Theory and Practice of Second-Order Rewriting: Foundation, Evolution, and SOL

Makoto Hamana

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

Abstract. We give an overview of the theory and practice of second-order rewriting. Second-order rewriting methods have been demonstrated as useful that is applicable to important notions of programming languages such as logic programming, algebraic effects, quantum computation, and cyclic computation. We explain foundation and evolution of second-order rewriting by presenting the framework of second-order computation systems. We also demonstrate our system SOL of second-order laboratory through various programming language examples.

1 Introduction

Computation rules such as the β-reduction of the λ-calculus and arrangement of let-expressions are fundamental mechanisms of functional programming. Computation rules for modern functional programming are necessarily higher-order and are presented as a λ-calculus extended with extra rules such as rules of let-expressions or first-order algebraic rules like "$0 + x \to x$".

Because of ubiquity of computation rules, a general framework to describe and reason about them is necessary. ***Second-order computation systems*** are a framework of second-order rewriting the present author has developed for recent years [Ham17b,Ham18,Ham19,HAK20]. A second-order computation system consists of rewrite rules that may involve second-order typed terms.

2 Foundation: Second-Order Computation Systems

We give the definition of monomorphic second-order computation systems. 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) $\mathcal{T}$ is generated by atomic types, and type constructors is the least set satisfying $\mathcal{T} = \mathcal{A} \cup \{T(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in \mathcal{T}, T \text{ is an } n\text{-ary type constructor}\}$. By a *type constructor* T of arity n, we mean that it takes n-mol types $a_1, \ldots, a_n$ and gives a mol type $T(a_1, \ldots, a_n)$. A *signature* Σ is a set of function symbols of the form

$$f : (\overline{a_1} \to b_1), \ldots, (\overline{a_m} \to b_m) \to c$$

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

$$\frac{y : b \in \Gamma}{\Theta \triangleright \Gamma \vdash \ y \ : b} \qquad \frac{(M : a_1, \ldots, a_m \to b) \in \Theta \quad \Theta \triangleright \Gamma \vdash \ t_i \ : a_i \quad (1 \le i \le m)}{\Theta \triangleright \Gamma \vdash \ M[t_1, \ldots, t_m] \ : b}$$

$$\frac{f : (\overline{a_1} \to b_1), \cdots, (\overline{a_m} \to b_m) \to c \in \Sigma \quad \Theta \triangleright \Gamma, \overline{x_i : a_i} \vdash \ t_i \ : b_i \quad (1 \le i \le m)}{\Theta \triangleright \Gamma \vdash \ f(\ \overline{x_1^{a_1}}.t_1, \ldots \overline{x_m^{a_m}}.t_m\ ) \ : c}$$

Fig. 1. Typing rules of meta-terms

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

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

The last form $M[t_1, \ldots, t_n]$ is called a *meta-application*, meaning that when we instantiate $M : \overline{a} \to b$ with a term s, free variables of s (which are of types $\overline{a}$) are replaced with (meta-)terms $t_1, \ldots, t_n$.

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. 1.

For meta-terms $\Theta \triangleright \Gamma \vdash \ \ell \ : b$ and $\Theta \triangleright \Gamma \vdash \ r \ : b$, a *computation rule* is of the form $\Theta \triangleright \Gamma \vdash \quad \ell \Rightarrow r \ : b$ satisfying: (i) ℓ is a deterministic second-order pattern [YHT04]. (ii) all metavariables in r appear in ℓ. We usually omit the context and type and simply write $\ell \Rightarrow r$.

A ***computation system*** is a set $\mathcal{C}$ of computation rules. We write $s \Rightarrow_{\mathcal{C}} t$ to be one-step computation using $\mathcal{C}$ obtained by the inference system in Fig. 2.

Example 1. 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 $\mathcal{T}$ of all mol types is the least set satisfying $\mathcal{T} = \mathrm{Ty} \cup \{\mathsf{Arr}(a, b) \mid a, b \in \mathcal{T}\}$, i.e., the set of all simple types in our encoding. The λ-terms are given by a signature

$$\Sigma_{\mathrm{lam}} = \left\{ \begin{array}{l} \mathsf{lam}_{a,b} : (a \to b) \to \mathsf{Arr}(a, b) \\ \mathsf{app}_{a,b} : \mathsf{Arr}(a, b), a \to b \end{array} \mid a, b \in \mathcal{T} \right\}$$

The β-reduction law is presented as for each $a, b \in \mathcal{T}$,

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

Note that $\mathsf{Arr}(a, b)$ is a mol type, but function types $a \to b$ are not mol types.

3 Evolution of Second-Order Computation

Using the second-order computation systems, we have formulated various higher-order calculi and have checked their decidability [Ham17b,Ham18,Ham19,HAK20].

$$\text{(Rule)}\ \frac{\Theta \triangleright \Gamma', \overline{x_i : a_i} \vdash\ s_i\ : b_i \quad (1 \le i \le k) \quad \theta = [\,\overline{M \mapsto \overline{x}.s}\,] \qquad (M_1 : (\overline{a_1} \to b_1), \ldots, M_k : (\overline{a_k} \to b_k) \triangleright\ \vdash\ \ell \Rightarrow r\ : c) \in \mathcal{C}}{\Theta \triangleright \Gamma' \vdash\ \ell\,[\,\overline{M \mapsto \overline{x}.s}\,] \Rightarrow_{\mathcal{C}} r\,[\,\overline{M \mapsto \overline{x}.s}\,]\ : c}$$

$$\text{(Fun)}\ \frac{f : (\overline{a_1} \to b_1), \cdots, (\overline{a_k} \to b_k) \to c \in \Sigma \qquad \Theta \triangleright \Gamma, \overline{x_i : a_i} \vdash\ t_i \Rightarrow_{\mathcal{C}} t_i'\ : b_i \quad (\text{some } i \text{ s.t. } 1 \le i \le k)}{\Theta \triangleright \Gamma \vdash\ f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i \ldots, \overline{x_1^{a_1}}.t_k) \Rightarrow_{\mathcal{C}} f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i' \ldots, \overline{x_1^{a_1}}.t_k)\ : c}$$

Fig. 2. Second-order computation (one-step)

The framework of second-order computation systems is founded on foundational studies on second-order computation. We describe below the foundations and succeeding developments how the theory of second-order computation systems is evolved.

Second-order Abstract Syntax. The syntactic structure of meta-terms and substitution for abstract syntax with variable binding was introduced by Aczel [Acz78] for his general framework of rewrite rules. Fiore, Plotkin, and Turi formulated *second-order abstract syntax* [FPT99,Fio08] as a mathematical structure of syntax with variable binding using algebras on presheaves. The abstract syntax and the associated algebraic structure have extended to second-order abstract syntax with extra feature or type discipline: having metavariables [Ham04], simple types [Fio02,Ham07], dependent types [Fio08], and polymorphic types [Ham11].

Second-order Algebraic Theories. Ordinary equational logic is logic for equations on first-order algebraic terms. Second-order algebraic theories and equational logic [FM10,FH10] are second-order extensions, which provide mathematical models of second-order equations and deduction based on second-order abstract syntax and its algebraic models. This algebraic modelling of syntax, theory, type system, or programming laguage has been actively investigated. Staton demonstrated that second-order algebraic theories are a useful framework that models various important notions of programming languages such as logic programming [Sta13a], algebraic effects [Sta13b,FS14], and quantum computation [Sta15]. We have also applied it to modelling cyclic structures [Ham10a] and cyclic datatypes modulo bisimulation [Ham17a]. This line of algebraic modelling is still active. Recently, Arkor and Fiore [AF20] gave algebraic models of simple type theories.

Second-order Computation Systems. Based on the structures of second-order abstract syntax and algebraic theories, the present author developed the framework of ***second-order computation systems*** and its algebraic semantics [Ham05]. Notably, the semantics established the first *sound and complete model* of second-order rewriting systems. It has been extended to *simply-typed second-order computation systems* and its algebraic semantics [Ham07]. This is the basis of our SOL system described below. We also applied the semantical

structures to develop termination proof techniques: the interpretation method [Ham05], higher-order semantic labelling [Ham07,Ham10b], and modular termination [Ham20].

Second-Order Rewriting. As the rewriting theoretic side, Aczel's formal language allowed him to consider a general framework of rewrite rules for calculi with variable binding, which influenced Klop's rewriting framework of combinatory reduction systems [Klo80]. Blanqui introduced a typed version of Klop's framework and provided a termination criterion of the General Schema [Bla00]. SOL implemented the General Schema criterion for termination checking.

Polymorphic Computation Systems with Call-by-Value. We have developed a general framework of *multiversal polymorphic algebraic theories* [FH13] based on polymorphic abstract syntax [Ham11]. It admits multiple type universes and higher-kinded polymorphic types. Based on it, we presented a new framework of polymorphic second-order computation systems [Ham18] that can accommodate a distinction between values and non-values [HAK20]. It is suitable for analysing fundamental calculi of programming languages. We developed a type inference algorithm and new criteria to check the confluence property.

4 SOL: Second-Order Laboratory

Based on the above foundations and evolution, we have implemented the system SOL [Ham17b,Ham18,Ham19,HAK20]. SOL is a tool to check confluence and termination of polymorphic second-order computation systems. The system works on top of the interpreter of Glasgow Haskell Compiler. SOL uses the feature of quasi-quotation (i.e. `[signature|..]` and `[rule|..]` are quasi-quotations) of Template Haskell, with a custom parser which provides a readable notation for signature, terms and rules. It makes the language of our formal computation rules available within a Haskell script. For example, the computation system of λ-calculus in Example 1 is described as

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

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

The web interface for SOL is available at the author's homepage:

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

References

[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978.

[AF20] N. Arkor and M. Fiore. Algebraic models of simple type theories: A polynomial approach. In *Proc. of LICS '20*, pages 88–101. ACM, 2020.

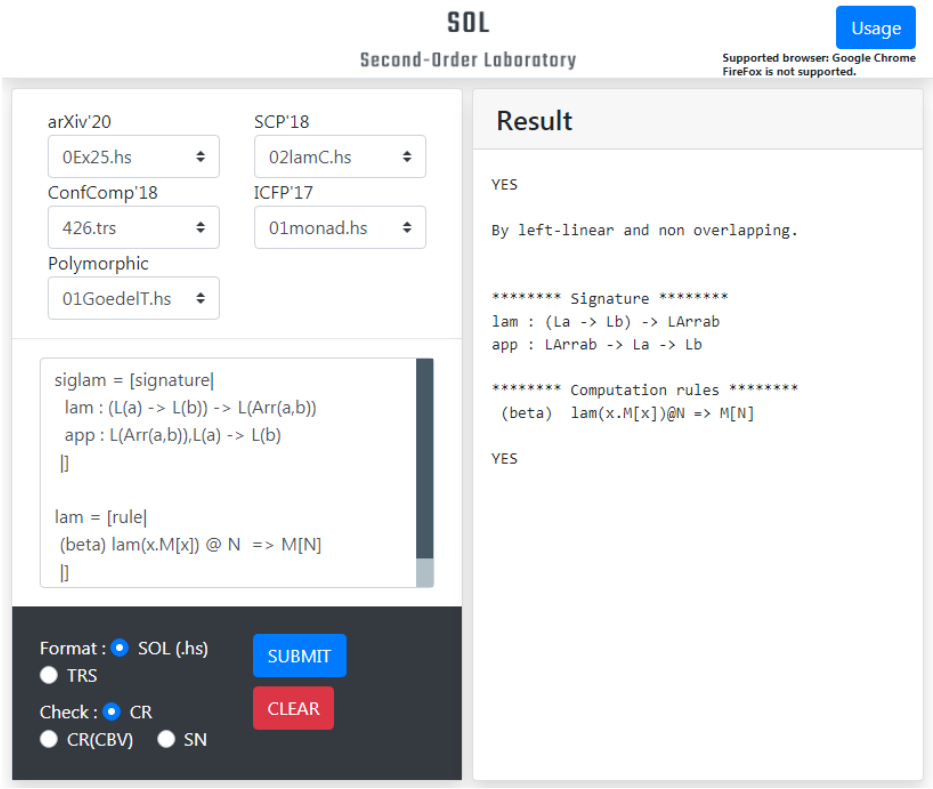

Fig. 3. Web interface of SOL

[Bla00] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.

[Bla16] F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theor. Comput. Sci.*, 611:50–86, 2016.

[FH10] M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.

[FH13] M. Fiore and M. Hamana. Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, pages 520–529, 2013.

[Fio02] M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proc. of PPDP'02*, pages 26–37. ACM Press, 2002.

[Fio08] M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of LICS'08*, pages 57–68, 2008.

[FM10] M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of LICS'99*, pages 193–202, 1999.

[FS14] M. Fiore and S. Staton. Substitution, jumps, and algebraic effects. In *Proc. CSL-LICS 2014*, pages 41:1–41:10, 2014.

[HAK20] M. Hamana, T. Abe, and K. Kikuchi. Polymorphic computation systems: Theory and practice of confluence with call-by-value. *Science of Computer Programming*, 187(102322), 2020.

[Ham04] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Proc. of APLAS'04*, LNCS 3302, pages 348–363, 2004.

[Ham05] M. Hamana. Universal algebra for termination of higher-order rewriting. In *Proc. of RTA'05*, LNCS 3467, pages 135–149, 2005.

[Ham07] M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of PPDP'07*, pages 97–108. ACM Press, 2007.

[Ham10a] M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6, Issue 3(15), 2010.

[Ham10b] M. Hamana. Semantic labelling for proving termination of combinatory reduction systems. In *Proc. WFLP'09*, LNCS 5979, pages 62–78, 2010.

[Ham11] M. Hamana. Polymorphic abstract syntax via Grothendieck construction. In *FoSSaCS'11*, LNCS3467, pages 381–395, 2011.

[Ham17a] M. Hamana. Cyclic datatypes modulo bisimulation based on second-order algebraic theories. *Logical Methods in Computer Science*, 2017.

[Ham17b] M. Hamana. How to prove your calculus is decidable: Practical applications of second-order algebraic theories and computation. *Proceedings of the ACM on Programming Languages*, 1(22):1–28, September 2017.

[Ham18] M. Hamana. Polymorphic rewrite rules: Confluence, type inference, and instance validation. In *Proc. of 14th International Symposium on Functional and Logic Programming (FLOPS'18)*, LNCS 10818, pages 99–115, 2018.

[Ham19] M. Hamana. How to prove decidability of equational theories with second-order computation analyser SOL. *Journal of Functional Programming*, 29(e20), 2019.

[Ham20] M. Hamana. Modular termination for second-order computation rules and application to algebraic effect handlers. arXiv:1912.03434, 2020.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. volume 127 of Mathematical Centre Tracts.

[Sta13a] S. Staton. An algebraic presentation of predicate logic. In *Proc. of FOSSACS 201*, pages 401–417, 2013.

[Sta13b] S. Staton. Instances of computational effects: An algebraic perspective. In *Proc. of LICS'13*, page 519, 2013.

[Sta15] S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of POPL'15*, pages 395–406, 2015.

[YHT04] T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns. *Inf. Process. Lett.*, 89(6):309–314, 2004.