case letter, e.g., `lam`, `app`
- metavariable: capital letter, e.g., `M`, `N1`
- (bound) variable: lower case letter, e.g., `x`, `y0`
- type constructor: capital letter, e.g., `Arr`, `Hom`
- type variable: lower case letter, e.g. `a`, `b`

8.2. The command line interface

The command line interface is the most fundamental use of PolySOL. The user invokes GHCi and loads a file containing the definition of signature and rules. For example, by invoking

```
% ghci 01need.hs
```

PolySOL starts with loading the file "`01need.hs`", which contains the line

```
import SOL
```

to import PolySOL system and definitions `siglamC` and `lamC` given in §6. The file "`01need.hs`" is also available in the web inference.

To check confluence or strong normalisation of the computation system, the user invokes a SOL's command implemented as a Haskell function such as

```
*SOL> criCBV lamC siglamC
```

for critical pair checking of computation using value/non-value distinction in §6. Another example is

```
*SOL> crity skew sigskew
```

for critical pair checking with types as done in §7.

PolySOL has the following commands implemented as Haskell functions:

Result	CSI^ho	PolySOL
YES	49	66
NO	15	16
MAYBE	30	12
TOTAL	94	94

The table shows the number of decided results by each tool. The result YES/NO is a decided result of CR. MAYBE is a result when the tool cannot decide. The total of a tool is the number of problems. The results and detailed outputs of each tool in the competition were recorded in the official site: http://cops.uibk.ac.at/results/?y=2018&c=HRS.

Fig. 9. Results of HRS category in Confluence Competition 2018.

- `criCBV <rules> <signature>`
  Enumerating critical pairs of a polymorphic computation system using value/non-value distinction.
- `crity <rules> <signature>`
  Enumerating critical pairs of a polymorphic computation system.
- `sn <rules> <signature>`
  Checking SN of a computation system using the General Schema criterion.
- `solsn <rules> <signature>`
  Checking SN of a computation system using multiple methods.
- `cr <rules> <signature>`
  Checking confluence of a computation system using multiple methods.

8.3. Web interface

There is also a web interface for PolySOL, which is available at the first author Makoto Hamana's homepage. The examples in this paper have been stored in the web interface and one can chose an example from the pull-down menu.

PolySOL supports two file formats:

- SOL format (`.hs`) implemented as a Haskell embedded domain specific language used throughout the paper, and
- TRS format (`.trs`) used in the Confluence Competition.

The TRS format is also used in the evaluation discussed in the next section.

9. Evaluation

9.1. Results in Confluence Competition 2018

PolySOL participated in the Higher-Order Rewriting (HRS) category of the International Confluence Competition 2018 (CoCo'18)[1] held at the Federated Logic Conference (FLoC 2018) in Oxford, U.K. [37].

This event had automatic tools competing for the number of decided results of confluence problems. We inferred that participation in the competition would demonstrate how PolySOL was more effective than existing tools.

In the HRS category, 94 problems of higher-order rewrite systems (which are only simply-typed, not polymorphic) [8, 9] were given. Because our polymorphic computation systems subsume the second-order fragment of higher-order rewrite systems, PolySOL is able to address the simply-typed confluence problems that are provided for the competition. We have adapted PolySOL to support the TRS format, which is the format of rewrite systems in the competition.

Two tools participated in the HRS category: CSI^ho [38] and PolySOL (named SOL there). Results show that PolySOL solved more problems than CSI^ho (see Fig. 9). This outcome derives from the fact that PolySOL implemented new criteria of confluence, strong closedness, and modular checking developed in §5.

9.2. Benchmark

To assess the behaviour of these tools more precisely, we again tested the tools against the problems of the Confluence Computation 2018 in our own environment. We conducted the benchmark test on a machine with Intel(R) Xeon E7-4809, 2.00 GHz 4 CPU (8 cores each), 256 GB memory, Red Hat Enterprise Linux 7.3, and timeout set to 120 s. The results are described on a relevant web page[2]. We excerpt the results of problems 514–789 in Fig. 10 with short comments in the final column of the table. Comments show the methods used by PolySOL to solve problems. The new confluence criteria

[1] Confluence Competition official site: http://project-coco.uibk.ac.at/2018/.
[2] http://www.cs.gunma-u.ac.jp/hamana/polysol/scpro.html

Prob. No.	CSI^ho	(sec.)	PolySOL	(sec.)	Used method
514	YES	1.448	YES	0.412	
515	YES	0.772	YES	0.032	
516	YES	1.459	YES	0.041	
517	MAYBE	0.482	NO	0.03	non-joinable CP check
518	MAYBE	0.617	NO	0.043	non-joinable CP check
723	MAYBE	58.633	MAYBE	0.489	
724	YES	0.734	YES	0.023	
725	NO	0.747	NO	0.041	
726	YES	0.676	YES	0.03	
727	NO	0.656	NO	0.023	
728	MAYBE	58.637	YES	0.33	joinable CP check
729	MAYBE	58.703	MAYBE	0.341	
730	YES	0.698	YES	0.614	
746	MAYBE	0.517	MAYBE	0.023	
747	MAYBE	0.586	MAYBE	0.029	
748	MAYBE	0.508	MAYBE	0.316	
749	MAYBE	0.559	NO	0.05	non-joinable CP check
750	MAYBE	0.669	YES	0.046	joinable CP check
751	MAYBE	0.689	YES	0.045	joinable CP check
752	MAYBE	0.671	MAYBE	0.319	
758	MAYBE	0.631	NO	0.079	non-joinable CP check
759	MAYBE	58.642	MAYBE	0.039	
764	NO	0.635	NO	0.03	
766	NO	0.566	MAYBE	0.023	
767	NO	0.594	MAYBE	0.302	
768	MAYBE	2.052	YES	0.279	joinable CP check
769	MAYBE	0.504	YES	0.038	joinable CP check
772	MAYBE	58.566	YES	0.027	strong closedness
773	MAYBE	0.59	NO	0.035	non-joinable CP check
775	MAYBE	58.705	YES	0.026	strong closedness
776	MAYBE	58.721	YES	15.117	modularity
777	NO	0.699	NO	0.038	
780	MAYBE	0.611	YES	0.239	joinable CP check
781	MAYBE	58.719	YES	0.186	strong closedness
782	MAYBE	58.571	YES	0.028	strong closedness
783	MAYBE	58.573	YES	0.027	strong closedness
784	MAYBE	0.656	MAYBE	0.054	
785	MAYBE	0.645	MAYBE	0.059	
789	MAYBE	0.5	MAYBE	0.027	

The unit of checking time shown in the next column of each result is second.

Fig. 10. Detailed results for problems used in Confluence Competition 2018 (excerpt).

are effective. CSI^ho reports counter-examples to confluence for some cases (problems 766 and 767) that PolySOL cannot find. After the competition, we improved PolySOL for more exact matching with the specifications of rewrite systems used in the competition. For this reason, some results of PolySOL in the above benchmark URL show changes or improvements from those in the competition.

9.3. On checking speed

Another interesting point in this benchmark is checking speed. Results show that PolySOL is more than 10 times faster than CSI^ho for almost all of the problems. One reason might be that PolySOL has its own termination checker, which is simple and fast [1]. However, CSI^ho sometimes calls an external higher-order termination checker, which is generally expensive. For PolySOL, we directly implemented the theory of confluence checking developed in this paper using no clever implementation trick. The benchmark result demonstrates that, for the case of confluence, such a simple and theoretically sound strategy is an effective method to implement an automated checking tool.

10. Summary and related work

10.1. Summary

We have presented a new framework of polymorphic computation rules that can accommodate a distinction between values and non-values. The framework was demonstrated to be suitable for formulating and analysing fundamental calculi of programming languages. We have given a type inference algorithm and criteria to check the confluence property of polymorphic rules. These have provided a handy method to prove confluence of polymorphic second-order computation rules with call-by-value. We have demonstrated the effectiveness of our methodology by examining sample calculi using our automated confluence checking tool, PolySOL.

10.2. Related work

Mayr and Nipkow studied critical pairs for the confluence of higher-order rewrite systems [8,9]. Their rewrite rule format was rules on simply-typed λ-terms modulo $\beta\eta$-equivalence. There are *no* polymorphic types nor value/non-value distinctions. Therefore, no example (λ_C, λ_{NEED} and skew monoidal category) which has confluence described in this paper can be formulated directly or can be checked in their framework.

The systems ACPH [39] and CSI^ho [38] can automatically check confluence of higher-order rules. They are based on the Mayr–Nipkow higher-order rule format. Therefore, these tools have neither features of polymorphic types nor value/non-value distinction. Therefore, all of our examples are beyond the scope of the existing confluence checking systems. In this respect, this report is the first description of our system, to the best of our knowledge, the first automatic tool that can check confluence of the call-by-value variants of the λ-calculus directly, as described in the paper.

A framework of polymorphic higher-order rewrite rules was presented by Jouannaud et al. [40,41]. Our framework is similar, but with several fundamentally important differences. For instance, our framework is based on (polymorphic) second-order algebraic theories [3,42], whereas theirs is based on a polymorphic λ-calculus. The main purpose of [40,41] was establishment of a termination criterion for higher-order rewrite rules. The issue of confluence of polymorphic rules has remained unaddressed. To the best of our knowledge, the present paper is the first to describe a study of confluence of a general kind of polymorphic second-order rewrite rules.

We presented a type inference algorithm of polymorphic computation rules, which has not been given in the context of rewriting theory, although it might 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 existing higher-order confluence tools such as ACPH and CSI^ho. These tools force the user to write more detailed type information in rewrite rule specifications than PolySOL. The user needs to declare all free and bound variables used in the rules, with their types. Actually, PolySOL's rule specification is simpler because of the type inference. Our algorithm might also be beneficial to other tools to improve this situation.

For our earlier work [1], we developed a simply-typed framework of second-order equational logic and computation rules. We also developed a tool SOL for checking methods of termination confluence. It lacked proper polymorphism and the treatment of call-by-value. For that reason, the examples examined in this paper could not be handled directly.

Our earlier paper [42] presented a general framework of multiversal polymorphic algebraic theories based on polymorphic abstract syntax [43], and developed polymorphic equational logic and their algebraic models. It admits multiple type universes and higher-kinded polymorphic types, hence it is richer than the present setting. However, we developed neither polymorphic computation rules, call-by-value, nor their confluence.

Acknowledgements

The first author is grateful to Masahito Hasegawa for his question about methods to check confluence of Moggi's computational λ-calculus using the earlier tool, SOL. The author was thereby enlightened to the necessity of proper treatment of call-by-value calculi, which led to the framework described in this paper. This work was partly supported by JSPS KAKENHI Grant Numbers JP17K00092, JP17K00005 and JP19K11891.

References

[1] M. Hamana, How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation, Proc. ACM Program. Lang. 1 (22) (2017) 1–28.
[2] M. Fiore, C.-K. Hur, Second-order equational logic, in: Proc. of CSL'10, in: LNCS, vol. 6247, 2010, pp. 320–335.
[3] M. Fiore, O. Mahmoud, Second-order algebraic theories, in: Proc. of MFCS'10, in: LNCS, vol. 6281, 2010, pp. 368–380.
[4] S. Staton, An algebraic presentation of predicate logic, in: Proc. of FoSSaCS'13, 2013, pp. 401–417.
[5] S. Staton, Instances of computational effects: an algebraic perspective, in: Proc. of LICS'13, 2013, pp. 519–528.
[6] S. Staton, Algebraic effects, linearity, and quantum programming languages, in: Proc. of POPL'15, 2015, pp. 395–406.
[7] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[8] T. Nipkow, Higher-order critical pairs, in: Proc. 6th IEEE, in: Symp. Logic in Computer Science, 1991, pp. 342–349.
[9] R. Mayr, T. Nipkow, Higher-order rewrite systems and their confluence, Theor. Comput. Sci. 192 (1) (1998) 3–29.
[10] G.D. Plotkin, Call-by-name, call-by-value and the lambda-calculus, Theor. Comput. Sci. 1 (2) (1975) 125–159.
[11] J. Maraist, M. Odersky, P. Wadler, The call-by-need lambda calculus, J. Funct. Program. 8 (3) (1998) 275–317.
[12] G. Huet, Confluent reductions: abstract properties and applications to term rewriting systems, J. ACM 27 (4) (1980) 797–821.
[13] Y. Ohta, M. Hasegawa, A terminating and confluent linear lambda calculus, in: Proc. of RTA'06, 2006, pp. 166–180.
[14] D. Knuth, P. Bendix, Simple word problems in universal algebras, in: Computational Problem in Abstract Algebra, Pergamon Press, Oxford, 1970, pp. 263–297.
[15] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, J. Log. Comput. 1 (4) (1991) 497–536.
[16] T. Sheard, S.P. Jones, Template metaprogramming for Haskell, in: Proc. Haskell Workshop, 2002, 2002.
[17] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, North Holland, 1984.
[18] M. Hamana, Polymorphic rewrite rules: confluence, type inference, and instance validation, in: Proc. of 14th International Symposium on Functional and Logic Programming (FLOPS'18), in: LNCS, vol. 10818, 2018, pp. 99–115.
[19] M. Hamana, Universal algebra for termination of higher-order rewriting, in: Proc. of RTA'05, in: LNCS, vol. 3467, 2005, pp. 135–149.
[20] M. Hamana, Higher-order semantic labelling for inductive datatype systems, in: Proc. of PPDP'07, ACM Press, 2007, pp. 97–108.
[21] M. Hamana, Semantic labelling for proving termination of combinatory reduction systems, in: Proc. WFLP'09, in: LNCS, vol. 5979, 2010, pp. 62–78.
[22] M. Hamana, Free Σ-monoids: a higher-order syntax with metavariables, in: Proc. of APLAS'04, in: LNCS, vol. 3302, 2004, pp. 348–363.

[23] M. Hamana, Correct looping arrows from cyclic terms: traced categorical interpretation in Haskell, in: Proc. of FLOPS'12, in: LNCS, vol. 7294, 2012, pp. 136–150.
[24] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proc. POPL'82, 1982, pp. 207–212.
[25] R. Milner, Communicating and Mobile Systems – The π-Calculus, CUP, 1999.
[26] T. Suzuki, K. Kikuchi, T. Aoto, Y. Toyama, Critical pair analysis in nominal rewriting, in: Proc. of SCSS'16, 2016, pp. 156–168.
[27] P. Aczel, A General Church-Rosser Theorem, Tech. rep., University of Manchester, 1978.
[28] M. Hamana, K. Matsuda, K. Asada, The algebra of recursive graph transformation language UnCAL: complete axiomatisation and iteration categorical semantics, Math. Struct. Comput. Sci. 28 (2) (2018) 287–337, https://doi.org/10.1017/S096012951600027X.
[29] E. Moggi, Computational Lambda-Calculus and Monads, LFCS ECS-LFCS-88-66, University of Edinburgh, 1988.
[30] A. Sabry, P. Wadler, A reflection on call-by-value, ACM Trans. Program. Lang. Syst. 19 (6) (1997) 916–941.
[31] S. Lindley, I. Stark, Reducibility and $\top\top$-lifting for computation types, in: Proc. of TLCA'05, 2005, pp. 262–277.
[32] K. Szlachányi, Skew-monoidal categories and bialgebroids, Adv. Math. 231 (3–4) (2012) 1694–1730.
[33] T. Uustalu, Coherence for skew-monoidal categories, in: Proc. 5th Workshop on Mathematically Structured Functional Programming, MSFP'14, 2014, pp. 68–77.
[34] S. Mac Lane, Categories for the Working Mathematician, Graduate Texts in Mathematics, vol. 5, Springer-Verlag, 1971.
[35] T. Altenkirch, J. Chapman, T. Uustalu, Monads need not be endofunctors, Log. Methods Comput. Sci. 11 (1) (2015).
[36] A. Yamada, K. Kusakari, T. Sakabe, Nagoya termination tool, in: Proc. Joint 25th RTA and 12th TLCA, in: LNCS, vol. 8560, 2014, pp. 466–475.
[37] T. Aoto, M. Hamana, N. Hirokawa, A.M.J. Nagele, N. Nishida, K. Shintani, H. Zankl, Confluence Competition 2018, in: Proc. of 3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, in: LIPIcs, vol. 108, 2018, chap. 32, 5 pp.
[38] J. Nagele, B. Felgenhauer, A. Middeldorp, CSI: new evidence – a progress report, in: Proc. of CADE'17, in: LNCS (LNAI), vol. 10395, 2017, pp. 385–397.
[39] K. Onozawa, K. Kikuchi, T. Aoto, Y. Toyama, ACPH: system description, in: 6th Confluence Competition (CoCo 2017), 2017, p. 76.
[40] J.-P. Jouannaud, A. Rubio, Polymorphic higher-order recursive path orderings, J. ACM 54 (1) (2007) 2, 48 pp.
[41] J.-P. Jouannaud, A. Rubio, Normal higher-order termination, ACM Trans. Comput. Log. 16 (2) (2015) 13, 38 pp.
[42] M. Fiore, 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, 2013, pp. 520–529.
[43] M. Hamana, Polymorphic abstract syntax via Grothendieck construction, in: FoSSaCS'11, in: LNCS, vol. 3467, 2011, pp. 381–395.