How to Prove Your Calculus Is Decidable: Practical Applications of Second-Order Algebraic Theories and Computation

MAKOTO HAMANA, Gunma University, Japan

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 SOL, 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 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 Stark's theory of π -calculus.

CCS Concepts: • **Theory of computation** → **Equational logic and rewriting**; **Algebraic semantics**; **Operational semantics**;

Additional Key Words and Phrases: Theory, λ -calculus, algebra, monad, equational logic, semantics, confluence, termination, second-order algebraic theory, Haskell

ACM Reference Format:

Makoto Hamana. 2017. How to Prove Your Calculus Is Decidable: Practical Applications of Second-Order Algebraic Theories and Computation. *Proc. ACM Program. Lang.* 1, ICFP, Article 22 (September 2017), 28 pages. <https://doi.org/10.1145/3110266>

1 INTRODUCTION

Equations and equational reasoning are ubiquitous in functional programming in pure [Bird and Moor 1996] and even impure setting [Gibbons and 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 Association for Computing Machinery.

2475-1421/2017/9-ART22

<https://doi.org/10.1145/3110266>

1990], especially in Haskell. A monad T is a structure with two operations

$$\text{return} : a \rightarrow Ta \qquad \gg : Ta \rightarrow (a \rightarrow Tb) \rightarrow Tb$$

satisfying three laws

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

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

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

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:

Given two well-typed terms s, t consisting of arbitrary combinations of return , $\gg$ and variables, is an equation

$$s = t$$

derivable from the three laws of monad?

This problem is decidable, i.e., there is a terminating algorithm answering yes or no for any two terms. How do we prove it? Below, we sketch theoretical and practical methods to prove the decidability of the theory of monad.

1.2 Theoretical method

1.2.1 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 [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. 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.

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

Fig. 1. Overlaps in monad laws

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 and Hur 2010; Fiore and Mahmoud 2010]. Second-order algebraic theories are founded on the mathematical theory of second-order abstract syntax [Fiore 2008; Fiore et al. 1999; Hamana 2004]. 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.

1.2.2 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:

$$\begin{aligned}
 (\text{unitL}) \quad & \text{return}(X) \gg y.K[y] \Rightarrow K[X] \\
 (\text{unitR}) \quad & E \gg y.\text{return}(y) \Rightarrow E \\
 (\text{assoc}) \quad & (E \gg x.K[x]) \gg y.L[y] \Rightarrow E \gg x.(K[x] \gg y.L[y])
 \end{aligned}$$

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 .

1.2.3 CR. 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 that it has, in all, 5 patterns of overlap situations as depicted in Fig. 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: $(\text{return}(X') \gg x.K[x]) \gg y.L[y]$, 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 (Def. 6.5). The pair of divergent terms is called a **critical pair** (**CP** for short) (Def. 6.6). The diagrams in Fig. 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 situations¹ are convergent. This convergence is ensured by Theorem 6.7 and the property is called *local confluence* (**WCR** for short, see §6.4), meaning that every possible one-step divergence is always joinable. By applying Newman’s lemma (Lemma 6.4), 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.

1.2.4 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 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 rule²

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

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 [Pol 1994] are known for such a “weight” based proof, which is not readily to apply 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 flavor to the β -rule, as X is applied to K 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 [Girard 1989] by Tait and Girard. Particularly in this paper, we use Blanqui’s General Schema criterion [Blanqui 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 [Moggi 1988] has been shown by translation into λ -calculus with sums [Benton et al. 1998], or using $\top\top$ -lifting method [Lindley and 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

1.3.1 Why SOL. What were the difficult parts of the proof of confluence and strong normalisation? Actually, the proof structure is not theoretically difficult³ 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

¹Because the 5 patterns above might occur in an arbitrary larger term context and because every metavariable can be instantiated by arbitrary term.

²Milner used the same notation in [Milner 1996], <https://books.google.co.jp/books?id=lGgPwQFeXSEC&pg=PA271>.

³Of course, the invention of $\top\top$ -lifting in [Lindley and Stark 2005] is highly non-trivial.

(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, 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 overlooking. 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.

1.3.2 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 interpreter 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.

1.3.3 Monad in SOL [Problem #1]. We define the signature by

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

and define the axioms by⁴

```
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 §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.

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

```
*SOL> cri monad
1: Overlap (unitL)-(unitR)--- N'|-> return(X), K|-> z1.return(z1) -----
   L: (return(X) >=>y.K[y]) => K[X]
   R: N' >=>y'.return(y') => N'
                                     return(X) >=>y.return(y)
   return(X) <-(unitL)-^-(unitR)-> return(X)
   ----> return(X) =OK= return(X) <----
```

⁴Several operators are reserved in SOL to be translated automatically to the corresponding function symbols for readability. Here the infix `>=>` is translated to the prefix `bind`.

```

2: Overlap (unitR)-(assoc)--- N|-> N' >=>x'.K'[x'], L'|-> z1.return(z1) -----
L: (N >=>y.return(y)) => N
R: (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' >=>x'.(K'[x'] >=>y'.return(y'))
      ---> N' >=>x'.K'[x'] =OK= N' >=>x'.K'[x'] <---
3: Overlap (assoc)-(unitL)--- N|-> return(X'), K'|-> z1.K[z1] -----
L: (N >=>x.K[x]) >=>y.L[y] => N >=>x.(K[x] >=>y.L[y])
R: 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] =OK= K[X'] >=>y.L[y] <---
4: Overlap (assoc)-(unitR)--- N'|-> N, K|-> z1.return(z1) -----
L: (N >=>x.K[x]) >=>y.L[y] => N >=>x.(K[x] >=>y.L[y])
R: 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] -----
L: (N >=>x.K[x]) >=>y.L[y] => N >=>x.(K[x] >=>y.L[y])
R: (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' >=>x'.(K'[x'] >=>y'.K[y'])) >=>y.L[y]
      ---> N' >=>x.(K'[x'] >=>y.(K[y] >=>y.L[y])) =E= N' >=>x.(K'[x'] >=>x.(K[x] >=>y.L[y])) <---
#Joinable! (Total 5 CPs)

```

The above five overlaps descriptions correspond exactly to the five diagrams shown in Fig. 1. The labels L: and R: indicate the rules used in the left and right paths of a divergence, and the highlight in L-rule shows that the subterm is unifiable with the root of left-hand side of R-rule.

For example, in the overlap 3, the subterm $(N \gg=x.K[x])$ in the L-rule is unifiable with the term $\text{return}(X') \gg=y'.K'[y']$ in the R-rule using the unifier $N|-> \text{return}(X')$, $K'|-> z1.K[z1]$ described at the immediate 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 underline term $(\text{return}(X') \gg=x.K[x]) \gg=y.L[y]$ which exactly corresponds to the source in a diagram of a divergence in Fig. 1. The lines involving $\wedge$ (indicating “divergence”) mimics the divergence diagram and the joinability test in text. The sign =OK= denotes syntactic equal, and =E= denotes equal modulo an equational theory, here just the α -equivalence.

1.3.5 SN. We prove termination of the axioms. The command `sn` of SOL tries to prove SN of the rules by checking all the conditions of the General Schema (§6.8):

```

*SOL> sn monad sigm
Found constructors: return
Checking type order >>OK
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! -- on this signature

```

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

$$\begin{array}{c}
\frac{y : b \in \Gamma}{\Theta \triangleright \Gamma \vdash y : b} \quad \frac{(M : a_1, \dots, a_m \rightarrow b) \in \Theta \quad \Theta \triangleright \Gamma \vdash t_i : a_i \quad (1 \leq i \leq m)}{\Theta \triangleright \Gamma \vdash M[t_1, \dots, t_m] : b} \quad \frac{f : (\overline{a_1} \rightarrow b_1), \dots, (\overline{a_m} \rightarrow b_m) \rightarrow c \in \Sigma \quad \Theta \triangleright \Gamma, \overline{x_i} : \overline{a_i} \vdash t_i : b_i \quad (1 \leq i \leq m)}{\Theta \triangleright \Gamma \vdash f(\overline{x_1^{a_1}.t_1}, \dots, \overline{x_m^{a_m}.t_m}) : c}
\end{array}$$

Fig. 2. Typing rules of meta-terms

1.4 Organisation

The paper is organised as follows. We first introduce the framework of second-order algebraic theories and computation rules in §2. From §3 to 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, and Part III: A Theory of π -calculus. We consider 8 problems tagged as “Problem [n]”. In §6, we present the technical foundations of our notions of computation systems and SOL, including second-order matching (§6.1, 6.2), unification (§6.3), rewriting (§6.4), critical pairs (§6.5-6.7), and the General Schema (§6.8). In §7, we summarise the paper.

The SOL system is available from the author’s homepage.

2 SECOND-ORDER ALGEBRAIC THEORIES WITH MOLECULAR TYPES

In this section, we introduce the framework of second-order algebraic theories [Fiore and Hur 2010], [Fiore and Mahmoud 2010] with minor modification on types (see also a related framework of polymorphich algebraic theories [Fiore and Hamana 2013]). We formulate axioms and computation rules of calculi for programming languages as second-order algebraic theories. It gives a formal framework to provide syntax, logic and computation.

NOTATION 2.1. We use the notation $\overline{A}$ for a sequence $A_1, \dots, A_n$, and $|\overline{A}|$ for its length. The notation $s[u]_p$ means replacing the position p of s at with u , and $s|_p$ means selecting a subterm of the position p . We use the abbreviations “lhs” and “rhs” to mean left-hand side and right-hand side, respectively.

2.1 Syntax

2.1.1 Types. We assume that $\mathcal{A}$ is a set of *atomic types* (e.g. Bool, Nat, etc.) We assume that the set of *molecular types* (or *mol types* for short) $\mathcal{B}$ is generated by atomic types, and type constructors is the least set satisfying

$$\mathcal{B} = \mathcal{A} \cup \{T(a_1, \dots, a_n) \mid a_1, \dots, a_n \in \mathcal{B}, T \text{ is an } n\text{-ary type constructor}\}$$

By a *type constructor* T of arity n , we mean that it takes n -mol types $a_1, \dots, a_n$ and gives a mol type $T(a_1, \dots, a_n)$. Note that we do not regard “ $\rightarrow$ ” as a type constructor (in our sense). We usually write mol types as $a, b, c, \dots$

REMARK 2.2. Molecular types work as “base types” in ordinary type theories. But in our usage, we need “base types” which are constructed from “more basic” types. Hence we first assume atomic types as the most atomic ones, and then generate molecular types from them. Molecular types exactly correspond to base types in [Blanqui 2000; Staton 2013b].

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

$$f : (\overline{a_1} \rightarrow b_1), \dots, (\overline{a_m} \rightarrow b_m) \rightarrow c$$

where all a_i, b_i, c are mol types (thus any function symbol is of up to second-order type).

$$\begin{array}{c}
\text{(Ax1)} \frac{(\Theta \triangleright \Gamma \vdash t_1 = t_2 : c) \in \mathcal{E}}{\Theta \triangleright \Gamma \vdash t_1 = t_2 : c} \quad \text{(Ax2)} \frac{(\Theta \triangleright \Gamma \vdash t_1 = t_2 : c) \in \mathcal{E}}{\Theta \triangleright \Gamma \vdash t_2 = t_1 : c} \\
\\
\text{(Fun)} \frac{f : (\overline{a_1} \rightarrow b_1), \dots, (\overline{a_k} \rightarrow b_k) \rightarrow c \in \Sigma \quad \Theta \triangleright \Gamma, \overline{x_i} : \overline{a_i} \vdash t_i = t'_i : b_i \quad (\text{some } i \text{ s.t. } 1 \leq i \leq k)}{\Theta \triangleright \Gamma \vdash f(\overline{x_1^{a_1}}.t_1, \dots, \overline{x_i^{a_i}}.t_i, \dots, \overline{x_1^{a_1}}.t_k) = f(\overline{x_1^{a_1}}.t_1, \dots, \overline{x_i^{a_i}}.t'_i, \dots, \overline{x_1^{a_1}}.t_k) : c} \\
\\
\text{(Sub)} \frac{\Theta \triangleright \Gamma', \overline{x_i} : \overline{a_i} \vdash s_i : b_i \quad (1 \leq i \leq k) \quad M_1 : (\overline{a_1} \rightarrow b_1), \dots, M_k : (\overline{a_k} \rightarrow b_k) \triangleright \Gamma \vdash t_1 = t_2 : c}{\Theta \triangleright \Gamma, \Gamma' \vdash t_1 [\overline{M} \mapsto \overline{x}.s] = t_2 [\overline{M} \mapsto \overline{x}.s] : c} \\
\\
\text{(Ref)} \frac{}{\Theta \triangleright \Gamma \vdash t = t : c} \quad \text{(Tra)} \frac{\Theta \triangleright \Gamma \vdash s = t \quad t = u : c}{\Theta \triangleright \Gamma \vdash s = u : c}
\end{array}$$

Fig. 3. Second-order equational logic

EXAMPLE 2.3. The simply typed λ -terms on base types Ty are modelled in our setting as follows. Suppose that Arr is a type constructor. The set $\mathcal{B}$ of all mol types is the least set satisfying $\mathcal{B} = \text{Ty} \cup \{\text{Arr}(a, b) \mid a, b \in \mathcal{B}\}$, i.e. the set of all simple types in our encoding. The λ -terms are given by a signature

$$\Sigma_{\text{lam}} = \left\{ \begin{array}{l} \text{lam}_{a,b} : (a \rightarrow b) \rightarrow \text{Arr}(a, b) \\ \text{app}_{a,b} : \text{Arr}(a, b), a \rightarrow b \end{array} \mid a, b \in \mathcal{B} \right\}$$

As this example, a signature is very often presented by function symbols parameterised by several molecular types. We often present a signature simply schematically by

$$\left\{ \begin{array}{l} \text{lam} : (a \rightarrow b) \rightarrow \text{Arr}(a, b) \\ \text{app} : \text{Arr}(a, b), a \rightarrow b \end{array} \right\}$$

to mean the signature Σ_{lam} . Moreover, simply writing function symbols lam and app , we mean $\text{lam}_{a,b}$ and $\text{app}_{a,b}$ in Σ_{lam} having appropriate mol type subscripts a, b .

Note that $\text{Arr}(a, b)$ is a mol type, but a function type $a \rightarrow b$ is not a mol type. Hereafter, we use the above convention for a presentation of arbitrary signature parameterised by mol types, also in SOL.

A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g., $b_1, \dots, b_m \rightarrow c$, or $() \rightarrow c$. The latter case is simply denoted by c . A *metavariable* is a variable of (at most) first-order function type, declared as $M : \overline{a} \rightarrow b$ (written as capital letters $M, N, K, \dots$). A *variable* of a molecular type is merely called variable (written usually $x, y, \dots$, or sometimes written x^b when it is of type b). The raw syntax is given as follows.

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

The last form is called a *meta-application*, meaning that when we instantiate $M : \overline{a} \rightarrow b$ with a term s , free variables of s (which are of types $\overline{a}$) are replaced with (meta)-terms $t_1, \dots, t_n$ (cf. Def. 2.4). We may write $x_1, \dots, x_n. t$ for $x_1 \dots x_n. t$, and we assume ordinary α -equivalence for bound variables. A second-order equational theory is a set of proved equations deduced from a set of axioms.

$$\begin{array}{c}
\text{(Rule)} \frac{(\Theta \triangleright \Gamma \vdash \ell \Rightarrow r : c) \in C}{\Theta \triangleright \Gamma \vdash \ell \Rightarrow_C r : c} \\
\\
\text{(Fun)} \frac{\begin{array}{c} f : (\overline{a_1} \rightarrow b_1), \dots, (\overline{a_k} \rightarrow b_k) \rightarrow c \in \Sigma \\ \Theta \triangleright \Gamma, \overline{x_i} : \overline{a_i} \vdash t_i \Rightarrow_C t'_i : b_i \quad (\text{some } i \text{ s.t. } 1 \leq i \leq k) \end{array}}{\Theta \triangleright \Gamma \vdash f(\overline{x_1^{a_1}}.t_1, \dots, \overline{x_i^{a_i}}.t_i, \dots, \overline{x_1^{a_1}}.t_k) \Rightarrow_C f(\overline{x_1^{a_1}}.t_1, \dots, \overline{x_i^{a_i}}.t'_i, \dots, \overline{x_1^{a_1}}.t_k) : c} \\
\\
\text{(Sub)} \frac{\begin{array}{c} \Theta \triangleright \Gamma', \overline{x_i} : \overline{a_i} \vdash s_i : b_i \quad (1 \leq i \leq k) \\ M_1 : (\overline{a_1} \rightarrow b_1), \dots, M_k : (\overline{a_k} \rightarrow b_k) \triangleright \Gamma \vdash t_1 \Rightarrow_C t_2 : c \end{array}}{\Theta \triangleright \Gamma, \Gamma' \vdash t_1 [\overline{M \mapsto \overline{x}.s}] \Rightarrow_C t_2 [\overline{M \mapsto \overline{x}.s}] : c}
\end{array}$$

Fig. 4. Second-order computation (one-step)

A metavariable context Θ is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:mol type)-pairs. A judgment is of the form $\Theta \triangleright \Gamma \vdash t : b$. A meta-term t is *well-typed* by the typing rules Fig. 2. Note that a raw meta-term of the form $x.t$ does not have a type. But we will use raw terms $x.t$ in various places, such as in substitution below.

2.2 Equational logic and computation

Given a meta-term t with free variables $x_1, \dots, x_n$, the notation $t \{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1, \dots, s_n$.

DEFINITION 2.4. (Substitution of terms for metavariables) Let $\Gamma = y_1 : b_1 \cdots, y_k : b_k$. Let $n_i = |\overline{a_i}|$ and $\overline{a_i} = a_i^1, \dots, a_i^{n_i}$. Suppose

$$\begin{array}{c}
\Theta \triangleright \Gamma', x_i^1 : a_i^1, \dots, x_i^{n_i} : a_i^{n_i} \vdash s_i : b_i \quad (1 \leq i \leq k) \\
\Theta, M_1 : \overline{a_1} \rightarrow b_1, \dots, M_k : \overline{a_k} \rightarrow b_k \triangleright \Gamma \vdash e : c
\end{array}$$

Then substitution of terms for metavariables $\Theta \triangleright \Gamma, \Gamma' \vdash e [\overline{M \mapsto \overline{x}.s}] : c$ is defined as:

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

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

2.2.1 Equations. 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, then write simply $s = t$. The second-order equational logic is a logic to deduce formally proved equations from a given set $\mathcal{E}$ of equations, regarded as *axioms*. The inference system of equational logic is given in Fig. 3.

2.2.2 Computation rules. For meta-terms $\Theta \triangleright \Gamma \vdash \ell : b$ and $\Theta \triangleright \Gamma \vdash r : b$, a *computation rule* is of the form $\Theta \triangleright \Gamma \vdash \ell \Rightarrow r : b$ satisfying: (i) ℓ is a deterministic second-order pattern. (ii) all metavariables in r appear in ℓ . The definition of the condition (i) will be given in §6.2. We usually omit the context and type and simply write $\ell \Rightarrow r$.

We call a set C of rules a *computation system*. We write $s \Rightarrow_C t$ to be one-step computation using C obtained by the inference system given in Fig. 4. Clearly, it is a subsystem of the second-order equational logic (Fig. 3), deleting (Ref)(Tra)(Ax2). We also regard $\Rightarrow_C$ to be a binary relation on meta-terms.

3 PROBLEM PART I: A THEORY OF EFFECT

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 and Power 2002; Staton 2009] for the single location. The signature

```
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:⁵

```
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 look-up the state, bind the value to v , then update the state to v , is equivalent to do nothing. The axiom (ul) says that update the state to V , then look-up and continue X with the look-uped value, is equivalent to update the state to V and continue with V . Plotkin and Power showed that the monad corresponding to the theory of global state (of the finitely many locations) is the state monad [Plotkin and 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. [Melliès 2010; Staton 2009]). We show the decidability of the second-order equational theory $gstate$ using SOL.

3.1.1 SN. First we show SN of $gstate$, regarded as left-to-right computation rules.

```
*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..
```

But SOL cannot prove it. This is because the General Schema checks that a recursive call at rhs must be with a subterm of an argument at lhs. In case of (ll), recursive call of lk happens with $v.X[v, v]$, which is not a subterm of $w.lk(v.X[v, w])$ at lhs. This seems harmless, because $v.X[v, v]$ looks “almost” a subterm of $lk(v.X[v, w])$. But there is another difficulty in (ul), where ud is called with $X[V]$, which is not a subterm of $lk(w.X[w])$. This is more serious, because $X[V]$ is an application of a term V to X , which is similar to the β -axiom of λ -calculus $\lambda(x.M[x])@N \Rightarrow M[N]$, where the rhs has an application $M[N]$. Hence it has the same level of difficulty to show SN of λ -calculus. Therefore we cannot hope for the existence of a simple “weight” (§1.2.4) to show termination of $gstate$. This problem can be solved by the higher-order semantic labelling method [Hamana 2007, 2010]. We label the rules to give more “hints” for a termination proof. Any second-order computation system have the term model MT with the relation $\Rightarrow^*$ [Hamana 2005]. It interprets every function symbol as an operation on terms as: $lk^{MT}(t) = lk(t)$, $ud^{MT}(u, t) = ud(u, t)$. Now, we choose the natural numbers $\mathbb{N}$ to be *labels* attached to function symbols. The *label constructions* are functions chosen (in this case) as counting the numbers of symbols in arguments

$$\langle\langle t \rangle\rangle_{lk} \triangleq \text{the total number of } lk \text{ and } ud \text{ in } t, \quad \langle\langle u, t \rangle\rangle_{ud} \triangleq \text{the total number of } lk \text{ and } ud \text{ in } t$$

where the former will be used for the label of lk , and the latter is for ud . Then we attach labels to all the instances of the rules using these. For example, (uu) is transformed to labelled rules

⁵In SOL, `[axiom|..|]`, `[rule|..|]` are treated as the same data. The distinction is only for representing the user’s intention.

$$(uu) \quad ud_{\langle\langle v, ud(w, t) \rangle\rangle_{ud}}(v, ud_{\langle\langle w, t \rangle\rangle_{ud}}(w, t)) \Rightarrow ud_{\langle\langle w, t \rangle\rangle_{ud}}(w, t)$$

by using assignments $V \mapsto v$, $W \mapsto w$, $X \mapsto t$. Since instances are infinitely many, the resulting labelled rule set is infinite, but there is a finite description. Suppose the number of ud and lk in t is n . Then we have

$$(uu) \quad ud_{n+1}(v, ud_n(w, t)) \Rightarrow ud_n(w, t)$$

Similarly, by using assignments $V \mapsto v$, $W \mapsto w$, $X \mapsto x.t$ (t may involve x), we have

$$(u1) \quad ud_{n+1}(v, lk_n(w.t\{x \mapsto w\})) \Rightarrow ud_n(v, t\{x \mapsto v\})$$

Since the first argument of ud is of type Val of values, $t\{v\}$ does not increase the number of ud and lk . A key fact is that if we can show the labelled rules to be terminating, then we ensure that *the original rules are terminating*. This is the higher-order semantic labelling method [Hamana 2007]. Intuitive reason is that since the label index just indicates “an invariant” (i.e. the number of ud and lk), the labelling does not essentially change the computation behavior of the rules, thus termination is preserved. Hence, we try to show SN of the labelled rules using SOL. We formulate the labelled rules

```
sigstL = [signature|
  lk-n:(Val -> E) -> E; lk-n+1:(Val -> E) -> E; ud-n:Val, E -> E; ud-n+1:Val, E -> E |]
gstateL = [rule|
  (lu-n)  lk-n(v.ud-n(v, X))          => X
  (ll-n)  lk-n+1(w.lk-n(v.X[v, w])) => lk-n(v.X[v, v])
  (uu-n)  ud-n(V, ud-n(W, X))         => ud-n(W, X)
  (u1-n)  ud-n+1(V, lk-n(w.X[w]))    => ud-n(V, X[V]) |]
```

Then the command `sn gstateL sigstL` reports SN. Hence we conclude the original axioms `gstate` is SN.

3.1.2 CR. Next consider CR of `gstate`. SOL reports 3 non-joinable critical pairs out of 8.

```
*SOL> cri gstate
1: Overlap (lu)-(uu)--- V' |-> v, X |-> ud(W', X') -----
  L: lk(v.ud(v, X)) => X
  R: ud(V', ud(W', X')) => ud(W', X')
      lk(v2.ud(v2.ud(W', X'))
      ud(W', X') <-(lu)-^-(uu)-> lk(v2.ud(W', X'))
      ----> ud(W', X') =# lk(v2.ud(W', X')) <----

2: Overlap (lu)-(u1)--- V' |-> v, X |-> lk(w.X'[w]) -----
  L: lk(v.ud(v, X)) => X
  R: ud(V', lk(w.X'[w])) => ud(V', X'[V'])
      lk(v6.ud(v6.lk(w.X'[w]))
      lk(w.X'[w]) <-(lu)-^-(uu)-> lk(v6.ud(v6.X'[v6]))
      ----> lk(w.X'[w]) =# lk(v6.ud(v6.X'[v6])) <----

3: Overlap (ll)-(lu)--- X |-> z1.z2.ud(z1, X') -----
  L: lk(w.lk(v.X[v, w])) => lk(v.X[v, v])
  R: lk(v.ud(v, X')) => X'
      lk(w.lk(v.ud(v, X'))
      lk(v1.ud(v1, X')) <-(ll)-^-(lu)-> lk(w.X')
      ----> X' =# lk(w.X') <----

..
8: Overlap (uu)-(ll)--- X |-> z1.lk(v.H1[v]), X' |-> z1.z2.H1[z1] -----
  L: ud(V, lk(w.X[w])) => ud(V, X[V])
  R: lk(w.lk(v.X'[v, w])) => lk(v.X'[v, v])
      ud(V, lk(w.lk(v.H1[v]))
      ud(V, lk(v.H1[v])) <-(uu)-^-(ll)-> ud(V, lk(v8.H1[v8]))
      ----> ud(V, H1[V]) =OK= ud(V, H1[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 $s \equiv t$ 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,

$$\begin{aligned} \text{lk}(v1.\text{ud}(W', X')) &= \text{ud}(W', X') \\ \text{lk}(v2.\text{ud}(v2, X'[v2])) &= \text{lk}(w'.X'[w']) \\ \text{lk}(w.X') &= X' \end{aligned}$$

we see that these are naturally understandable theorems. The first says that look-up the state, then update it by a value W' is equivalent to just update the state by W' . The second says that look-up the state, then update it by the looked-up value, and continue with the value, is equivalent to look-up the state and continue 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'])
                  (1)    lk(v.X')              => X'                |]
```

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)
...
#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, we need to show SN of $(\text{gstate} ++ \text{gstateEx})$. Again, it does not follow the General Schema, but the same labelling used for gstate works, and we finally have SN. Hence, we conclude that the theory of global state is decidable, and the SN and CR computation system $(\text{gstate} ++ \text{gstateEx})$ provides a decision procedure.

4 PROBLEM PART II: VARIATIONS ON THE λ -CALCULUS

This section discusses 5 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 #3] — molecular HOAS

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

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

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

SOL automatically translates the infix operator $@$ to the prefix function symbol app . We defined the rule straightforwardly, but our choice of the signature should be commented. We chose a particular representation of types, which is different from the ordinary higher-order abstract syntax (HOAS) [Pfenning and Elliott 1988]. We strictly distinguish a molecular type $\text{Arr}(a, b)$ representing arrow type in the “object level” simply-typed λ -calculus from a function type $a \rightarrow b$ of the “meta-level” second-order algebraic theory. In second-order algebraic theory with molecular types, a function type $a \rightarrow b$, where a and b are molecular types, always represents “variable binding”. While written $a \rightarrow b$ in our notation, it is semantically corresponds to a presheaf $\delta_a b \cong V_a \Rightarrow b$, roughly

considered as a function type from the type of “variables” [Fiore 2002; Fiore and Hur 2010]. 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 in simple types. The type of lam

$$\text{lam} : (a \rightarrow b) \rightarrow \text{Arr}(a, b)$$

does not involve a negative occurrence of the target type $\text{Arr}(a, b)$ for any mol types a, b (see Def. 6.14, 6.15). Even in the case $a = b$, we have $\text{lam} : (a \rightarrow a) \rightarrow \text{Arr}(a, a)$, where the molecular type $\text{Arr}(a, a)$ cannot appear negatively in $(a \rightarrow a)$ because of $a \neq \text{Arr}(a, a)$. We call this encoding *molecular HOAS* because of the use of molecular types.

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 cbn 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! -- on this signature
```

where SOL automatically recognizes 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.

REMARK 4.1. If one badly chose types for the signature (with the exactly the same rule cbn), the General Schema may not pass. For example, the ordinary HOAS encoding $\text{lam} : (a \rightarrow a) \rightarrow a$ is not positive, and is regarded as the representation of *untyped* λ -calculus. Hence cbn under this “untyped” signature is actually non-terminating, and the General Schema wisely rejects it. (Try `sn cbn sigunLam` in SOL using the above untyped λ signature).

CR of the call-by-name λ is now clear, because the rule cbn does not have a critical pair, hence WCR. With SN, we have CR. Hence the theory of simply-typed λ -calculus is decidable.

4.2 The call-by-value λ -calculus [Problem #4]

— 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 and 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 $V ::= y \mid \lambda w.M$

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 tell it to SOL by the following meta-programming on rules using the benefit of Haskell DSL. We can easily define the expansion function `expandV` that replaces V with a variable and an abstraction (using the internal data structures of SOL) in Haskell, then 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 $\text{lam}(x.M[x])@y \Rightarrow M[y]$, 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 8 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 #5] — difficulty on non-positively typed commuting conversion

We consider Moggi’s computational meta-language [Moggi 1988] in the calculus form [Sabry and 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 $\text{let } x = s \text{ in } t$ is represented by $\text{let}(s, x. t)$ using $\text{let}:T(a), (a \rightarrow T(b)) \rightarrow T(b)$, which is more concise and is nothing but “bind”. Hence, the relationship between this and the laws of monad considered in §1.3.3 is now evident, namely lamML is the combination of the λ -calculus with monad.

4.3.1 Partial result. Let’s first see SOL’s output on SN.

```

*SOL> sn lamML sigML
...
Checking (assoc) let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
(fun let=let) subterm comparison of args
(meta L)[is acc in let(L,x.M[x]),y.N[y]] [is positive in let(L,x.M[x])] [is acc in L]
(fun let=let) subterm comparison of args
(meta M)[is acc in let(L,x.M[x]),y.N[y]] [is positive in let(L,x.M[x])] [is acc in M[x]]
(meta N)[is acc in let(L,x.M[x]),y.N[y]] [is positive in let(L,x.M[x])] [is acc in N[y]] >>True
#SN! -- on this signature

```

However, we should regard that this shows a partial result on termination. The message “-- on this signature” designates the signature sigML, in particular, the declaration $\text{let}:T(a), (a \rightarrow T(b)) \rightarrow T(b)$. (assoc) is terminating for this let, but not sure for the case that a is taken to be $T(b)$ as $\text{let} : T(T(b)), (T(b) \rightarrow T(b)) \rightarrow T(b)$. The current SOL does not automatically vary a in checking SN. This case involves the negative occurrence of $T(b)$ in the type $(T(b) \rightarrow T(b))$ (see Def. 6.14, 6.15). This is problematic. Consider the modified signature using this:

```
sigML' = [signature|
  app : Arr(a,b),a -> b      ;   let   : T(T(b)),(T(b) -> T(b)) -> T(b)
  lam : (a -> b) -> Arr(a,b) ;   return : a -> T(a)                                ]]
```

Then, SOL reports a problem:

```
*SOL> sn lamML sigML'
Found constructors: return
Checking positivity of constructors >>OK
...
Checking (assoc) let(let(L,x.M[x]),y.N[y]) => let(L,x.let(M[x],y.N[y]))
...
(fun let=let) subterm comparison of args
(meta M)[is acc in let(L,x.M[x]),y.N[y]] [is positive in let(L,x.M[x])] >>False
#No idea..
```

In checking (assoc), the General Schema checks the “accessibility” of the metavariable M in the clause (meta M) (Def. 6.16). It requires that the type $T(b)$ must occur at a positive position of the types of the arguments of let (see (a3) in Def. 6.16), but the type $T(b)$ occurs negatively in $(T(b) \rightarrow T(b))$, hence the General Schema does not pass. This is a limitation of the General Schema criterion, prohibiting a potential danger, which is correct, but this sensitive case is out of the scope.

It seems that this is a known difficulty of showing SN of *commuting conversion*. Lindley and Stark overcome this difficulty by defining reducibility on computation types using the method of $\tau\tau$ -lifting, and showed SN of the computational meta-language λ_{ml} as well as the computational λ -calculus λ_c [Lindley and Stark 2005]. SN of λ_{ml} has also been proved in [Benton and Hyland 2003]. Hence anyway, we can conclude that the full λ_{ml} is SN due to these results.

In summary, SOL shows SN of a sublanguage of λ_{ml} in which disallowing the let of the type $\text{let}: T(T(b)), (T(b) \rightarrow T(b)) \rightarrow T(b)$. Nevertheless, such a sublanguage is meaningful. As noted in [Lindley and Stark 2005, §2], often one does not require the full power of λ_{ml} , allows only pure and Kleisli $b \rightarrow T(b)$ functions, and disallows *meta-computations* of the forms $T(T(b))$ and $T(b) \rightarrow T(b)$.

REMARK 4.2. This non-positivity problem does not happen in other function symbols such as app and lam , and any other examples in this paper other than the monadic let and $\gg$.

4.3.2 CR. SOL automatically shows WCR of lamML by enumerating 7 CPs and checking their joinability. Combining SN of λ_{ml} [Benton and Hyland 2003; Lindley and Stark 2005], this proves CR of the full λ_{ml} . The result on CR of λ_{ml} was stated in [Sabry and Wadler 1997, Prop. 4.2] but without proof or any reference. We will touch this issue in Remark 4.3.

4.4 The simplified monadic calculus λ_{ml*} [Problem #6] — clarification of CR

We consider a variation called the simplified monadic calculus λ_{ml*} in [Sabry and 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 $\text{cbv } \lambda$, 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. SOL also reports SN of lamMLstar , but the caution of partial result of SN on λ_{ml} is also applied to this case. Full SN has been shown in [Lindley and Stark 2005], hence combining it with SOL’s WCR result, we have CR, hence decidable.

REMARK 4.3. In [Sabry and 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 and 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 Moggi [1988]” [Sabry and Wadler 1997, Prop.5.1]. But as far as the present author examined Moggi’s LFCS technical report [Moggi 1988], Moggi did not give a proof of confluence.

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

4.5 Hasegawa’s yet simpler linear λ -calculus [Problem #7]

— necessity of deterministic second-order patterns and FCU unification algorithm

Hasegawa proposed a linear λ -calculus in [Hasegawa 2005]. It has the !-type constructor that corresponds to a modality in linear logic. We represent it as a type constructor Bang. The formalisation of the syntax and typing in [Ohta and 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 criticized in [Ohta and Hasegawa 2006, §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
3: Overlap (eta!)-(beta-o)--- L|-> lam(x'.M'[x']), N'|-> bang(x) -----
L: let(M,x.L@bang(x)) => L@M
R: lam(x'.M'[x'])@N' => M'[N']
      let(M,x2.lam(x'.M'[x'])@bang(x2))
lam(x'.M'[x'])@M <-(eta!)-^-(beta-o)-> let(M,x2.M'[bang(x2)])
      ----> M'[M] =# let(M,x2.M'[bang(x2)]) <---
#NON 1 joinable... (Total 4 CPs)
```

Although SOL does not have linear/intuitionistic distinction of contexts, it does not affect to confluence problem as long as considering well-typed terms. SOL reports 4 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 has 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)
6: Overlap (beta!)-(eta'!)--- M'|-> bang(M), %FLEX3|-> x.N[x]<->x.C'[bang(x)] -----
L: let(bang(M),x.N[x]) => N[M]
R: 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)
```

SOL reports 6 CPs and finally all are joinable. Hence we conclude that lamHas is WCR. The above overlap is particularly interesting. SOL reports an overlap between $(\text{beta}!)$ and $(\text{eta}'!)$ using a unifier involving a flex-flex pair $x.N[x] \leftarrow x.C'[bang(x)]$ meaning that this unification problem is frozen and should be regarded as an equality constraint (§6.3.2). Hence, the middle term $\text{let}(bang(M), x.C'[bang(x)])$ of the critical pair can be rewritten by $(\text{beta}!)$ under the constraint $x.N[x] = x.C'[bang(x)]$ at the left path of divergence. The flex-flex pair constraint is crucial here. Why does the flex-flex solution happen in unification? This is due to the fact that the lhs of $(\text{eta}'!)$

$\text{let}(M, x.C[bang(x)])$

is *beyond* the class of Miller's higher-order patterns, but is within the class of Yokoyama et al.'s *deterministic second-order pattern* [Yokoyama et al. 2003, 2004a], i.e. the metavariable C takes a constructor term $bang(x)$ (cf. §6.2).

What about SN? Unfortunately, SOL cannot prove SN of the calculus lamHas . This is due to the fact that $(\text{eta}'!)$'s lhs involves a deterministic second-order pattern. The General Schema cannot cope with a term $C[bang(x)]$ for termination check. Attempting to extend the General Schema to cover second-order deterministic patterns is under investigation.

REMARK 4.4. SOL's mechanical check found also a few missing critical pairs for a finer linear λ -calculus in [Ohta and Hasegawa 2006], which were not described in Ohta-Hasegawa's analysis. It seems these were overlooked.

5 PROBLEM PART III: A THEORY OF π -CALCULUS [PROBLEM #8]

As the final example, we consider a fairly large problem, a theory of π -calculus given by Stark [Stark 2008]. The π -calculus of Milner is one of the most fundamental concurrent calculi [Milner 1999]. Stark gave a free algebra model of π -calculus [Stark 2008] and Staton examined algebraic and categorical properties of this algebraic theory [Staton 2009, 2013b]. 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 show the decidability of it with assistance of SOL.

A theory of π consists of 12 axioms. We use Huet's idea of partitioning axioms into rules and equations (§6.6). A reason of doing this is that commutativity axioms ((sum-com) , (new-com) below) cannot not be oriented. The signature and the partitioned axioms are written 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[b,a]))                []

```

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

- In (new-uni) , X cannot contain the variable a (since it is not written as $X[a]$).
- The rhss of $(\text{newout}0)$, (newout) seem overlapped. However, they are not, because B in $\text{new}(a.\text{out}(B, C, X[a]))$ cannot contain a (since it is not written as $B[a]$), hence cannot unify with $\text{new}(a.\text{out}(a, B, X[a]))$.

5.1 Proof with SOL

5.1.1 UCR~. We check that the computation rules `pical` has uniformly Churh-Rosser modulo the equational theory (§6.6) generated by `pieq` by the command:

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

then SOL reports 53 CPs. This large number of CPs shows that it is hard for human to enumerate all the critical pairs manually without overlooking. SOL also tells that the following three CPs are non-joinable ((sum-ascr) , (sum-comr) are the reversed equations):

```
Overlap (sum-ascr)-(sum-uni)  Overlap (sum-com)-(sum-uni)  Overlap (sum-comr)-(sum-uni)
sum(sum(X,nil),Z) == sum(X,Z)  sum(Y,nil) == Y          sum(X,nil) == X
```

We can find that clearly a missing computation rule to make these joinable is

```
pieq = [rule| (sum-uniR)  sum(X,nil()) => X |]
```

We check again by putting “`crimod (pieq++pical) pieq`”, SOL reports 67 CPs, and all are joinable modulo `pieq`.

5.1.2 Decidability. By the command “`sn (pieq ++ pical) pisig`”, SOL shows SN without trouble. The equational theory generated by `pieq` is decidable, because the $(\text{sum-}*)$ 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. Therefore, SN and CR modulo `pieq` properties of `pical` provide a decidable proof method of the whole equational theory (`pical ++ pieq`) (§6.6.1). Hence, we conclude that this algebraic theory of π -calculus is decidable.

5.1.3 Note on two function spaces. In algebraic formulation of π -calculus, it has been understood that two kinds of function spaces are needed [Fiore et al. 1996; Stark 1996], one $[N \rightarrow A]$ 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 $[N \rightarrow A]$ 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 FOUNDATIONS

In this section, we present the technical foundations of our computation systems and SOL. We explain known basic results on second-order matching (§6.1, §6.2), unification (§6.3), rewriting (§6.4) and our new developments on critical pairs and CR modulo second-order equational theory (§6.6, §6.7), the General Schema for termination criterion (§6.8), and what are the connections between them.

6.1 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 for metavariables (Def. 2.4).

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 for rewrite (NB. s' is a candidate of *reducible expression*, “*redex*”).
2. Select a computation rule $\ell \Rightarrow r$ from C .
3. Try *second-order matching* between ℓ and s' .
If it succeeds, get a matcher θ such that $\ell \theta = s'$ holds.
Otherwise, go to 2 (or give up if there is no other possible rule to match).
4. Replace s' in s with an 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 3 of the phase of second-order matching. In general, second-order matching is expensive (the known algorithm 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 for which there exists a single most general matcher. A well-known idea to recover these benefits in the higher-order case [Miller 1991; Nipkow 1991] is to restrict the left-hand sides ℓ of computation rules to be Miller’s *higher-order patterns* [Miller 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 the form

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

where $x_1, \dots, x_n$ are distinct bound variables. Thus meta-terms such as $M[N]$, $M[\text{cons}(x, y)]$ 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 §4.5.

6.2 Deterministic second-order patterns

Yokoyama et al. [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 (§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 $s \neq t$.

A *deterministic second-order pattern* is a meta-term p in which every occurrence of meta-application $M[t_1, \dots, t_n]$, 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) $t_i \not\leq t_j$ for every $1 \leq i, j \leq n$.

For example, $M[\text{cons}(x, y)]$ is a deterministic second-order pattern. Yokoyama et al.'s deterministic second-order patterns extends the second-order fragment of Miller's higher-order patterns, because "metavariables with distinct bound variables" are ensured by (ii)(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. Below we use the letter p to denote a deterministic second-order pattern.

THEOREM 6.1. ([Yokoyama et al. 2003, 2004a]) *Any deterministic second-order pattern matching problem $p \stackrel{?}{=} t$ 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. [Yokoyama et al. 2004b, Sec. 2] have shown such an example. The unification problem between deterministic second-order patterns

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

has at least 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.

6.3 Modified FCU unification

6.3.1 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 and 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 and Miller 2016, Def. 14]).

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

COROLLARY 6.2. *Second-order FCU unification problem is decidable and ensures the existence of a most general unifier if solvable.*

The higher-order FCU unification algorithm has been given in [Libal and Miller 2016], which is terminating and returns a most general unifier. It is a basis of our implementation of unification in SOL.

6.3.2 Step 2: Freezing a flex-flex pair for a unification problem violating non-global restriction. But this is still not enough. When examining the example of a linear λ -calculus (§4.5), we need to check overlapping between the following two rules

$$\begin{aligned} \text{let}(\text{bang}(M), x. N[x]) & \Rightarrow N[M] \\ \text{let}(M1, x. L[\text{bang}(x)]) & \Rightarrow L[M1] \end{aligned}$$

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

Modified FCU algorithm

- When the algorithm encounters a problem $M[s_1, \dots, s_n] \stackrel{?}{=} N[t_1, \dots, t_m]$ which violates the **global restriction** and one side of the equation is a linear Miller pattern, then stops and returns the flex-flex pair $(M[s_1, \dots, s_n], N[t_1, \dots, t_m])$ as the result. Otherwise, fail.
 - Other form of unification problem is handled by the original algorithm.
-

“Flex-flex” [Huet 1975] means the the heads of the pair are “flexible” (i.e. metavariables). A flex-flex pair $(M[s_1, \dots, s_n], N[t_1, \dots, t_m])$ is regarded as a constraint, as in higher-order pre-unification [Huet 1975]. The constraint actually suffices for computing overlaps between rules. A frozen flex-flex pair only has finite numbers of unifiers due to the following result:

PROPOSITION 6.3. ([Prehofer 1995] Thm. 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.*

Hence the modified algorithm is correct, and outputs a single solution if succeeds, where a flex-flex pair is regarded as a representative of a finite minimal complete set of unifiers.

6.4 Properties of abstract rewriting

This subsection reviews classical results on abstract rewriting [Baader and Nipkow 1998; Huet 1980]. 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 $(A, \rightarrow)$ of a set A and a binary relation $\rightarrow$ on A . We write $\rightarrow^*$ for the reflexive transitive closure, $\rightarrow^+$ for the transitive closure, and $\leftarrow$ for the converse of $\rightarrow$. We define $\leftrightarrow \triangleq \rightarrow \cup \leftarrow$. We say:

1. $a, b \in A$ are *joinable*, written $a \downarrow b$, if $\exists c \in A. a \rightarrow^* c \ \& \ b \rightarrow^* c$.
2. $\rightarrow$ is *confluent* if $\forall a, b \in A. a \rightarrow^* b \ \& \ a \rightarrow^* c$ implies $b \downarrow c$.
3. $\rightarrow$ is *Church-Rosser* (CR) if $\forall a, b \in A. a \leftrightarrow^* b$ implies $a \downarrow b$.
4. $\rightarrow$ is *locally confluent* (WCR) if $\forall a, b \in A. a \rightarrow b \ \& \ a \rightarrow c$ implies $b \downarrow c$.
5. $\rightarrow$ is *strongly normalising* (SN) if $\forall a \in A$, there is no infinite sequence $a \rightarrow a_1 \rightarrow a_2 \rightarrow \dots$.
6. a is a *normal form* if there is no $b \in A$ such that $a \rightarrow b$.
7. We write $t \xrightarrow{!} a'$ if $a \rightarrow^* a'$ and a' is a normal form, meaning rewriting to a normal form.

We identify an ARS A with its relation $\rightarrow$ (e.g. we say A is CR to mean $\rightarrow$ 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 6.4. (**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.

6.5 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 and Bendix 1970].

Suppose a computation system 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 C as the ARS. We say two rules $l_1 \Rightarrow r_1, l_2 \Rightarrow r_2$ in C are *variant* if $l_1 \Rightarrow r_1$ is obtained by injectively renaming variables and metavariables of $l_2 \Rightarrow r_2$. We say that a position p in a term t is a *metavariable position* if $t|_p$ is a metavariable or meta-application.

DEFINITION 6.5. An *overlap* between two rules $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ of a computation system C is a tuple $\langle l_1 \Rightarrow r_1, p, l_2 \Rightarrow r_2, \theta, FL \rangle$ satisfying the following properties:

- $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ are variants of rules in C without common variables and metavariables.
- p is a non-metavariable position of l_1 .
- If p is the root position, $l_1 \Rightarrow r_1$ and $l_2 \Rightarrow r_2$ are not variants.
- FL is a set of flex-flex equations (possibly empty).
- θ is a unifier between $l_1|_p$ and l_2 modulo FL ,
i.e. $(l_1|_p)\theta = l_2\theta$ is derivable from FL using second-order equational logic.
(In case of $FL = \emptyset$, it is just α -equivalence).

This is a mathematical definition. Algorithmically, the components θ and FL in an overlap are expected to be outputs of the modified FCU unification algorithm. An overlap represents an overlapping situation of computation, meaning that a term t is rewritten to the two different ways as follows.

DEFINITION 6.6. The *critical pair (CP)* generated from an overlap $\langle l_1 \Rightarrow r_1, p, l_2 \Rightarrow r_2, \theta, FL \rangle$ is a triple $\langle r'_1, t, r'_2 \rangle$ where t is obtained from $l_1\theta$ by replacing every u_i in $l_1\theta$ with w_i for every $(u_i \stackrel{?}{=} w_i)$ in FL , and

- $t \Rightarrow_C r'_1$ which rewrites the root position of t using $l_1 \Rightarrow r_1$
- $t \Rightarrow_C r'_2$ which rewrites the position p of t using $l_2 \Rightarrow r_2$.

Ordinary Knuth-Bendix critical pairs lack the middle t , 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

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

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

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

Finally, we obtain all critical pairs of C by generating the critical pair of each overlap in O . We say that a critical pair $\langle r'_1, t, r'_2 \rangle$ is *joinable* if $r'_1 \downarrow r'_2$. It is depicted as the right figure. The following is an important theorem that extends the first-order case.

THEOREM 6.7. *A computation system C is WCR if all its critical pairs are joinable.*

PROOF. (sketch) We show “if $s \Leftarrow_C u \Rightarrow_C t$ then $s \downarrow t$ ” by induction on the proof of $s \Leftarrow_C u$ using Fig. 4. The proof of Thm. 6.13, which will be established later on, uses the same strategy. $\square$

COROLLARY 6.8. *If a computation system C is SN, checking “ C is CR or not” is decidable.*

PROOF. All constructions until 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. $\square$

REMARK 6.9. Nipkow has considered critical pairs for higher-order rewrite rules on the simply-typed λ -calculus [Nipkow 1991],[Mayr and Nipkow 1998]. The development in this section is similar, but there are differences. Our computation system has distinction between variables and metavariables, cover 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 $\beta\eta$ -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 §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 (η !) rule needed in the linear λ -calculus (§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 and Nipkow 1998].

6.6 Uniformly 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 §5. This and next subsections explain the background theories used there (i.e. SOL’s `crimod` command).

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

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

1. $a, b \in A$ are *uniformly joinable modulo $\sim$* if $\exists c, c' \in A. a \xrightarrow{!} c \ \& \ b \xrightarrow{!} c' \ \& \ c \sim c'$, which is denoted by $a \downarrow_{\sim}^! b$,
2. $\rightarrow$ is *uniformly Church-Rosser modulo $\sim$* (UCR $\sim$) [Jouannaud et al. 1983] if for all $a, b \in A$, $a (\sim \cup \leftrightarrow)^* b$ implies $a \downarrow_{\sim}^! b$.

6.6.1 A decidable proof method of “partitioned” algebraic theory. We explain the reason why UCR $\sim$ is useful below. Suppose that an ARS satisfies SN and UCR $\sim$, and $\sim$ is decidable. Now UCR $\sim$ 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

$$s \xrightarrow{!} s_0 \sim t_0 \xleftarrow{!} t \quad (1)$$

which means that first rewriting terms s, t to the normal forms 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 criteria to check UCR $\sim$. For UCR $\sim$, the following (perhaps less known) theorem is particularly useful,

more so than the Huet's often used criteria [Huet 1980] (for instance in [Mayr and Nipkow 1998; Ohta and Hasegawa 2006]), because local check for “one-step” equality $\vdash$ (rather than transitive “many-step” $\sim$) suffices. Below, “ $\circ$ ” denotes the composition of relations.

THEOREM 6.10. [Jouannaud et al. 1983, Prop. 1,2] *Let $\sim$ be an equivalence relation and $\rightarrow$ a binary relation on the same set. Suppose $\vdash$ is a symmetric relation such that $\vdash^* = \sim$. If $\rightarrow$ is SN and*

- (i) $s \leftarrow \circ \rightarrow t$ implies $s \downarrow_{\sim}^! t$ (called $\rightarrow$ is uniformly locally confluent modulo $\sim$), and
- (ii) $s \vdash \circ \rightarrow t$ implies $s \downarrow_{\sim}^! t$ (called $\rightarrow$ is uniformly locally coherent modulo $\sim$)

then $\rightarrow$ is uniformly Church-Rosser modulo $\sim$.

6.7 Critical pairs between second-order computation rules and equations

A *partitioned computation system* $(C, \mathcal{E})$ consists of a computation system C and a set of equational axioms $\mathcal{E}$ satisfying for each axiom $s = t$, the set of all meta-variables of s and that of t are exactly the same. For a partitioned computation system $(C, \mathcal{E})$, we extend the notion of critical pairs to the one between C and $\mathcal{E}$ to check UCR $\sim$, where $\sim$ is the equivalence relation on meta-terms generated by $\mathcal{E}$. We establish an effective method to show UCR $\sim$.

EXAMPLE 6.11. In the case of a theory of π -calculus in §5, $C = \text{pical}$, $\mathcal{E} = \text{pieq}$ and $s (\sim \cup \leftrightarrow)^* t$ means that $s = t$ is derivable from the original “unpartitioned” axioms. If we have UCR $\sim$ of the partitioned algebraic theory $(C, \mathcal{E})$, then we can decide $s = t$ by the method (1).

DEFINITION 6.12. An *overlap between an equation $s = t \in \mathcal{E}$ 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 an equation as a bidirectional computation rule.

All the overlaps for $(C, \mathcal{E})$ are now defined by

$$O = \bigcup \left(\{ \text{overlap}(l_1 \Rightarrow r_1, l_2 \Rightarrow r_2) \mid l_1 \Rightarrow r_1, l_2 \Rightarrow r_2 \in C \} \cup \right. \\ \left. \{ \text{overlap}(s \Rightarrow t, l \Rightarrow r), \text{overlap}(t \Rightarrow s, l \Rightarrow r), \right. \\ \left. \text{overlap}(l \Rightarrow r, s \Rightarrow t), \text{overlap}(l \Rightarrow r, t \Rightarrow s) \mid s = t \in \mathcal{E}, l \Rightarrow r \in C \} \right)$$

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

THEOREM 6.13. *Let $(C, \mathcal{E})$ be a partitioned computation system, and assume C is left-linear and SN. The relation $\Rightarrow_C$ is uniformly Church-Rosser modulo $\sim$ if for every critical pair $\langle r, t, r' \rangle$ of $(C, \mathcal{E})$, $r \downarrow_{\sim}^! r'$.*

PROOF. (sketch) We apply the criterion Thm. 6.10 to deduce UCR $\sim$. We define $s \vdash t$ if $s = t$ is derived from $\mathcal{E}$ by the second-order equational logic without using (Ref) and (Tra) in Fig. 3. Namely, the symmetric relation $\vdash$ is the congruence closure of a one-step application of an instance of axiom of $\mathcal{E}$. Thus $\vdash^* = \sim$. Local confluence modulo $\sim$ follows from joinability modulo $\sim$ of all critical pairs of overlaps of C . To show local uniform coherence, we show “if $s \vdash u \Rightarrow_C t$ then $s \downarrow_{\sim}^! t$ ” by induction on the proof of $s \vdash u$. $\square$

Note that this theorem is the basis of SOL's crimod command used §5.

6.8 The General Schema

The General Schema is a criterion for proving strong normalisation of higher-order rules given in [Blanqui 2000] and improved in [Blanqui 2016]. In this subsection, we summarise the definitions and properties of the General Schema needed for our SOL system. For more details and the proofs,

see the original papers [Blanqui 2000, 2016]. The General Schema is formulated for a framework of higher-order type system with rewrite rules called inductive datatype system, whose second-order fragment is essentially the same as the present computation system given in §2. The General Schema has succeeded to prove 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 the right-hand side of a rewrite rule are "smaller" than the left-hand sides' ones. It is similar to Coquand's notion of "structurally smaller" [Coquand 1992], but more relaxed and extended.

A minor adaptation from the original presentation is about our use of molecular types (that correspond to "base types" in the original definition). Mol types are not decomposable here. In the following, b and c denote mol types.

We first define the notion of positions in a type and term. 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 $p.p' = q$.

DEFINITION 6.14. Given a type τ , the set $\text{Posi}(\tau)$ of *positive positions* and the set $\text{Nega}(\tau)$ of *negative positions* are defined by $\text{Posi}(b) = \{\varepsilon\}$, $\text{Nega}(b) = \emptyset$,

$$\text{Posi}(\bar{\tau}) = \{i.w \mid i = 1, \dots, |\bar{\tau}|, w \in \text{Posi}(\tau_i)\} \quad \text{Nega}(\bar{\tau}) = \{i.w \mid i = 1, \dots, |\bar{\tau}|, w \in \text{Nega}(\tau_i)\}$$

$$\text{Posi}(\bar{\sigma} \rightarrow \tau) = \text{Nega}(\bar{\sigma}) \cup \{0w \mid w \in \text{Posi}(\tau)\} \quad \text{Nega}(\bar{\sigma} \rightarrow \tau) = \text{Posi}(\bar{\sigma}) \cup \{0w \mid w \in \text{Nega}(\tau)\}$$

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

$$\mathcal{Pos}(b, b) = \{\varepsilon\} \quad \mathcal{Pos}(b, \bar{\sigma}) = \{i.w \mid i = 1, \dots, |\bar{\sigma}|, w \in \mathcal{Pos}(b, \sigma_i)\}$$

$$\mathcal{Pos}(b, c) = \emptyset \text{ if } b \neq c \quad \mathcal{Pos}(b, \bar{\sigma} \rightarrow \tau) = \mathcal{Pos}(b, \bar{\sigma}) \cup \{0w \mid w \in \mathcal{Pos}(b, \tau)\}.$$

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

DEFINITION 6.15. A *constructor* is a function symbol $c : \bar{\tau} \rightarrow b$ which does not occur at the root symbol of the left-hand side of any rule. The set of all constructors defines a preorder $\leq_{\mathcal{B}}$ on the set $\mathcal{B}$ of mol types by $b \leq_{\mathcal{B}} \bar{\tau}$ if b occurs in $\bar{\tau}$ for a constructor $c : \bar{\tau} \rightarrow b$. Let $<_{\mathcal{B}}$ be the strict part and $=_{\mathcal{B}}$ the equivalence relation generated by $\leq_{\mathcal{B}}$. A type b is *positive* if for each constructor $c : \bar{\sigma} \rightarrow b$ of it, $\mathcal{Pos}(b', \bar{\sigma}) \subseteq \text{Posi}(\bar{\sigma})$ for any type b' s.t. $b =_{\mathcal{B}} b'$. A constructor $c : \bar{\sigma} \rightarrow b$ is *positive* if b is positive.

DEFINITION 6.16. A metavariable M is *accessible* in a meta-term t if there are distinct bound variables $\bar{x}$ such that $M[\bar{x}] \in \text{Acc}(t)$, where $\text{Acc}(t)$ is the least set satisfying the following clauses:

- (a1) $t \in \text{Acc}(t)$.
- (a2) If $x.u \in \text{Acc}(t)$ then $u \in \text{Acc}(t)$.
- (a3) Suppose $f(u_1, \dots, u_n) \in \text{Acc}(t)$ with $f : \tau_1, \dots, \tau_n \rightarrow b$ (where b is a mol type). If $\mathcal{Pos}(b', \tau_i) \subseteq \text{Posi}(\tau_i)$ for any type b' s.t. $b =_{\mathcal{B}} b'$ and some $i = 1, \dots, n$, then $u_i \in \text{Acc}(t)$ ($1 \leq i \leq n$) [Blanqui 2016, Def.15].

DEFINITION 6.17. Given a meta-term t , $\mathcal{Pos}(t)$ denotes the set of all positions of t , $t|_p$ denotes a subterm of t at a position p , and $t[s]_p$ the term obtained from t by replacing the subterm at the position p by s . A meta-term u is a *covered-subterm* of t , written $t \hat{\supseteq} u$, if there are two positions $p \in \mathcal{Pos}(t)$, $q \in \mathcal{Pos}(t|_p)$ 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, $f(a, c(x)) \hat{\supseteq} c(x)$, and $\lambda(x.M[x]) \hat{\supseteq} M[x]$, and $y.\lambda(x.M[x]) \hat{\supseteq} y.x.M[x]$.

DEFINITION 6.18. 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 left-hand side is headed by f) in the right-hand side of which g occurs. Its transitive closure is denoted by $>_{\Sigma}$.

DEFINITION 6.19. Given $f : \tau_1, \dots, \tau_n \rightarrow \tau \in \Sigma$, the *computable closure* $CC_f(\bar{t})$ of a meta-term $f(\bar{t})$ is the least set CC satisfying the following clauses. All the meta-terms below are well-typed.

1. **(meta M)** If $M : \tau_1, \dots, \tau_p \rightarrow \tau$ is accessible in some of $\bar{t}$, and $\bar{u} \in CC$, then $M[\bar{u}] \in CC$.
2. For any variable x , $x \in CC$.
3. If $u \in CC$ then $x.u \in CC$.
4. **(fun $f > g$)** If $f >_{\Sigma} g$ and $\bar{w} \in CC$, then $g(\bar{w}) \in CC$.
5. **(fun $f = f$)** If $\bar{u} \in CC$ such that $\bar{t} \succ^{\text{lex}} \bar{u}$, then $f(\bar{u}) \in CC$, where $\succ^{\text{lex}}$ is the lexicographic extension of the strict part of $\widehat{\succeq}$.

Here, **(meta M)** etc. are used as labels for references in a termination proof using SOL (see e.g. §1.3.5). The lexicographic extension $\succ^{\text{lex}}$ can be left-to-right or right-to-left comparison according to f (so-called “status”).

DEFINITION 6.20. A rule $f(\bar{t}) \Rightarrow r$ *satisfies the General Schema* if $CC_f(\bar{t}) \ni r$.

THEOREM 6.21. *Suppose that given a signature Σ and rules C satisfy:*

- (i) $>_{\mathcal{B}}$ is well-founded,
- (ii) every constructor is positive, and
- (iii) $>_{\Sigma}$ 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 Thm. 6.21,

7 SUMMARY AND CONCLUDING REMARK

7.1 Summary

We have presented a general methodology of proving decidability of equational theory of programming language concepts in the framework of second-order algebraic theories. We proposed a Haskell-based analysis tool SOL, 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 have combined and extend both syntactical and semantical results of second-order computation in non-trivial manner. 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 8 examples of algebraic theories in the literature using SOL.

7.2 Limitation of SOL, the General Schema and prospect

From a variety of examples we have examined, we found the usefulness and limitations of SOL. The most desirable issue is to enhance the applicability of the General Schema, which now does not cope with non-positively typed commuting conversions and deterministic second-order patterns.

It would be interesting to develop a combination of $\top\top$ -lifting and the General Schema’s computability predicate, which may increase the power of proving SN. Prawitz’s method [Prawitz 1971] to handle commuting conversion may also be worth considering.

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 the stimulated discussion in the meeting of the Cooperative Research Project “Logical Approach to Formalization of Mathematics” at RIEC, Tohoku University, and in the PPL workshop series. 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

- F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- L. Baxter. *The complexity of unification*. PhD thesis, Department of Computer Science, University of Waterloo, 1977.
- N. Benton and M. Hyland. Traced premonoidal categories. *Theoretical Informatics and Applications*, 37(4):273–299, 2003.
- P. N. Benton, Gavin M. Bierman, and Valeria de Paiva. Computational types from a logical perspective. *J. Funct. Program.*, 8(2):177–193, 1998.
- R. Bird and O. D. Moor. *Algebra of Programming*. Prentice-Hall, 1996.
- F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.
- F. Blanqui. Termination of rewrite relations on λ -terms based on Girard's notion of reducibility. *Theor. Comput. Sci.*, 611:50–86, 2016.
- M. M. T. Chakravarty, G. Keller, and S. L. Peyton Jones. Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 241–253, 2005a.
- M. M. T. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 1–13, 2005b.
- T. Coquand. Pattern matching with dependent types. In *Proc. of the 3rd Work. on Types for Proofs and Programs*, 1992.
- M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proc. of PPDP'02*, pages 26–37. ACM Press, 2002.
- M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of LICS'08*, pages 57–68, 2008.
- M. Fiore and M. Hamana. Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In *Twenty-Eighth Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, pages 520–529, 2013.
- M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.
- M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.
- M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of LICS'99*, pages 193–202, 1999.
- M.P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the pi-calculus. In *11th Logic in Computer Science Conf. (LICS'96)*, pages 43–54. IEEE, 1996. Information and Computation 179, 76–117 (2002).
- J. Gibbons and R. Hinze. 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*, pages 2–14, 2011.
- J.-Y. Girard. *Proofs and types*. Cambridge University Press, 1989. translated and with appendices by Paul Taylor, Yves Lafont.
- M. Hamana. Free Σ -monoids: A higher-order syntax with metavariables. In *Asian Symposium on Programming Languages and Systems (APLAS'04)*, LNCS 3302, pages 348–363, 2004.
- M. Hamana. Universal algebra for termination of higher-order rewriting. In *Rewriting Techniques and Application (RTA'05)*, LNCS 3467, pages 135–149, 2005.
- M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of 9th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'07)*, pages 97–108. ACM Press, 2007.
- M. Hamana. Semantic labelling for proving termination of combinatory reduction systems. In *Functional and (Constraint) Logic Programming (WFLP'09)*, LNCS 5979, pages 62–78, 2010.

- M. Hamana. Strongly normalising cyclic data computation by iteration categories of second-order algebraic theories. In *Proc. of FSCD'16*, volume 52 of the *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:18, 2016.
- M. Hasegawa. Classical linear logic of implications. *Mathematical Structures in Computer Science*, 15(2):323–342, 2005.
- G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980.
- G. P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.
- J. Jouannaud, H. Kirchner, and J. Remy. Church-Rosser properties of weakly terminating term rewriting systems. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 909–915, 1983.
- D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problem in abstract algebra*, pages 263–297. Pergamon Press, Oxford, 1970. included also in *Automation of reasoning 2*, Springer (1983), pp.342–376.
- T. Libal and D. Miller. Functions-as-Constructors Higher-Order Unification. In *Proc. of FSCD 2016*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, 2016.
- S. Lindley and I. Stark. Reducibility and $\top\top$ -lifting for computation types. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21–23, 2005, Proceedings*, pages 262–277, 2005.
- R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 192(1):3–29, 1998.
- P.-A. Mellies. 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, United Kingdom*, pages 150–159, 2010.
- D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
- R. Milner. Semantic ideas in computing. In *Computing tomorrow*. Cambridge University Press, 1996.
- R. Milner. *Communicating and mobile systems - the π -calculus*. CUP, 1999.
- E. Moggi. Computational lambda-calculus and monads. LFCS ECS-LFCS-88-66, University of Edinburgh, 1988.
- E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
- T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.
- Y. Ohta and M. Hasegawa. A terminating and confluent linear lambda calculus. In *Rewriting Techniques and Application (RTA'06)*, LNCS 1833, pages 166–180, 2006.
- S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, 2001.
- F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, 1988.
- G. Plotkin and J. Power. Notions of computation determine monads. In *Proc. of FoSSaCS'02*, pages 342–356, 2002.
- G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
- J. van de Pol. Termination proofs for higher-order rewrite systems. In *the First International Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA'93)*, LNCS 816, pages 305–325, 1994.
- D. Prawitz. Ideas and results in proof theory. In *Proceedings of the 2nd Scandinavian Logic Symposium*, number 63 in *Studies in Logics and the Foundations of Mathematics*, pages 235–307, 1971.
- C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, Technische Universität München, 1995.
- A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.
- I. Stark. A fully abstract domain model for the π -calculus. In *Proc. of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 36–42, 1996.
- I. Stark. Free-algebra models for the π -calculus. *Theor. Comput. Sci.*, 390(2-3):248–270, 2008.
- S. Staton. Two cotensors in one: Presentations of algebraic theories for local state and fresh names. *Electr. Notes Theor. Comput. Sci.*, 249:471–490, 2009.
- S. Staton. An algebraic presentation of predicate logic. In *Proc. of FoSSaCS'13*, LNCS 7794, pages 401–417, 2013a.
- S. Staton. Instances of computational effects: An algebraic perspective. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, page 519, 2013b.
- S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of POPL'15*, pages 395–406, 2015.
- P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990.
- T. Yokoyama, Z. Hu, and M. Takeichi. 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*, pages 128–142, 2003.
- T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns. *Inf. Process. Lett.*, 89(6):309–314, 2004a.
- T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns for program transformation. *Computer Software*, 21(5):71–76, 2004b. in Japanese.