JFP **29**, e20, 53 pages, 2019.
doi:10.1017/S0956796819000157

How to prove decidability of equational theories with second-order computation analyser SOL

MAKOTO HAMANA
Gunma University, Kiryu, Gunma, Japan
(*e-mail:* hamana@cs.gunma-u.ac.jp)

Abstract

We present a general methodology of proving the decidability of equational theory of programming language concepts in the framework of second-order algebraic theories. We propose a Haskell-based analysis tool, i.e. Second-Order Laboratory, which assists the proofs of confluence and strong normalisation of computation rules derived from second-order algebraic theories. To cover various examples in programming language theory, we combine and extend both syntactical and semantical results of the second-order computation in a non-trivial manner. We demonstrate how to prove decidability of various algebraic theories in the literature. It includes the equational theories of monad and λ-calculi, Plotkin and Power's theory of states and bits, and Stark's theory of π-calculus. We also demonstrate how this methodology can solve the coherence of monoidal categories.

A video abstract can be found at: https://vimeo.com/365486403

1 Introduction

Equations and equational reasoning are ubiquitous in functional programming in pure (Bird & Moor, 1996) and even impure setting (Gibbons & Hinze, 2011). In a programming language paper, one often defines one's own calculus or algebra by giving a set of equational laws or reduction rules. Then, the decidability of the equational theory of a calculus or algebra of programming language is useful for reasoning, verification, and program transformation. If one knows the decidability, one can use a decidable test of an equation for type checking, compilation, optimisation, etc. Therefore, the decidability of equational theory is important for programming languages in theory and practice. The purpose of this paper is to present a general methodology of proving the decidability of equational theory with the assistance of our Haskell-based analysis tool ***SOL*, *Second-Order Laboratory***.

1.1 Monad [Problem #1]

First, we demonstrate how to prove that a sample equational theory is decidable. We consider monads (Moggi, 1991), important and indispensable features in functional programming (Wadler, 1990), especially in Haskell. A monad T is a structure with two operations

$$\mathsf{return} : a \to Ta \qquad \gg\!= : Ta \to (a \to Tb) \to Tb$$

satisfying three laws

$$\text{(unitL)} \qquad \mathsf{return}(x) \gg\!= \lambda y.k\,y \;=\; k\,x$$

$$\text{(unitR)} \qquad e \gg\!= \lambda y.\mathsf{return}(y) \;=\; e$$

$$\text{(assoc)} \qquad (e \gg\!= \lambda x.k\,x) \gg\!= \lambda y.\ell\,y \;=\; e \gg\!= \lambda x.(k\,x \gg\!= \lambda y.\ell\,y)$$

This is well known. However, the following is perhaps less known:

The theory of monad is *decidable*.

By theory, we mean *equational theory*, which is the set of all equationally provable equations (i.e. *theorems*) under a given set of axioms. In this case, axioms are the three laws of monad, and the problem is the following:

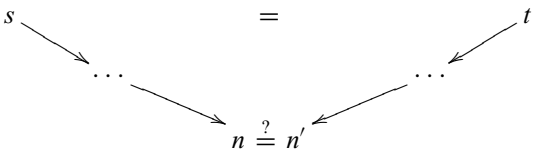
> Given two well-typed terms s, t consisting of return, $\gg\!=$, and variables, is an equation
>
> $$s = t$$
>
> derivable from the three laws of monad?

This problem is decidable; that is, there is a terminating algorithm answering yes or no for any two terms. How do we prove it? In the following, we sketch a rewriting theoretic proof and a practical method to prove the decidability of the theory of monad.

1.2 Theoretical method

Proof by rewriting. A principal idea to obtain the decidability of equational theory in this paper is to extend a rewriting method originated by Knuth and Bendix (1970). They solved the decidability of the theory of a group, known as a *word problem*, by obtaining confluent and terminating rewrite rules to compute normal forms. Their method for deciding an equality between expressions of a group is first to orient the given equational axioms E, as left-to-right rewrite rules, and next to transform them to a set of confluent and terminating rewrite rules R. Finally, for a given equation $s = t$, the proof method is to rewrite both sides of the equation to normal forms using R, and to compare them. If two normal forms are the same term, then we conclude $s = t$ is derivable from the axioms E. Otherwise, it is not derivable by the equational logic. To make this method correct and decidable, we must ensure that the rewriting rules are terminating and confluent. Termination is to reach the normal form in finite time. Confluence ensures the existence of unique normal forms. A given set E of equational axioms is not always terminating or confluent, hence they developed an algorithm to transform the rules to have these properties.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press



Normal forms

In this paper, we follow this described general methodology of proving equations by rewriting as a foundational method. Our principal interest is to give a methodology that is applicable to programming language theory. In this respect, our method is *not* merely a straightforward application of the classical Knuth and Bendix's approach. In the case of Knuth and Bendix, a group is axiomatised as an ordinary algebraic theory. In other words, all the axioms are built on algebraic terms without having any higher-order terms. The theory of programming languages absolutely requires higher-order terms, such as λ-terms and let-expressions. Therefore, we extended the methodology to cover second-order algebraic theories. One important observation related to our development is that calculi and algebras for programming languages are often described naturally as *second-order algebraic theories* (Fiore & Hur, 2010; Fiore & Mahmoud, 2010). Second-order algebraic theories are founded on the mathematical theory of second-order abstract syntax (Fiore et al., 1999; Hamana, 2004; Fiore, 2008). Staton has shown that second-order algebraic theories are a useful framework that models various important notions of programming languages, such as logic programming (Staton, 2013a), algebraic effects (Staton, 2013b), and quantum computation (Staton, 2015). Recently, the present author modelled cyclic computation using second-order algebraic theories (Hamana, 2016). This paper presents another practical application of second-order algebraic theories to investigate decidability.

To cover various examples in programming language theory, we combine and extend the results of computation on second-order algebraic theories, including both rewriting and semantical methods, in a non-trivial manner. The main problems are how to prove the termination and confluence of a given second-order equational theory, regarded as *second-order computation rules*, correctly, effectively, and in a simple manner. This paper presents a solid and useful methodology for them using our tool: SOL.

Monad laws as computation rules. We turn to the problem of the theory of monad. To use the rewriting method, first we orient each axiom from left to right:

$$\text{(unitL)} \quad \mathsf{return}(X) \gg\!= y.K[y] \;\Rightarrow\; K[X]$$

$$\text{(unitR)} \quad E \gg\!= y.\mathsf{return}(y) \;\Rightarrow\; E$$

$$\text{(assoc)} \quad (E \gg\!= x.K[x]) \gg\!= y.L[y] \;\Rightarrow\; E \gg\!= x.(K[x] \gg\!= y.L[y])$$

We use these as a schema of computation. Therefore, free variables are important as the targets of instantiation by pattern matching. To clarify the distinction of free and bound variables, we write the free variables in capitals, and maintain bound variables as small letters. To simplify and clarify the term structure for further syntactic analysis of computation, we omit the λ symbol and use the square bracket $K[y]$ to denote an application of a term to a free variable K.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

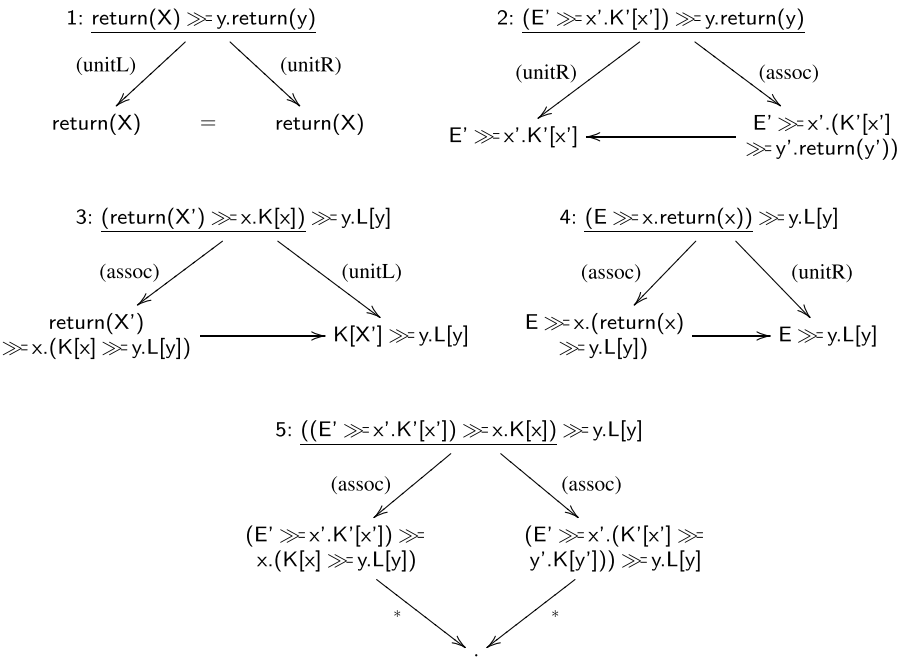


Fig. 1. Overlaps in monad laws.

Confluence. Next, we prove confluence of this computation rule set, named C. To the best of our knowledge, confluence of the monad laws has not been explicitly proved (even stated) in the literature. Our proof is the following. Careful inspection of C reveals that it has, in all, five patterns of overlap situations as depicted in Figure 1, each of which admits two ways of reductions. The underline represents the rewritten part (i.e. redex) by the right path of rewriting. The whole term (involving an underlined subterm) is rewritten by the left path. For example, in the overlap number 3: $\underline{(\mathsf{return}(\mathsf{X}') \gg\!= \mathsf{x}.\mathsf{K}[\mathsf{x}])} \gg\!= \mathsf{y}.\mathsf{L}[\mathsf{y}]$, the underlined term is rewritten by (unitL), and the whole term (i.e. the root position) is rewritten by (assoc).

An *overlap* is such a situation that matches the left-hand sides of rules in two ways (Definition 7.7). The pair of divergent terms is called a ***critical pair*** (***CP*** for short) (Definition 7.10). The diagrams in Figure 1 show that all the critical pairs are joinable by further rewriting. Importantly, this finite number of checks is sufficient to conclude that all other infinite numbers of instances of the divergent situations[1] are convergent. This convergence is ensured by Theorem 7.11 and the property is called *local confluence* (***WCR*** for short, see Section 7.5), meaning that every possible one-step divergence is always joinable. By applying Newman's lemma (Lemma 7.6), stating "termination and local confluence imply confluence", we can conclude that C is confluent (***CR*** for short). Because it requires termination, we consider it next.

SN. We consider termination (***SN*** for short, meaning strong normalisation) of the computation rules C for monad. We warn that a naive proof by assigning some "weight" to each

[1] Because the five patterns above might occur in an arbitrary larger term context and because every metavariable can be instantiated by arbitrary term.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

term (such as a natural number calculated using a certain polynomial), which is decreasing in each rule, is typically dangerous for proving the termination of higher-order rules. For example, can we assign some simple "weight" to terms to prove termination of the β-reduction rule[2]

$$\lambda(x.M[x])@N \Rightarrow M[N]$$

of the simply typed λ-calculus? No simple weight is known. One reason is that M can contain many x. Therefore, N might be copied many times at the right-hand side, which makes the term larger. Also, there is a case that M does not contain x. Then the term size becomes smaller. It is not an easy problem. Only quite involved weights using ordinals (van de Pol, 1994) are known for such a "weight"-based proof, which is not readily applicable to other variations of rules.

The reason why we refer to the β-reduction rule here is that in case of monad, (unitL) has a similar flavour to the β-rule, as K is applied to X at the right-hand side. Therefore, we must regard that the termination of the monad laws is almost as difficult as termination of the typed λ-calculus.

The usual way of proving termination of a higher-order calculus is to employ the notion of reducibility or computability by Tait and Girard (1989). Particularly in this paper, we use Blanqui's General Schema criterion (2000; 2016) as a main proof method of termination of second-order rules. The General Schema is based on Tait's computability method.

To provide a quick overview, we skip presentation of a proof of SN. Instead, we borrow the existing result. Actually, SN of Moggi's computational meta-language (1988) has been shown by a translation into λ-calculus with sums (Benton et al., 1998), or using ⊤⊤-lifting method (Lindley & Stark, 2005). The rules C is a part of strongly normalising computational meta-language. Therefore, C is also strongly normalising. We conclude that the theory of monad is decidable.

1.3 Proposed method: How to prove decidability with **SOL**

Why SOL. What were the difficult parts of the proof of confluence and strong normalisation? Actually, the proof structure is not theoretically difficult for experienced researchers, but *practically difficult* because one must try to do the following:

i. enumerate all the overlaps of the rules without oversight;
ii. rewrite terms to check joinability, which is tedious when choosing suitable rules and redexes;
iii. define a suitable reducibility predicate and check the many syntactic conditions to prove SN.

These are often quite confusing and prone to error because of similar rules, terms, and variable names. Moreover, one often needs trial and error to develop a better calculus. Changing just a single rule might produce a huge number of overlaps, which must be checked again. Adding a new rule also produces overlaps and might affect termination,

[2] Milner used the same notation in Milner (1996, p. 271), https://books.google.co.jp/books?id=lGgPwQfeXSEC&pg=PA271 .

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

and might globally change the whole proof, such as a reducibility predicate. The redexes to check joinability are often a mixture of various binding/infix/prefix notations, which appear to be confusing and a source of mistakes. Calculation of critical pairs between higher-order rules needs *higher-order unification* (Huet, 1980; Miller, 1991), which is complicated for a human to check manually. Therefore, this paper presents a proposal of the tool SOL to eliminate such pains.

A usage scenario for SOL. We show how proofs can be completed more easily with the assistance of SOL. Our system SOL is implemented as an embedded domain-specific language (DSL) in Haskell. An intended scenario of the usage of SOL is the following. First, the user creates a Haskell script for a given problem by specifying a signature and axioms (or rules). Then in the Glasgow Haskell interpreter (GHCi), the user attempts several commands to analyse CR and SN. These commands are realised as Haskell functions. Therefore, the GHCi provides an interactive user interface for SOL. If the user is lucky, then this attempt just finishes by invoking a few commands. If not, the user modifies the specification of rules, and repeats the checking process of SN and CR until the user obtains sufficient computation rules.

Monad in SOL. We define the signature by

```
sigm = [signature|
 return : a -> T(a) ;  bind : T(a),(a -> T(b)) -> T(b) |]
```

and define the axioms by[3]

```
monad = [axiom|
 (unitL)  return(X) >>= y.K[y]       = K[X]
 (unitR)  E >>= y.return(y)          = E
 (assoc)  (E >>= x.K[x]) >>= y.L[y] = E >>= x.(K[x] >>= y.L[y]) |]
```

The reader might find that the description presented above in SOL exactly and naturally matches the mathematical definition of monads presented in Section 1.1. The keyword-headed bracket `[signature|..|]` or `[axiom|..|]` indicates the beginning and end of SOL's DSL using a feature of Template Haskell. Other than the bracketed parts are normal Haskell, but the inside of the brackets is completely the world of SOL, which has its own syntax designed to be close, to the greatest extent possible, to the ordinary mathematical meta-language used in formulating laws and calculi of programming languages. SOL regards each axiom as a computation rule transforming a term from left to right in textual order.

Confluence. Next we try to prove confluence (CR) using SOL. Invoking the command `cri` (which is short for **cri**tical pairs), SOL enumerates all possible overlap situations and checks the joinability of critical pairs.

[3] Several operators are reserved in SOL to be translated automatically to the corresponding function symbols for readability. Here the infix operator `>>=` is translated to the prefix function `bind`.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
*SOL> cri monad sigm
1: Overlap (unitL)-(unitR)--- N'|-> return(X), K|-> z1.return(z1) -------------
   (unitL) (return(X) >>=y.K[y]) => K[X]
   (unitR) N' >>=y'.return(y') => N'
                       return(X) >>=y.return(y)
             return(X) <-(unitL)-∧-(unitR)-> return(X)
                 ---> return(X) =OK= return(X) <---
2: Overlap (unitR)-(assoc)--- N|-> N' >>=x'.K'[x'], L'|-> z1.return(z1) -------
   (unitR) (N >>=y.return(y)) => N
   (assoc) (N' >>=x'.K'[x']) >>=y'.L'[y'] => N' >>=x'.(K'[x'] >>=y'.L'[y'])
                    (N' >>=x'.K'[x']) >>=y.return(y)
        N' >>=x'.K'[x'] <-(unitR)-∧-(assoc)-> N' >>=xd.(K'[xd] >>=yd.return(yd))
            ---> N' >>=x'.K'[x'] =E= N' >>=xd.K'[xd] <---
3: Overlap (assoc)-(unitL)--- N|-> return(X'), K'|-> z1.K[z1] -----------------
   (assoc) (N >>=x.K[x]) >>=y.L[y] => N >>=x.(K[x] >>=y.L[y])
   (unitL) return(X') >>=y'.K'[y'] => K'[X']
                    (return(X') >>=x.K[x]) >>=y.L[y]
return(X') >>=x.(K[x] >>=y.L[y]) <-(assoc)-∧-(unitL)-> K[X'] >>=y.L[y]
        ---> K[X'] >>=y.L[y] =E= K[X'] >>=y.L[y] <---
4: Overlap (assoc)-(unitR)--- N'|-> N, K|-> z1.return(z1) ---------------------
   (assoc) (N >>=x.K[x]) >>=y.L[y] => N >>=x.(K[x] >>=y.L[y])
   (unitR) N' >>=y'.return(y') => N'
                      (N >>=x.return(x)) >>=y.L[y]
N >>=x.(return(x) >>=y.L[y]) <-(assoc)-∧-(unitR)-> N >>=y.L[y]
            ---> N >>=x.L[x] =E= N >>=y.L[y] <---
5: Overlap (assoc)-(assoc)--- N|-> N' >>=x'.K'[x'], L'|-> z1.K[z1] ------------
   (assoc) (N >>=x.K[x]) >>=y.L[y] => N >>=x.(K[x] >>=y.L[y])
   (assoc) (N' >>=x'.K'[x']) >>=y'.L'[y'] => N' >>=x'.(K'[x'] >>=y'.L'[y'])
              ((N' >>=x'.K'[x']) >>=x.K[x]) >>=y.L[y]
(N' >>=x'.K'[x']) >>=x.(K[x] >>=y.L[y])
        <-(assoc)-∧-(assoc)-> (N' >>=xd.(K'[xd] >>=yd.K[yd])) >>=y.L[y]
        ---> N' >>=x.(K'[x] >>=y.(K[y] >>=y.L[y]))
         =E= N' >>=x29.(K'[x29] >>=x30.(K[x30] >>=y30.L[y30])) <---
#Joinable! (Total 5 CPs)
```

The above five overlaps descriptions correspond exactly to the five diagrams shown in Figure 1. The two rules after the line number: `Overlap (..)-(..)--` indicate the rules used in the left and right paths of a divergence, and 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 `(N >>=x.K[x])` in the rule `(assoc)` is unifiable with the term `return(X') >>=y'.K'[y']` in the rule `(unitL)` using the unifier `N|-> return(X'), K'|-> z1.K[z1]` described at the immediately above. This is obtained by *higher-order unification* (Huet, 1980; Miller, 1991), which is crucial but difficult for a human. Then using this information, SOL generates the underlined term `(return(X') >>=x.K[x]) >>=y.L[y]`, which exactly corresponds to the source in a diagram of a divergence in Figure 1. The lines involving ∧ (indicating "divergence") mimics the divergence diagram and the joinability test in text. The sign `=OK=` denotes syntactic equality, and `=E=` denotes equal modulo an equational theory, here just the α-equivalence.

SN. We prove termination of the axioms of monad. The command `sn` of SOL tries to prove SN of the rules by checking all the conditions of the General Schema (Section 7.9):

```
*SOL> sn monad sigm
Found constructors: return
Checking type order >>OK
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
Checking positivity of constructors >>OK
Checking no loop in function dependency >>OK
Checking  (unitL) return(X) >>=y.K[y] => K[X]
 (meta K)[is acc in return(X),y.K[y]] [is positive in return(X)]
 (meta X)[is acc in return(X),y.K[y]] [is positive in return(X)] [is acc in X]  >>True
Checking  (unitR) N >>=y.return(y) => N
 (meta N)[is acc in N,y.return(y)] [is acc in N]  >>True
Checking  (assoc) (N >>=x.K[x]) >>=y.L[y] => N >>=x.(K[x] >>=y.L[y])
 (fun bind=bind) subterm comparison of args
 (meta N)[is acc in N >>=x.K[x],y.L[y]] [is positive in N >>=x.K[x]] [is acc in N]
 (fun bind=bind) subterm comparison of args
 (meta K)[is acc in N >>=x.K[x],y.L[y]] [is positive in N >>=x.K[x]] [is acc in K[x]]
 (meta L)[is acc in N >>=x.K[x],y.L[y]] [is positive in N >>=x.K[x]] [is acc in L[y]]
 >>True
#SN!
```

It reports SN. Therefore, we know that `monad`, regarded as left-to-right computation rules, is strongly normalising on the signature `sigm`.

1.4 Contributions

This paper is the fully reworked and extended version of the paper (Hamana, 2017) with the following new contributions:

1. We extend the framework to be polymorphic, i.e. polymorphic second-order algebraic theory and computation systems (Section 2).
2. We add new examples: theory of reading a bit (Section 3.2), theory of writing a bit (Section 3.3), and coherence of monoidal categories (Section 6).
3. We complete the π-calculus example (Section 5). In Hamana (2017), there was a problematic formulation in a rule, which is corrected in this paper. The proof becomes more difficult and longer, which requires detailed analysis to show Church–Rosser modulo equivalence.
4. We give the complete proofs of the theorem that critical pair checking implies local confluence for computation systems (Theorem 7.11) and the theorem establishing Church–Rosser modulo equational theory by critical pair checking for partitioned computation systems (Theorem 7.19).

1.5 Organisation

The paper is organised as follows. We first introduce the framework of second-order algebraic theories and computation rules in Section 2. From Sections 3–5, we examine various algebras and calculi and try to show their decidability with SOL. These are Part I: Theory of effect, Part II: Variations on the λ-calculus, Part III: A Theory of π-calculus, and Part VI: On Coherence of Skew-Monoidal Categories. We consider 11 problems tagged as "Problem [#n]". In Section 7, we present the technical foundations of our notions of computation systems and SOL, including second-order matching (Sections 7.1 and 7.2), unification (Section 7.3), rewriting (Section 7.5), critical pairs (Sections 7.6–7.8), and the General Schema (Section 7.9). In Section 9, we summarise the paper.

The GHCi interface and the web interface of the SOL system are available; see Section 8.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

2 Polymorphic second-order algebraic theories and computation

In this section, we introduce the framework of polymorphic second-order algebraic theories and computation rules. It gives a formal unified framework to provide syntax, types, and computation for various simply typed computational structures. It is a simplified framework of general polymorphic framework (Hamana, 2011; Fiore & Hamana, 2013). It is also regarded as a polymorphic extension of second-order abstract syntax with metavariables (Hamana, 2004), second-order algebraic theories (Fiore & Hur, 2010; Fiore & Mahmoud 2010), and its rewriting system (Hamana, 2005, 2007, 2010).

In a previous paper (Hamana, 2017), we use the type system called molecular types, which was intended to mimic polymorphic types in a simple-type setting. However, this mimic setting provided an awkward framework to address polymorphic typed rules. The present framework first developed in Hamana (2018) is more direct and introduces type variables into types. The polymorphism in this framework is essentially ML polymorphism, i.e. predicative and only universally quantified at the outermost level, and has type constructors on types. The necessity of polymorphism for formulating simply typed systems is illustrated in Section 4.1.

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

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

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

We call $\overline{\sigma} \to \tau$ with $|\overline{\sigma}| > 0$ a *function type*. A type having no type variables is a *simple type*. We usually write types as $\sigma, \tau, \ldots$. A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g. $() \to \tau$, or simply τ. For example, Bool is an atomic type, $\mathsf{List}^{(1)}$ is a type constructor, and $\mathsf{Bool} \to \mathsf{List}(\mathsf{Bool})$ is a type. We assume that there is at least one atomic type in $\mathscr{A}$.

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

$$a_1, \ldots, a_n \rhd f : (\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m) \to \tau$$

where $(\overline{\sigma_1} \to \tau_1), \ldots, (\overline{\sigma_m} \to \tau_m), \tau \in \mathscr{T}$, and type variables $a_1, \ldots, a_n$ may occur in these types. Any function symbol is of up to second-order type.

Remark 2.2. Staton's parameterised algebraic theories (Staton, 2013a) are similar, which are second-order algebraic theories, where the target type of every function type is always a fixed atomic type ι. Therefore, a signature there consists of function symbols of the form $f : (\overline{a_1} \to \iota), \ldots, (\overline{a_m} \to \iota) \to \iota$.

Parameterised algebraic theories have been shown to be a useful framework that models various important notions of programming languages, such as logic programming (Staton, 2013a), algebraic effects (Staton, 2013b), and quantum computation (Staton, 2015).

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

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

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

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.

Because our framework encompasses them, we can analyse such useful examples of parameterised algebraic theories using our method and the tool SOL. We consider some of them in Sections 3 and 5. ■

A ***metavariable*** is a variable of (at most) first-order function type, declared as $M:\overline{\sigma}\to\tau$ (written as capital letters $M,N,K,\ldots$). A ***variable*** (of a 0-order type) is written usually $x,y,\ldots$, and sometimes written x^{τ} when it is of type τ. The raw syntax is given as follows:

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

where $f\in\Sigma$. 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 2.3). 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. A metavariable context Θ is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:type in $\mathscr{T}_0$)-pairs. A judgement is of the form $\Theta\triangleright\Gamma\vdash\ t\ :\tau$. A ***type substitution*** $\rho:S\to\mathscr{T}$ is a mapping that assigns a type $\sigma\in\mathscr{T}$ to each type variable a in S. We write $\tau\,\rho$ (resp. $t\,\rho$) to be the one obtained from a type τ (resp. a meta-term t) by replacing each type variable in τ (resp. t) with a type using the type substitution $\rho:S\to\mathscr{T}$. A meta-term t is *well-typed* by the typing rules Figure 2. In every well-typed function term, a function symbol is officially 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 $\sigma\triangleq((\overline{\sigma_1}\to\tau_1),\ldots,(\overline{\sigma_m}\to\tau_m)\to\tau)\xi$. Practically, we often omit the type annotation. The type annotation can be recovered by the type inference algorithm in Section 7.4. The notation $t\{x_1\mapsto s_1,\ldots,x_n\mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1,\ldots,s_n$.

Definition 2.3. (Substitution of meta-terms for metavariables) Let $n_i=|\overline{\tau_i}|$ and $\overline{\tau_i}=\tau_i^1,\ldots,\tau_i^{n_i}$. Suppose

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

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

$$\text{(Ax1)}\frac{(\Theta \triangleright \Gamma \vdash\ t_1 = t_2\ : \tau) \in E}{\Theta \triangleright \Gamma \vdash\ t_1 = t_2\ : \tau} \qquad \text{(Ax2)}\frac{(\Theta \triangleright \Gamma \vdash\ t_1 = t_2\ : \tau) \in E}{\Theta \triangleright \Gamma \vdash\ t_2 = t_1\ : \tau}$$

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

$$\text{(Sub)}\frac{\begin{array}{c} S \text{ is the set of all type variables in } \overline{\tau_i}, \overline{\sigma_i}, \tau \qquad \xi : S \to \mathscr{T} \\ \Theta \triangleright \Gamma', \overline{x_i : \tau_i} \vdash\ s_i\ : \sigma_i\xi \quad (1 \le i \le k) \\ (M_1 : (\overline{\tau_1} \to \sigma_1), \ldots, M_k : (\overline{\tau_k} \to \sigma_k) \triangleright \Gamma \vdash\ t_1 = t_2\ : \tau) \in E \end{array}}{\Theta \triangleright \Gamma, \Gamma' \vdash\ t_1\xi\,[\overline{M \mapsto \bar{x}.s}] = t_2\xi\,[\overline{M \mapsto \bar{x}.s}]\ : \tau\xi}$$

$$\text{(Ref)}\frac{}{\Theta \triangleright \Gamma \vdash\ t = t\ : \tau} \qquad \text{(Tra)}\frac{\Theta \triangleright \Gamma \vdash\ s = t\ : \tau \quad \Theta \triangleright \Gamma \vdash\ t = u\ : \tau}{\Theta \triangleright \Gamma \vdash\ s = u\ : \tau}$$

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

Fig. 3. Polymorphic second-order equational logic.

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

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

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

Definition 2.4. For meta-terms $\Theta \triangleright \Gamma \vdash\ s\ : b$ and $\Theta \triangleright \Gamma \vdash\ t\ : b$, an ***equation*** is of the form

$$\Theta \triangleright \Gamma \vdash\ s = t\ : b$$

We usually omit contexts and type information, and write simply $s = t$. The polymorphic second-order equational logic is a logic to deduce formally proved equations from a given set E of equations, regarded as ***axioms***. The inference system of equational logic is given in Figure 3. The second-order polymorphic equational theory (or second-order polymorphic algebraic theory) is the set of all proved equations deduced from a set of axioms.

Definition 2.5. 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 the following:

i. ℓ is a deterministic second-order pattern.
ii. All metavariables in r appear in ℓ.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

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

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

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

Fig. 4. Polymorphic second-order computation (one-step).

The definition of the condition (i) will be given in Section 7.2. We usually omit the context and type and simply write $\ell \Rightarrow r$.

A ***polymorphic computation system*** is a pair (Σ, C) consisting of a signature Σ and a set C of rules. We write $s \Rightarrow_C t$ to be one-step computation using (Σ, C) obtained by the inference system given in Figure 4. The (RuleSub) instantiates a polymorphic computation rule $\ell \Rightarrow r$ in C by substitution $[\overline{M \mapsto \overline{x}.s}]$ of meta-terms for metavariables and substitution ξ on types. The (Fun) means that the computation step is closed under polymorphic function symbol contexts. Clearly, it is a subsystem of the second-order equational logic (Figure 3), by deleting (Ref)(Tra)(Ax2). We regard $\Rightarrow_C$ to be a binary relation on meta-terms.

Example 2.6. The simply typed λ-calculus is modelled as a polymorphic second-order algebraic theory as follows. We suppose a binary type constructor Arr, and regard each $\iota \in \mathrm{Ty}$ as a type constant of a polymorphic second-order algebraic theory. The signature Σ_{lam} for the λ-terms is

$$a, b \triangleright \mathsf{lam} : (a \to b) \to \mathsf{Arr}(a, b)$$
$$a, b \triangleright \mathsf{app} : \mathsf{Arr}(a, b), a \to b$$

Here, a and b are type variables. The β-reduction law is presented as an equation

$$M\colon a \to b,\ N\colon a \triangleright \vdash \mathsf{lam}^{(a\to b)\to \mathsf{Arr}(a,b)}(x^a.\,M[x])\ @^{\mathsf{Arr}(a,b),a\to b}\ N \ = \ M[N]\colon b$$

where app is denoted by the infix operator @. The β-reduction rule is similar. ■

3 Problem part I: Theories of computation and effects

Using the basic framework of second-order algebraic theories, we examine various examples.

3.1 Global state [Problem #2]

We consider the theory of global state (Plotkin & Power, 2002; Staton, 2009) for the single location. Suppose there are the type `Val` of values and a type `A`, which is different from `Val`. The signature

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
sigst = [signature|  lk : (Val -> A) -> A;  ud : Val,A -> A   |]
```

consists of two operations `lk(v.t)` (looking-up the state, binding the value to `v`, and continuing `t`) and `ud(v,t)` (updating the state to `v` and continuing `t`), and the axioms are[4]

```
gstate = [axiom|
 (lu)  lk(v.ud(v,X))       = X
 (ll)  lk(w.lk(v.X[v,w])) = lk(v.X[v,v])
 (uu)  ud(V,ud(W,X))       = ud(W,X)
 (ul)  ud(V,lk(w.X[w]))    = ud(V,X[V]) |]
```

These axioms have intuitive reading. For example, the axiom `(lu)` says that looking-up the state, binding the value to `v`, then updating the state to `v`, is equivalent to doing nothing. The axiom `(ul)` says that updating the state to `V`, then looking-up and continuing `X` with the looked-up value, is equivalent to updating the state to `V` and continuing with `V`. Plotkin and Power showed that the monad corresponding to the theory of global state (of finitely many locations) is the state monad (Plotkin & Power, 2002). As far as we examined their paper, they did not touch the decidability issue of the theory, and to the best of our knowledge, it has not been explicitly proved in the literature so far (e.g. Staton, 2009; Melliès, 2010). We give a rewriting theoretic proof of the decidability of the second-order equational theory `gstate` using SOL.

SN. First we try to show SN of `gstate` regarded as left-to-right computation rules by using SOL.

```
*SOL> sn gstate sigst
Checking  (ll) lk(w.lk(v.X[v,w])) => lk(v.X[v,v])
 (fun lk=lk) subterm comparison of args  >>False
No idea..
```

SOL cannot prove it. This is because the General Schema checks that a recursive call at the rhs must be with a subterm of an argument at the lhs. In case of `(ll)`, the recursive call of `lk` happens with `v.X[v,v]`, which is not a subterm of `w.lk(v.X[v,w])` at the lhs. Moreover, in `(ul)`, where `ud` is called with `X[V]`, which is not a subterm of `lk(w.X[w])`. However, we can show SN by using another interpretation method. Since in each rule, the total numbers of `lk` and `ud` symbols in the lhs and in the rhs are strictly decreasing, we conclude the original axioms `gstate` is SN. Note that since `Val` $\neq$ `A`, the number of `lk` and `ud` in `X[v,v]` and `X[V]` are the same as the number of these in `X`.

CR. Next consider confluence (CR) of `gstate`. SOL reports three non-joinable critical pairs out of eight.

```
*SOL> cri gstate sigst
1: Overlap (lu)-(uu_v)--- V'|-> z1.z1, X|-> ud(H1,H2), W'|-> z1.H1, X'|-> z1.H2 --
   (lu) lk(v.ud(v,X)) => X
   (uu_v) ud(V'[v],ud(W'[v],X'[v])) => ud(W'[v],X'[v])
                        lk(v.ud(v,ud(H1,H2)))
                 ud(H1,H2) <-(lu)-∧-(uu_v)-> lk(v.ud(H1,H2))
                  ---> ud(H1,H2) =#= lk(v.ud(H1,H2)) <---
```

[4] In SOL, `[axiom|..|]`, `[rule|..|]` are treated as the same data. The distinction is only for representing the user's intention.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
2: Overlap (lu)-(ul_v)--- V'|-> z1.z1, X|-> lk(w'.H6[w']), X'|-> z1.z2.H6[z2] -
   (lu) lk(v.ud(v,X)) => X
   (ul_v) ud(V'[v],lk(w'.X'[v,w'])) => ud(V'[v],X'[v,V'[v]])
                      lk(v.ud(v,lk(w'.H6[w'])))
             lk(w'.H6[w']) <-(lu)-∧-(ul_v)-> lk(v.ud(v,H6[v]))
              ---> lk(w'.H6[w']) =#= lk(v.ud(v,H6[v])) <---
3: Overlap (ll)-(lu_w)--- X|-> z1.z2.ud(z1,X'[z2]) ----------------------------
   (ll) lk(w.lk(v.X[v,w])) => lk(v.X[v,v])
   (lu_w) lk(v'.ud(v',X'[w])) => X'[w]
                        lk(w.lk(v.ud(v,X'[w])))
   lk(v13.ud(v13,X'[v13])) <-(ll)-∧-(lu_w)-> lk(w.X'[w])
    ---> lk(v13.ud(v13,X'[v13])) =#= lk(w.X'[w]) <---
4: Overlap (ll)-(ll_w)--- X|-> z1.z2.lk(v'.X'[z2,v',z1]) ----------------------
   (ll) lk(w.lk(v.X[v,w])) => lk(v.X[v,v])
   (ll_w) lk(w'.lk(v'.X'[w,v',w'])) => lk(v'.X'[w,v',v'])
                     lk(w.lk(v.lk(v'.X'[w,v',v])))
lk(v19.lk(v'.X'[v19,v',v19])) <-(ll)-∧-(ll_w)-> lk(w.lk(vd20.X'[w,vd20,vd20]))
    ---> lk(v21.X'[v21,v21,v21]) =E= lk(v24.X'[v24,v24,v24]) <---
5: Overlap (uu)-(uu)--- V'|-> W, X|-> ud(W',X') --------------------------------
   (uu) ud(V,ud(W,X)) => ud(W,X)
   (uu) ud(V',ud(W',X')) => ud(W',X')
                          ud(V,ud(W,ud(W',X')))
          ud(W,ud(W',X')) <-(uu)-∧-(uu)-> ud(V,ud(W',X'))
                  ---> ud(W',X') =OK= ud(W',X') <---
6: Overlap (uu)-(ul)--- V'|-> W, X|-> lk(w'.X'[w']) ---------------------------
   (uu) ud(V,ud(W,X)) => ud(W,X)
   (ul) ud(V',lk(w'.X'[w'])) => ud(V',X'[V'])
                       ud(V,ud(W,lk(w'.X'[w'])))
      ud(W,lk(w'.X'[w'])) <-(uu)-∧-(ul)-> ud(V,ud(W,X'[W]))
               ---> ud(W,X'[W]) =OK= ud(W,X'[W]) <---
7: Overlap (ul)-(lu)--- X|-> z1.ud(z1,X') ---------------------------------------
   (ul) ud(V,lk(w.X[w])) => ud(V,X[V])
   (lu) lk(v'.ud(v',X')) => X'
                          ud(V,lk(w.ud(w,X')))
           ud(V,ud(V,X')) <-(ul)-∧-(lu)-> ud(V,X')
                   ---> ud(V,X') =OK= ud(V,X') <---
8: Overlap (ul)-(ll)--- X|-> z1.lk(v'.X'[v',z1]) -------------------------------
   (ul) ud(V,lk(w.X[w])) => ud(V,X[V])
   (ll) lk(w'.lk(v'.X'[v',w'])) => lk(v'.X'[v',v'])
                       ud(V,lk(w.lk(v'.X'[v',w])))
     ud(V,lk(v'.X'[v',V])) <-(ul)-∧-(ll)-> ud(V,lk(vd32.X'[vd32,vd32]))
              ---> ud(V,X'[V,V]) =OK= ud(V,X'[V,V]) <---
#NON 3 joinable... (Total 8 CPs)
```

This is unfortunate, but the output gives us useful information about how we should proceed next. Although `gstate` is not CR, by adding new theorems to `gstate`, we may recover CR. We should add *correct* theorems, otherwise, it becomes inconsistent. Interestingly, the three non-joinable indications s `=#=` t are actually *correct theorems* $s = t$, because s and t are obtained by unification between rules (i.e. instances of axioms) and rewriting (i.e. equational reasoning) through different paths. Listing them,

```
lk(v1.ud(W',X'))      = ud(W',X')
lk(v2.ud(v2,X'[v2])) = lk(w'.X'[w'])
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

we see that these are naturally understandable theorems. The first says that looking-up the state and then updating it by a value `W'` is equivalent to just updating the state by `W'`. The second says that looking-up the state and then updating it by the looked-up value and continuing with the value is equivalent to looking-up the state and continuing with the looked-up value. These must be the same. These are regarded as *missing theorems* to get confluent axioms of global state. We define them as extra rules

```
gstateEx = [rule| (lu1)  lk(v1.ud(W',X'))      => ud(W',X')
                  (lu2)  lk(v2.ud(v2,X'[v2])) => lk(w'.X'[w']) |]
```

We next try to show CR of the combination of the two sets of rules. Here, we use a benefit of SOL as a DSL in Haskell. Meta-theoretic operations of manipulating syntax, axioms, rules, etc., can be programmed using Haskell. In this case, since a rule declaration set is implemented as a list, we can use the append "++" for the combination:

```
*SOL> cri (gstate ++ gstateEx) sigst
..
#Joinable! (Total 21 CPs)
```

This addition generates new overlaps, and now all are successfully joinable. Note again that since `gstateEx` are theorems deduced from `gstate`, this addition does not change the equational theory, i.e. regarded as lemmas. Hence we have confluent reduction rules for `gstate`.

Finally, since the number of `lk` and `ud` symbols in the lhs and in the rhs in each rule are decreasing, `(gstate ++ gstateEx)` is SN. Hence, we conclude that the theory of global state is decidable, and the SN and CR computation system `(gstate ++ gstateEx)` provides a decision procedure.

The method we have done is nothing but the process of *Knuth–Bendix completion algorithm* (Knuth & Bendix, 1970). The "completion" means "to make the rewrite system confluent and terminating". The original Knuth–Bendix completion algorithm repeats automatically the "critical pair checking" and "adding new rules" process (for first-order rewrite rules) using an order to orient an equation to add to the rule set.

Instead, we have manually chosen new computation rules from the critical pair checking for second-order computation rules.

3.2 Reading a bit: Rectangular band [Problem #3]

Staton examined various algebraic theories of computational effects in Staton (2013b). We consider examples about bits.

A theory of reading a bit consists of a binary operator qu (also written as an infix operator "?")

```
sigrect = [signature| qu : a,a -> a |]
```

and the following axioms:

```
rect = [axiom|
 (idem-Q)  X ? X = X
 (dup-Q)   (U ? V) ? (X ? Y) = U ? Y    |]
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

The infix operator ? is automatically translated to the function qu in SOL. The idea is that X ? Y is a computation that first reads the bit and then proceeds as X (if the read bit is 0) or as Y (if the read bit is 1) depending on the result. The axiom (idem-Q) says that if the result is ignored, then the read is not observable. The axiom (dup-Q) says that the same result is obtained no matter how many times the reading is tested. This theory rect has been studied in universal algebra under the name "rectangular band". The monad associated to this theory is the reader monad.

We consider CR of rect. SOL reports five critical pairs, and only one is joinable.

```
*SOL> cri rect sigrect
1: Overlap (idem-Q)-(dup-Q)--- X|-> U' ? V', X'|-> U', Y'|-> V' ---------------
   (idem-Q) (X ? X) => X
   (dup-Q) (U' ? V') ? (X' ? Y') => U' ? Y'
                          (U' ? V') ? (U' ? V')
               U' ? V' <-(idem-Q)-∧-(dup-Q)-> U' ? V'
                     ---> U' ? V' =OK= U' ? V' <---
2: Overlap (dup-Q)-(idem-Q)--- X'|-> V, U|-> V -------------------------------
   (dup-Q) (U ? V) ? (X ? Y) => U ? Y
   (idem-Q) X' ? X' => X'
                          (V ? V) ? (X ? Y)
                  V ? Y <-(dup-Q)-∧-(idem-Q)-> V ? (X ? Y)
                      ---> V ? Y =#= V ? (X ? Y) <---
3: Overlap (dup-Q)-(idem-Q)--- X'|-> Y, X|-> Y -------------------------------
   (dup-Q) (U ? V) ? (X ? Y) => U ? Y
   (idem-Q) X' ? X' => X'
                          (U ? V) ? (Y ? Y)
                  U ? Y <-(dup-Q)-∧-(idem-Q)-> (U ? V) ? Y
                      ---> U ? Y =#= (U ? V) ? Y <---
..
#NON 4 joinable... (Total 5 CPs)
```

Analysing the output, we see that the following two equations are missing:

```
rectEx = [axiom|
 (ex1)  V ? (X ? Y) = V ? Y
 (ex2)  (U ? V) ? Y = U ? Y   |]
```

These equations are naturally understandable. Since the same result is obtained no matter how many times the reading is tested, in (ex1), if the read bit is 1, always the second argument is tested. Therefore, in the lhs of (ex1), firstly (X ? Y) is tested, and next Y is test, which is equivalent to the result of rhs V ? Y. Adding these equations to rect, we check again the critical pairs

```
*SOL> cri (rectEx++rect) sigrect
...
#Joinable! (Total 22 CPs)
```

SOL reports 22 critical pairs, and now all are successfully joinable. For SN, by invoking the command sn (rectEx++rect) sigrect, SOL successfully check SN. This is immediate because in every axiom, the length of rhs is smaller than that of lhs. Therefore, we have CR of rectEx++rect, and the theory of rectangular band is decidable.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

The decidability is useful to prove a new law. For example, we can show the following equation is derivable from `rectEx++rect`:

```
(U ? V) ? (X ? Y) = (U ? X) ? (V ? Y)
```

Since the theory is CR and SN, it suffices to check whether the unique normal forms are equal. Normalising both sides of equation using a SOL's command `normalise`,

```
*SOL> normalise [o| (U ? V) ? (X ? Y) |] (rectEx++rect)
Just U ? Y
*SOL> normalise [o| (U ? X) ? (V ? Y) |] (rectEx++rect)
Just U ? Y
```

we see that the normal forms are the same, hence the equation is proved.

3.3 Theory of writing a bit [Problem #4]

A theory of writing a bit of memory has two additional function symbols `w0` and `w1` for writing `0` and `1`.

```
sigw = [signature|
 qu : a,a -> a   ;   w0 : a -> a   ;   w1 : a -> a  |]
```

The idea is that `w0(X)` sets the bit to `0` and continues as `X`. It is subject to the following equations:

```
wbit = [axiom|
 (w0w0)  w0(w0(X)) = w0(X)  ;  (w1w0)  w1(w0(X)) = w0(X)
 (w0w1)  w0(w1(X)) = w1(X)  ;  (w1w1)  w1(w1(X)) = w1(X)
 (w0Q)   w0(X ? Y) = w0(X)  ;  (w1Q)   w1(X ? Y) = w1(Y)
 (_Qw)   w0(Z) ? w1(Z) = Z                              |]
```

By invoking the command "`cri wbit sigw`", SOL reports 20 CPs and 6 non-joinable. By repeating the completion process manually as the previous examples, we obtain the following new rules:

```
wbitEx = [rule|
 (idem-Q) X ? X       => X
 (ex1)    V ? (X ? Y) => V ? Y
 (ex2)    (U ? V) ? Y => U ? Y
 (e0)     w0(Z) ? Y   => Z ? Y
 (e1)     w1(Z) ? Z   => w1(Z)
 (e2)     Z ? w0(Z)   => w0(Z)
 (e3)     Y ? w1(Z)   => Y ? Z   |]
```

Then, `(wbitEx++wbit)` has 76 CPs and all of these are joinable. SN of `(wbitEx++wbit)` is also immediate by SOL. Hence, the theory of writing a bit is decidable.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

4 Problem part II: Variations on the λ-calculus

This section discusses five variations of λ-calculus. We try to show the decidability of them by checking SN and CR with SOL. During the examination of each λ-calculus, we will also discuss a few notable sub-issues. The subtitle "— something" of each section denotes it.

4.1 The call-by-name λ-calculus [Problem #5] – Polymorphism is necessary for formulating a simply typed system

We consider first the most fundamental calculus of typed functional programming, i.e. the simply typed λ-calculus in call-by-name.

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

Describing this simply typed system requires a schematic type notation that is best formulated in a *polymorphic* typed framework. An important point is that σ and τ are not fixed types, but schemata of types. Therefore, (β) actually describes a *family of actual computation rules*. Namely, it represents various instances of rules by varying σ and τ, such as the following:

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

From the viewpoint of meta-theory, the (β)-rule should be formulated in a polymorphic typed framework, where types τ and σ vary over simple types. Therefore, we represent the λ-calculus by the following signature and rules:

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

beta   = [rule|      (beta) lam(x.M[x]) @ N  => M[N] |]
```

Here, `a`,`b` are type variables, and `Arr(a,b)` encodes the arrow type of the "object level" simply typed λ-calculus. SOL automatically translates the infix operator `@` to the prefix function symbol `app`. SOL also automatically infers the types of terms in the rule (such as `beta`) and its typing context. This is the type inference algorithm we will give in Section 7.4.

Our choice of the signature is different from the ordinary higher-order abstract syntax (HOAS) (Pfenning & Elliott, 1988). We strictly distinguish a type `Arr(a,b)` from a function type `a -> b` of the "meta-level" second-order algebraic theory.

In second-order algebraic theory, a function type $a \to b$, where a and b are types, always represents "variable binding". While written $a \to b$ in our notation, it semantically corresponds to a presheaf $\delta_a b \cong \mathrm{V}_a \Rightarrow b$, roughly considered as a function type from the type of "variables". The above signature is exactly the syntactic counterpart of Fiore's algebra structure of simply typed terms in Fiore (2002).

Type theoretically, this choice is also suitable, because it is strictly positive. The type of `lam`

$$\mathtt{lam} : (a \to b) \to \mathsf{Arr}(a, b)$$

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

does not involve a negative occurrence of the target type $\mathsf{Arr}(a, b)$ for any types a, b (see Definitions 7.20 and 7.21). Even in the case $a = b$, we have $\texttt{lam} : (a \to a) \to \mathsf{Arr}(a, a)$, where the type $\mathsf{Arr}(a, a)$ cannot appear negatively in $(a \to a)$ because of $a \neq \mathsf{Arr}(a, a)$.

This fact is crucial in applying the General Schema's criterion for SN, because the General Schema requires positivity of constructors. For instance, SOL reports SN

```
*SOL> sn beta siglam
Found constructors: lam
Checking positivity of constructors >>OK
Checking  (beta) lam(x.M[x])@N => M[N]
 (meta M)[is acc in lam(x.M[x]),N] [is positive in lam(x.M[x])] [is acc in M[x]]
 (meta N)[is acc in lam(x.M[x]),N] [is positive in lam(x.M[x])] [is acc in N]  >>True
#SN!
```

By varying the type variables in the signature over simple types, SOL automatically checks that `lam` is a positive constructor. Hence we could show the termination of simply typed λ-calculus using the general method of the General Schema.

Confluence (CR) of the call-by-name λ is immediate. Since the rule `beta` does not have a critical pair, `beta` is locally confluent (WCR). With SN, we have CR. Hence the equational theory of the simply typed λ-calculus generated by (β) is decidable.

Remark 4.1. If one badly chose types for the signature (with the exactly the same rule `beta`), the General Schema may not pass. For example, consider the following definition:

```
sigunLam = [signature| lam : (I -> I) -> I
                       app : I,I -> I                |]

beta     = [rule|      (beta) lam(x.M[x]) @ N  => M[N] |]
```

This is the ordinary HOAS encoding $\texttt{lam} : (\iota \to \iota) \to \iota$, which is not positive, and is regarded as the representation of *untyped* λ-calculus. Therefore, `beta` with the untyped signature `sigunLam` is actually non-terminating, and the General Schema wisely rejects it. (Try `sn beta sigunLam` in SOL). ■

4.2 The call-by-value λ-calculus [Problem #6] – Meta-programming on rules and importance of "variables"

Next we consider Plotkin's call-by-value λ-calculus (Plotkin, 1975). The ordinary style of definition (Sabry & Wadler, 1997) can be straightforwardly defined in SOL:

```
cbv0 = [rule| (beta-v) lam(x.M[x])@V  => M[V]
              (eta-v)  lam(x.V@x)     => V    |]
```

with the signature `siglam`. But this is not enough. This style additionally imposes the grammatical restriction on the form of values V:

Values $\quad V ::= y \mid \lambda w.M$

saying that values are either variables or abstractions. But the definition `cbv0` in SOL did not reflect it. The metavariable `V` had no difference with other metavariables. We can

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

tell it to SOL by the following meta-programming on rules using the benefit of Haskell DSL. We can define an expansion function `expandV` that replaces `V` with a variable and an abstraction (using the internal data structures of SOL) in Haskell. We omit this straightforward but intricate definition for readability. Then, we define `cbv` as the expansion.

```
cbv = expandV cbv0
```

Then printing command `pr` of SOL shows us the actual rules of CBV λ:

```
*SOL> pr cbv
(beta-v_v)   lam(x.M[x])@y           => M[y]
(beta-v_b)   lam(x.M[x])@lam(w.K[w]) => M[lam(w.K[w])]
(eta-v_v)    lam(x.y@x)              => y
(eta-v_b)    lam(x.lam(w.K[w])@x)    => lam(w.K[w])
```

There is one important point in this formulation. In `(beta-v_v)` (resp. `(eta-v_v)`), a variable `y` is used as `lam(x.M[x])@y => M[y]`, which is *not* the same as writing

```
(bad_beta-v) lam(x.M[x])@Y           => M[Y]
```

using a ***metavariable*** `Y`. While very similar, there is a big difference. In case of the latter, the metavariable `Y` can match *any* term including a non-value, hence it does not reflect the intention. In the formulation `cbv`, the variable `y` does not match with e.g. `M@N`. Hence unintended overlapping between `(beta-v_v)` and `(beta-v_b)` does not happens. The clear distinction between variables and metavariables in second-order algebraic theories is important in formulating axioms having free "variables". SOL successfully proves SN of `cbv`. SOL also reports joinable eight CPs; therefore, `cbv` is CR. Hence the theory of the call-by-value λ is decidable. There may be cases that use finer classifications of term structures other than values. This example shows that meta-programming on rules is useful to generate rules from a schematic definition for those cases.

4.3 The computational meta-language λ_{ml} [Problem #7]

We consider Moggi's computational meta-language (Moggi, 1988) in the calculus form (Sabry & Wadler, 1997).

```
sigML = [signature|
 app : Arr(a,b),a -> b      ;   let    : T(a),(a -> T(b)) -> T(b)
 lam : (a -> b) -> Arr(a,b) ;   return : a -> T(a)                |]
lamML = [rule|
 (beta)     lam(x.M[x])@N             => M[N]
 (eta)      lam(x.M@x)                => M
 (beta-let) let(return(M),x.N[x])     => N[M]
 (eta-let)  let(M,x.return(x))        => M
 (assoc)    let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y])) |]
```

Here, a `let`-expression `let x = s in t` is represented by `let(s,x.t)` using `let:T(a),(a -> T(b)) -> T(b)`, which is more concise and is nothing but "bind".

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

Hence, the relationship between this and the laws of monad considered in Section 1.3 is now evident, namely `lamML` is the combination of the λ-calculus with `monad`.

WCR. SOL automatically shows WCR of `lamML` by enumerating seven CPs and checking their joinability.

SN. SOL's termination criteria does not work for this example because of the commuting conversion rule `assoc`, but SN of λ_{ml} has also been proved in Benton & Hyland (2003), Lindley & Stark (2005).

Decidability. With this SN result, we have CR of λ_{ml}. The result on CR of λ_{ml} was stated in Sabry & Wadler (1997, Prop. 4.2) but without proof or any reference. We will touch this issue in Remark 4.2.

4.4 The simplified monadic calculus λ_{ml*} [Problem #8] – Clarification of CR

We consider a variation called the simplified monadic calculus λ_{ml*} in Sabry & Wadler (1997, Fig. 4). The signature is the same as `sigML` before, and the rules are defined as the combination of cbv λ-calculus with value version of `let`-axioms.

```
lamMLstar0 = cbv0 ++ [rule|
  (beta-let) let(return(V),x.N[x])     => N[V]
  (eta-let)  let(M,x.return(x))        => M
  (assoc)    let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y])) |]
lamMLstar  = expandV lamMLstar0
```

As in the case of cbv λ, we apply the expansion of value metavariable `V` to get actual rules. Checking CR, SOL reports a fairly large number of 17 CPs and all are joinable. Therefore, SOL successfully shows WCR. SN has been shown in Lindley & Stark (2005), hence combing it with SOL's WCR result, we have CR, hence decidable.

Remark 4.2. In Sabry & Wadler (1997), CR of λ_{ml*} was stated (after Prop. 5.2) as a corollary of CR of λ_{ml} using the correspondence between of λ_{ml} and λ_{ml*}. This simulation result is no problem, but there is one unclear point: CR of λ_{ml} had no proof in the paper (Sabry & Wadler, 1997). It may have to rely on some result in Moggi's original paper, because they also mentioned "The system (the computational λ-calculus λ_c) is confluent as was shown by Moggi (1988)" (Sabry & Wadler, 1997, Prop. 5.1). But as far as the present author examined Moggi's LFCS technical report (1988), Moggi did not give a proof of confluence. ■

We make clear this point: λ_{ml*} is certainly confluent.

4.5 Hasegawa's yet simpler linear λ-calculus [Problem #9] – Necessity of deterministic second-order patterns and FCU unification algorithm

Hasegawa (2005) proposed a linear λ-calculus. It has the !-type constructor that corresponds to a modality in linear logic. We represent it as a type constructor `Bang`.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

The formalisation of the syntax and typing in Ohta & Hasegawa (2006) is straightforward in SOL.

```
sigHas = [signature|
  app : Arr(a,b),a -> b          ; lam : (a -> b) -> Arr(a,b)
  bang : a -> Bang(a)            ; let : Bang(a),(a -> b) -> b          |]
lamHas = [rule|
  (beta-o) lam(x.M[x])@N => M[N] ; (eta!)   let(M, x.L@bang(x))  => L@M
  (eta-o)  lam(x.M@x)    => M    ; (beta!)  let(bang(M), x.N[x]) => N[M]|]
```

This calculus is criticised in Ohta & Hasegawa (2006, Section 2) as *"While this is very compact, it does not immediately hint a terminating and confluent system."* Hence, Ohta and Hasegawa developed another finer linear λ-calculus and tried to prove SN and CR modulo equational theory with considerable effort.

From our point of view, the above comment sounds an interesting challenge. Does SOL "immediately prove termination and confluence" of this system? Let's try.

```
*SOL> cri lamHas sigHas
..
3: Overlap (eta!)-(beta-o_x)--- L|-> lam(x'.H10[x']), M'|-> z1.z2.H10[z2], N'|-
   (eta!) let(M,x.(L@bang(x))) => L@M
   (beta-o_x) lam(x'.M'[x,x'])@N'[x] => M'[x,N'[x]]
                  let(M,x.(lam(x'.H10[x'])@bang(x)))
       lam(x'.H10[x'])@M <-(eta!)-∧-(beta-o_x)-> let(M,x.H10[bang(x)])
                     ---> H10[M] =OK= H10[M] <---
..
#NON 1 joinable... (Total 4 CPs)
```

Although SOL does not have linear/intuitionistic distinction of contexts, it does not affect the confluence problem as long as we are considering well-typed terms. This is because starting from a well-typed linear term, rewriting using `lamHas` never breaks linearity and the intended property of `bang`.

SOL reports four CPs, and the above *only one* critical pair (the number 3) is non-joinable. Hence we add it as a new rule

```
etaDashBang = [rule|  (eta'!)   let(M, x.C[bang(x)]) => C[M]   |]
```

Interestingly, Ohta and Hasegawa have also considered the same `(eta'!)` rule for their finer linear λ-calculus from a different source. They tried to overcome a more complicated situation of non-joinability using a finer rule set. SOL's finding is algorithmic, just a consequence of critical pair checking. Checking again,

```
*SOL> cri (lamHas ++ etaDashBang) sigHas
..
6: Overlap (beta!)-(eta'!)--- M'|-> bang(M), N|-> x.C'[bang(x)] ---------------
   (beta!) let(bang(M),x.N[x]) => N[M]
   (eta'!) let(M',x'.C'[bang(x')]) => C'[M']
                     let(bang(M),x.C'[bang(x)])
           C'[bang(M)] <-(beta!)-∧-(eta'!)-> C'[bang(M)]
               ---> C'[bang(M)] =OK= C'[bang(M)] <---
#Joinable! (Total 6 CPs)
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

SOL reports six CPs and finally all are joinable. Hence we conclude that `lamHas` is WCR. The above overlap is particularly interesting. SOL reports an overlap between `(beta!)` and `(eta'!)` using a unifier `M'|-> bang(M), N|-> x.C'[bang(x)]` using the modified FCU algorithm (Section 7.3). Hence, the middle term `let(bang(M),x.C'[bang(x)])` of the critical pair can be rewritten by `(beta!)` at the left path and `(eta!)` at the right path of divergence. Why does the solution happen in unification? This is due to the fact that the lhs of `(eta'!)`

```
let(M, x.C[bang(x)])
```

is *beyond* the class of Miller's higher-order patterns, but is within the class of Yokoyama et al.'s (2003, 2004a) *deterministic second-order patterns* – i.e. the metavariable `C` takes a constructor term `bang(x)` (cf. Section 7.2). Unification between this and a term is only solvable using the recent Function-as-Constructor Unification (FCU) algorithm.

5 Problem part III: A theory of π-calculus [Problem #10]

As a fairly large problem, we consider a theory of π-calculus given by Stark (2008). The π-calculus of Milner is one of the most fundamental concurrent calculi (Milner, 1999). Stark (2008) gave a free algebra model of π-calculus and Staton (2009, 2013b) examined algebraic and categorical properties of this algebraic theory. However, as far as the present author examined, none of the above mentioned papers touched the decidability issue of this algebraic theory of π-calculus. We try to show the decidability of it with assistance of SOL.

A theory of π-calculus consists of 12 axioms. We use Huet's idea of partitioning axioms into rules and equations (Section 7.7). A reason for doing this is that commutativity axioms (`(sum-com)`,`(new-com)` below) cannot be oriented. We define the signature and the partitioned axioms of the π-calculus in SOL as follows:

```
pisig = [signature|
 nil : A        ; in  : N,(N -> A) -> A ; tau : A -> A
 sum : A,A -> A ; out : N,N,A -> A      ; new : (N -> A) -> A   |]
pical = [rule|
 (new-uni)  new(a.X)              => X
 (sum-uniL) sum(nil,X)            => X
 (new-sum)  new(a.sum(X[a],Y[a])) => sum(new(a.X[a]),new(a.Y[a]))
 (new-out0) new(a.out(a,B,X[a]))  => nil
 (new-out)  new(a.out(B,C,X[a]))  => out(B,C,new(a.X[a]))
 (new-in)   new(a.in(B,c.X[a,c])) => in(B,c.new(a.X[a,c]))
 (new-tau)  new(a.tau(X[a]))      => tau(new(a.X[a]))
 (new-in0)  new(a.in(a,b.X[a,b])) => nil                      |]

pieq  = [axiom|
 (sum-idem) sum(X,X)              = X
 (sum-com)  sum(X,Y)              = sum(Y,X)
 (sum-asc)  sum(sum(X,Y),Z)       = sum(X,sum(Y,Z))
 (new-com)  new(a.new(b.X[a,b]))  = new(b.new(a.X[a,b]))       |]
```

None of the axioms are changed from the original presentation (Stark, 2008; Staton, 2009). We just partitioned them and wrote in the notation of second-order algebraic theory.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

The notation of meta-applications concisely reflects the intention without writing side-conditions of variables. For example,

- In `(new-uni)`, `X` cannot contain the variable `a` (since it is not written as `X[a]`).
- The rhss of `(newout0),(newout)` seem overlapped. However, they are not, because `B` in `new(a.out(B,C,X[a]))` cannot contain `a` (since it is not written as `B[a]`), hence it cannot unify with `new(a.out(a,B,X[a]))`.

To prove the decidability, we show that the computation rules `pical` have the property called Church–Rosser modulo the equational theory ($CR_{\sim}$) generated by `pieq` (see Section 7.7).

We show it by the critical pair checking on the rules `pical` and axioms `pieq`, and the SN of the computation rules (Theorem 7.19). These entail the decidability (Section 7.7).

Critical pair checking. SOL has the command `crimod`, which is short for **cri**tical pair checking **mod**ulo equational theory. It enumerates all the critical pairs of a polymorphic second-order computation rules modulo equational theory, and checks their joinability, which is needed to establish Church–Rosser modulo equational theory. For the case of π-calculus, we invoke

```
*SOL> crimod pical pieq
```

Then SOL reports 53 critical pairs (CPs) and 11 CPs are non-joinable. This large number of CPs shows that it is hard for a human to enumerate all the critical pairs manually without oversight. We list the CPs by omitting duplication.

```
12: Overlap (sum-ascr)-(sum-uniL)   14: Overlap (sum-com)-(sum-uniL)
  sum(sum(X,nil),Z) =#= sum(X,Z)      sum(Y,nil) =#= Y

36: Overlap (new-com)-(new-out0_a)
  new(b.new(a.out(b,B[a,b],X[a,b]))) =#= nil

37: Overlap (new-comr)-(new-out0_b)
  new(a.new(b.out(a,B[b,a],X[b,a]))) =#= nil

40: Overlap (new-com)-(new-out_a)
new(b.new(a.out(B[a],C[a],X[a,b]))) =#= new(a.out(B[a],C[a],new(b.X[a,b])))

41: Overlap (new-comr)-(new-out_b)
new(a.new(b.out(B[b],C[b],X[b,a]))) =#= new(b.out(B[b],C[b],new(a.X[b,a])))

44: Overlap (new-com)-(new-in_a)
  new(b.new(a.in(B[a],c.X[a,b,c]))) =#= new(a.in(B[a],c.new(b.X[a,b,c])))

45: Overlap (new-comr)-(new-in_b)
  new(a.new(b.in(B[b],c.X[b,a,c]))) =#= new(b.in(B[b],c.new(d.X[b,d,c])))
```

Here `(sum-ascr)` and `(sum-comr)` are the reversed versions of the equations `(sum-asc)` and `(sum-com)`, respectively. The notation like "`_b`" in a label of an axiom indicates an extension of the axiom by a variable. For example, `(new-out0_b)` is a variation of the axiom `(new-out0)`:

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
(new-out_b) new(a.out(B[b],C[b],X[b,a])) => out(B[b],C[b],new(a.X[b,a]))
```

which extends the axiom by allowing a free variable b.

Now a missing computation rule to make joinable the first two of the non-joinable CPs is the right unit law of sum, and for the other three, we just add them:

```
piex = [rule|
  (sum-uniR)  sum(X,nil)                          => X
  (new1-out0) new(a.new(b.out(a,B[b,a],X[b,a])))  => nil
  (new1-out)  new(a.new(b.out(B[b],C[b],X[b,a])))
          => new(b.out(B[b],C[b],new(a.X[b,a])))
  (new1-in)   new(a.new(b.in(B[b],c.X[b,a,c])))
          => new(b.in(B[b],c.new(a.X[b,a,c]))) |]
```

We check again by putting "`crimod (piex++pical) pieq`", and SOL reports 104 CPs (!), and 6 non-joinable. Carefully looking at one of them:

```
  34: Overlap (new-com)-(new1-out_a)
      new(b.new(a.new(b'.out(B[a,b'],C[a,b],X[a,b',b]))))
  =#= new(a.new(bd.out(B[a,bd],C[a,bd],new(ad.X[a,bd,ad]))))
```

we see that this is just a variant of the previous overlap number 40, where new-binder in lhs is nested three times, rather than twice (in 40). Other non-joinable CPs are also variants of the previous case, where new is nested three times. This observation leads us to the additional rules of three times version of piex:

```
piex2 = [axiom|
  (new2-out)  new(a.new(b.new(c.out(B[b,c],C[b,c],X[b,c,a]))))
          => new(b.new(c.out(B[b,c],C[b,c],new(a.X[b,c,a]))))
  (new2-out0) new(a.new(b.new(c.out(a,B[b,c,a],X[b,c,a])))) => nil
  (new2-in)   new(a.new(b.new(c.in(B[b,c],q.X[b,c,a,q]))))
          => new(b.new(c.in(B[b,c],q.new(a.X[b,c,a,q])))) |]
```

We check again by putting "`crimod (piex++piex2++pical) pieq`", and SOL reports 157 CPs, and 6 non-joinable. The non-joinable CPs are again variants of the previous case, but now new-binder in lhs is nested four times, rather than three.

Clearly, this "checking and adding missing rules" process (known as Knuth–Bendix completion) does not end in finite times in this example. Since (new-com) can swap any two nested new, (new-com) and the added new-nested rules (such as (new2-*)) are overlapped again.

But now we see that what we actually need are schematic rules for every $(n+1)$-time nested new-case, which we call $\mathtt{piex}_n$:

$$
\begin{array}{l}
(\mathtt{new}_n\text{-}\mathtt{out0})\ \mathtt{new}(\mathtt{a}.\mathtt{new}_n(\overline{\mathtt{b}}.\mathtt{out}(\mathtt{a},\mathtt{B}[\overline{\mathtt{b}},\mathtt{a}],\mathtt{X}[\overline{\mathtt{b}},\mathtt{a}]))) \Rightarrow \mathtt{nil} \\
(\mathtt{new}_n\text{-}\mathtt{out})\ \ \mathtt{new}(\mathtt{a}.\mathtt{new}_n(\overline{\mathtt{b}}.\mathtt{out}(\mathtt{B}[\overline{\mathtt{b}}],\mathtt{C}[\overline{\mathtt{b}}],\mathtt{X}[\overline{\mathtt{b}},\mathtt{a}]))) \\
\qquad\qquad \Rightarrow \mathtt{new}_n(\overline{\mathtt{b}}.\mathtt{out}(\mathtt{B}[\overline{\mathtt{b}}],\mathtt{C}[\overline{\mathtt{b}}],\mathtt{new}(\mathtt{a}.\mathtt{X}[\overline{\mathtt{b}},\mathtt{a}]))) \\
(\mathtt{new}_n\text{-}\mathtt{in})\ \ \ \mathtt{new}(\mathtt{a}.\mathtt{new}_n(\overline{\mathtt{b}}.\mathtt{in}(\mathtt{B}[\overline{\mathtt{b}}],\mathtt{q}.\mathtt{X}[\overline{\mathtt{b}},\mathtt{a},\mathtt{q}]))) \\
\qquad\qquad \Rightarrow \mathtt{new}_n(\overline{\mathtt{b}}.\mathtt{in}(\mathtt{B}[\overline{\mathtt{b}}],\mathtt{q}.\mathtt{new}_n(\mathtt{a}.\mathtt{X}[\overline{\mathtt{b}},\mathtt{a},\mathtt{q}])))
\end{array}
$$

where $\overline{\mathtt{b}}$ denotes a sequence $\mathtt{b}_1,\ldots,\mathtt{b}_n$ of variables, and $\mathtt{new}_n(\overline{\mathtt{b}},X)$ denotes n-time nesting of new. Then we have that every critical pair of $\mathtt{pical}\cup\{(\mathtt{sum\text{-}uniR})\}\cup$

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

$\{\texttt{piex}_n \mid n \in \mathbb{N}\}$ is joinable modulo `pieq` by meta-theoretical reasoning. Note that theoretically, a computation system need not be finite, hence this is meaningful.

SN. We show SN of $\texttt{pical} \cup \{\texttt{(sum-uniR)}\} \cup \{\texttt{piex}_n \mid n \in \mathbb{N}\}$. Although SOL could not show it automatically (because it does not follow the General Schema), this system fits into the class of binding term rewriting systems (Hamana, 2005, Section 7), meaning that there is no proper meta-application. We can use the polynomial interpretation method. Interpreting the function symbols using the following monotone polynomials over natural numbers $\mathbb{N}$,

$$[\![\texttt{in}]\!](x,f) = x + f + 2 \quad [\![\texttt{new}]\!](f) = 2f + 1 \quad [\![\texttt{nil}]\!] = 0$$
$$[\![\texttt{out}]\!](x,y,z) = 2x + 2y + z + 2 \quad [\![\texttt{sum}]\!](x,y) = x + y + 2 \quad [\![\texttt{tau}]\!](x) = x + 1$$

the semantics of the rules in `pical` are strictly decreasing. By Hamana (2005, Section 7), we have SN.

Decidability. We prove that `pical` has the property of Church–Rosser modulo the equational theory `pieq`. Formally, we apply Theorem 7.19 to deduce it. We take a binary relation $\mapsto$ to be the equational theory generated by `pieq` and a sub-relation $\rightsquigarrow$ to be the equational theory generated by $\{\texttt{(sum-com)}, \texttt{(sum-asc)}, \texttt{(new-com)}\}$. Why we do not include `(sum-idem)` is that `(sum-idem)` is problematic to prove well-foundedness of $\Rightarrow_C \circ \rightsquigarrow^*$ (cf. Section 7.7). The well-foundedness of $\Rightarrow_C \circ \rightsquigarrow^*$ follows from SN of $\Rightarrow_C$ and the fact that in each axiom in $\{\texttt{(sum-com)}, \texttt{(sum-asc)}, \texttt{(new-com)}\}$, the lhs and rhs do not change the number of function symbols `sum` and `new`. Since we have checked that all critical pairs are joinable, we can conclude Church–Rosser modulo equational theory.

The equational theory generated by `pieq` is decidable, because the `(sum-*)` axioms in `pieq` axiomatise that `sum`-terms are "set"-like data (i.e. unordered sequences satisfying idempotency), and `(new-com)` just swaps the order of `new`. Hence enumeration is finite and there is an evident algorithm to decide the equality on `sum` and `new` terms. Hence, the algebraic theory of π-calculus is decidable.

Note on two function spaces. In algebraic formulations of π-calculus, it has been understood that two kinds of function spaces are needed (Stark, 1996; Fiore et al., 1996), one $[N \to A]$ for giving a location/channel and the other $[N \multimap A]$ for giving a *new* location/channel (cf. a detailed analysis Staton, 2009). This distinction affects substitution of variables in equational logic deduction. Our formalisation loosely uses $[N \to A]$ for both, but it is no problem in proving CR and SN on meta-terms of the theory in SOL, because (1) the above mentioned notational benefit of meta-applications, (2) SOL's unification in critical pair checking does not identify differently named variables – i.e. injectivity of substitution for variables is ensured, and (3) SN of a (loosely larger) theory implies SN of its sub-theory.

6 Problem part IV: Coherence of monoidal categories [Problem #11]

We show another example, which is different from computational calculi. We consider the coherence problem of monoidal categories with SOL.

A *monoidal category* (Mac Lane, 1971) is a category equipped with a "monoidal" product ⊗ and three natural isomorphisms λ, ρ, α, considered as the unit and associative "laws"

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

for the monoidal product. A typical example of monoidal category is a category with cartesian products and terminal object, but a monoidal category is more general than that (e.g. the smash product $A \otimes B$ of cpos is a monoidal product, and is not cartesian because it is not decomposable). There are many examples of monoidal categories in programming language theory and theoretical computer science, such as in formulating directed (acyclic) graphs (Gibbons, 1995; Fiore & Campos, 2013).

Definition 6.1. A *monoidal category* is a category $\mathbb{C}$ together with a distinguished object I, a bifunctor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, and natural transformations

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

which are isomorphisms parameterised by objects A, B, C of $\mathbb{C}$, subject to the axioms

(a)
$$\begin{array}{ccc} \mathrm{I} \otimes \mathrm{I} & & \\ {\scriptstyle \rho_{\mathrm{I}}} \swarrow & \searrow {\scriptstyle \lambda_{\mathrm{I}}} & \\ \mathrm{I} & = & \mathrm{I} \end{array}$$

(b)
$$\begin{array}{ccc} (A \otimes B) \otimes \mathrm{I} & \xrightarrow{\alpha_{A,B,\mathrm{I}}} & A \otimes (B \otimes \mathrm{I}) \\ & {\scriptstyle \rho_{A \otimes B}} \searrow \quad \swarrow {\scriptstyle A \otimes \rho_B} & \\ & A \otimes B & \end{array}$$

(c)
$$\begin{array}{ccc} (A \otimes \mathrm{I}) \otimes B & \xrightarrow{\alpha_{A,\mathrm{I},B}} & A \otimes (\mathrm{I} \otimes B) \\ & {\scriptstyle \rho_A \otimes B} \searrow \quad \swarrow {\scriptstyle A \otimes \lambda_B} & \\ & A \otimes B & \end{array}$$

(d)
$$\begin{array}{ccc} (\mathrm{I} \otimes A) \otimes B & \xrightarrow{\alpha_{\mathrm{I},A,B}} & \mathrm{I} \otimes (A \otimes B) \\ & {\scriptstyle \lambda_A \otimes B} \searrow \quad \swarrow {\scriptstyle \lambda_{A \otimes B}} & \\ & A \otimes B & \end{array}$$

(e)
$$\begin{array}{ccccc} (A \otimes (B \otimes C)) \otimes D & & \xrightarrow{\alpha_{A,B \otimes C,D}} & & A \otimes ((B \otimes C) \otimes D) \\ {\scriptstyle \alpha_{A,B,C} \otimes D} \uparrow & & & & \downarrow {\scriptstyle A \otimes \alpha_{B,C,D}} \\ ((A \otimes B) \otimes C) \otimes D & \xrightarrow{\alpha_{A \otimes B,C,D}} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\alpha_{A,B,C \otimes D}} & A \otimes (B \otimes (C \otimes D)) \end{array}$$

A monoidal category has the following remarkable property called *coherence*.

Theorem 6.2. *Every diagram in a monoidal category made up of* λ, ρ, α *commutes.*

This theorem means that any two formal arrows having the same domain and codomains and consisting of composable λ, ρ, α and $\otimes, \mathrm{id}$ are equal in $\mathbb{C}$ by using axioms (a)–(e) and bi-functoriality of $\otimes$. The proof of coherence theorem by Mac Lane (1963) is by a complicated combinatory argument. We show another proof based on a rewriting technique.

Our proof. The first step is considering the following question: ***where do the diagrams of the axioms (a)–(e) of a monoidal category come from?*** We now see that the diagrams look like the joinability of critical pairs we have considered (e.g. see Figure 1). To make sense of this understanding, we regard the objects of a monoidal category as meta-terms, the arrows as one-step rewriting using some computation rule. This leads us to defining the computation rules of a computation system for this problem from the natural isomorphisms of a monoidal category.

```
sigmcat = [signature|
 i: C  ;  times : C,C -> C          |]
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

```
mcat = [rule|
 (lmd)   i * A       => A
 (rho)   A * i       => A
 (alpha) (A * B) * C => A * (B * C)  |]
```

The infix operator * (for monoidal product) is automatically translated to the function symbol `times` in SOL.

The second step is to consider the following question: In this setting, what are the critical pairs? We try to check them by SOL.

```
*SOL> cri mcat sigmcat
1: Overlap (rho)-(lmd)--- A|-> i, A'|-> i -----------------------------------
   (rho) (A * i) => A
   (lmd) i * A' => A'
                                  i * i
                          i <-(rho)-∧-(lmd)-> i
                            ---> i =OK= i <---
2: Overlap (rho)-(alpha)--- A|-> A' * B', C'|-> i ---------------------------
   (rho) (A * i) => A
   (alpha) (A' * B') * C' => A' * (B' * C')
                              (A' * B') * i
                  A' * B' <-(rho)-∧-(alpha)-> A' * (B' * i)
                    ---> A' * B' =OK= A' * B' <---
3: Overlap (alpha)-(rho)--- A'|-> A, B|-> i ---------------------------------
   (alpha) (A * B) * C => A * (B * C)
   (rho) A' * i => A'
                               (A * i) * C
            A * (i * C) <-(alpha)-∧-(rho)-> A * C
                          ---> A * C =OK= A * C <---
4: Overlap (alpha)-(lmd)--- A|-> i, A'|-> B ---------------------------------
   (alpha) (A * B) * C => A * (B * C)
   (lmd) i * A' => A'
                               (i * B) * C
            i * (B * C) <-(alpha)-∧-(lmd)-> B * C
                          ---> B * C =OK= B * C <---
5: Overlap (alpha)-(alpha)--- A|-> A' * B', C'|-> B -------------------------
   (alpha) (A * B) * C => A * (B * C)
   (alpha) (A' * B') * C' => A' * (B' * C')
                          ((A' * B') * B) * C
    (A' * B') * (B * C) <-(alpha)-∧-(alpha)-> (A' * (B' * B)) * C
        ---> A' * (B' * (B * C)) =OK= A' * (B' * (B * C)) <---
#Joinable! (Total 5 CPs)
```

Five critical pairs arise and all are joinable. Interestingly, the automatically obtained critical pairs (1)–(5) precisely correspond to Mac Lane's original axioms (a)–(e) in Definition 6.1. We now understand that the axioms are the minimal requirements to ensure that if an object can be transformed in two ways by laws consisting λ, ρ, or α and possibly $\otimes$, id then the divergence is finally joinable by applying other laws.

By Theorem 7.11 for establishing local confluence from critical pair checking, we have that `sigmcat` is locally confluent.

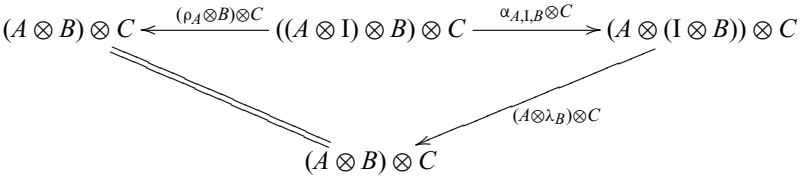
The third step is to analyse the structure of the proof of Theorem 7.11. The proof is by induction on the construction of computation relation $\Rightarrow_C$ given in Figure 4. This means that any (large) locally confluent diagram is made of

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

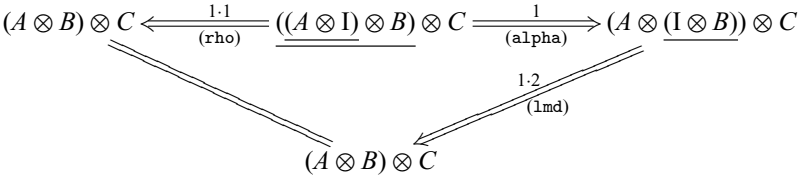
- smaller "atomic" locally confluent diagrams, i.e. instances of critical pairs by (RuleSub), and
- these smaller diagrams are composed and extended by context closure by (Fun).

Therefore, any rewriting diagram of `mcat` can be viewed as a categorical diagram in $\mathbb{C}$ because any one-step rewrite rewritten at some position of a meta-term by `(lmd),(rho),(alpha)` can be viewed as an arrow of $\mathbb{C}$ consisting of λ, ρ, or α and possibly $\otimes$ and id's (which is for context closure).

For example, the commutativity of the following categorical diagram



in a monoidal category $\mathbb{C}$ is ensured by the joinability of the rewriting diagram



due to the case (Fun) (iii) of Theorem 7.11 (which corresponds to bi-functoriality of $\otimes$) with the case (RuleSub) using the joinable critical pair `4:` of `mcat` (which corresponds to the use of the axiom (c) of monoidal category). The notation such as $\overset{1}{\underset{\texttt{(alpha)}}{\Longrightarrow}}$ means rewriting a term at the position 1 by the rule `(alpha)`, and the underline in a term designates a redex.

Remark 6.3. The axioms (a)–(e) in Definition 6.1 are the original axioms (Mac Lane, 1963). Later Kelly (1964) discovered that (a), (b), and (d) can be derived from (c) and (e). ■

7 Foundations

In this section, we present the technical foundations of our computation systems and SOL. We explain known basic results on second-order matching (Sections 7.1 and 7.2), unification (Section 7.3), type inference (Section 7.4), rewriting (Section 7.5), and our new developments on critical pairs and CR modulo second-order equational theory (Sections 7.7 and 7.8), the General Schema for termination criterion (Section 7.9), and what are the connections between them.

7.1 An algorithm for second-order computation

A one-step computation by a computation system can be understood algorithmically, as we now demonstrate. A *matcher* θ is a substitution of terms for metavariables (Definition 2.3).

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

Second-order one-step computation

Input: a target term s for rewrite, and a computation system C
Output: a one-step computed result term t

1. Select a subterm s' of s and a computation rule $\ell \Rightarrow r$ from C for rewrite.
   (NB. This selection should be fair – i.e. it does not select always the same pair of a subterm and a rule. s' is a candidate of *red*ucible *ex*pression, "*redex*").
2. Try *second-order matching* between ℓ and s'.
   If it succeeds, get a matcher θ such that $\ell\,\theta = s'$ holds.
   Otherwise, go to 1 to select again a subterm and a rule.
   If there are no other possible subterm and rule to match, then give up.
3. Replace s' in s with the instance $r\theta$ of the rhs of the computation rule. It is the output t.

Then we write $s \Rightarrow_C t$.

A key point of the above algorithm is the item 2 of the phase of second-order matching. In general, second-order matching is expensive (i.e. the second-order matching problem is NP-complete Baxter, 1977) and may have incomparable matchers between ℓ and s' (i.e. the existence of *single* most general matcher is not ensured), which means computation is inefficient and may be non-deterministic. These are in contrast to first-order matching, which is cheap and the existence of single most general matcher is ensured. A well-known idea to recover these benefits in the higher-order case (Miller, 1991; Nipkow, 1991) is to restrict the lhss ℓ of computation rules to be Miller's *higher-order patterns* (1991). We call the second-order fragments of them *second-order Miller-patterns*. In our term syntax, a second-order Miller-pattern is a meta-term ℓ in which every occurrence of meta-application in ℓ is restricted to be of the form

$$M[x_1, \ldots, x_n],$$

where $x_1, \ldots, x_n$ are distinct bound variables. Thus meta-terms such as $M[N]$, $M[\mathsf{cons}(x, y)]$ are not Miller-patterns. Second-order Miller-patterns have nice properties: there exists the most general unifier (*mgu*) for a unification problem, and an efficient algorithm is known (Miller, 1991). Thus, a computationally reasonable idea is to restrict the lhs ℓ of a rule $\ell \Rightarrow r$ to be a second-order Miller-pattern. But this is a bit too restrictive as shown in Section 4.5.

7.2 Deterministic second-order patterns

Yokoyama et al. (2003, 2004a) have found a slightly wider class of decidable second-order matching, which we have found suitable for Hasegawa's linear λ-calculus (Section 4.5). It is called *deterministic second-order patterns*, because it ensures the existence of unique most general matchers (hence called "deterministic", while general second-order matching may have incomparable matchers). We present their result by adapting it to the language of meta-terms. We denote by $s \trianglelefteq t$ if s is a subterm of t, and $s \triangleleft t$ if $s \trianglelefteq t$ and $s \neq t$.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

A ***deterministic second-order pattern*** is a meta-term p in which for every occurrence of meta-application $M[t_1, \ldots, t_n]$, the following conditions are satisfied:

i. Every t_i is a term without binders, metavariables, or free variables, but it can contain function symbols with arity $n > 0$ and bound variables.
ii. Every t_i contains at least one bound variable.
iii. $t_i \not\trianglelefteq t_j$ for every $1 \leq i, j \leq n$.

For example, $M[\mathsf{cons}(x, y)]$ is a deterministic second-order pattern. Yokoyama et al.'s deterministic second-order patterns extend the second-order fragment of Miller's higher-order patterns, because "metavariables with distinct bound variables" are ensured by ii. and iii.. We say: a *matching problem* $s \stackrel{?}{=} t$ between meta-terms asks whether there exists a matcher θ such that $s\,\theta = t$ holds. A *unification problem* $s \stackrel{?}{=} t$ between meta-terms asks whether there exists a unifier θ such that $s\,\theta = t\,\theta$ holds.

Theorem 7.1. **(Yokoyama et al., 2003, 2004a)** *Any deterministic second-order pattern matching problem $p \stackrel{?}{=} t$, where p is a deterministic second-order pattern, is decidable and has a single most general matcher if matchable. There exists an efficient algorithm for matching.*

Hence deterministic second-order patterns are computationally suitable for the syntax of lhss of second-order computation rules. What about unification? Unification between the lhss of rules is also needed when we compute overlaps of rules used for establishing local confluence. To make computation of overlaps deterministic, we expect assurance of the existence of most general unifiers. In contrast to matching, deterministic second-order pattern *unification* problem $p \stackrel{?}{=} t$ may not have a single most general *unifier*. Yokoyama et al. (2004b, Section 2) have shown such an example. The unification problem between deterministic second-order patterns

$$x.y.M[\mathsf{c}(x), \mathsf{c}(y)] \stackrel{?}{=} x.y.\mathsf{c}(N[y, x])$$

has two incomparable unifiers: $\{M \mapsto x.y.y,\ N \mapsto x.y.x\}$ and $\{M \mapsto x.y.x,\ N \mapsto x.y.y\}$. We overcome this problem by the following two steps.

7.3 Modified FCU unification

Step 1: FCU unification. Recently, Libal and Miller considered a new decidable class of higher-order unification problems called *Functions-as-Constructors unification (FCU)* (Libal & Miller, 2016). Although they did not mention a connection to Yokoyama et al.'s work, we found that the class was quite close to deterministic second-order patterns, if we restrict it to the second-order fragment. This is a key point of our design of the syntax of computation system. We connect these two works and adapt their result to our setting. A second-order unification problem $s \stackrel{?}{=} t$ is called *FCU unification* if s and t are deterministic second-order patterns and it satisfies the following condition (Libal & Miller, 2016, Definition 14).

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

- **Global restriction:** in a unification problem $s \stackrel{?}{=} t$, for every two different occurrences of meta-applications $M[s_1, \ldots, s_n]$ and $N[t_1, \ldots, t_m]$, $s_i \not\lhd t_j$ holds for every $1 \leq i \leq n$, $1 \leq j \leq m$.

Yokoyama et al.'s example actually violates the global restriction. As a corollary of a general result on higher-order FCU unification, we have the following.

Corollary 7.2. *Second-order FCU unification problem is decidable and ensures the existence of a most general unifier if solvable.*

A higher-order FCU unification algorithm has been given in Libal & Miller (2016), which is sound, complete, and terminating and returns a most general unifier. It is the basis of our implementation of unification in SOL.

Step 2: Checking a solvable unification problem violating the global restriction. But this is still not enough. When examining the example of a linear λ-calculus (Section 4.5), we need to check an overlap between the following two rules:

```
let(bang(M),x.N[x])        => N[M]
let(M1,     x.L[bang(x)]) => L[M1]
```

Trying unification between the lhss of these, we encounter a unification problem $\texttt{N[x]} \stackrel{?}{=} \texttt{L[bang(x)]}$. This violates the **global restriction**, because $\texttt{x} \lhd \texttt{bang(x)}$. We regard this as a general phenomenon when allowing deterministic second-order patterns at lhs of a rule. To cope with this pattern, we modify the algorithm:

Modified FCU algorithm

- When the algorithm encounters a problem $M[s_1, \ldots, s_n] \stackrel{?}{=} N[t_1, \ldots, t_m]$ which violates the **global restriction**, we check whether it comes from a problem of the form $x_1, \ldots, x_n.M[x_1, \ldots, x_n] \stackrel{?}{=} x_1, \ldots, x_n.N[t_1, \ldots, t_m]$. If so, returns the unifier $\{M \mapsto x_1, \ldots, x_n.N[t_1, \ldots, t_m]\}$ for it. Otherwise, fail.
- Other forms of unification problems are handled by the original algorithm.

For example, the original FCU algorithm decomposes the unification problem

$$\texttt{let(bang(M),x.N[x])} \stackrel{?}{=} \texttt{let(M1,x.L[bang(x)])}$$

to

$$\texttt{bang(M)} \stackrel{?}{=} \texttt{M1}, \quad \texttt{x.N[x]} \stackrel{?}{=} \texttt{x.L[bang(x)]}$$

and then, the second unification problem becomes $\texttt{N[x]} \stackrel{?}{=} \texttt{L[bang(x)]}$. This is a problem treated by the modified part of FCU algorithm and the algorithm returns the unifier $\{\texttt{N} \mapsto \texttt{x.L[bang(x)]}\}$.

The modified FCU algorithm is sound, but not complete. For example, the algorithm returns fail for Yokoyama et al.'s counterexample. But it is enough for the critical pair checking of second-order computation. For a unification problem $p \stackrel{?}{=} t$, if the algorithm

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

succeeds, then the output is the most general unifier of the problem. The reasons are as follows. Suppose $p \stackrel{?}{=} t$ is solvable.

i. Case that $p \stackrel{?}{=} t$ satisfies the **global restriction**. Then by Corrollary 7.2, the algorithm returns the most general unifier.
ii. Case that $p \stackrel{?}{=} t$ violates the **global restriction**, but in the scope of the modified FCU algorithm and p is a linear. Then the algorithm returns a unifier involving $\{M \mapsto x_1, \ldots, x_n.N[t_1, \ldots, t_m]\}$, which is the most general and correct by Proposition 7.3 presented below.
iii. Otherwise, the algorithm fails. Therefore, the algorithm is not complete.

Proposition 7.3. **(Prehofer, 1995, Theorem 5.2.2)** *A unification problem $p \stackrel{?}{=} t$, where p is a linear (w.r.t. metavariables) Miller-pattern and t is a second-order term such that p and t share no variables, is decidable and has a finite number of minimal complete sets of unifiers.*

7.4 Type inference for polymorphic computation rules

We defined in Section 2 that polymorphic computation rules were explicitly typed. But to insist that the user writes fully annotated type and context information for computation rules in our confluence and termination checker SOL is not a good system design. Hence we give a type inference algorithm.

For example, in the case of the call-by-name λ-calculus, the user only provides the signature `siglam` and the "plain" rule `(beta)` as in Section 4.1. The type inference algorithm infers the missing context and type annotations (highlights) as

$$M\colon a \to b,\ N\colon a \ \triangleright\ \vdash \mathsf{lam}^{(a\to b)\to \mathsf{Arr}(a,b)}(x^{a}.\, M[x])\ @^{\mathsf{Arr}(a,b),\, a\to b}\ N \ \Rightarrow\ M[N] : b$$

Algorithm. Our algorithm (Hamana, 2018) is given in Figure 5, which is a modification of Damas–Milner type inference algorithm W (Damas & Milner, 1982). It has several modifications to cope with the language of meta-terms and to return enough type information for confluence checking. The algorithm takes a signature Σ and an un-annotated meta-term t. A sub-function $\mathscr{W}$ returns $(\theta,\ \Theta \triangleright u : \tau)$, which is a pair of type substitution θ and an inferred judgement. The types in it still need to be unified. The context Θ may contain unifiable declarations, such as $M : \sigma$ and $M : \tau$ with $\sigma \neq \tau$, and these σ and τ should be unified. The main function $\mathsf{infer}(\Sigma, t)$ does it, and returns the form

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

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

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

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

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

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

$$
\begin{array}{lcl}
\mathsf{infer}(\Sigma, t) & = & \mathsf{let}\ (\theta,\ \Theta \triangleright u:\tau) = \mathscr{W}(\Sigma, t) \\
 & & \qquad \theta' = \mathsf{unify}(\mathsf{mkMatch}(\Theta\theta)) \circ \theta \\
 & & \mathsf{in}\ \Theta\theta' \triangleright u\theta' : \tau\theta'
\end{array}
$$

$$
\mathsf{infer}(\Sigma,\ s \Rightarrow t) = \mathsf{infer}(\{\mathsf{rule} : s, s \to t\} \cup \Sigma,\ \mathsf{rule}(s,t))
$$

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

Fig. 5. Type inference algorithm.

We denote by $|\, t \,|$ a meta-term obtained from t by erasing all type annotations in the variables and the function symbols of t.

Theorem 7.4. (Soundness) *If* $\mathit{infer}(\Sigma, t) = (\Theta \triangleright t' : \tau)$*, then there exists* Γ *such that* $\Theta \triangleright \Gamma \vdash t' : \tau$.

Theorem 7.5. (Completeness) *If* $\Theta \triangleright \Gamma \vdash t : \tau$ *holds under a signature* Σ *and* $\mathit{infer}(\Sigma, |\, t \,|) = (\Theta' \triangleright t' : \tau')$*, 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* $(\overline{\sigma'} \to \tau')\theta = \overline{\sigma} \to \tau$.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

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. 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 $\overline{\sigma'} \to \tau'$.

7.5 Properties of abstract rewriting

This subsection reviews classical results on abstract rewriting (Huet, 1980; Baader & Nipkow, 1998). 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 the following:

1. $a, b \in A$ are *joinable*, written $a \downarrow b$, if $\exists c \in A.\ a \to^* c \ \&\ b \to^* c$.
2. $\to$ is *confluent* if $\forall a, b \in A.\ a \to^* b \ \&\ a \to^* c$ 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 $\forall a, b \in A.\ a \to b \ \&\ a \to c$ 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 $a \to b$.
7. We write $a \xrightarrow{!} a'$ if $a \to^* a'$ and a' is a normal form, meaning rewriting to a normal form.

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. The properties of SN and CR are important for establishing decidability of equational theories. Thus one is interested in how to deduce CR from other (possibly easier to check) properties. Newman's lemma is a well-known good candidate, provided SN is established.

Lemma 7.6. (Newman's lemma) *If an ARS A is SN and WCR, then A is CR.*

To deduce WCR, we next consider the concrete one, i.e. our second-order computation system.

7.6 Critical pairs for second-order computation rules

In this subsection, we develop our notion of critical pairs for second-order computation rules, which generalises Knuth and Bendix's "joinability test" for the finite set of critical pairs (Knuth & Bendix, 1970).

Suppose that a computation system (Σ, C) is given. We assert an important fact that the pair "(the set of all meta-terms, $\Rightarrow_C$)" forms an ARS. Any notion and result on ARS are applicable to second-order computation. Henceforth, we may regard a computation system (Σ, C) as an ARS. We first give some preliminary definitions.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

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 $\mathrm{Pos}(t)$ of the positions of a meta-term t is defined by

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

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

We say that a position p in a 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 7.7. An ***overlap*** between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ of a polymorphic computation system (Σ, C) is a tuple

$$\langle l_1 \Rightarrow r_1,\ p,\ l_2 \Rightarrow r_2,\ \xi,\ \theta\rangle$$

satisfying the following properties:

i. $l_1 \Rightarrow r_1,\ l_2 \Rightarrow r_2$ are variants of rules in C without common (meta)variables.
ii. (ξ, θ) is a unifier between $l_{1|p}$ and l_2.
iii. a. p is a non-metavariable position of l_1, or
   b. if p is a metavariable position of l_1 as $l_{1|p} = M[c_1,\ldots,c_n]$ such that at least one of $c_1,\ldots,c_n$ is not a bound variable, and θ has the assignment

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

   then for all $i = 1,\ldots,n$, if c_i is not a bound variable then x_i occurs in u.
iv. If p is the root position, $l_2 \Rightarrow r_2$ is not a variant of $l_1 \Rightarrow r_1$.

Algorithmically, the components θ in an overlap are obtained by the modified FCU unification algorithm.

The condition (iii)-(b) may need explanation. Ordinary definition of overlap requires that the position p of l_1 must 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 present setting, l is a deterministic second-order pattern, which makes the situation differ, because a metavariable M may be instantiated as a meta-term with free variables,

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

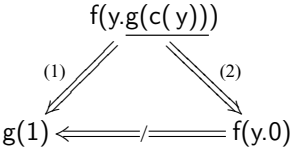
which are filled by terms (not by bound variables) c_i. The following examples illustrate this situation.

Example 7.8. Consider the computation system.

```
sigFCU = [signature|
 f:(I -> I),(I -> I) -> I ; g:I -> I ; c:I -> I ; d:I -> I
 0:I ; 1:I                                                  |]

exFCU = [rule|
 (1) f(y.M[c(y)]) => M[1]
 (2) g(c(X)) => 0                                           |]
```

This system is non-overlapping in the ordinary sense, but is not locally confluent:



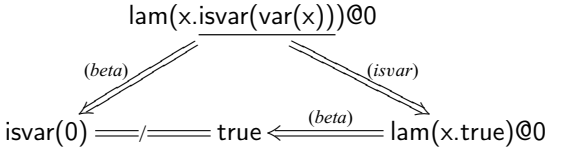
■

Example 7.9. Consider the computation system.

```
sigFCU = [signature|
 lam : (Atom(a) -> b) -> Arr(a,b)  ;  app : Arr(a,b),a -> b
 var : Atom(a) -> a  ;  isvar : a -> Bool  ;  true : Bool  ;  0 : Nat |]

exFCU = [rule|
 (beta)  lam(x.M[var(x)])@N => M[N]
 (isvar) isvar(var(X)) => true      |]
```

This system is non-overlapping in the ordinary sense, but is not locally confluent:



■

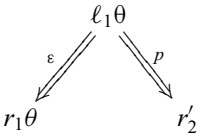
By these example, we see that we need to formulate a new notion of overlap such that these cases are regarded as non-joinable critical pairs. The case (iii)-(b) in Definition 7.7 treats these kinds of cases.

Definition 7.10. The *critical pair (CP)* generated from an overlap $\langle \ell_1 \Rightarrow r_1,\ p,\ \ell_2 \Rightarrow r_2, \theta \rangle$ is a triple $\langle r_1\theta,\ \ell_1\theta,\ r_2' \rangle$ where

- $\ell_1\theta \Rightarrow_C r_1\theta$ which rewrites the root position ε of $\ell_1\theta$ using $\ell_1 \Rightarrow r_1$
- $\ell_1\theta \Rightarrow_C r_2'$ which rewrites the position p of $\ell_1\theta$ using $\ell_2 \Rightarrow r_2$.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

This is depicted as



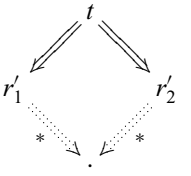
This is a critical situation that admits two ways of reduction, hence called a critical pair. Ordinary Knuth–Bendix critical pairs lack the middle $\ell_1\theta$, hence "pairs". But including "the source of divergence" designates a situation more clearly (especially in the implementation), hence our 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 where the modified FCU unification algorithm can be used. We then collect all overlaps in C by

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

Finally, we obtain all critical pairs of C by generating the critical pair of each overlap in $\mathscr{O}$. We say that a critical pair $\langle r'_1, t, r'_2\rangle$ is joinable if $r'_1 \downarrow r'_2$, depicted as



The following is an important theorem that extends the first-order case.

Theorem 7.11. *A computation system* (Σ, C) *is WCR if all its critical pairs are joinable.*

Proof. We show "if $u \Leftarrow_C w \Rightarrow_C s$ then $u \downarrow s$" by induction on the proof of $u \Leftarrow_C w$, using the inference rules in Figure 4.

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

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

for a 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 $r\theta \downarrow s$.

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 $l_{|_p} = M[c_1, \ldots, c_n]$. Suppose θ has the following assignment.

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

By the fact that l is a deterministic second-order pattern and Definition 7.7 of overlaps, the following two cases are possible.

a. Case that if c_i is not a bound variable then x_i occurs in t in θ, for each $i = 1, \ldots, n$. Then it is an instance of a critical pair, hence $r\theta \downarrow s$.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

b. Not the previous case.
(i.e. the case that all $\overline{c}$ are distinct bound variables, or, the case that c_i is not a bound variable and x_i does not occur in t for some i.)
Let t_p be a one-step reduct of t by contracting the position q as

$$t\{\overline{x} \mapsto \overline{c}\} \Rightarrow_C t_p$$

where t_p does not involve variables $\overline{x}$, but may involve $\overline{c}$. 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).
Suppose that z_i's are new distinct variables. For each $i = 1, \ldots, n$, we define a variable

$$y_i \triangleq \begin{cases} c_i & \text{if } c_i \text{ is a bound variable} \\ z_i & \text{otherwise} \end{cases}$$

Define θ' to be $M \mapsto \overline{y}.t_p,\ N \mapsto \theta(N)$ for $N \neq M$. We have

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

- (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 $l \Rightarrow r \in C$ and θ such that

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

  Flipping the left and right rewrites, $r\theta \downarrow s$ is proved as the case for (RuleSub).

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

$$\begin{array}{ccccc} f(\ldots, \overline{x_i}.u_i', \ldots, \overline{x_j}.u_j, \ldots) & \xleftarrow{ip} & f(\overline{\overline{x}.u}) & \xrightarrow{jq} & f(\ldots, \overline{x_i}.u_i, \ldots, \overline{x_j}.u_j'', \ldots) \\ {\scriptstyle jq}\Big\downarrow & & & & \Big\downarrow{\scriptstyle ip} \\ f(\ldots, \overline{x_i}.u_i', \ldots, \overline{x_j}.u_j'', \ldots) & & \xleftarrow{\qquad\qquad} & & f(\ldots, \overline{x_i}.u_i', \ldots, \overline{x_j}.u_j'', \ldots) \end{array}$$

  iii. Case the ith argument of f is rewritten by two ways as

$$\begin{array}{ccccc} f(\ldots, \overline{x_i}.u_i', \ldots) & \xleftarrow{ip} & f(\overline{\overline{x}.u}) & \xrightarrow{iq} & f(\ldots, \overline{x_i}.u_i'', \ldots) \\ {\scriptstyle *}\Big\downarrow & & & & \Big\downarrow{\scriptstyle *} \\ f(\ldots, \overline{x_i}.s, \ldots) & & \xleftarrow{\qquad\qquad} & & f(\ldots, \overline{x_i}.s, \ldots) \end{array}$$

  The above diagram commutes by the induction hypothesis:
  $u_i' \Rightarrow_C^* s \Leftarrow_C^* u_i''$ and closedness of reduction by contexts. □

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

Fig. 6. CR$_\sim$.

Corollary 7.12. *If a computation system* (Σ, C) *is SN and* C *is finite, checking "*C *is CR or not" is decidable.*

Proof. All constructions up to the theorem and checking joinability of all critical pairs are done in finite time because C is SN, and the modified FCU unification algorithm is decidable. □

Remark 7.13. Nipkow has considered critical pairs for higher-order rewrite rules over the simply typed λ-calculus (Nipkow, 1991; Mayr & Nipkow, 1998). The development in this section is similar, but there are differences. Our computation system has distinction between variables and metavariables, covers deterministic second-order patterns, and overlap checks are based on the recent FCU unification. Nipkow's rules are based on the simply typed λ-terms in βη-long normal forms, Miller's higher-order patterns and pattern unification. There is no distinction between free variables and metavariables in the rule syntax of Nipkow. As we mentioned in Section 4.2 in the rules, it may be crucial in modelling the object level variables such as in the call-by-value λ-calculus. The `(eta'!)` rule needed in the linear λ-calculus (Section 4.5) is beyond the scope of Nipkow's formalism. Therefore, the developments in this subsection are not a consequence of existing results (Nipkow, 1991; Mayr & Nipkow, 1998). ■

7.7 Church–Rosser modulo equational theory

A given computation system may not be SN and CR, or it might be difficult to prove them, as we have seen in Section 5. This and next subsections explain the background theories used there (i.e. SOL's `crimod` command).

Huet (1980) observed that the axioms of a theory are usually partitioned into two forms: "structural axioms" (such as associativity and commutativity of operators), and "simplification rules" such as "`if true then` x `else` $y \to x$". Huet and Jouannaud et al. (1983) developed general theories of abstract rewriting modulo equivalence. We follow this approach, and define a partitioned computation system as (Σ, C, E) consisting of a computation system (Σ, C) and equational axioms (Σ, E).

We set up the approach firstly at the abstract rewriting level. We consider an ARS $(A, \to)$ equipped with an equivalence relation $\sim$ on A. We say

1. $a, b \in A$ are *joinable modulo* $\sim$ if $\exists c, c' \in A.\, a \to^* c \;\&\; b \to^* c' \;\&\; c \sim c'$, which is denoted by $a \downarrow_\sim b$.
2. $\to$ is ***Church–Rosser modulo*** $\sim$ (CR$_\sim$) (Jouannaud et al., 1983) if for all $a, b \in A$, $a\,(\sim \cup \leftrightarrow)^*\,b$ implies $a \downarrow_\sim b$ (see Figure 6).

A decidable proof method for "partitioned" algebraic theories. We explain the reason why CR$_\sim$ is useful below. Suppose that an ARS satisfies SN and CR$_\sim$, and $\sim$ is

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

decidable. Now $\mathrm{CR}_{\sim}$ means that a proof of $s\,(\sim \cup \leftrightarrow)^*\,t$, where s and t are connected by a combination of $\sim$, $\rightarrow$, $\leftarrow$, is always transformed to a "tidy" proof

$$s \rightarrow^* s_0 \sim t_0 \leftarrow^* t \tag{1}$$

which means first rewriting terms s, t some terms s_0, t_0 and then comparing them by the equality $\sim$. The reasons why this method is effective are (i) $\rightarrow$ is SN, and (ii) the equational theory $\sim$ is decidable. Hence (1) gives a decidable proof method. We use the following criterion to check $\mathrm{CR}_{\sim}$. In the following, "$\circ$" denotes the composition of relations.

Theorem 7.14. (Aoto & Toyama, 2012, Theorem 2.2) *Let $\sim$ be an equivalence relation and $\rightarrow$ a binary relation on a set A. Suppose $\vdash\!\dashv$ is a symmetric relation such that $\vdash\!\dashv^* = \sim$, and take a relation $\rightsquigarrow \subseteq \vdash\!\dashv$. Define $\overset{=}{\vdash\!\dashv} \triangleq \vdash\!\dashv \cup \mathrm{id}_A$ and $\Rrightarrow \triangleq \rightarrow \cup \rightsquigarrow$. If $\rightarrow \circ \rightsquigarrow^*$ is well-founded and*

(A) $s \leftarrow \circ \rightarrow t$ implies $s \Rrightarrow^* \circ \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$,
(B) $s \vdash\!\dashv \circ \rightarrow t$ implies $s \rightarrow \circ \Rrightarrow^* \circ \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$ or $s \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$,

then $\rightarrow$ is Church–Rosser modulo $\sim$.

For later use, we name the conclusions of the conditions of the above theorem.

Definition 7.15. We call that s and t are "*joinable for* $\leftarrow\rightarrow$" if $s \Rrightarrow^* \circ \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$, and "*joinable for* $\vdash\!\dashv\rightarrow$" if $s \rightarrow \circ \Rrightarrow^* \circ \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$ or $s \overset{=}{\vdash\!\dashv} \circ \Lleftarrow^* t$.

We will use this abstract theorem to prove our critical pair checking method for a partitioned computation system in Theorem 7.19. This less known theorem is practically useful for various examples, more so than Huet's often used criterion (Huet, 1980, Lemma 2.7) (for instance in Ohta & Hasegawa, 2006; Lindley, 2007; Mayr & Nipkow, 1998), or other criteria to deduce $\mathrm{CR}_{\sim}$ requiring well-foundedness of $\rightarrow \circ \vdash\!\dashv^*$ (such as Huet, 1980, Lemma 2.8). The following points are superior points compared with other criteria.

1. Checking a peak by "one-step" equality $\vdash\!\dashv$ and rewrite suffices in (B), rather than a peak by "many-step" $\sim = \vdash\!\dashv^*$ equality and rewrite required in Huet (1980, Lemma 2.7).
2. The well-foundedness of $\rightarrow \circ \rightsquigarrow^*$ holds more likely than the well-foundedness of $\rightarrow \circ \vdash\!\dashv^*$ required in Huet (1980, Lemma 2.8) by choosing and orienting appropriate axioms $\rightsquigarrow$ from the "equational axioms" $\vdash\!\dashv$.
3. Some equational axioms taken from $\vdash\!\dashv$ can be used *many times* to check the closing part by using $\Rrightarrow^*$ or $\Lleftarrow^*$ in (A) and (B), rather just $\overset{=}{\vdash\!\dashv}$, i.e. a zero-or-one time use of an axiom, as in Jouannaud et al. (1983, Prop. 1,3), Aoto & Toyama (2012, Cor. 2.3).

What 2. means that, for example, if a partitioned computation system has an idempotency axiom $x \otimes x = x$, then $\vdash\!\dashv$ involves it. In this case, well-foundedness of $\rightarrow \circ \vdash\!\dashv^*$ cannot hold because using the idempotency axiom in reverse direction can infinitely copy a redex. Therefore, one chooses some other axioms $\rightsquigarrow \subseteq \vdash\!\dashv$ avoiding the idempotency axiom and tries to prove the well-foundedness of $\rightarrow \circ \rightsquigarrow^*$. This is more likely to hold.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

7.8 Critical pairs between second-order computation rules and equations

A *partitioned computation system* (Σ, C, E) consists of a computation system C and a set of equational axioms E satisfying for each axiom $s = t$,

- the set of all metavariables of s and that of t are exactly the same, and
- s and t are second-order Miller patterns.

For a partitioned computation system (Σ, C, E), we extend the notion of critical pairs to the one between C and E to check $\mathrm{CR}_\sim$, where $\sim$ is the equivalence relation on meta-terms generated by E. We establish an effective method to show $\mathrm{CR}_\sim$.

Example 7.16. In the case of a theory of π-calculus in Section 5, $C = \mathtt{pical}, E = \mathtt{pieq}$ and $s\ (\sim \cup \leftrightarrow)^*\ t$ means that $s = t$ is derivable from the original "unpartitioned" axioms. If we have $\mathrm{CR}_\sim$ of the partitioned algebraic theory (Σ, C, E), then we can decide $s = t$ by the method (1). ■

Definition 7.17. An *overlap between an equation $s = t \in E$ and a rule $l \Rightarrow r \in C$* is an overlap between $s \Rightarrow t$ and $l \Rightarrow r$, or, an overlap between $t \Rightarrow s$ and $l \Rightarrow r$. Namely, the overlaps are generated by regarding each equation as a bidirectional computation rule.

All the overlaps in (Σ, C, E) are now defined by

$$\begin{aligned}\mathscr{O} = \bigcup (\{&\mathsf{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \mid l_1 \Rightarrow r_1,\ l_2 \Rightarrow r_2 \in C\} \cup \\ &\{\mathsf{overlap}(s \Rightarrow t, l \Rightarrow r),\ \mathsf{overlap}(t \Rightarrow s, l \Rightarrow r), \\ &\ \mathsf{overlap}(l \Rightarrow r, s \Rightarrow t),\ \mathsf{overlap}(l \Rightarrow r, t \Rightarrow s) \mid s = t \in E,\ l \Rightarrow r \in C\})\end{aligned}$$

The *critical pairs* of a partitioned computation system (Σ, C, E) are generated by $\mathscr{O}$. We call a meta-term *linear* if no metavariable occurs more than once, and C is *linear* if for every $\ell \Rightarrow r$ in C, both ℓ and r are linear.

Definition 7.18. A critical pair $\langle s, u, t\rangle$ of (Σ, C, E) is *joinable* if one of the following holds:

i. If it is generated by an overlap between two rules in C, then s and t are joinable for $\Leftarrow_C \Rightarrow_C$ (see Definition 7.15).
ii. If it is generated by an overlap between an equation in E and a rule in C, then s and t are joinable for $\vdash\!\!\dashv \Rightarrow_C$.
iii. If it is generated by an overlap between a rule in C and an equation in E, then t and s are joinable for $\vdash\!\!\dashv \Rightarrow_C$.

Theorem 7.19. *Let (Σ, C, E) be a partitioned computation system. Assume C is linear and every rhs is a second-order Miller pattern, and there is $E' \subseteq E$ such that $\Rightarrow_C \circ \rightsquigarrow^*$ is SN, where $\rightsquigarrow$ is the computation relation generated by $E' \cup E'^{-1}$. If every critical pair of (Σ, C, E) is joinable, then $\Rightarrow_C$ is Church–Rosser modulo $\sim$, where $\sim$ the equivalence relation on meta-terms generated by E.*

Proof. We use the criterion Theorem 7.14 to deduce $\mathrm{CR}_\sim$. Now A is taken to be the set of all meta-terms. We first prove the condition (B):

$$s \vdash\!\!\dashv u \Rightarrow_C t \text{ implies } s \Rightarrow_C \circ \Rrightarrow^* \circ \overset{=}{\vdash\!\!\dashv} \circ \Lleftarrow^* t \text{ or } s \overset{=}{\vdash\!\!\dashv} \circ \Lleftarrow^* t$$

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

$$\text{(Ax1Sub)}\ \frac{\begin{array}{c}S \text{ is the set of all type variables in } \overline{\tau_i}, \overline{\sigma_i}, \tau \qquad \xi : S \to \mathscr{T} \\ \Theta \triangleright \Gamma', \overline{x_i : \tau_i} \vdash\ s_i\ : \sigma_i \xi \quad (1 \le i \le k) \\ (M_1 : (\overline{\tau_1} \to \sigma_1), \ldots, M_k : (\overline{\tau_k} \to \sigma_k) \triangleright \Gamma \vdash\ \ell \Rightarrow r\ : \tau) \in E\end{array}}{\Theta \triangleright \Gamma, \Gamma' \vdash\ \ell\xi\,[\overline{M \mapsto \overline{x}.s}] = r\xi\,[\overline{M \mapsto \overline{x}.s}]\ : \tau\xi}$$

$$\text{(Ax2Sub)}\ \frac{\begin{array}{c}S \text{ is the set of all type variables in } \overline{\tau_i}, \overline{\sigma_i}, \tau \qquad \xi : S \to \mathscr{T} \\ \Theta \triangleright \Gamma', \overline{x_i : \tau_i} \vdash\ s_i\ : \sigma_i \xi \quad (1 \le i \le k) \\ (M_1 : (\overline{\tau_1} \to \sigma_1), \ldots, M_k : (\overline{\tau_k} \to \sigma_k) \triangleright \Gamma \vdash\ \ell \Rightarrow r\ : \tau) \in E\end{array}}{\Theta \triangleright \Gamma, \Gamma' \vdash\ r\xi\,[\overline{M \mapsto \overline{x}.s}] = \ell\xi\,[\overline{M \mapsto \overline{x}.s}]\ : \tau\xi}$$

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

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

Fig. 7. Polymorphic second-order equational logic (one-step).

by induction on the proof of $s \vdash\!\!\mid u$, where the relation $\vdash\!\!\mid$ is defined by: $s \vdash\!\!\mid u$ if $\Theta \triangleright \Gamma \vdash s = u\ : \tau$ is derived from E by the polymorphic second-order equational logic (one-step) in Figure 7. Namely, the symmetric relation $\vdash\!\!\mid$ is the congruence closure of a one-step application of an instance of axiom of E. Thus $\vdash\!\!\mid^* = \sim$. We define $\overset{=}{\vdash\!\!\mid} \triangleq \vdash\!\!\mid \cup \mathrm{id}$ and $\Rrightarrow \triangleq \Rightarrow_C \cup \rightsquigarrow$.

We write $t \overset{p}{\vdash\!\!\mid} s$ if s is rewritten at a position p of s using an axiom, and $s \overset{p}{\Rrightarrow}_C t$ if s is rewritten at a position p of s using a computation rule.

- (Ax1sub) Let $E \ni u = s$ and consider the situation $u\theta \vdash\!\!\mid s\theta \Rrightarrow_C t$ for a substitution θ for metavariables and type variables.

  i. If the rewrite position p is not "a metavariable position of s or below it",

  $$u\theta \overset{\varepsilon}{\vdash\!\!\mid} s\theta \overset{p}{\Longrightarrow} t$$

  then it is an instance of a critical pair between an equation and a rule, hence $u\theta$ and t are joinable for $\vdash\!\!\mid \Rrightarrow_C$.

  ii. If p is a metavariable position in s, or below it. Note that s is a Miller pattern. In this case, the metavariable rewritten by the rewriting appears in s. Since C is linear, p determines a rewriting of substitution $\theta \Rightarrow_C \theta'$, meaning that $\theta(M) \Rightarrow_C \theta'(M)$ for a metavariable in the domain of θ, θ'. Then we have

  $$\begin{array}{ccc} u\theta & \overset{\varepsilon}{\vdash\!\!\mid} & s\theta \overset{p}{\Longrightarrow} s\theta' \\ {\scriptstyle *}\Big\Vert & & \Big\vert \\ u\theta' & = & u\theta' \end{array}$$

  Note that if u does not involve metavariables in the domain of θ, then $u\theta \Rrightarrow_C^* u\theta'$ becomes 0-step. Therefore, $u\theta$ and $s\theta'$ are joinable for $\vdash\!\!\mid \Rrightarrow_C$.

  (Ax2sub) is similar.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

- (Fun): Consider the situation $f^{\xi}(\overline{\overline{x}.s}) \overset{p}{\longmapsto\!\!\mid} f^{\xi}(\overline{\overline{x}.u}) \Rightarrow_C t$. Hereafter, we omit the superscript of f. In this case, the position p must not be the root because (Fun) is applied.

  i. If the rewrite happens at the root of $f(\overline{\overline{x}.u})$ by $\Rightarrow_C$, and $\vdash\!\!\mid$-step happens at a non-metavariable position p of l as

  $$s \overset{p}{\longmapsto\!\!\mid} f(\overline{\overline{x}.u}) = l\theta \overset{\varepsilon}{\Longrightarrow} r\theta$$

  then it is an instance of a critical pair, hence it is joinable.

  ii. Case $\vdash\!\!\mid$-step happens in the meta-substitution θ and the rewrite happens at the root. Then, it becomes

  $$\begin{array}{ccc} l\theta' & \overset{p}{\longmapsto\!\!\mid} f(\overline{\overline{x}.u}) = l\theta & \overset{\varepsilon}{\Longrightarrow} r\theta \\ \Big\Downarrow & & \Big| \vdash\!\!\mid \cup = \\ r\theta' & = & r\theta' \end{array}$$

  Since l is linear, p determines a one-step equality of substitutions as

  $$\theta(M) \vdash\!\!\mid \theta'(M)$$

  for every metavariable M. Since r is linear and a Miller pattern, $r\theta\ (\vdash\!\!\mid \cup =)\ r\theta'$. Therefore, we have that $l\theta'$ and $r\theta$ are joinable.

  iii. Case $u'_i \vdash\!\!\mid u_i$ and $u_j \Rightarrow_C u''_j$ and without loss of generality, assume $i < j$.

  $$\begin{array}{ccc} f(\ldots,\overline{x_i}.u'_i,\ldots,\overline{x_j}.u_j,\ldots) & \overset{i}{\longmapsto\!\!\mid} f(\overline{\overline{x}.u}) & \overset{j}{\Longrightarrow} f(\ldots,\overline{x_i}.u_i,\ldots,\overline{x_j}.u''_j,\ldots) \\ {\scriptstyle j}\Big\Downarrow & & \Big]{\scriptstyle i} \\ f(\ldots,\overline{x_i}.u'_i,\ldots,\overline{x_j}.u''_j,\ldots) & = & f(\ldots,\overline{x_i}.u'_i,\ldots,\overline{x_j}.u''_j,\ldots) \end{array}$$

  iv. Case $u'_i \vdash\!\!\mid u_i$ and $u_i \Rightarrow_C u''_i$. We have

  $$f(\ldots,\overline{x_i}.u'_i,\ldots) \overset{i}{\longmapsto\!\!\mid} f(\overline{\overline{x}.u}) \overset{i}{\Longrightarrow} f(\ldots,\overline{x_i}.u''_i,\ldots)$$

  By induction hypothesis, u'_i and u''_i are joinable for $\vdash\!\!\mid \Rightarrow_C$, hence so are $f(\ldots,\overline{x_i}.u'_i,\ldots)$ and $f(\ldots,\overline{x_i}.u''_i,\ldots)$.

The condition (iii) is also shown similarly by induction on the proof of $u \Leftarrow_C s$.

The condition (A) "$s \Leftarrow_C u \Rightarrow_C t$ implies $s \Rightarrow^* \circ \overset{=}{\vdash\!\!\mid} \circ \Leftarrow^* t$" is also proved similarly by induction on $s \Leftarrow_C t$. By Thm. 7.14, we conclude that $\Rightarrow_C$ is $\mathrm{CR}_{\sim}$. □

This theorem is the basis of SOL's `crimod` command used in Section 5.

7.9 Strong normalisation

The General Schema. The General Schema is a criterion for proving strong normalisation of higher-order rules given in Blanqui (2000). We summarise the definitions and properties of the General Schema needed for our SOL system (Blanqui, 2000, 2016). The General Schema is formulated for a framework of higher-order type systems with rewrite rules called inductive data-type systems, whose second-order fragment is equivalent to our framework of computation systems with simple types. The General Schema has

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

succeeded in proving SN of various rewrite rules such as Gödel's System T. The basic idea of the General Schema is to check whether the arguments of recursive calls in the rhs of a rewrite rule are "smaller" than the left-hand sides' ones. It is similar to Coquand's notion of "structurally smaller" (1992), but more relaxed and extended.

Definition 7.20. Given a type τ, the set $\mathsf{Posi}(\tau)$ of *positive positions* and the set $\mathsf{Nega}(\tau)$ of *negative positions* are defined by $\mathsf{Posi}(b) = \{\varepsilon\}$, $\mathsf{Nega}(b) = \varnothing$,

$$\begin{aligned}\mathsf{Posi}(\overline{\tau}) &= \{i.w \mid i = 1, \ldots, |\overline{\tau}|, w \in \mathsf{Posi}(\tau_i)\}\\ \mathsf{Posi}(\overline{\sigma} \to \tau) &= \mathsf{Nega}(\overline{\sigma}) \cup \{0w \mid w \in \mathsf{Posi}(\tau)\}\\ \mathsf{Nega}(\overline{\tau}) &= \{i.w \mid i = 1, \ldots, |\overline{\tau}|, w \in \mathsf{Nega}(\tau_i)\}\\ \mathsf{Nega}(\overline{\sigma} \to \tau) &= \mathsf{Posi}(\overline{\sigma}) \cup \{0w \mid w \in \mathsf{Nega}(\tau)\}\end{aligned}$$

The set $\mathrm{Pos}(b, \tau)$ of *positions* of a type b in a type τ is defined by

$$\begin{array}{ll}\mathrm{Pos}(b, b) = \{\varepsilon\} & \mathrm{Pos}(b, \overline{\sigma}) = \{i.w \mid i = 1, \ldots, |\overline{\tau}|, w \in \mathrm{Pos}(b, \sigma_i)\}\\ \mathrm{Pos}(b, c) = \varnothing \text{ if } b \neq c & \mathrm{Pos}(b, \overline{\sigma} \to \tau) = \mathrm{Pos}(b, \overline{\sigma}) \cup \{0w \mid w \in \mathrm{Pos}(b, \tau)\}.\end{array}$$

Here we regard that every function type of the form $\sigma \to \tau$ is formally a type from a singleton sequence, i.e. $\langle\sigma\rangle \to \tau$.

Definition 7.21. A *constructor* is a function symbol $c : \overline{\tau} \to b$ which does not occur at the root of the lhs of any rule. The set of all constructors defines a preorder $\leq_{\mathscr{B}}$ on the set $\mathscr{B}$ of types by $b \leq_{\mathscr{B}} \overline{\tau}$ if b occurs in $\overline{\tau}$ for a constructor $c : \overline{\tau} \to b$. We write $\geq_{\mathscr{B}}$ to be the inverse of $\leq_{\mathscr{B}}$. Let $<_{\mathscr{B}}$ be the strict part of $\leq_{\mathscr{B}}$ and $=_{\mathscr{B}} \triangleq \leq_{\mathscr{B}} \cap \geq_{\mathscr{B}}$ the equivalence relation. A type b is *positive* if for each constructor $c : \overline{\sigma} \to b$ of it, $\mathrm{Pos}(b', \overline{\sigma}) \subseteq \mathsf{Posi}(\overline{\sigma})$ for any type b' s.t. $b =_{\mathscr{B}} b'$. A constructor $c : \overline{\sigma} \to b$ is *positive* if b is positive.

Definition 7.22. A metavariable M is *accessible* in a meta-term t if there are distinct bound variables $\overline{x}$ such that $M[\overline{x}] \in \mathsf{Acc}(t)$, where $\mathsf{Acc}(t)$ is the least set satisfying the following clauses:

a1. $t \in \mathsf{Acc}(t)$.
a2. If $x.u \in \mathsf{Acc}(t)$ then $u \in \mathsf{Acc}(t)$.
a3. If $c(s_1, \ldots, s_n) \in \mathsf{Acc}(t)$ then each $s_i \in \mathsf{Acc}(t)$ for a constructor c.
a4. Let $f : \tau_1, \ldots, \tau_n \to b$ and $f(u_1, \ldots, u_n) \in \mathsf{Acc}(t)$. Then $u_i \in \mathsf{Acc}(t)$ $(1 \leq i \leq n)$ if for all types $b' \leq_{\mathscr{T}} b$, $\mathrm{Pos}(b', \tau_i) \subseteq \mathsf{Posi}(\tau_i)$ (Blanqui, 2016, Definition 15).

Definition 7.23. A meta-term u is a *covered subterm* of t, written $t \mathrel{\widehat{\trianglerighteq}} u$, if there are two positions $p \in \mathrm{Pos}(t), q \in \mathrm{Pos}(t_{|p})$ such that $u = t[t_{|pq}]_p$, (i) $\forall r < p.\, t_{|r}$ is headed by an abstraction, and (ii) $\forall r < q.\, t_{|pr}$ is headed by a function symbol.

For example, $\mathtt{f}(a, \mathtt{c}(x)) \mathrel{\widehat{\trianglerighteq}} \mathtt{c}(x)$, $\mathtt{lam}(x.M[x]) \mathrel{\widehat{\trianglerighteq}} x.M[x]$, and $y.\mathtt{lam}(x.M[x]) \mathrel{\widehat{\trianglerighteq}} y.x.M[x]$.

Definition 7.24. A set of rules $\ell \Rightarrow r$ induces the following relation on function symbols in a signature Σ: *f depends on g* if there is a rule defining f (i.e. whose lhs is headed by f) in the rhs of which g occurs. Its transitive closure is denoted by $>_\Sigma$, and the associated equivalence relation is denoted by $=_\Sigma$.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

Definition 7.25. Given $f : \tau_1, \ldots, \tau_n \to \tau \in \Sigma$, the *computable closure* $\mathsf{CC}_f(\bar{t})$ of a meta-term $f(\bar{t})$ is the least set CC satisfying the following clauses. All the meta-terms below are supposed to be well-typed.

1. **(meta M)** If $M : \tau_1, \ldots, \tau_p \to \tau$ is accessible in some of $\bar{t}$, and $\overline{u} \in \mathsf{CC}$, then $M[\overline{u}] \in \mathsf{CC}$.
2. For any variable x, $x \in \mathsf{CC}$.
3. If $u \in \mathsf{CC}$ then $x.u \in \mathsf{CC}$.
4. **(fun $f >_\Sigma g$)** If $f >_\Sigma g$ and $\overline{w} \in \mathsf{CC}$, then a well-typed $g(\overline{w}) \in \mathsf{CC}$.
5. **(fun $f =_\Sigma g$)** If $\overline{u} \in \mathsf{CC}$ such that $\bar{t} \mathrel{\widehat{\rhd}^{\mathsf{lex}}} \overline{u}$, then a well-typed $g(\overline{u}) \in \mathsf{CC}$, where $\widehat{\rhd}^{\mathsf{lex}}$ is the lexicographic extension of the strict part of $\widehat{\trianglerighteq}$.

The labels **(meta M)** etc. are used for references in a termination proof using SOL (see, e.g., Section 1.3). The lexicographic extension $\widehat{\rhd}^{\mathsf{lex}}$ can be left-to-right or right-to-left comparison. This option sometimes called "status" is supposed for each $f \in \Sigma$ before applying the General Schema, and for every $f =_\Sigma g$, the statuses of f and g must be the same.

Using this, the item 5. **(fun $f =_\Sigma g$)** is expanded as follows. Suppose the status of f and g are left-to-right comparison. Let $\overline{u} = u_1, \ldots, u_n \in \mathsf{CC}$ and $t = t_1, \ldots, t_n$. If there exists $1 \leq i \leq n$ such that $t_1 = u_1, \ldots, t_i = u_i$ and $t_{i+1} > u_{i+1}$ then $g(\overline{u}) \in \mathsf{CC}$.

Definition 7.26. A rule $f(\bar{t}) \Rightarrow r$ *satisfies the General Schema* if $\mathsf{CC}_f(\bar{t}) \ni r$.

Theorem 7.27. *Suppose that given a signature Σ and rules C satisfy:*

i. All the types in Σ are simple types,
ii. $>_{\mathscr{B}}$ is well-founded,
iii. every constructor is positive, and
iv. $>_\Sigma$ is well-founded.

If all the rules of C satisfy the General Schema, then C is strongly normalising.

The SOL's command `sn` implements checking all the conditions of Theorem 7.27.

Associated simply typed systems. We say that a type substitution $\xi : S \to \mathscr{T}$ is a *simple type instantiation* if for all $t \in S$, $\xi(t)$ does not involve type variables. For each function symbol $f : \overline{\sigma} \to \tau \in \Sigma$ and polymorphic rule $\Theta \vdash l \Rightarrow r : \tau \in C$, we define the following computation system:

$$\Sigma_{\mathsf{inst}} \triangleq \{f : \overline{\sigma}\xi \to \tau\xi \mid f\overline{\sigma} \to \tau \in \Sigma,\ \xi \text{ is a simple type instantiation}\}$$
$$C_{\mathsf{inst}} \triangleq \{\Theta\xi \triangleright \Gamma\xi \vdash l\xi \to r\xi : \alpha\xi \mid \xi \text{ is a simple type instantiation}\}$$

and call it *the associated simply typed instance*.

The following are immediate by the definition of $\Rightarrow_C$.

Proposition 7.28. *A polymorphic computation system (Σ, C) is SN if and only if $(\Sigma_{inst}, C_{inst})$ is SN.*

Proposition 7.29. *Assume a polymorphic computation system C with a polymorphic signature Σ. If C_{inst} with Σ_{inst} satisfies the General Schema, then C is SN.*

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

8 Implementation of SOL

In this section, we describe some details of the system SOL.

The system SOL consists of about 7,000 lines Haskell code, and works on Glasgow Haskell Compiler (tested on version 7.6.2 and 8.0.2). SOL uses Template Haskell (Sheard & Jones, 2002) with a custom parser generated by Alex (for lexer) and Happy (for paper) to realise readable notation for signatures and rules. There are a command line interface using GHCi (Section 8.2), and a web interface (Section 8.3).

8.1 Design and syntax

Design principles. The design principles of SOL are as follows:

- One of the purposes of SOL is to assist the user to develop better and suitable axioms and rules for a problem.
- Hence, SOL outputs enough information of how the checking proceeds and where the methods fail.
- The syntax should be natural and as close as possible to the ordinary mathematical meta-language of computation rules and axioms, to let the users quickly test their own rules and axioms written in a theory paper, using SOL without embarrassing encoding.

Syntax. SOL's definitions are realised by the feature of quasi-quotation of Template Haskell. For example, the notations `[signature|..]` and `[rule|..]` are the quasi-quotations, and the keywords `signature` and `rule` are implemented as quasi-quoters of Template Haskell. SOL's syntax has a layout rule. Newline is regarded as a separator of declarations in signatures and rules. For example,

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

If the user wants to write the declarations sequentially, the user can 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 enclosed by round brackets, such as "`(beta)`", which is placed before a rule. Any blank line is not allowed in the SOL's bracket. The user should write like

```
sig = [signature|
  f : a -> b
  ;
  g : b -> c       |]
```

The user makes sure writing a space before and after the arrow type constructor "`->`". "`a->b`" is bad, the user should write "`a -> b`". Rules and axioms are internally the same; for example, the following definitions give internally the same data:

```
lambdaCal = [rule|  (beta) lam(x.M[x])@N => M[N] |]
lambdaCal = [axiom| (beta) lam(x.M[x])@N =  M[N] |]
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

This surface distinction may be useful for the user to represent one's intention. A meta-term can be written using the `[o|..|]`-bracket, such as `[o| lam(x.M[x]) |]` (meaning: **o**bject for term). In GHCi, the user should first enable the Template Haskell feature as

```
*SOL> :set -XTemplateHaskell -XQuasiQuotes
```

and then the user can use the notation freely. For example, we can invoke FCU unification between meta-terms.

```
*SOL> unify [o| f(lam(x.M[x]),Y) |]  [o| f(lam(x.x@x),a()) |]
"M|-> z1.(z1@z1), Y|-> a"
```

8.2 Commands

SOL has the following commands implemented as Haskell functions:

- `cri <rules> <signature>`
  Enumerating critical pairs of a computation system and check their joinability.
- `crimod <rules> <equations>`
  Enumerating critical pairs of a partitioned algebraic theory and checking their joinability modulo equational theory.
- `sn <rules> <signature>`
  Checking SN of a computation system.
- `unify <meta-term> <meta-term>`
  Unifying two meta-terms using the extended FCU algorithm.
- `match <meta-term> <meta-term>`
  Trying to match two meta-terms
- `normalise <meta-term> <rules>`
  Normalising a meta-term to get a normal form (cf. Section 3.2).
- `pr <any>`: a generic printing command

As we have demonstrated, checking SN is by the command `sn` that takes two arguments of a rule set and signature. Checking the critical pairs is by the command `cri` that takes two arguments of a rule set and signature.

The command `pr` is a generic command for printing, e.g.

```
*SOL> pr cbn
 (beta) lam(x.M[x])@N => M[N]
 (eta)  lam(x.(M@x)) => M

*SOL> pr sigcbn
 app : (Arr(a,b),a) -> b
 lam : (a -> b) -> Arr(a,b)
```

Without `pr`, we can see the internal data structure

```
*SOL> [o| lam(x.M[x]) |]
"lam" :< ["x" :. "M" :$ [V "x"]]
```

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

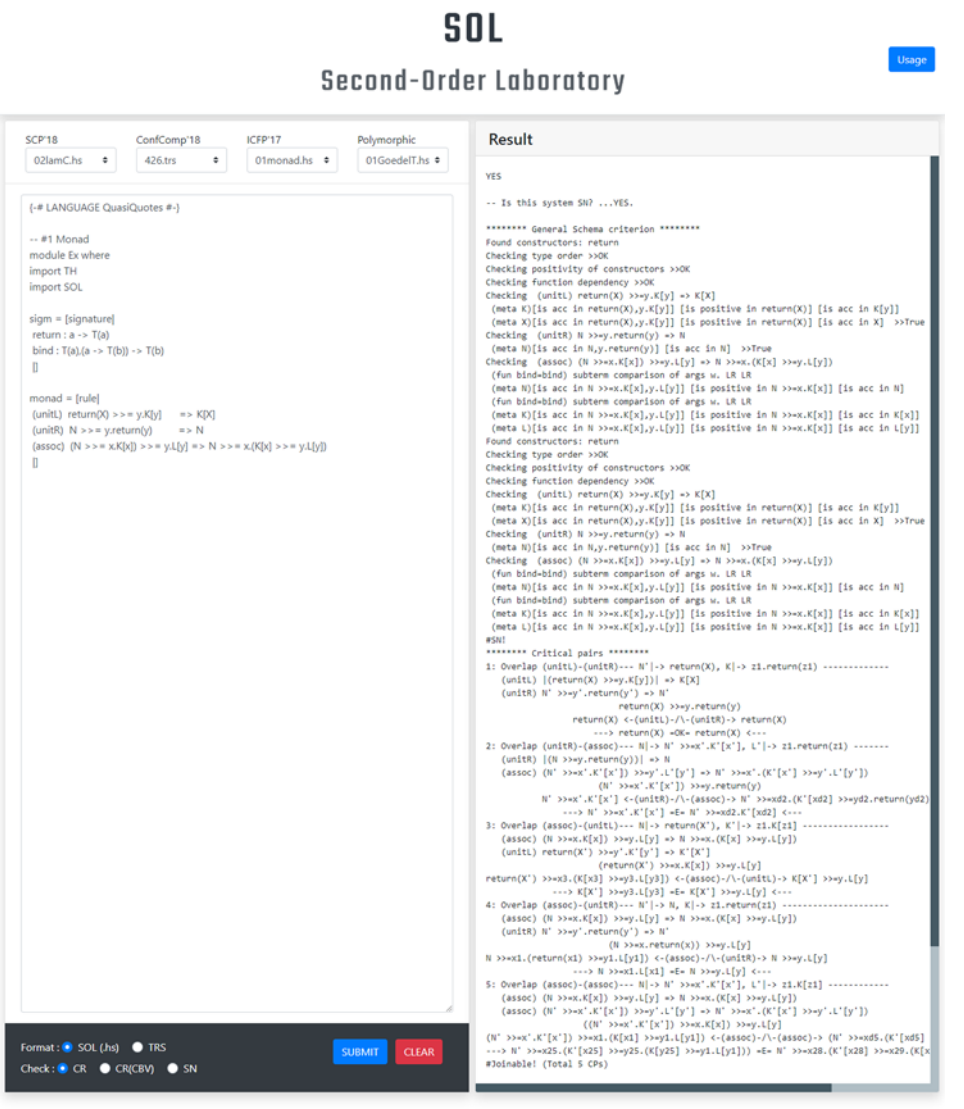


Fig. 8. Web interface of SOL.

```
*SOL> [rule| (sample) f(X) => a() |]
["sample" :! "f" :< ["X" :$ []] :=> "a" :< []]
```

which may be useful to get information for meta-programming on rules.

8.3 Web interface

To ease the use of SOL, we have also constructed a web interface for SOL (Figure 8), which is available at

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

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

The usage is as follows:

- The user selects a file using the topmost of the pull-down menu, and checks CR or SN radio button. Push [SUBMIT] button for checking.
- The user can also submit own computation rules, firstly pushing [CLEAR], secondly writing rules by filling the form, and finally checking CR or SN radio button. Push [SUBMIT] button for checking.

All the examples in this paper have been stored in the web interface and one can choose an example from the pull-down menu.

9 Summary

We have presented a general methodology of proving decidability of equational theories of programming language concepts in the framework of second-order algebraic theories. We proposed a Haskell-based analysis tool SOL which assists the proofs of confluence and strong normalisation of computation rules derived from second-order algebraic theories.

To cover various examples in programming language theory, we have combined and extended both syntactical and semantical results on second-order computation in non-trivial directions. In particular, our choice of Yokoyama's deterministic second-order patterns as a syntactic construct of rules is important to cover a wide range of examples, which makes our computation system useful for programming language theory. We demonstrated how to prove decidability of 11 examples of algebraic theories in the literature using SOL.

As related systems, there are equational specification systems based on rewriting, OBJ (Goguen et al., 1996), Maude (Clavel et al., 2007), and CafeOBJ (Diaconescu & Futatsugi, 2002). But these systems can only deal with order-sorted *first-order* equational theories, and cannot describe higher-order equational theories, such as the λ-calculus directly. In this respect, SOL is unique. SOL is the first system that can automatically check both confluence and strong normalisation of second-order polymorphic computation systems in a single system.

SOL's foundational technologies may be applicable to other systems, such as termination or confluence checker of Haskell's rewrite rule pragma (Peyton Jones et al., 2001), and type functions (Chakravarty et al., 2005a,b), which have currently no sufficient checking mechanism for rules.

Acknowledgments

I acknowledge Kazuki Fuju and Dateyao Faustin Dieudonne for developing the Vue.js-based web interface of SOL. It has been helpful to discuss this work with many people, including Yoshihito Toyama, Kentaro Kikuchi, Tetsuo Yokoyama, Tatsuya Abe, Kazuyuki Asada, Kazuhiko Sakaguchi, Yuito Murase, and Masahito Hasegawa. I also thank to Kazutaka Matsuda for advice on parsing in Haskell, Frédéric Blanqui for clarifying details of the General Schema, and the reviewers for their constructive comments. This work was supported in part by JSPS KAKENHI Grant Number 17K00092.

References

Aoto, T. & Toyama, Y. (2012) A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logic. Method. Comput. Sci.* **8**(1), 1–29.

Baader, F. & Nipkow, T. (1998) *Term Rewriting and All that*. Cambridge University Press.

Baxter, L. (1977) *The Complexity of Unification*. PhD Thesis, Department of Computer Science, University of Waterloo.

Benton, N. & Hyland, M. (2003) Traced premonoidal categories. *Theor. Inform. Appl.* **37**(4), 273–299.

Benton, P. N., Bierman, G. M. & de Paiva, V. (1998) Computational types from a logical perspective. *J. Funct. Program.* **8**(2), 177–193.

Bird, R. & Moor, O. D. (1996) *Algebra of Programming*. Prentice-Hall.

Blanqui, F. (2000) Termination and confluence of higher-order rewrite systems. In Rewriting Techniques and Application (RTA 2000). LNCS, vol. 1833. Springer, pp. 47–61.

Blanqui, F. (2016) Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theor. Comput. Sci.* **611**, 50–86.

Chakravarty, M. M. T., Keller, G. & Peyton Jones, S. L. (2005a) Associated type synonyms. In Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26–28, 2005, pp. 241–253.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S. L. & Marlow, S. (2005b) Associated types with class. In Proceedings of POPL'05, Long Beach, California, USA, January 12–14, 2005, pp. 1–13.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. & Talcott, C. L. (2007) *All About Maude - A High-Performance Logical Framework, How to Specify, program and Verify Systems in Rewriting Logic*. LNCS, vol. 4350. Springer.

Coquand, T. (1992) Pattern matching with dependent types. In Proceedings of the 3rd Workshop on Types for Proofs and Programs.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Proceedings of POPL'82, pp. 207–212.

Diaconescu, R. & Futatsugi, K. (2002) Logical foundations of CafeOBJ. *Theor. Comput. Sci.* **285**(2), 289–318.

Fiore, M. (2002) Semantic analysis of normalisation by evaluation for typed lambda calculus. In Proceedings of PPDP'02. ACM, pp. 26–37.

Fiore, M. (2008) Second-order and dependently sorted abstract syntax. In Proceedings of LICS'08, pp. 57–68.

Fiore, M. & Hamana, M. (2013) Multiversal polymorphic algebraic theories: syntax, semantics, translations, and equational logic. In 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS, 2013, pp. 520–529.

Fiore, M. & Hur, C.-K. (2010) Second-order equational logic. In Proceedings of CSL'10. LNCS, vol. 6247, pp. 320–335.

Fiore, M. & Mahmoud, O. (2010) Second-order algebraic theories. In Proceedings of MFCS'10. LNCS, vol. 6281, pp. 368–380.

Fiore, M., Plotkin, G. & Turi, D. (1999) Abstract syntax and variable binding. In Proceedings of LICS'99, pp. 193–202.

Fiore, M. P. & Campos, M. D. (2013) The algebra of directed acyclic graphs. In Coecke B., Ong, L., & Panangaden, P. (eds), *Computation, Logic, Games, and Quantum Foundations*. LNCS, vol. 7860, pp. 37–51.

Fiore, M. P., Moggi, E. & Sangiorgi, D. (1996) A fully-abstract model for the pi-calculus. In 11th Logic in Computer Science Conference (LICS'96). IEEE, pp. 43–54. Information and Computation 179, 76–117 (2002).

Gibbons, J. (1995) An initial-algebra approach to directed acyclic graphs. In Proceedings of MPC'95. LNCS, vol. 947, pp. 282–303.

Gibbons, J. & Hinze, R. (2011) Just do it: simple monadic equational reasoning. In Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011, pp. 2–14.

https://doi.org/10.1017/S0956796819000157 Published online by Cambridge University Press

Girard, J.-Y. (1989) *Proofs and Types*. Cambridge University Press. translated and with appendices by Paul Taylor, Yves Lafont.

Goguen, J., Winkler, T., Meseguer, J. & Futatsugi, K. (2000) *Introducing OBJ.* In Goguen, J. & Malcolm, G. (eds), Software Engineering with OBJ. ADFM, vol. 2, pp. 3–167.

Hamana, M. (2004) Free Σ-monoids: a higher-order syntax with metavariables. In Asian Symposium on Programming Languages and Systems (APLAS'04). LNCS, vol. 3302, pp. 348–363.

Hamana, M. (2005) Universal algebra for termination of higher-order rewriting. In Rewriting Techniques and Application (RTA'05). LNCS, vol. 3467, pp. 135–149.

Hamana, M. (2007) Higher-order semantic labelling for inductive datatype systems. In Proceedings of 9th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'07). ACM, pp. 97–108.

Hamana, M. (2010) Semantic labelling for proving termination of combinatory reduction systems. In Proceedings of WFLP'09. LNCS, vol. 5979, pp. 62–78.

Hamana, M. (2011) Polymorphic abstract syntax via Grothendieck construction. In FoSSaCS'11. LNCS, vol. 3467, pp. 381–395.

Hamana, M. (2016) Strongly normalising cyclic data computation by iteration categories of second-order algebraic theories. In Proceedings of FSCD'16. The Leibniz International Proceedings in Informatics (LIPIcs), vol. 52, 21:1–21:18.

Hamana, M. (2017) How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation. *Proc. ACM Program. Lang.* **1**(22), 1–28.

Hamana, M. (2018) Polymorphic rewrite rules: confluence, type inference, and instance validation. In Proceedings of 14th International Symposium on Functional and Logic Programming (FLOPS'18). LNCS, vol. 10818, 99–115.

Hasegawa, M. (2005) Classical linear logic of implications. *Math. Struct. Comput. Sci.* **15**(2), 323–342.

Huet, G. (1980) Confluent reductions: abstract properties and applications to term rewriting systems. *J. ACM* **27**(4), 797–821.

Jouannaud, J., Kirchner, H. & Remy, J. (1983) Church–Rosser properties of weakly terminating term rewriting systems. In Proceedings of the 8th International Joint Conference on Artificial Intelligence, pp. 909–915.

Kelly, G. M. (1964) On mac lane's conditions for coherence of natural associativities, commutativities, etc. *J. Alg.* **1**(4), 397–402.

Knuth, D. & Bendix, P. (1970) Simple word problems in universal algebras. In *Computational Problem in Abstract Algebra*. Oxford: Pergamon, pp. 263–297. Included also in Automation of reasoning 2, Springer (1983), pp. 342–376.

Libal, T. & Miller, D. (2016) Functions-as-constructors higher-order unification. In Proceedings of FSCD 2016. Leibniz International Proceedings in Informatics (LIPIcs), vol. 52, pp. 26:1–26:17.

Lindley, S. (2007) Extensional rewriting with sums. In *Typed Lambda Calculi and Applications*, 8th International Conference, TLCA 2007, Paris, France, June 26–28, 2007. Proceedings, pp. 255–271.

Lindley, S. & Stark, I. (2005) Reducibility and ⊤⊤-lifting for computation types. In *Typed Lambda Calculi and Applications*, 7th International Conference, TLCA 2005, Nara, Japan, April 21–23, 2005. Proceedings, pp. 262–277.

Mac Lane, S. (1963) Natural associativity and commutativity. *Rice Univ. Stud.* **49**(4), 28–46. Available at: http://hdl.handle.net/1911/62865.

Mac Lane, S. (1971) *Categories for the Working Mathematician*. Graduate Texts in Mathematics, vol. 5. Springer-Verlag.

Mayr, R. & Nipkow, T. (1998) Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.* **192**(1), 3–29.

Melliès, P.-A. (2010) Segal condition meets computational effects. In Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11–14 July 2010, Edinburgh, UK, 150–159.

Miller, D. (1991) A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic Comput.* **1**(4), 497–536.
Milner, R. (1996) Semantic ideas in computing. In Wand, I. and Milner, R. (eds), *Computing Tomorrow*. Cambridge University Press.
Milner, R. (1999) *Communicating and Mobile Systems - The π-Calculus*. Cambridge University Press.
Moggi, E. (1988) *Computational Lambda-Calculus and Monads*. LFCS ECS-LFCS-88-66. University of Edinburgh.
Moggi, E. (1991) Notions of computation and monads. *Inform. Comput.* **93**, 55–92.
Nipkow, T. (1991) Higher-order critical pairs. In Proceedings of 6th IEEE Symposium on Logic in Computer Science, pp. 342–349.
Ohta, Y. & Hasegawa, M. (2006) A terminating and confluent linear lambda calculus. In *Rewriting Techniques and Application (RTA'06)*. LNCS, vol. 1833, pp. 166–180.
Peyton Jones, S. L., Tolmach, A. & Hoare, T. (2001) Playing by the rules: rewriting as a practical optimisation technique in GHC. In Haskell Workshop 2001.
Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. In Proceedings of the ACM SIGPLAN'88 Symposium on Language Design and Implementation, pp. 199–208.
Plotkin, G. & Power, J. (2002) Notions of computation determine monads. In Proceedings of FoSSaCS'02, pp. 342–356.
Plotkin, G. D. (1975) Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.* **1**(2), 125–159.
Prehofer, C. (1995) *Solving Higher-Order Equations : From Logic to Programming*. PhD Thesis, Technische Univerität München.
Sabry, A. & Wadler, P. (1997) A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.* **19**(6), 916–941.
Sheard, T. & Jones, S. P. (2002) Template metaprogramming for Haskell. In Proceedings of the Haskell Workshop 2002.
Stark, I. (1996) A fully abstract domain model for the π-calculus. In Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, pp. 36–42.
Stark, I. (2008) Free-algebra models for the π-calculus. *Theor. Comput. Sci.* **390**(2–3), 248–270.
Staton, S. (2009) Two cotensors in one: presentations of algebraic theories for local state and fresh names. *Electr. Notes Theor. Comput. Sci.* **249**, 471–490.
Staton, S. (2013a) An algebraic presentation of predicate logic. In Proceedings of FoSSaCS'13. LNCS, vol. 7794, pp. 401–417.
Staton, S. (2013b) Instances of computational effects: an algebraic perspective. In 28th Annual ACM/IEEE Symposium on Logic in Computer Science. LICS, vol. 2013, p. 519.
Staton, S. (2015) Algebraic effects, linearity, and quantum programming languages. In Proceedings of POPL'15, pp. 395–406.
van de Pol, J. (1994) Termination proofs for higher-order rewrite systems. In the First International Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA'93). LNCS, vol. 816, pp. 305–325.
Wadler, P. 1990 (June) Comprehending monads. In ACM Conference on Lisp and Functional Programming, pp. 61–78.
Yokoyama, T., Hu, Z. & Takeichi, M. (2003) Deterministic higher-order patterns for program transformation. In Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25–27, 2003, Revised Selected Papers, 128–142.
Yokoyama, T., Hu, Z. & Takeichi, M. (2004a) Deterministic second-order patterns. *Inf. Process. Lett.* **89**(6), 309–314.
Yokoyama, T., Hu, Z. & Takeichi, M. (2004b) Deterministic second-order patterns for program transformation. *Comput. Soft.* **21**(5), 71–76. In Japanese.