Logical Methods in Computer Science
Volume 18, Issue 2, 2022, pp. 18:1–18:32
https://lmcs.episciences.org/

Submitted Jun. 26, 2020
Published Jun. 14, 2022

MODULAR TERMINATION FOR SECOND-ORDER COMPUTATION RULES AND APPLICATION TO ALGEBRAIC EFFECT HANDLERS

MAKOTO HAMANA

Faculty of Informatics, Gunma University
e-mail address: hamana@gunma-u.ac.jp

Abstract. We present a new modular proof method of termination for second-order computation, and report its implementation SOL. The proof method is useful for proving termination of higher-order foundational calculi. To establish the method, we use a variation of the semantic labelling translation and Blanqui's General Schema: a syntactic criterion of strong normalisation. an application, we show termination of a variant of call-by-push-value calculus with algebraic effects, an effect handler and effect theory. We also show that our tool SOL is effective to solve higher-order termination problems.

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 "$0 + x \to x$".

The termination property is one of the most important properties of such a calculus because it is a key to ensuring the decidability of its properties. A powerful termination checking method is important in theory and in practice. For instance, Agda and Coq perform termination checking before type checking for decidable type checking. Haskell's type families [CKJM05, CKJ05] have several syntactic restrictions on the form of type instances to ensure termination, but a more flexible and powerful termination checking method is desirable. Although currently Haskell's rewrite rule pragma [JTH01] does not have any restriction on the rules, ideally some conditions or termination checking are necessary, because the compiler may go into an infinite loop without termination assurance.

In the situations listed above, termination is ideally checked *modularly*. Several reasons for it can be given.

(i) A user program is usually built on various library programs. Rather than checking termination of the union of the user and library program at once, we prove termination

Key words and phrases: Theory, λ-calculus, algebra, modularity, termination, second-order algebraic theory, functional programming language, Haskell.

DOI:10.46298/LMCS-18(2:18)2022

of library programs beforehand. And then we just prove termination of the user program at compile time to ensure termination of the whole program.

(ii) For a well-known terminating calculus (such as the typed λ-calculus), one often extends it by adding extra computation rules to enrich the computation power. Usually, proving termination of the extended calculus directly by hand is difficult. If merely proving termination of the extra rule part suffices to conclude termination of the extended calculus, then the termination proof becomes much easier.

(iii) Effectful λ-calculus with effect handlers [Lev06, PP13, KLO13, FKLP19] can accommodate computational effects and ordinary values. An effect handler interprets effect terms as actual effects. Moreover, each effect has an *effect theory* required to satisfy [PP02]. Such an effect theory can be regarded as optimization rules of effectful programs. We must ensure the termination of the combination of such optimization rules and effectful λ-calculus with effect handlers. Because the effectful λ-calculus with effect handler is generic, its termination can be established beforehand, but an effect theory varies according to used effects in a program. Ideally, we prove termination of effect theories merely to ensure the termination of the whole calculus: the effectful λ-calculus with effect theories[1].

However, in general, strong normalisation is not modular. There are examples which show that, given two separated terminating computation rule sets $\mathbf{A}$ and $\mathbf{B}$, the disjoint union $\mathbf{A} \uplus \mathbf{B}$ is not always terminating [Toy87]. Therefore, the above mentioned modular termination checking is not immediately available.

Although unrestricted modular termination checking fails, for various restricted classes, modular termination results have been obtained. This includes the cases of first-order algebraic rules [TKB95, Ohl94, Gra94, GAO02] and the λ-calculus extended with first-order algebraic rules [Oka89, BFG97, BTG91, Pol96]. There are various termination criteria for higher-order pattern rules: [Bla00, Bla16, JR15, Pol94]. By a higher-order pattern rule, we mean that the left-hand side is a Miller's higher-order pattern [Mil91], and the rule is matched by higher-order matching, not just by matching modulo α-renaming. This class of rules is more general than ordinary higher-order functional programs, such as Haskell's programs and rewrite rules, where the left-hand sides are first-order patterns. This class is important to faithfully formalise foundational calculi such as the typed λ-calculus and its variants as higher-order pattern rules. For example, the class includes the encoding of the β-reduction rule $\mathsf{lam}(x.M[x])@N \Rightarrow M[N]$ and the η-reduction rule $\mathsf{lam}(x.(L@x)) \Rightarrow L$ as sample higher-order pattern rules, where the metavariable L cannot contain the variable x, although M can contain x because of higher-order patterns and matching.

1.1. **The framework.** We use the framework of ***second-order computation systems***, which are computational counterparts of ***second-order algebraic theories*** [FM10, FH10]. They formalise pattern rules consisting of second-order typed terms. The limitation to second order is irrelevant to the type structure of the object language. Second-order abstract syntax [FPT99, Fio08] has been demonstrated capable to encode higher-order terms of any order. Also, a well-developed model theory exists for second-order syntax: algebras on presheaves and Σ-monoids [FPT99, Ham04, Ham05].

[1] In Section 4, we demonstrate the termination of this using our result of a modular termination theorem.

Using this framework, we have formulated various higher-order calculi as second-order algebraic theories and have checked their decidability using second-order computation systems and the tool SOL, the second-order laboratory [Ham17, Ham18, Ham19, HAK20]. 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]. Our modular termination method is applicable to algebraic theories of these applications.

1.2. Contributions.

In this paper, we use the word ***termination*** to mean strong normalisation (in short, ***SN***, meaning that any computation path is finite). We establish a modular termination proof method of the following form: if $\mathbf{A}$ is SN and $\mathbf{B}$ is SN with some suitable conditions, then $\mathcal{C} = \mathbf{A} \uplus \mathbf{B}$ is SN. More precisely, the contributions of this paper can be summarised as described below.

1. We present a new modular proof method of termination for second-order computation and prove its correctness (Section 3, Section B).
2. As an application of the modular termination proof method, we give a termination proof of effectful calculus using a variant of Levy's call-by-push value (CBPV) calculus [Lev06] called multi-adjunctive metalanguage (MAM) [FKLP19] (Section 4) with effect handlers and effect theory [PP13].
3. As another application of the modular termination proof method, we give a termination proof method for combinations of first-order and higher-order rules (Section 5).
4. We report the implementation SOL and experiments over a collection of termination problems (Section 5.2, Section 5.3).

1.3. Organisation.

This paper is organised as follows. We first introduce the framework of this paper, and key techniques in Section 2. In Section 3, we prove the main theorem. In Section B, we prove a version of higher-order semantic labelling. In Section 4 and Section 5, we show applications of the main theorem through several examples. In Section 6, we discuss related work and give a summary. The computation systems in examples of this paper are available at the arXiv′20 pull-down menu of the web interface of SOL system `http://solweb.mydns.jp/` .

2. Preliminaries

2.1. Second-Order Computation Systems.

We introduce a formal framework of second-order computation based on second-order algebraic theories [FH10, FM10]. This framework has been used in [Ham17].

Notation 2.1. The notation $A \uplus B$ denotes the disjoint union of two sets, where $A \cap B = \varnothing$ is supposed. We use the notation $\overline{A}$ for a sequence $A_1, \cdots, A_n$, and $|\overline{A}|$ for its length. We abbreviate the words left-hand side as "lhs", right-hand side as "rhs", first-order as "FO" and higher-order as "HO".

For a binary relation $\to$, we write $\to^*$ for the reflexive transitive closure, $\to^+$ for the transitive closure, and $\leftarrow$ for the inverse of $\to$.

For a preorder $\succeq$, we write $s \succ t$ if $s \succeq t$ and $t \not\succeq s$, and the transitive and irreflexive relation $\succ$ is called the strict part of $\succeq$.

2.2. **Types.** We assume that $\mathcal{A}$ is a set of *atomic types* (e.g. Bool, Nat, etc.). We also assume a set of *type constructors* together with arities $n \in \mathbb{N}$, $n \geq 1$. The sets of ***molecular types*** *(* mol types, for short) $\mathcal{T}_0$ and ***types*** $\mathcal{T}$ are generated by the following rules:

$$\frac{b \in \mathcal{A}}{b \in \mathcal{T}_0} \qquad \frac{\begin{array}{c} b_1, \ldots, b_n \in \mathcal{T}_0 \\ T \ n\text{-ary type constructor} \end{array}}{T(b_1, \ldots, b_n) \in \mathcal{T}_0} \qquad \frac{a_1, \ldots, a_n, b \in \mathcal{T}_0}{a_1, \ldots, a_n \to b \in \mathcal{T}}$$

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 [Bla00, Sta13b].

2.3. **Terms.** 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). A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g., $b_1, \ldots, b_m \to c$, or $() \to c$. The latter case is simply denoted by c. We assume two disjoint syntactic classes of letters, called ***metavariables*** (written as capital letters $M, N, K, \ldots$) and ***variables*** (written usually $x, y, \ldots$). The raw syntax is given as follows.

- ***Terms*** have the form $\quad t ::= x \mid x^a.t \mid f(t_1, \ldots, t_n)$.
- ***Meta-terms*** extend terms to $\quad s ::= x \mid x^a.s \mid f(s_1, \ldots, s_n) \mid M[s_1, \ldots, s_n]$.

These forms are respectively *variables*, *abstractions*, and *function terms*, and the last form is called a *meta-application*. We may write $x_1^{a_1}, \ldots, x_n^{a_n}.\, t$ (or $\overline{x^a}.t$) for $x_1^{a_1}. \cdots .x_n^{a_n}.\, t$, and we assume ordinary α-equivalence for bound variables. Hereafter, we often omit the superscript of variables $x_i^{a_i}$. We also assume that all bound variables and free variables are mutually disjoint in computation steps to avoid α-renaming during computation. If computation rules do not satisfy this property, we consider suitable variants of the rules by renaming free/bound (meta)variables. A metavariable context Z is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of pairwise distinct (variable:mol type)-pairs. Thus writing a context Γ, Γ', we implicitly mean that Γ and Γ' are disjoint. A judgment is of the form

$$Z \triangleright \Gamma \vdash t : b.$$

A meta-term t is called *well-typed* if $Z \triangleright \Gamma \vdash t : c$ is derived by the typing rules in Fig. 1 for some Z, Γ, c.

$$\frac{y : b \in \Gamma}{Z \rhd \Gamma \vdash \ y \ : b} \qquad \frac{\begin{array}{c}(M : a_1, \ldots, a_m \to b) \in Z \\ Z \rhd \Gamma \vdash \ t_i \ : a_i \quad (1 \le i \le m)\end{array}}{Z \rhd \Gamma \vdash \ M[t_1, \ldots, t_m] \ : b} \qquad \frac{Z \rhd \Gamma, \overline{x : a} \vdash \ t \ : b}{Z \rhd \Gamma \vdash \ \overline{x : a}.t \ : \overline{a} \to b}$$

$$\frac{\begin{array}{c}f : (\overline{a_1} \to b_1), \cdots, (\overline{a_m} \to b_m) \to c \in \Sigma \\ Z \rhd \Gamma \vdash \ \overline{x_i^{a_i}}.s_i \ : b_i \quad (1 \le i \le m)\end{array}}{Z \rhd \Gamma \vdash \ f(\overline{x_1^{a_1}}.s_1, \ldots, \overline{x_m^{a_m}}.s_m) \ : c}$$

Figure 1: Typing rules of meta-terms

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

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

Figure 2: Second-order computation (one-step)

2.4. **Contextual sets of meta-terms.** In the proofs of this paper, we will use the structure of type and context-indexed sets. A ***contextual set*** A is a family $\{A_b(\Gamma) \mid b \in \mathcal{T},\ \text{context } \Gamma\}$ of sets indexed by types and variable contexts. Set operations such as $\cup, \uplus, \cap$ are extended to contextual sets by index-wise constructions, such as $A \cup B$ by $\{A_b(\Gamma) \cup B_b(\Gamma) \mid b \in \mathcal{T},\ \text{context } \Gamma\}$. Throughout this paper, for a contextual set A, we simply write $a \in A$ if there exist b, Γ such that $a \in A_b(\Gamma)$. The indices are usually easily inferred from context. A ***map*** $f : A \to B$ ***between contextual sets*** is given by indexed functions $\{f_b(\Gamma) : A_b(\Gamma) \to B_b(\Gamma) \mid b \in \mathcal{T},\ \text{context } \Gamma\}$. Examples of contextual sets are ***the contextual sets of meta-terms*** $M_\Sigma Z$ and ***of terms*** $T_\Sigma \mathrm{V}$ defined by

$$M_\Sigma Z_b(\Gamma) \triangleq \{t \mid Z \rhd \Gamma \vdash t : b\}, \qquad T_\Sigma \mathrm{V}_b(\Gamma) \triangleq \{t \mid \ \rhd \Gamma \vdash t : b\}.$$

for a given signature Σ. We call a meta-term t Σ-*meta-term* if t is constructed from Σ and (meta)variables, i.e., $t \in M_\Sigma Z$ for some Z.

The notation $t\,\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1, \ldots, s_n$.

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

$$\rhd \Gamma',\, x_i^1 : c_i^1, \ldots, x_i^{n_i} : c_i^{n_i} \vdash \ s_i \ : b_i \qquad (1 \le i \le k),$$
$$M_1 : \overline{c_1} \to b_1, \ldots, M_k : \overline{c_k} \to b_k \ \rhd \ \Gamma \vdash \ e \ : c$$

For an assignment $\theta = \{M_1 \mapsto \overline{x_1}.s_1, \ldots, M_k \mapsto \overline{x_k}.s_k\} : Z \to T_\Sigma \mathrm{V}$,
the map $\theta^\sharp : M_\Sigma Z \to T_\Sigma \mathrm{V}$ is *a substitution for metavariables* defined by

$$\begin{aligned} \theta^\sharp(x) &\triangleq x \\ \theta^\sharp(M_i[t_1, \ldots, t_{n_i}]) &\triangleq s_i\ \{x_i^1 \mapsto \theta^\sharp(t_1), \ldots, x_i^{n_i} \mapsto \theta^\sharp(t_{n_i})\} \\ \theta^\sharp(f(\overline{y_1}.t_1, \ldots, \overline{y_k}.t_k)) &\triangleq f(\overline{y_1}.\theta^\sharp(t_1), \ldots, \overline{y_k}.\theta^\sharp(t_k)) \end{aligned}$$

Lemma 2.4. *Under the situation of the above definition, the substituted term is well-typed as* $\triangleright \Gamma, \Gamma' \vdash \ \theta^\sharp(e) \ : c$.

Proof. This is proved by straightforward induction on the typing derivations [FH10]. □

Remark 2.5. The map $\theta^\sharp : M_\Sigma Z \to T_\Sigma \mathrm{V}$ is the substitution operation of terms for metavariables, and is a homomorphic extension of $\theta : Z \to T_\Sigma \mathrm{V}$, depicted as the diagram of Figure 3, where η_Z embeds metavariables to meta-terms. It shows that $\theta^\sharp$ is a unique

$$\begin{tikzcd} Z \arrow[r, "\eta_Z"] \arrow[dr, "\theta"'] & M_\Sigma Z \arrow[d, "\theta^\sharp"] \\ & T_\Sigma \mathrm{V} \end{tikzcd}$$

Figure 3: The map $\theta^\sharp$

Σ-monoid morphism that extends θ [Ham04, Def. 11, θ^*] [FH13, III]. The syntactic structure of meta-terms and substitution for abstract syntax with variable binding was first introduced by Aczel [Acz78]. This formal language allowed him to consider a general framework of rewrite rules for calculi with variable binding. The structure of meta-terms has clean algebraic properties. The contextual set $T_\Sigma \mathrm{V}$ forms an initial Σ-monoid [FPT99] and $M_\Sigma Z$ forms a free Σ-monoid over Z [Ham04, Fio08]. These algebraic characterisations have been applied to the complete algebraic charactersisation of termination of second-order rewrite systems [Ham05] and higher-order semantic labelling [Ham07]. The polymorphic and precise algebraic characterisation of abstract syntax with binding and substitution were given in [FH13].

2.5. **Computation rules.** First we need the notion of Miller's second-order pattern [Mil91]. A ***second-order pattern*** is a meta-term in which every occurrence of meta-application is of the form $M[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n$ are distinct bound variables.

For meta-terms $Z \triangleright \ \vdash \ \ell \ : b$ and $Z \triangleright \ \vdash \ r \ : b$ using a signature Σ, a *computation rule* is of the form

$$Z \triangleright \ \vdash \ \ell \Rightarrow r \ : b$$

satisfying:

(i) ℓ is a function term and a second-order pattern.

(ii) all metavariables in r appear in ℓ.

Note that ℓ and r are meta-terms without free variables, but may have free ***meta***variables. A ***computation system*** (CS) is a pair $(\Sigma, \mathcal{C})$ of a signature and a set $\mathcal{C}$ of computation rules consisting of Σ-meta-terms. We write $s \Rightarrow_{\mathcal{C}} t$ to be one-step computation using $\mathcal{C}$ obtained by the inference system given in Fig. 2. We may omit some contexts and type information of a judgment, and simply write it as $Z \triangleright \ell \Rightarrow r : b$, $\ell \Rightarrow_{\mathcal{C}} r$, or $\ell \Rightarrow r$ if they are clear from the context. From the viewpoint of pattern matching, (Rule) means that a computation system uses the decidable second-order pattern matching [Mil91] for one-step computation (cf. [Ham17, Sec.6.1]) not just syntactic matching. We regard $\Rightarrow_{\mathcal{C}}$ to be a binary relation on terms.

A function symbol $f \in \Sigma$ is called ***defined*** if it occurs at the root of the lhs of a rule in $\mathcal{C}$. Other function symbols in Σ are called ***constructors***.

Example 2.6. The simply-typed λ-terms on the set LamTy of simple types generated by a set of base types BTy are modeled in our setting as follows. Let $\mathcal{A} = \mathrm{BTy}$. We suppose type constructors L, Arr. The set of LamTy of all simple types for the λ-calculus is the least set satisfying

$$\mathrm{LamTy} = \mathrm{BTy} \cup \{\mathsf{Arr}(a, b) \mid a, b \in \mathrm{LamTy}\}.$$

We use the mol type $\mathsf{L}(a)$ for encoding λ-terms of type $a \in \mathrm{LamTy}$. The λ-terms are given by a signature

$$\Sigma_{\mathrm{stl}} = \left\{ \begin{array}{ll} \mathsf{lam}_{a,b} & : (\mathsf{L}(a) \to \mathsf{L}(b)) \to \mathsf{L}(\mathsf{Arr}(a,b)) \\ \mathsf{app}_{a,b} & : \mathsf{L}(\mathsf{Arr}(a,b)), \mathsf{L}(a) \to \mathsf{L}(b) \end{array} \quad \middle| \ a, b \in \mathrm{LamTy} \right\}$$

The β-reduction law is presented as

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

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

We use the following notational convention throughout the paper. We will present a signature by omitting mol type subscripts a, b (see also more detailed account [Ham18]). For example, simply writing function symbols lam and app, we mean $\mathsf{lam}_{a,b}$ and $\mathsf{app}_{a,b}$ in Σ_{stl} having appropriate mol type subscripts a, b.

2.6. **The General Schema.** The General Schema is a criterion for proving strong normalisation of higher-order rules developed by Blanqui, Jouannaud and Okada [BJO02] and refined by Blanqui [Bla00, Bla16]. We summarise the definitions and properties of GS in [Bla00, Bla16]. The General Schema has succeeded in proving SN of various rewrite rules such as Gödel's System T. The basic idea of GS is to check whether the arguments of recursive calls in 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" [Coq92], but more relaxed and extended. This section reviews the definitions and the property of GS criterion [Bla00, Bla16].

We give a summary of the General Schema (GS) criterion [Bla00, Bla16] of the second-order case. Suppose that

- a well-founded preorder $\leq_{\mathcal{T}}$ on the set $\mathcal{T}$ of types and
- a well-founded preorder $\leq_{\Sigma}$ on a signature Σ

are given. Let $<_{\mathcal{T}}$ be the strict part of $\leq_{\mathcal{T}}$, and $=_{\mathcal{T}} \triangleq \leq_{\mathcal{T}} \cap \geq_{\mathcal{T}}$ the associated equivalence relation. Similarly for $\leq_{\Sigma}$.

The *stable sub-meta-term ordering* $\trianglelefteq_{\mathsf{s}}$ is defined by $s \trianglelefteq_{\mathsf{s}} t$ if s is a sub-meta-term of t and all the free variables in s appear in t.

Definition 2.7. A metavariable M is *accessible in a meta-term* t if there are distinct bound variables $\overline{x}$ such that $M[\overline{x}] \in \mathsf{Acc}(t)$, where $\mathsf{Acc}(t)$ is the least set satisfying the following clauses:

(a1) $t \in \mathsf{Acc}(t)$.

(a2) If $x.u \in \mathsf{Acc}(t)$ then $u \in \mathsf{Acc}(t)$.

(a3) Let $f : \tau_1, \dots, \tau_n \to b$ and $f(u_1, \dots, u_n) \in \mathsf{Acc}(t)$,
where $\tau_i = \overline{a_i} \to c_i$ for each $i = 1, \dots, n$ ($|\overline{a_i}|$ is possibly 0, and then $\tau_i = c_i$).
If (for all $a \in \{\overline{a_i}\}$, $a <_{\mathcal{T}} b$) & $c_i \le_{\mathcal{T}} b$, then $u_i \in \mathsf{Acc}(t)$.

Note that (a3) also says that any of $\overline{a_i}$ must not be equivalent (w.r.t. $=_{\mathcal{T}}$) to b.

Definition 2.8. Given $f \in \Sigma$, the *computable closure* $\mathsf{CC}_f(\bar{t})$ of a meta-term $f(\bar{t})$ is the least set CC satisfying the following clauses. All the meta-terms and abstractions below are assumed to be well-typed.

1. **(meta M)** If $M : \tau_1, \dots, \tau_p \to b$ is accessible in some of $\bar{t}$, and $\overline{u} \in \mathsf{CC}$, then $M[\overline{u}] \in \mathsf{CC}$.
2. For any variable x, $x \in \mathsf{CC}$.
3. If $u \in \mathsf{CC}$ then $x.u \in \mathsf{CC}$.
4. **(fun $f >_\Sigma g$)** If $f >_\Sigma g$ and $\overline{w} \in \mathsf{CC}$, then $g(\overline{w}) \in \mathsf{CC}$.
5. **(fun $f =_\Sigma g$)** If $\overline{u} \in \mathsf{CC}$ such that $\bar{t} \rhd_{\mathsf{s}}^{\mathsf{lex}} \overline{u}$, then $g(\overline{u}) \in \mathsf{CC}$, where $\rhd_{\mathsf{s}}^{\mathsf{lex}}$ is the lexicographic extension of the stable sub-meta-term ordering $\rhd_{\mathsf{s}}$.

The labels **(meta M)** etc. are used for references in a termination proof using GS.

Theorem 2.9 [Bla00, Bla16]**.** Let $(\Sigma, \mathcal{C})$ be a computation system. Suppose that $\le_{\mathcal{T}}$ and $\le_\Sigma$ are well-founded. If for all $f(\bar{t}) \Rightarrow r \in \mathcal{C}$, $\mathsf{CC}_f(\bar{t}) \ni r$, then $\mathcal{C}$ is strongly normalising.

Example 2.10. We consider a computation system `recursor` of a recursor on natural numbers. The signature Σ_{rec} [Bla16] is given by

$\texttt{zero : L(Nat)}, \quad \texttt{succ : L(Nat)} \to \texttt{L(Nat)}$
$\texttt{rec}_a \texttt{ : L(Nat),L(}a\texttt{),(L(Nat),L(}a\texttt{)} \to \texttt{L(}a\texttt{))} \to \texttt{L(}a\texttt{)}$

where $a \in$ LamTy, `Nat` $\in$ BTy, and BTy and LamTy are the ones used in Example 2.6. We take preorders $\le_{\mathcal{T}}, \le_\Sigma$ to be the identities. The rules are

```
(recZ) rec(zero, U,x.y.V[x,y])    ⇒ U
(recS) rec(succ(X),U,x.y.V[x,y]) ⇒ V[X, rec(X,U,x.y.V[x,y])]
```

We check $\mathsf{CC}_{\texttt{rec}}(\texttt{succ(X),U,x.y.V[x,y]}) \ni \texttt{V[X, rec(X,U,x.y.V[x,y])]}$.

- **(meta `V`)** is applicable. We check that `V` is accessible and `X` $\in \mathsf{CC}$. Now `V[x,y]` $\in \mathsf{Acc}(\texttt{x.y.V[x,y]})$ holds by (a1)(a2). `X` $\in \mathsf{CC}$ holds because `X` is accessible in `succ(X)`.
- To check `rec(X,U,x.y.V[x,y])` $\in \mathsf{CC}$, **(fun `rec`=`rec`)** is applicable.
Since `succ(X)` $\rhd_{\mathsf{s}}$ `X` and `U,x.y.V[x,y]`$\in \mathsf{CC}$, which is easily checked, we are done.

We also check $\mathsf{CC}_{\texttt{rec}}(\texttt{zero,U,x.y.V[x,y]}) \ni \texttt{U}$ by **(meta `U`)**. Hence the computation system is SN.

2.7. **Higher-Order Semantic Labelling.** As we have seen in Example 2.10, GS checks syntactical decreasing (`succ(X)` $\rhd_{\mathsf{s}}$ `X`) of an argument in each recursive call. But sometimes recursion happens with syntactically larger but semantically smaller arguments. For example, consider the following computation system of computing the prefix sum of a list

```
map(y.F[y],nil)         ⇒ nil
map(y.F[y],cons(X,XS)) ⇒ cons(F[X],map(y.F[y],XS))
```

```
ps(nil)            ⇒ nil
ps(cons(X,XS))     ⇒ cons(X,ps(map(y.X+y,XS)))
```

In the final rule, `ps` in rhs is called with a shorter list than `cons(X,XS)` in lhs, but syntactically, `map(y.X+y,XS)` is not a sub-meta-term of `cons(X,XS)`. Therefore, the rule does not satisfy GS. The *higher-order semantic labelling method* [Ham07] extending [Zan95] solves this problem, which is founded on the presheaf models of second-order syntax [FPT99, Fio08, Ham04] and computation [Ham05]. We will use a version of the method in the proof of the main theorem, hence we sketch the idea of it. The following notion is needed: a ***quasi-model*** $(A, \geq)$ of a second-order computation system $\mathcal{C}$ is a second-order algebra equipped with a preorder (i.e. a weakly monotone Σ-monoid [Ham05, Ham07]) in which, for every rule $Z \triangleright\ \vdash \ell \Rightarrow r$ of $\mathcal{C}$, $[\![\ell]\!]\varphi \geq [\![r]\!]\varphi$ holds for every assignment $\varphi : Z \to A$, where $[\![-]\!]\varphi$ is an interpretation of meta-terms using φ. It is called *well-founded* (or SN) if $\geq$ is well-founded. If one finds a quasi-model for a given $\mathcal{C}$, then one attaches semantic elements in the quasi-model to the function symbols in the rules of $\mathcal{C}$. For the case of the rules of the prefix sum, one chooses a quasi-model of natural numbers with the usual order (where `ps` is interpreted as counting the length of the argument list). Applying the higher-order semantic labelling method, one obtains labelled rules [Ham10]:

$$\mathtt{ps}_{n+1}\mathtt{(cons(X,XS))} \ \Rightarrow\ \mathtt{cons(X,ps}_{n}\mathtt{(map(y.X+y,XS)))}$$

for all $n \in \mathbb{N}$, where n is a label. The labelling, in principle, does not change the computational behavior, but it is effective to prove SN because the call relation $\mathtt{ps}_{n+1} > \mathtt{ps}_n$ is well-founded, the rule can satisfy GS.

The main theorem [Ham07, Thm.3.7] of HO semantic labelling ensures that if the labelled second-order computation system combined with additional *decreasing rules*

$$\mathtt{ps}_{n+1}\mathtt{(X)} \ \Rightarrow\ \mathtt{ps}_{n}\mathtt{(X)}$$

(which expresses compatibility of the computation relation $\Rightarrow_{\mathcal{C}}$ with the order $n+1 > n$ of the quasi-model) is proved to be SN, then the original system is SN.

3. A Modular Termination Theorem for Second-Order Computation

In this section, we prove the main theorem of this paper. In the rest of this paper, we assume the following. $\mathsf{Fun}(-)$ denotes the set of all function symbols in its argument.

Assumption 3.1. Let $(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A})$ and $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ be computation systems satisfying:

(i) $\Sigma_{\mathbf{A}}$ is the set of defined function symbols of $\mathbf{A}$.

(ii) $\Sigma_{\mathbf{B}}$ is the set of defined function symbols of $\mathbf{B}$.

(iii) Θ is the signature for constructors of $\mathcal{C}$, where $\mathcal{C} \triangleq \mathbf{A} \uplus \mathbf{B}$.

(iv) $\mathcal{C}$ is finitely branching.

(v) Both sides of each rule in $\mathcal{C}$ satisfy the $\Sigma_{\mathbf{A}}$-layer condition.

Definition 3.2. We say that ***a meta-term*** u ***satisfies the*** $\Sigma_{\mathbf{A}}$***-layer condition*** if for every $f(\overline{\overline{x}.t}) \lhd u$ with $f \in \Sigma_{\mathbf{A}}$, $\mathsf{Fun}(f(\overline{\overline{x}.t})) \subseteq \Sigma_{\mathbf{A}} \uplus \Theta$ and $\overline{t}$ are second-order patterns.

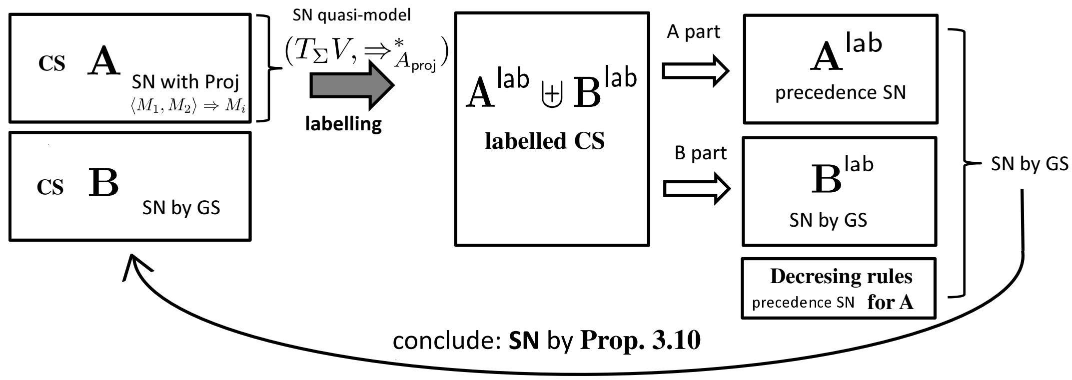

Figure 4: Proof strategy of Thm. 3.11

Note that it does not mean u is a pattern. The condition merely requires $\Sigma_{\mathbf{A}}$-headed sub-meta-terms of t to be patterns, and u may contain $\Sigma_{\mathbf{B}}$-symbols whose arguments need not to be patterns.

Note also that every sub-meta-term of a second-order pattern is again a second-order pattern. A bound variable (e.g. x) might become free in a sub-meta-term (e.g. $M[x]$) of a second-order pattern (e.g. $f(x.M[x])$), but x is "originally" a bound variable, therefore it is regarded as a bound variable in the condition of "distinct bound variables". So, $M[x]$ is a second-order pattern.

Remark 3.3. As a consequence of this assumption,

(1) the rhs of each rule in $\mathbf{A}$ is a second-order pattern, and

(2) $\Sigma_{\mathbf{A}}$-function symbols may appear in $\mathbf{B}$.

This kind of combination of two systems is called *shared constructors* [KO92, Gra94] (i.e. constructors Θ are shared in $\mathbf{A}$ and $\mathbf{B}$), and *hierarchical combination* [Rao95] (because the computation system $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ can involve function symbols defined in $\mathbf{A}$, but $(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A})$ cannot involve $\Sigma_{\mathbf{B}}$-symbols). The technical assumption (v) is needed to establish a variation of the HO semantic labelling method. Technically, it is essential to make Lemma B.2 and B.4 hold. In practice, this assumption admits effect handler examples (Section 4).

3.1. **Proof method.** The modular termination theorem we want to establish is of the form: under Assumption 3.1, if $\mathbf{A}$ is SN and $\mathbf{B}$ is SN (with some more conditions on both SN), then $\mathcal{C} = \mathbf{A} \uplus \mathbf{B}$ is SN. Since in general, the assumption of SN (of each of $\mathbf{A}$ and $\mathbf{B}$) is not necessarily established by GS, an idea to prove SN of $\mathcal{C}$ is

(i) to use the precedence termination (of $\mathbf{A}$ or $\mathbf{B}$, or both) by HO labelling [Ham07, Thm. 5.6], and

(ii) then to use GS to prove SN of the labelled $\mathcal{C}$.

Precedence termination is a notion of termination: if $\mathcal{C}$ is accessible and a well-founded relation $\succ$ exists such that for every rule $f(\overline{t}) \Rightarrow r \in \mathcal{C}$, every function symbol g in r, $f \succ g$ holds, then $\mathcal{C}$ is called precedence terminating, which implies SN.

But to apply HO semantic labelling, one must first seek a *well-founded quasi-model* of the whole system $\mathcal{C}$. A natural candidate of quasi-model for $\mathcal{C}$ is $(T_\Sigma \mathrm{V}, \Rightarrow^*_{\mathbf{A}\uplus\mathbf{B}})$, i.e., all terms with the many-step computation relation using $\mathbf{A}$ and $\mathbf{B}$, but proving that it is *well-founded* is difficult. In fact, it is nothing but the termination property (i.e. $\mathcal{C}$ is SN) we are seeking to prove. This means that the original HO semantic labelling is not quite appropriate. To overcome the difficulty, in this paper, we use a variation of the semantic labelling (Section B). Instead of requiring a quasi-model of the whole rules, we now require only a quasi-model of $\mathbf{A}$. This gives a well-founded quasi-model $(T_\Sigma \mathrm{V}, \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}})$ of $\mathbf{A}$ (see Fig. 4), where $\mathbf{A}_{\mathsf{Proj}}$ is an extension of $\mathbf{A}$ with projections of pairs $\langle M_1, M_2\rangle \Rightarrow M_i$ $(i = 1, 2)$ (Def. 3.4).

Attaching a $\Sigma_{\mathbf{A}}$-headed term to each $\Sigma_{\mathbf{A}}$-function symbol as labels in $\mathcal{C}$, we construct a labelled computation system $\mathbf{A}^{\mathsf{lab}} \uplus \mathbf{B}^{\mathsf{lab}}$. Then we try to prove SN of it by GS. To do so, we presuppose that $\mathbf{B}$ should be SN by GS. Then the $\mathbf{A}^{\mathsf{lab}}$-part is precedence terminating by labelling, and the $\mathbf{B}^{\mathsf{lab}}$-part is SN by GS by assumption.

Finally, a variation of the HO semantic labelling theorem (Prop. 3.10) ensures that SN of the labelled computation system $\mathbf{A}^{\mathsf{lab}} \uplus \mathbf{B}^{\mathsf{lab}}$ (with additional decreasing rules) implies SN of the original $\mathbf{A} \uplus \mathbf{B}$. This establishes a modular termination theorem (Thm. 3.11).

3.2. **Labelled rules.**

Definition 3.4 (Projection rules)**.** We extend the computation system $\mathbf{A}$ with pairs and the "projection rules" on the pairs. For every mol type $b \in \mathcal{T}$, we use pairing $\langle -, - \rangle_b$ and a bottom element $\perp_b$.

- $\Sigma_{\mathsf{Proj}} \triangleq \{\perp_b : b \mid b \in \mathcal{T}\} \uplus \{\langle -, -\rangle_b : b, b \to b \mid b \in \mathcal{T}\}$
- $\mathsf{Proj} \triangleq \{M_1, M_2 : b \triangleright\ \vdash \langle M_1, M_2\rangle_b \Rightarrow M_1 : b,$
  $M_1, M_2 : b \triangleright\ \vdash \langle M_1, M_2\rangle_b \Rightarrow M_2 : b \mid b \in \mathcal{T}\}.$

We define

$$\Sigma \triangleq \Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}, \ \ \mathbf{A}_{\mathsf{Proj}} \triangleq \mathbf{A} \uplus \mathsf{Proj}.$$

We construct a labelled $\mathcal{C}$. We label function symbols in $\Sigma_{\mathbf{A}}$, and do not label other function symbols, where labels are taken from $T_\Sigma \mathrm{V}$. We define the labelled signature by

$$\Sigma^{\mathsf{lab}} = \{f_u \mid f \in \Sigma_{\mathbf{A}},\ u \in T_\Sigma \mathrm{V}\} \cup \Sigma_{\mathbf{B}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}.$$

Labelled terms are constructed by the typing rules in Fig. 1 using Σ^{lab}, instead of Σ. The contextual set $M_{\Sigma^{\mathsf{lab}}} Z$ of labelled meta-terms is defined by collecting all labelled meta-terms.

Next we define a labelling map lab_φ that attachs labels to plain meta-terms using a function $\varphi^\sharp$ to calculate the labels.

Let $\varphi : Z \to T_\Sigma \mathrm{V}$ be an assignment. The *term labelling* lab_φ is defined using $\varphi^\sharp$, i.e., $\mathsf{lab}_\varphi : M_\Sigma Z \longrightarrow M_{\Sigma^{\mathsf{lab}}} Z$ is a map defined by

$$\begin{array}{lll}
\mathsf{lab}_\varphi(x) & = x & \\
\mathsf{lab}_\varphi(M[t_1, \ldots, t_n]) & = M[\mathsf{lab}_\varphi(t_1), \ldots, \mathsf{lab}_\varphi(t_n)] & \\
\mathsf{lab}_\varphi(f(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n)) & = f_{\varphi^\sharp(f(\overline{x_1}.t_1,\ldots,\overline{x_n}.t_n))}(\overline{x_1}.\mathsf{lab}_\varphi(t_1), \ldots, \overline{x_n}.\mathsf{lab}_\varphi(t_n)) & \text{if } f \in \Sigma_{\mathbf{A}} \\
\mathsf{lab}_\varphi(f(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n)) & = f(\overline{x_1}.\mathsf{lab}_\varphi(t_1), \ldots, \overline{x_n}.\mathsf{lab}_\varphi(t_n)) & \text{if } f \notin \Sigma_{\mathbf{A}}
\end{array}$$

We define labelled rules

$$\mathcal{C}^{\mathsf{lab}} \triangleq \{Z \ \triangleright \ \mathsf{lab}_\varphi(l) \Rightarrow \mathsf{lab}_\varphi(r) : b \ \mid Z \ \triangleright \ l \Rightarrow r : b \in \mathcal{C},\ \text{assignment } \varphi : Z \longrightarrow T_\Sigma \mathrm{V}\}$$

We only attach labels to $\Sigma_{\mathbf{A}}$-symbols.

Definition 3.5. The set of *decreasing rules* $\mathsf{Decr}(\mathbf{A})$ consists of rules

$$f_v(\overline{x_1}.M_1[\overline{x_1}],\ \ldots,\ \overline{x_n}.M_n[\overline{x_n}]) \ \Rightarrow\ f_w(\overline{x_1}.M_1[\overline{x_1}],\ \ldots,\ \overline{x_n}.M_n[\overline{x_n}])$$

for all $f \in \Sigma_{\mathbf{A}}$, $v, w \in T_\Sigma \mathrm{V}$ with all $v\ (\Rightarrow^+_{\mathbf{A}_{\mathsf{Proj}}} \cup \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \circ \rhd)\ w$. Here $\lhd$ is the strict subterm relation.

3.3. Traces and labelled systems.

Definition 3.6. We define a list-generating function list_b taking a finite set $\{t_1, \ldots, t_n\}$ of terms of type b and returning a tuple using the pairing as

$$\mathsf{list}_b(\varnothing) \triangleq \bot_b; \quad \mathsf{list}_b(S) \triangleq \langle\, t,\ \mathsf{list}_b(S - \{t\})\,\rangle_b$$

where we pick a term $t \in S$ by some fixed order on terms (such as lexicographic order of alphabetical order on symbols). This returns a tuple as $\mathsf{list}_b(\{t_1, \ldots, t_n\}) = \langle t_1, \langle t_2, \ldots, \langle t_n, \bot\rangle\rangle\rangle$. If S is an infinite set, $\mathsf{list}_b(S)$ is undefined. Hereafter, we will omit writing the type subscripts of $\bot, \langle -, - \rangle, \mathsf{list}$, which can be recovered from context.

Applying Proj rules, we have

$$\mathsf{list}(\{t_1, \ldots, t_n\}) = \langle t_1, \langle t_2, \ldots, \langle t_n, \bot\rangle\rangle\rangle \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} t_i,$$
$$\mathsf{list}(\{t_1, \ldots, t_n\}) = \langle t_1, \langle t_2, \ldots, \langle t_n, \bot\rangle\rangle\rangle \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \bot.$$

The reason why we need the pairs is to make a term constructed by a trace of reduction, i.e., by collecting terms appearing in a reduction. Projection rules are used to pick a term from such a trace.

We use another function $\mathsf{tr}(t)$ to calculate a label, which collects all the traces of reductions by $\mathcal{C}$ starting from a $\Sigma_{\mathbf{B}}$-headed term. The *trace map* $\mathsf{tr} : T_\Sigma \mathrm{V} \longrightarrow T_\Sigma \mathrm{V}$ is defined by

$$\begin{array}{ll} \mathsf{tr}(x) = x & \\ \mathsf{tr}(f(\overline{\overline{x}.t})) = f(\overline{x_1}.\mathsf{tr}(t_1), \ldots, \overline{x_n}.\mathsf{tr}(t_n)) & \text{if } f \notin \Sigma_{\mathbf{B}} \\ \mathsf{tr}(f(\overline{\overline{x}.t})) = \mathsf{list}(\{\,\mathsf{tr}(u) \mid f(\overline{\overline{x}.t}) \Rightarrow_{\mathcal{C}} u\,\}) & \text{if } f \in \Sigma_{\mathbf{B}} \end{array}$$

Remark 3.7. The Assumption 3.1 (iv) that $\mathcal{C}$ is finitely branching is necessary to ensure a finite term constructed by the second clause. If $f(\overline{\overline{x}.t})$ with $f \in \Sigma_{\mathbf{B}}$ is a normal form, then $\mathsf{tr}(f(\overline{\overline{x}.t})) = \bot$.

The notion of tr and the method of transforming a minimal non-terminating $\mathcal{C}$-reduction sequence to a reduction sequence using the projection rules has been used in proving modularity of termination [Gra94, Def. 3][Ohl94].

The *trace labelling* $\mathsf{labtr} : T_\Sigma \mathrm{V} \longrightarrow T_{\Sigma^{\mathsf{lab}}} \mathrm{V}$ is defined by $\mathsf{labtr} \triangleq \mathsf{lab}_{\mathsf{tr} \circ \psi}$ with the unique map $\psi : \varnothing \to T_\Sigma \mathrm{V}$. It satisfies:

$$\begin{array}{lll} \mathsf{labtr}(x) = x & & \\ \mathsf{labtr}(f(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n)) = f_{f(\overline{x_1}.\mathsf{tr}(t_1), \ldots, \overline{x_n}.\mathsf{tr}(t_n))}(\overline{x_1}.\mathsf{labtr}(t_1), \ldots, \overline{x_n}.\mathsf{labtr}(t_n)) & \text{if } f \in \Sigma_{\mathbf{A}} \\ \mathsf{labtr}(f(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n)) = f(\overline{x_1}.\mathsf{labtr}(t_1), \ldots, \overline{x_n}.\mathsf{labtr}(t_n)) & \text{if } f \notin \Sigma_{\mathbf{A}} \end{array}$$

Definition 3.8 (Labelled substitution for metavariables)**.** We define $t \bullet \rho = \mathsf{labtr}\, \| t\rho \|$. Given an assignment for labelled meta-terms $\theta : Z \longrightarrow T_{\Sigma^{\mathsf{lab}}} \mathrm{V}$, labelled substitution for

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

$$
\text{(Abs)}\frac{\Gamma, \overline{x : a} \vdash t \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} t' \;\; : b}{\Gamma \vdash \overline{x^a}.t \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} \overline{x^a}.t' \;\; : \overline{a} \to c}
$$

$$
\text{(Fun)}\frac{\begin{array}{c} f : (\overline{a_1} \to b_1), \cdots, (\overline{a_k} \to b_k) \to c \in \Sigma^{\mathsf{lab}} \qquad (f \text{ may be labelled}) \\ \Gamma \vdash \overline{x_i^{a_i}}.t_i \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} \overline{x_i^{a_i}}.t_i' \;\; : \overline{a_i} \to b_i \quad (\text{some single } i \text{ s.t. } 1 \le i \le k)\end{array}}{\Gamma \vdash f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i \ldots, \overline{x_k^{a_k}}.t_k) \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} f(\overline{x_1^{a_1}}.t_1, \ldots, \overline{x_i^{a_i}}.t_i' \ldots, \overline{x_k^{a_k}}.t_k) \;\; : c}
$$

Figure 5: Labelled reduction

metavariables $\theta^{\sharp} : M_{\Sigma^{\mathsf{lab}}} Z \longrightarrow T_{\Sigma^{\mathsf{lab}}} \mathrm{V}$ is defined by

$$
\begin{array}{rcl}
\theta^{\sharp}(x) & = & x \\
\theta^{\sharp}(f(\overline{y_1}.t_1, \ldots, \overline{y_m}.t_m)) & = & f(\overline{y_1}.\theta^{\sharp}(t_1), \ldots, \overline{y_m}.\theta^{\sharp}(t_m)) \quad \text{ if } f \in \Sigma^{\mathsf{lab}} \\
\theta^{\sharp}(M[\overline{t}])) & = & u \bullet \{\overline{x} \mapsto \overline{t}\} \quad \text{for } \theta : M \mapsto \overline{x}.u.
\end{array}
$$

Here, the term $\|t\|$ is obtained by deleting all labels in a labelled term t.

The labelled reduction on labelled terms $s \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} t$ is defined by the inference system given in Fig. 5.

We can also apply the GS defined in Section 2.6 to prove SN of labelled systems. Appendix A proves the validity.

Proposition 3.9. *Let* $(\Sigma^{\mathsf{lab}}, \mathcal{C}^{\mathsf{lab}})$ *be a labelled computation system. Suppose that* $\le_{\mathcal{T}}$ *and* $\le_{\Sigma^{\mathsf{lab}}}$ *are well-founded. If for all* $f(\overline{t}) \Rightarrow r \in \mathcal{C}^{\mathsf{lab}}$, $\mathsf{CC}_f(\overline{t}) \ni r$, *then* $\mathcal{C}^{\mathsf{lab}}$ *with* $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$ *is strongly normalising. Note that* f *may be labelled.*

The following is a variation of the higher-order semantic labelling explained in Section 2.7 and proved in Appendix B.

Proposition 3.10. *If* $\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})$ *is SN, then* $\mathcal{C}$ *is SN.*

3.4. **Proving a modular termination theorem.** Then we prove the main theorem. We say that $\mathbf{A}$ *is accessible* if for each $f(\overline{\overline{x}.t}) \Rightarrow r \in \mathbf{A}$, every metavariable occurring in r is accessible in some of $\overline{t}$.

Theorem 3.11 (Modular Termination)**.** *Let* $(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A})$ *and* $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ *be computation systems satisfying Assumption 3.1 and the following.*

(i) $\mathbf{A}$ *is accessible.*

(ii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}, \mathbf{A}_{\mathsf{Proj}})$ *is SN (not necessarily by GS).*

(iii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B})$ *is SN by the General Schema.*

Then $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta,\ \mathbf{A} \uplus \mathbf{B})$ *is SN.*

Proof. We show SN of $\mathcal{C} = \mathbf{A} \uplus \mathbf{B}$ by using Prop. 3.10. Suppose that $\mathbf{B}$ is SN by the General Schema with a well-founded order $>_{\Sigma_\mathbf{B}}$. We define an order on Σ by

$$h \quad >_\Sigma \quad k \quad >_\Sigma \quad f_v \quad >_\Sigma \quad f_w,\ g_w \quad >_\Sigma \quad c,\ \langle -,- \rangle_b,\ \bot_b \tag{3.1}$$

- for all $h, k \in \Sigma_\mathbf{B}$ with $h >_{\Sigma_\mathbf{B}} k$,
- for all $f, g \in \Sigma_\mathbf{A}$, $v, w \in T_\Sigma \mathrm{V}$ with $v\ (\Rightarrow^+_{\mathbf{A}_\mathsf{Proj}} \cup \Rightarrow^*_{\mathbf{A}_\mathsf{Proj}} \circ \rhd)\ w$ and f is the head of the lhs of a $\mathbf{A}$-rule and g appears in the corresponding rhs (possibly $g = f$),
- for all $c \in \Theta,\ b \in \mathcal{T}$.

The order $>_\Sigma$ is well-founded because $>_{\Sigma_\mathbf{B}}$ is well-founded. We show $\mathcal{C}^\mathsf{lab} \uplus \mathsf{Decr}(\mathbf{A}) = \mathbf{A}^\mathsf{lab} \uplus \mathbf{B}^\mathsf{lab} \uplus \mathsf{Decr}(\mathbf{A})$ satisfies GS with $>_\Sigma$.

1. $\mathbf{A}^\mathsf{lab}$ satisfies GS. Take a labelled rule $f_v(\overline{x.t}) \Rightarrow r \in \mathbf{A}^\mathsf{lab}$.
   We show that for all $r' \unlhd r$, $\mathsf{CC} \ni r'$ by induction on the structure of r'.
   - Case $r' = M[\overline{x}]$. By assumption, M is accessible in some of $\bar{t}$. Variables $\overline{x} \in \mathsf{CC}$. Therefore, **(meta** M**)** is applied, we have $M[\overline{x}] \in \mathsf{CC}$.
   - Case r' is a variable x. Then $\mathsf{CC} \ni x$.
   - Case $r' = g_w(\overline{y.s})$ with $g \in \Sigma_\mathbf{A}$ (possibly $g = f$). Since r' is a sub-meta-term of the rhs r of a $\mathbf{A}^\mathsf{lab}$-rule,
   $$v = \varphi^\sharp(f(\overline{x.\,\|t\|}))\ \ (\Rightarrow_{\mathbf{A}_\mathsf{Proj}} \cup \Rightarrow_{\mathbf{A}_\mathsf{Proj}} \circ \rhd)\ \ \varphi^\sharp(g(\overline{y.\,\|s\|})) = w$$
   where the term $\|t\|$ is obtained by deleting all labels in a labelled term t. Then we have $f_v >_\Sigma g_w$. By I.H., $\mathsf{CC} \ni \overline{s}$. Applying **(fun** $f_v >_\Sigma g_w$**)**, $\mathsf{CC} \ni r'$.
   - Case $r' = c(\overline{y.s})$ with $c \in \Theta$. Then $f_v >_\Sigma c$. By I.H., $\mathsf{CC} \ni \overline{s}$, hence **(fun** $f_v >_\Sigma c$**)** is applied, $\mathsf{CC} \ni r'$.

   Therefore, $\mathsf{CC} \ni r$.
2. $\mathbf{B}^\mathsf{lab}$ satisfies GS with $>_\Sigma$. In $\mathbf{B}^\mathsf{lab}$, any $\Sigma_\mathbf{B}$-symbol is not labelled and any labelled $\Sigma_\mathbf{A}$-symbol is smaller (w.r.t. $<_\Sigma$) than a $\Sigma_\mathbf{B}$-symbol. These facts and the assumptions (ii) and (iii) imply the desired result.
3. $\mathsf{Decr}(\mathbf{A})$ satisfies GS: We need to show that each rule
   $$f_v(\overline{x_1}.M_1[\overline{x_1}], \ldots, \overline{x_n}.M_n[\overline{x_n}]) \ \Rightarrow\ f_w(\overline{x_1}.M_1[\overline{x_1}], \ldots, \overline{x_n}.M_n[\overline{x_n}])$$
   with $v\ (\Rightarrow^+_{\mathbf{A}_\mathsf{Proj}} \cup \Rightarrow^*_{\mathbf{A}_\mathsf{Proj}} \circ \rhd)\ w$ satisfies GS. This holds because GS checks the root symbols by **(fun** $f_v >_\Sigma g_w$**)** for $f, g \in \Sigma_\mathbf{A}$, and each metavariable is accessible.

Therefore, $\mathcal{C}^\mathsf{lab} \uplus \mathsf{Decr}(\mathbf{A})$ satisfies GS, hence SN by Proposition 3.9. Finally, applying Prop. 3.10, we conclude that $\mathcal{C}$ is SN. □

3.5. **Variations.**

3.5.1. *GS with other term orders.* In [Bla16], a general and comprehensive account of GS is presented, where a preorder on function symbols and a preorder to compare the arguments of equivalent function symbols are abstracted to the notion of *valid (status) $\mathcal{F}$-quasi-ordering* [Bla16, Def.7,9]. GS reviewed in Section 2.6 is merely one instance of it. Importantly, Thm. 3.11 holds for any variation of GS because the proof does not depended on any particular valid $\mathcal{F}$-quasi-ordering. These changes can enhance applicability of the theorem.

Corollary 3.12. *Thm. 3.11 holds by changing the General Schema to another version of the General Schema using a different valid $\mathcal{F}$-quasi-ordering. For instance, the use of covered-subterm ordering* [BJO02, Def.5]*, or structural subterm ordering* [Bla16, Def.13] *is possible.*

For instance, GS with the structural subterm ordering can check SN of a rule

```
lim(x.M[x])+Y ⇒ lim(x.M[x]+Y)
```

[Bla05, Sec. 4.6] for addition of ordinals, while the stable subterm ordering cannot. Changing the General Schema, $\mathcal{C}$ can involve this kind of rules.

Similarly, the General Schema might be changed to the computability path ordering (CPO) with accessible subterms [BJR15, Sec.7.2] because the proof of Thm. 3.11 only uses the features of GS on comparison of function symbols and accessible variables, and CPO with accessible subterms also has them. We will pursue these directions elsewhere and will clarify detailed conditions on these variations.

3.5.2. *Polymorphic and dependently typed cases.* Furthermore, changing the General Schema to GS defined in [Bla05], and generalising the computation systems to the polymorphic computation systems developed in [Ham18], we can obtain a polymorphic version of Thm. 3.11. This setting is more natural than the present molecular typed rules (see the discussion on difference between molecular types and polymorphic types in [Ham18]) in practical examples because we do not need to check parameterised instances of function symbols and rules as in Example 2.6. But since in the framework of [Bla05], the lhss of rules cannot involve binders or HO patterns, we need to restrict the form of lhss to apply this version of GS. For the dependently typed case, the recent development [BGH19] will also be useful.

4. Application 1: Algebraic Effect Handlers and Effect Theory

As an application, we demonstrate that our theorem is useful to prove the termination of a calculus with algebraic effects. The background of this section is as follows. Plotkin and Power introduced the *algebraic theory of effects* to axiomatise various computational effects [PP02] and they correspond to computational monads of Moggi [Mog88]. Plotkin and Pretnar formulated *algebraic effects and handlers* [PP13], which provided an alternative to monads as a basis for effectful programming across a variety of functional programming languages.

We prove termination of an effectful λ-calculus with effect handlers that respects an effect algebraic theory. First, we formulate the *multi-adjunctive metalanguage* (MAM), for an effectful λ-calculus (Section 4.2). Secondly, we provide an effect theory (Section 4.3). Thirdly, we give an effect handler (Section 4.4). Two novelties arise in this section.

(i) We use a single framework to formalise MAM, an effectful handler, and an effect theory: they form second-order computation systems.

(ii) We prove the termination of the combination of these by the modular termination theorem: Thm. 3.11.

4.1. **Type constructors.** We first define type constructors for an effectful calculus.

- atomic type `Unit` (for unit type ())
- binary `Pair` (for product type $a_1 \times a_2$)
- binary `Sum` (for sum type $a_1 + a_2$)
- binary `CPair` (for product of computation types $a_1 \& a_2$)
- unary `U` (for thunked type $U_E\ a$)
- unary `F` (for computation type $F\ a$)
- binary `Arr` (for arrow type $a_1 \to a_2$)

4.2. **A calculus for algebraic effects.** We use the core calculus MAM for effectful computation given in [FKLP19], which is an extension of Levy's call-by-push-value (CBPV) calculus [Lev06]. We formulate MAM as a second-order computation system ($\Sigma_{\text{MAM}} \uplus \Theta$, `MAM`). The signature Σ_{MAM} consists of the following defined function symbols

```
bang  : U(c) → c
caseP : Pair(a1,a2),(a1,a2 → c) → c
case  : Sum(a1,a2),(a1 → c),(a2 → c) → c
let : F(a),(a → c) → c
app : Arr(a,c),a → c
prj1 : CPair(c1,c2) → c1     ; prj2 : CPair(c1,c2) → c2
```

and the set of constructors Θ consists of

```
unit  : Unit              ; pair : a1,a2 → Pair(a1,a2)
inj1  : a1 → Sum(a1,a2) ; inj2 : a2 → Sum(a1,a2)
cpair : c1,c2 → CPair(c1,c2)
thunk : c → U(c)
return : a → F(a)
lam : (a → b) → Arr(a,b)
```

The set `MAM` of MAM's computation rules is given by[2]

```
(beta)  lam(x.M[x])@V                    ⇒ M[V]
(u)     bang(thunk(M))                   ⇒ M
(prod1) prj1(cpair(M1,M2))               ⇒ M1
(prod2) prj2(cpair(M1,M2))               ⇒ M2
(caseP) caseP(pair(V1,V2),x1.x2.M[x1,x2]) ⇒ M[V1,V2]
(case1) case(inj1(V),x.M1[x],y.M2[y])    ⇒ M1[V]
(case2) case(inj2(V),x.M1[x],y.M2[y])    ⇒ M2[V]
(f)     let(return(V),x.M[x])            ⇒ M[V]
```

Using this formulation, SN of `MAM` is immediate because it satisfies GS. More precisely, every rhs of `MAM` involves no function symbol and every metavariable in `MAM` is accessible. The original paper proposing MAM [FKLP19, Thm. 2] described a sketch of proof of SN of MAM, but the details were omitted. Our formulation by a computation system is generic and the system SOL automatically proves SN of `0Ex52_MAM.hs` in arXiv'20 of the SOL web interface [Ham20].

Note that SOL merely checks SN of the finite number of function symbols and rules defined in the file `0Ex52_MAM.hs`. To conclude actual SN of the computation system MAM, we use meta-theoretic reasoning. Looking at the output of checking process of GS, we see that all the accessibility conditions are satisfied even if we replace the type letters `a,a1,a2,c,c1,c2`

[2] `t@s` is the abbreviation of `app(t,s)`.

used in the signature with concrete mol types generated by the type constructors defined in Section 4.1, because there are no constructors violating the positivity condition (written as "`is positive`" in SOL's output) (a3) of the accessibility predicate in Def. 2.7. Therefore, the infinite number of MAM' computation rules obtained by instantiating the type letters `a,a1,a2,c,c1,c2` with concrete types satisfy GS.

4.3. **Effect theory.** Next we extend `MAM` to have an effect handler with effect theory. To keep the discussion simple, we consider a particular theory, i.e. the theory of global state [PP02, Sta09] for a single location. We take the type `N` of natural numbers for the state. We define the signature Σ_{Gl} by

```
get : (N → F(N)) → F(N)
put : N,F(N) → F(N)
sub : (N → F(N)), N → F(N)
```

It consists of the operations `get(v.t)` (looking-up the state, binding the value to `v`, and continuing `t`) and `put(v,t)` (updating the state to `v` and continuing `t`), and the substitution operation `sub(x.t,s)` that replaces `x` in `t` with `s`. The theory of global state [PP02, FS14] can be stated as a computation system ($\Sigma_{\mathrm{Gl}} \uplus \{\texttt{return}\}, \texttt{gstate}$) defined by

```
(lu)   get(v.put(v,X))          ⇒ X
(ll)   get(w.get(v.X[v,w]))     ⇒ get(v.X[v,v])
(uu)   put(V,put(W,X))          ⇒ put(W,X)
(ul)   put(V,get(w.X[w]))       ⇒ put(V,sub(w.X[w],V))
(sub1) sub(x.return(x),K)       ⇒ return(K)
(sub2) sub(x.M,K)               ⇒ M
(sub3) sub(x.get(v.M[v,x]),K)   ⇒ get(v.sub(x.M[v,x],K))
(sub4) sub(x.put(V,M[x]),K)     ⇒ put(V,sub(x.M[x],K))
```

These axioms have intuitive reading. For example, the axiom `(lu)` says that looking-up the state, binding the value to `v`, then updating the state to `v`, is equivalent to doing nothing. The axiom `(ul)` says that updating the state to `V`, then looking-up and continuing `X` with the looked-up value, is equivalent to updating the state to `V` and continuing `X` with `V`.

Plotkin and Power showed that the monad corresponding to the theory of global state (of finitely many locations) is the state monad [PP02].

Crucially, ($\Sigma_{\mathrm{Gl}} \uplus \{\texttt{return}\}, \texttt{gstate}$) does not satisfy GS. The General Schema checks that a recursive call at the rhs must be with a strict sub-meta-term of an argument at the lhs. In case of `(ll)`, the recursive call of `get` happens with `v.X[v,v]`, which is not a sub-meta-term of `w.get(v.X[v,w])` at the lhs. Moreover, `(ul)` requires a precedence `put` $>$ `sub` while `(sub4)` requires `sub` $>$ `put`, which violates well-foundedness.

Using a different method, the computation system ($\Sigma_{\mathrm{Gl}} \uplus \{\texttt{return}\}, \texttt{gstate}$) is shown to be SN. We count the number of symbols `get,put,sub` using the weights defined by

$$[\![\texttt{put}]\!](v,x) = x+1 \quad [\![\texttt{get}]\!](x) = 2x+2 \quad [\![\texttt{sub}]\!](x,v) = 2x+1$$

In each rule, the weights are decreasing such as

$$\begin{array}{ll} \texttt{(ul)} & (2x+2)+1 > (2x+1)+1 \\ \texttt{(sub4)} & 2(x+1)+1 > (2x+1)+1 \end{array}$$

therefore it is SN. Note that since $\texttt{N} \neq \texttt{F(N)}$ and there is no function symbol of type $\texttt{F(N)} \to \texttt{N}$ in Σ_{Gl}, the function symbols `get,put,sub` cannot occur in the first argument of `put` and the second argument of `sub` as an instance of the argument `V`. Therefore, the parameter v is not used in the weights.

4.4. **Effect handler.** *An effect handler* provides an implementation of effects by interpreting algebraic effects as actual effects. The handler [KLO13, FKLP19] for effect terms for global states can be formulated as a computation system ($\Sigma_{\mathrm{Gl}} \uplus \Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}}$, `Handle`) as follows.

```
handler : (N → F(N)), (Arr(N,F(N)) → F(N)), (N,F(N) → F(N)), F(N) → F(N)

(h_r) handler(RET,GET,PUT,return(X))  ⇒ RET[X]
(h_g) handler(RET,GET,PUT,get(x.M[x]))⇒ GET[  lam(x.handler(RET,GET,PUT,M[x]))]
(h_p) handler(RET,GET,PUT,put(P,M))   ⇒ PUT[P,lam(x.handler(RET,GET,PUT,M))]
```

Note that `RET`,`GET`,`PUT` are metavariables. For brevity, the first three arguments of `handler` are abbreviated, and they are formally η-expanded forms:

`handler(y.RET[y],k.GET[k],p.k.PUT[p,k],···).`

We consider a standard interpretation of global states using parameter-passing [PP13, KLO13, FKLP19]. Taking states and values to be `N`, we have the computation type `F(N)` = `Arr(N,N)`. Then running the handler can be given by

```
runState : Arr(N,N) → Arr(N,N)
(run) runState(t) ⇒ handler(y.lam(z.y), k.lam(n.(k@n)@n), p.k.lam(n.k@p), t)
```

This interprets `return`,`get`,`put` in an effect term `t` as the corresponding arguments of `handler`, which are interpretation of global state in the state-passing style.

The computation system ($\Sigma_{\mathrm{Gl}} \uplus \Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}}$, `Handle`) is immediately shown to be SN by GS. SOL automatically proves it (try `0Ex54_Handle.hs` in arXiv′20 of the SOL web interface [Ham20]). From it, we can conclude actual SN of `Handle` by using meta-theoretic reasoning as mentioned in Section 4.2.

4.5. **Proof of SN.** An effect term expresses an effectful program. For example,

$$tm \triangleq \texttt{get(x.put(inc(x),get(y.put(y,get(z.return(z))))))}$$

expresses the imperative program [KLO13]

```
x := get; put inc(x); y:= get; put y; z := get; return z
```

where $\texttt{inc} : \texttt{N} \to \texttt{N} \in \Sigma_{\mathrm{Handle}}$ is intended to be the increment operation. If the initial store is set to `0`, then clearly this program returns `inc(0)`. Formally, it is computed using the term tm with the handler as

```
runState(tm)@0
⇒ handler(y.lam(z.y), k.lam(n.(k@n)@n), p.k.lam(n.k@p),tm)@0
⇒* lam(z.inc(z))@0 ⇒ inc(0)
```

This can be computed using `MAM` ⊎ `Handle`, or `gstate` ⊎ `MAM` ⊎ `Handle`. The latter is more efficient than the former, because `gstate` expresses program equivalences and the application of them to an effect term optimizes the program.

Now, we consider the main problem of this section: SN of the whole computation system

$$(\Sigma_{\mathrm{Gl}} \uplus \Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}} \uplus \Theta,\ \texttt{gstate} \uplus \texttt{MAM} \uplus \texttt{Handle}) \tag{4.1}$$

The General Schema does not work to show SN of it because `gstate` does not satisfy GS. Therefore, we divide it into

$$(\Sigma_{\mathbf{A}} \uplus \Theta, \mathbf{A}) = (\Sigma_{\mathrm{Gl}} \uplus \Theta,\ \mathtt{gstate}),$$
$$(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta, \mathbf{B}) = (\Sigma_{\mathrm{Gl}} \uplus (\Sigma_{\mathrm{MAM}} \uplus \Sigma_{\mathrm{Handle}}) \uplus \Theta,\ \mathtt{MAM} \uplus \mathtt{Handle}).$$

The computation system (4.1) is not a disjoint union of $\mathbf{A}$ and $\mathbf{B}$, and is actually a *hierarchical combination* (cf. Remark 3.3) that *shares constructors* Θ. The lhss of `Handle` ($\subseteq \mathbf{B}$) involve defined function symbols `get`,`put` in $\Sigma_{\mathbf{A}}$.

We apply the modularity Thm. 3.11. Assumption 3.1 is satisfied because the computation system (4.1) is finitely branching and satisfies the the $\Sigma_{\mathbf{A}}$-layer condition. We check the assumptions. We define the well-founded order on types by

$$T(a_1, \ldots, a, \ldots, a_n) >_{\mathcal{T}} a$$

for every n-ary type constructor T, and every type a, where the lhs's a is placed at the i-th argument of T for every $i = 1, \ldots, n$.

(i) $\mathbf{A} =$ `gstate` is accessible. This is immediate because the crucial case (a3) in Def. 2.7 checks the type comparison for the arguments of `get`,`put`,`return`, which holds by $\mathtt{N} <_{\mathcal{T}} \mathtt{F(N)}$.

(ii) $(\Sigma_{\mathbf{A}} \uplus \Theta \uplus \Sigma_{\mathsf{Proj}}, \mathtt{gstate} \uplus \mathsf{Proj})$ is SN. This is again established by the weights given in Section 4.3.

(iii) $(\Sigma_{\mathbf{A}} \uplus \Sigma_{\mathbf{B}} \uplus \Theta,\ \mathtt{MAM} \uplus \mathtt{Handle})$ is SN by GS. This is immediate by applying GS with the precedence

$$\mathtt{handler} >_{\Sigma} \mathtt{lam}$$

to the computation system. The rhss of `MAM` involve no function symbols and the rhss of `Handle` involve `handler`,`lam`. To check that each recursive call of `handler` happens with a smaller argument, here we use the structural subterm ordering [Bla16, Def.13] to establish that `M[x]` is smaller than `get(x.M[x])`. Every metavariable is accessible.

Hence we conclude that the computation system (4.1) is SN.

This termination result is general. Although in this section, we consider a particular handler for `get` and `put`, any effect handler is shown to be SN by GS along this way. MAM is SN regardless of used effects. To the best of our knowledge, this is the first report on how to prove the termination of the combination of an effectful λ-calculus, an effect handler and effect theory.

5. Application 2: Splitting a System into FO and HO parts

If a computation system contains first-order computation rules, it is often useful to split the system into the higher-order part and the first-order part to which we can apply a powerful termination checker for first-order term rewriting systems, such as AProVE [GAB+17]. The Modular Termination Theorem (Thm. 3.11) can be used for this purpose.

A FO computation system $(\Sigma, \mathcal{C})$ is a computation system if the type of every function symbol $f \in \mathcal{C}$ is of the form $f : a_1, \ldots, a_n \to b$, where a_i's and b are mol types. Suppose that a computation system $\mathcal{C}$ can be split as $\mathcal{C} = \mathbf{A} \uplus \mathbf{B}$, where $\mathbf{A}$ is a set of FO rules without $\Sigma_{\mathbf{B}}$-symbols, $\mathbf{B}$ is a set of second-order rules and Assumption 3.1 is satisfied. For every FO computation system, the rhss of rules are second-order patterns because no meta-application

The table shows the number of judged results by each tool. The result YES/NO is a judged result of SN. MAYBE is a result when the tool cannot judge. The score of a tool is the number of judged answers. (SOL is named as "sol 37957")

Result	SOL	WANDA	SizeChangeTool
YES	199	151	93
NO	20	14	0
MAYBE	37	28	163
error	5	66	3
timeout	0	2	2
Score	219	165	93

Score: `http://group-mmm.org/termination/competitions/Y2018/`
Details: `http://group-mmm.org/termination/competitions/Y2018/caches/termination_30047.html`
Termination Competition Official:
`http://termination-portal.org/wiki/Termination_Competition_2018`
and also proceed [Status]-[Results]

Note that there is one conflict of judgement in the problem `Hamana_17/restriction.xml` (which swaps two `new`-binders and is a problem the present author submitted) due to different interpretations of the higher-order rule format among the tools.

Figure 6: Results and scores in Termination Competition 2018

exists in the arguments of a function symbol. Hence if the FO computation system $\mathbf{A}$ is accessible, $\mathbf{A}$ with Proj is SN (using any method), and $\mathbf{B}$ is SN by GS, then we can conclude that $\mathbf{A} \uplus \mathbf{B}$ is SN by Thm. 3.11.

5.1. **Implementation.** We have implemented this FO splitting method by extending the tool SOL [Ham17]. We will also refer to this extended version as SOL. The system SOL consists of about 8000 line Haskell codes. The web interface of SOL is available at the author's homepage.

5.2. **Benchmark.** The Termination Problem Database (TPDB)[3] stores a collection of various rewrite systems for termination. In the TPDB, "`Applicative_first_order`" problems and some other problems in the higher-order category contain such examples, which are a mixture of difficult FO systems and HO systems. To show effectiveness of this method, we selected 62 problems from TPDB and did a benchmark to solve these problems by the previously proposed system SOL (2017) [Ham17] and the extended SOL (current version) for comparison. We conducted the benchmark on a machine with Intel(R) Xeon E7-4809, 2.00GHz 4CPU (8core each), 256GB memory, Red Hat Enterprise Linux 7.3, and set timeout 400 seconds.

SOL (2017) solved 31 problems, and SOL (current version) solved 57 problems out of 62, which clearly improves SOL (2017). SOL (current version) could solve 26 more problems than SOL (2017). The output details are available[4] and shown in Table 7 in the Appendix, where the use of a modularity method is indicated at the final column.

[3] `http://termination-portal.org/wiki/TPDB`
[4] `http://solweb.mydns.jp/bench/`

5.3. **Example.** We show one of these problems: `Applicative_05__mapDivMinusHard`.

```
(1)  map(x.F[x],nil)        ⇒ nil
(2)  map(x.Z[x],cons(U,V))⇒ cons(Z[U],map(x.Z[x],V))
(3)  minus(W,0)             ⇒ W
(4)  minus(s(P),s(X1))      ⇒ minus(p(s(P)),p(s(X1)))
(5)  p(s(Y1))               ⇒ Y1
(6)  div(0,s(U1))           ⇒ 0
(7)  div(s(V1),s(W1))       ⇒ s(div(minus(V1,W1),s(W1)))
```

This `mapDivMinusHard` does not satisfy GS because of (7), which is not structural recursive. SOL splits it into the FO part $\mathbf{A} = \{$(3)-(7)$\}$ and HO part $\mathbf{B} = \{$(1)-(2)$\}$. Then $\mathbf{A}$ with Proj can be proved to be SN by an external FO termination checker and $\mathbf{B}$ satisfies GS. Then SOL concludes that `mapDivMinusHard` is SN by Thm. 3.11.

As a more comprehensive evaluation, SOL participated to the higher-order union beta category of the International Termination Competition 2018 held at the Federated Logic Conference (FLoC 2018) in Oxford, U.K. This event was a competition for the number of checked results of termination problems by automatic tools. In the higher-order category, 263 problems of higher-order rewrite systems taken from TPDB were given. Three tools participated in the higher-order category: WANDA [Kop12], SizeChangeTool [BGH19], and SOL. SOL judged the greatest number of problems among three tools as shown in Fig. 6. A main reason derives from the fact that SOL has a modular SN checking method based on Theorem 3.11.

6. Related Work and Summary

6.1. **Related work.** A rich body of work describes termination and modularity of first-order and higher-order rewrite systems including [Zan95, Ohl94, AG00, GAO02, Bla00, BJO02, JR07, Kus18, Kop12, BG18, Bla16, Toy81, Ohl94, GAO02, Gra94, BFG97]. The modular termination theorem of the form of Thm. 3.11, i.e., the combination of (restricted classes of) two second-order computation systems with shared constructors, has not been reported in the literature to date. [FK11] provides a dependency pair method to split a HO termination problem into FO and HO parts, which might be regarded as a modularity of SN for the combination of FO and HO rewrite systems. An important difference is that our result covers strictly more than the combination of FO and HO systems as described in Section 5.

Originally, our proof of modularity was inspired by a method of proving modularity of SN for a rewrite system $\mathcal{R}$ and a recursive program scheme $\mathcal{P}$ by semantic labelling ([MOZ96] for the FO case, and [Ham07] for the second-order case). In this method, the normal forms computed by $\mathcal{P}$ are taken as the "semantics", and are used as labels to show termination of $\mathcal{R} \uplus \mathcal{P}$, where SN of $\mathcal{P}$ is crucially important to determine the semantics.

For the present work, we chose $\Sigma_{\mathbf{A}}$-terms computed by a $\Sigma_{\mathbf{A}}$-substitution $\varphi^\sharp$ as the "semantics". Also, they are used as labels to show termination of $\mathbf{A} \uplus \mathbf{B}$, where SN of $\mathbf{A}$ is crucially important to determine the semantics[5].

The dependency pair (DP) method is a successful method of proving termination of various first-order [Art96, AG00] and higher-order rewrite systems [Kus18, Kop12, BGH19]. One may find that several similarities exist between the DP method and the present proof

[5]It forms a quasi-model of $\mathbf{A}$ consisting of Σ-terms with a well-founded preorder ($\Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \circ \trianglerighteq$).

strategy. A reason might be that the DP method is a general method to prove termination by conducting modular analysis of the corresponding dependency pairs [GAO02].

More fundamentally, a similarity between DP and the present work can be found at the foundational level. At the early stage [Art96, AG96], the DP technique was *based on semantic labelling*, where the normal forms by a ground-convergent rewrite system $\mathcal{E}$[6] are taken as the "semantics", and are used as labels of rules to establish the DP method. The ground convergence of $\mathcal{E}$ is crucially important to determine the semantics. In the later refined DP method, the use of that semantic labelling was dropped [AG00]. But this fact illustrates that semantic labelling is a natural starting point to tackle the modularity problem of termination.

Our modularity result and its proof might be reformulated using the DP method, where the $\mathbf{A}_{\mathsf{Proj}}$-part might be regarded as usable rules. However, the static DP method [Kus18] might be insufficient to simulate our theorem completely. A mismatch seems to exist between SN of higher-order rules and non-loopingness of the corresponding dependency pairs. For example, while the second-order computation rule $\mathtt{f(0)}\Rightarrow\mathtt{g(x.f(x))}$ is terminating (and accessible, the lhs is a pattern), the corresponding dependency pair $\mathtt{f(0)}\rightarrow\mathtt{f(x)}$ is looping. Therefore, one cannot replace our assumption of SN of $\mathbf{A}_{\mathsf{Proj}}$ completely with non-loopingness of the corresponding DPs. But employing some ideas of the DP method might increase the power of Thm. 3.11, which is left as a future work.

6.2. **Summary.** We have presented a new modular proof method of termination for second-order computation. The proof method is useful for proving termination of higher-order foundational calculi. To establish the method, we have used a version of the semantic labelling translation and Blanqui's General Schema: a syntactic criterion of strong normalisation. As an application, we have applied this method to show termination of a variant of call-by-push-value calculus with algebraic effects, an effect handler and effect theory. We have also shown that our tool SOL is effective to solve higher-order termination problems.

References

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

[AG96] T. Arts and J. Giesl. Termination of constructor systems. In *Proc. of RTA'96*, pages 63–77, 1996.

[AG00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[Art96] T. Arts. Termination by absence of infinite chains of dependency pairs. In *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*, pages 196–210, 1996.

[BFG97] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalization in the algebraic-lambda-cube. *J. Funct. Program.*, 7(6):613–660, 1997.

[BG18] F. Blanqui and G. Genestier. Termination of $\lambda\pi$ modulo rewriting using the size-change principle. In *Proc. 16th International Workshop on Termination, WST 2018*, pages 10–14, 2018.

[BGH19] F. Blanqui, G. Genestier, and O. Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, 2019.

[BJO02] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive data type systems. *Theoretical Computer Science*, 272:41–68, 2002.

[6] A target rewrite system $\mathcal{R}$ to prove SN is assumed to be contained in $\mathcal{E}$.

[BJR15] F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015. URL: `https://doi.org/10.2168/LMCS-11(4:3)2015`, `doi:10.2168/LMCS-11(4:3)2015`.

[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.

[Bla05] F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

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

[BTG91] Val Breazu-Tannen and Jean H. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theor. Comput. Sci.*, 83(1):3–28, 1991.

[CKJ05] M. M. T. Chakravarty, G. Keller, and S. L. Peyton Jones. Associated type synonyms. In *Proc. of ICFP'05*, pages 241–253, 2005.

[CKJM05] M. M. T. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. of POPL'05*, pages 1–13, 2005.

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proc. of the 3rd Work. on Types for Proofs and Programs*, 1992.

[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.

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

[FK11] C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS 2011*, pages 147–162, 2011.

[FKLP19] Y. Forster, O. Kammar, S. Lindley, and M. Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.*, 29:e15, 2019.

[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 of CSL-LICS'14*, pages 41:1–41:10, 2014.

[GAB+17] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *J. Autom. Reasoning*, 58(1):3–31, 2017. `doi:10.1007/s10817-016-9388-y`.

[GAO02] J. Giesl, T. Arts, and Enno Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34(1):21–58, 2002.

[Gra94] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Appl. Algebra Eng. Commun. Comput.*, 5:131–158, 1994.

[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.

[Ham10] M. Hamana. Semantic labelling for proving termination of combinatory reduction systems. In *Proc. WFLP'09*, LNCS 5979, pages 62–78, 2010.

[Ham17] 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. SOL, Second-Order Laboratory, 2020. available at `http://www.cs.gunma-u.ac.jp/hamana/sol/`.

[JR07] J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *Journal of ACM*, 54(1):2:1–2:48, 2007.

[JR15] J.-P. Jouannaud and A. Rubio. Normal higher-order termination. *ACM Trans. Comput. Log.*, 16(2):13:1–13:38, 2015.

[JTH01] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, 2001.

[KLO13] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013.

[KO92] M. Kurihara and A. Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theor. Comput. Sci.*, 103(2):273–282, 1992.

[Kop12] C. Kop. *Higher Order Termination*. PhD thesis, Vrije Universiteit, Amsterdam, 2012.

[Kus18] K. Kusakari. Static dependency pair method in functional programs. *IEICE Transactions*, 101-D(6):1491–1502, 2018.

[Lev06] P. B. Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.*, 19(4):377–414, 2006.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mog88] E. Moggi. Computational lambda-calculus and monads. LFCS ECS-LFCS-88-66, University of Edinburgh, 1988.

[MOZ96] A. Middeldorp, H. Ohsaki, and H. Zantema. Transforming termination by self-labelling. In *Proceedings of the 13th International Conference on Automated Deduction*, LNCS 1104, pages 373–386, 1996.

[Ohl94] E. Ohlebusch. On the modularity of termination of term rewriting systems. *Theor. Comput. Sci.*, 136(2):333–360, 1994.

[Oka89] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proc. of ISSAC'89*, pages 357–363. ACM Press, 1989.

[Pol94] 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.

[Pol96] J. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, Universiteit Utrecht, 1996.

[PP02] G. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS'02*, pages 342–356, 2002.

[PP13] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. `doi:10.2168/LMCS-9(4:23)2013`.

[Rao95] M. R. K. Krishna Rao. Modular proofs for completeness of hierarchical term rewriting systems. *Theor. Comput. Sci.*, 151(2):487–512, 1995.

[Sta09] 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. `doi:10.1016/j.entcs.2009.07.103`.

[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.

[TKB95] Y. Toyama, J. W. Klop, and H. P. Barendregt. Termination for direct sums of left-linear complete term rewriting systems. *Journal of the ACM*, 42(6):1275–1304, 1995.

[Toy81] Y. Toyama. On the Church-Rosser property of term rewriting systems. Technical report, NTT, ECL, 1981. Technical Report 17672, in Japanese.

[Toy87] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Inf. Process. Lett.*, 25(3):141–143, 1987.

[Zan95] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1/2):89–105, 1995.

Problem	SOL(2017)	(sec.)	SOL	(sec.)	Used method
Applicative_05__Ex3Lists	YES	0.021	YES	0.022	
Applicative_05__Ex4MapList	YES	0.019	YES	0.019	
Applicative_05__Ex5Folding	YES	0.025	YES	0.022	
Applicative_05__Ex5Sorting	YES	0.025	YES	0.025	
Applicative_05__Ex6Folding	YES	0.024	YES	0.018	
Applicative_05__Ex6Recursor	YES	0.019	YES	0.018	
Applicative_05__Ex7Sorting	YES	0.024	YES	0.025	
Applicative_05__Ex7_9	MAYBE	0.025	MAYBE	0.025	
Applicative_05__Ex9Maps	YES	0.022	YES	0.019	
Applicative_05__Hamming	MAYBE	0.035	NO	0.044	
Applicative_05__mapDivMinus	MAYBE	0.027	YES	0.101	modularity
Applicative_05__mapDivMinusHard	MAYBE	0.022	YES	0.12	modularity
Applicative_05__termMonTypes	MAYBE	0.021	NO	0.016	
Applicative_first_order_05__#3.16	YES	0.023	YES	0.025	
Applicative_first_order_05__#3.18	MAYBE	0.03	YES	0.142	modularity
Applicative_first_order_05__#3.2	MAYBE	0.023	YES	0.102	modularity
Applicative_first_order_05__#3.22	MAYBE	0.022	YES	0.107	modularity
Applicative_first_order_05__#3.25	MAYBE	0.02	YES	0.114	modularity
Applicative_first_order_05__#3.27	YES	0.022	YES	0.021	
Applicative_first_order_05__#3.32	MAYBE	0.021	YES	0.095	modularity
Applicative_first_order_05__#3.36	YES	0.025	YES	0.026	
Applicative_first_order_05__#3.38	YES	0.028	YES	0.035	
Applicative_first_order_05__#3.40	MAYBE	0.025	YES	5.061	
Applicative_first_order_05__#3.45	MAYBE	0.022	YES	0.099	modularity
Applicative_first_order_05__#3.48	MAYBE	0.025	YES	0.119	modularity
Applicative_first_order_05__#3.52	MAYBE	0.021	YES	0.093	modularity
Applicative_first_order_05__#3.55	MAYBE	0.028	YES	0.221	modularity
Applicative_first_order_05__#3.57	MAYBE	0.026	YES	0.17	modularity
Applicative_first_order_05__#3.6	YES	0.027	YES	0.032	
Applicative_first_order_05__#3.8	MAYBE	0.023	YES	0.149	modularity
Applicative_first_order_05__01	YES	0.025	YES	0.026	
Applicative_first_order_05__02	YES	0.024	YES	0.025	
Applicative_first_order_05__06	YES	0.021	YES	0.021	
Applicative_first_order_05__08	YES	0.022	YES	0.028	
Applicative_first_order_05__11	YES	0.028	YES	0.047	
Applicative_first_order_05__12	YES	0.026	YES	0.029	
Applicative_first_order_05__13	MAYBE	0.023	YES	17.03	
Applicative_first_order_05__17	YES	0.024	YES	0.027	
Applicative_first_order_05__18	YES	0.022	YES	0.022	
Applicative_first_order_05__21	MAYBE	0.022	YES	0.128	modularity
Applicative_first_order_05__29	YES	0.025	YES	0.025	
Applicative_first_order_05__30	MAYBE	0.022	YES	0.313	modularity
Applicative_first_order_05__31	MAYBE	0.024	MAYBE	0.024	
Applicative_first_order_05__33	MAYBE	0.023	TIMEOUT	400.003	
Applicative_first_order_05__hydra	YES	0.024	YES	0.025	
Applicative_first_order_05__minsort	MAYBE	0.028	TIMEOUT	400.005	
Applicative_first_order_05__motivation	MAYBE	0.022	YES	0.084	modularity
Applicative_first_order_05__perfect	MAYBE	0.024	YES	0.122	modularity
Applicative_first_order_05__perfect2	MAYBE	0.03	YES	0.152	modularity
h21	MAYBE	0.023	YES	0.249	
h22	MAYBE	0.026	YES	0.337	modularity
h23	MAYBE	0.027	YES	0.376	modularity

Figure 7: Appendix: Comparison of SOL (2017)[Ham17] and SOL (this paper)

Appendix

Appendix A. The General Schema for Labelled Computation Systems

We show how the method of proving SN by GS is equally established for a labelled system $\mathcal{C}^{\mathsf{lab}}$.

A.1. β-CS.

We need the following kind of CS. Given a CS $(\Sigma, \mathcal{C})$, the β-CS [Bla00] is a CS $(\Sigma_\beta, \mathcal{C}_\beta)$ where

- Σ_β is an extension of Σ extended by the function symbols for all $\overline{a}, b \in \mathcal{T}_0$
$$@_{\overline{a},b} : (\overline{a} \to b), \overline{a} \to b$$
- $\mathcal{C}_\beta$ is obtained from $\mathcal{C}$ by adding the *β-rules*
$$@_{\overline{a},b}(\overline{x^a}.M[\overline{x}],\ \overline{N}) \Rightarrow M[\overline{N}]$$
for all $\overline{a}, b \in \mathcal{T}_0$.

The β-CS preserves and reflects SN of the original CS by GS. Hereafter, we will omit the subscripts of @.

Lemma A.1. *A CS $(\Sigma, \mathcal{C})$ satisfies GS if and only if the β-CS $(\Sigma_\beta, \mathcal{C}_\beta)$ satisfies GS.*

Proof. For both sides, take the same order $\leq_{\Sigma_\beta}$ which is an extension of $\leq_\Sigma$, where @ is smaller than any Σ-symbol, and apply GS. Note that M, N are accessible in lhs of the β-rule. □

Therefore, to apply GS to $\mathcal{C}^{\mathsf{lab}} = \mathbf{A}^{\mathsf{lab}} \uplus \mathbf{B}^{\mathsf{lab}}$, without loss of generality, we assume that $(\Sigma_{\mathbf{B}^{\mathsf{lab}}} \uplus \Theta, \mathbf{B}^{\mathsf{lab}})$ is a β-CS. Namely, $\mathbf{B}^{\mathsf{lab}}$ includes the β-rules, and $\Sigma_{\mathbf{B}^{\mathsf{lab}}}$ includes @. Every $\Sigma_{\mathbf{B}^{\mathsf{lab}}}$-symbol is not labelled.

A.2. Outline.

A key idea of GS is to the use of well-known notion of Tait's *computability* to show SN. Computability implies SN. The computable closure $\mathsf{CC}_f(\overline{t})$ (Def. 2.8) is a set of meta-terms, which approximate possible reducts of the lhs $f(\overline{t})$ of a rule. Thus, if all term instances of meta-terms in $\mathsf{CC}_f(\overline{t})$ are computable, we can conclude the target rewrite system is SN.

Following this idea, we prove SN of $\mathcal{C}^{\mathsf{lab}}$ with $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$ by GS. Our proof for the labelled case is actually the same as the original one [Bla00], except for the use of substitutions, i.e., to use labelled substitutions $t \bullet \rho$ for variables and labelled substitutions $\theta^{\sharp}_{-}(t)$ for metavariables. instead of normal substitutions. The difference between the normal computation $\Rightarrow_{\mathcal{C}}$ and the labelled computation $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$ is the use of $\theta^{\sharp}_{-}(t)$ for meta-applications, which is defined in Def. 3.8 as
$$\theta^{\sharp}_{-}(M[\overline{t}])) = u \bullet \{\overline{x} \mapsto\ \theta^{\sharp}_{-}(t_1), \ldots, \theta^{\sharp}_{-}(t_m)\} \quad \text{for } \theta : M \mapsto \overline{x}.u.$$

The original proof [Bla00] actually works in the case of the labelled case $\mathcal{C}^{\mathsf{lab}}$, by just replacing normal substitutions with labelled substitutions. A main reason of it is that labelled meta-terms with labelled substitutions forms a ***Σ-monoid*** as shown in [Ham07, Appendix A], which is an abstract algebraic model of syntax with variable binding and substitutions [FPT99, Ham04]. Because of this well-behaved algebraic structure, basic properties of substitutions are automatically ensured even in the labelled case, including the substitution lemma, interaction between substitution for variables and that for metavariables.

To make explicit how the proof works for the labelled case, in the following, we exhibit crucial parts of the proof, i.e.

(i) The definition of computability.
(ii) Key properties of computability.
(iii) A key lemma for computability closure correctness in [Bla00, Lemma 13].

A.3. **Computability.** Let $\mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$ be the set of all strongly normalising terms by the computation system $\mathcal{C}^{\mathsf{lab}}$ with $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$. We use the following definition of computability. For a type $\tau \in \mathcal{T}$, we now define $[\![\tau]\!]$ as follows:

$$\begin{aligned} [\![b]\!] &= \{t \in T_{\Sigma^{\mathsf{lab}}}\mathrm{V}_b \mid t \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})\} \quad \text{for } b \in \mathcal{T}_0 \\ [\![a_1, \ldots, a_n]\!] &= [\![a_1]\!] \times \cdots \times [\![a_n]\!] \\ [\![\overline{a} \to b]\!] &= \{\overline{x}.t \in T_{\Sigma^{\mathsf{lab}}}\mathrm{V}_{\overline{a} \to b} \mid \text{for all } \overline{s} \in [\![\overline{a}]\!],\ @(\overline{x}.t\ , \overline{s}) \in [\![b]\!]\} \end{aligned}$$

A term t of type τ is *computable* if $t \in [\![\tau]\!]$.

The above definition is standard [Bla00, BJO02], except for that SN is considered with respect to $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$, which uses labelled substitution, and @-terms are actually evaluated through labelled substitution. Nevertheless, the surface of definition is the same, which is a reason why our proof is essentially the same as the original in [Bla00].

A.4. **Key properties of computability.** The following are crucial properties, which is proved by the same way as in [Bla00].

(i) Every computable term is SN by $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$ (Lemma A.2).
(ii) Every one-step reduct of a computable term is computable (Lemma A.3).

Lemma A.2. *Every $t \in [\![\tau]\!]$ is SN by $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$.*

Proof. By induction on $[\![\tau]\!]$.

(i) Case $t \in [\![b]\!]$ for $b \in \mathcal{T}_0$. $t \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$ by definition of computability.
(ii) Case $\overline{x^a}.t \in [\![\overline{a} \to b]\!]$. Since variables $\overline{x^a}$ are SN, $\overline{x^a} \in [\![\overline{a}]\!]$. By definition of $[\![\overline{a} \to b]\!]$, $@(\overline{x}.t\ , \overline{x}) \in [\![b]\!]$. Hence $@(\overline{x}.t\ , \overline{x}) \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$, which follows $\overline{x}.t \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$. □

Lemma A.3. *Let $[\![\tau]\!] \ni u$. If $u \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} t$, then $t \in [\![\tau]\!]$.*

Proof. By induction on the type τ.

(i) Case $[\![b]\!] \ni u \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} t$. $t \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$ since $u \in \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$ by definition.
(ii) Case $[\![\overline{a} \to b]\!] \ni \overline{x^a}.u$. Suppose $u \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} t$. Take $\overline{s} \in [\![\overline{a}]\!]$. By definition, $[\![b]\!] \ni @(\overline{x}.u\ , \overline{s})$. Applying I.H., we have

$$[\![b]\!] \ni @(\overline{x}.u\ , \overline{s}) \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} @(\overline{x}.t\ , \overline{s}) \in [\![b]\!].$$

By definition of $[\![\overline{a} \to b]\!]$, we have $\overline{x}.t \in [\![\overline{a} \to b]\!]$. □

A.5. **Key lemma.** We say that a substitution $\rho : X \to T_{\Sigma^{\mathsf{lab}}}\mathrm{V}$ for variables is computable if for every $x \in X$, $\rho(x)$ is computable. We say a substitution $\theta : Z \to T_{\Sigma^{\mathsf{lab}}}\mathrm{V}$ for metavariables is computable if for every $\theta : M \mapsto x.u$, $x.u$ is computable.

Let $t \in M_{\Sigma^{\mathsf{lab}}}Z$. For a substitution $\rho : \mathsf{FV}(t) \to T_{\Sigma^{\mathsf{lab}}}\mathrm{V}$ for variables and a substitution θ for metavariables, where θ's codomain does not involve variables in $\mathsf{FV}(t)$ as free, $\theta^{\sharp}_{-}(t) \bullet \rho = \theta^{\sharp}_{-}(t\rho)$ holds, which is proved by easy induction of meta-terms.

To prove a key lemma [Bla00, Lemma 13] for computability closure correctness, an order $\succeq$ is used to comparing function terms defined for [Bla00, Lemma 13]. Since the proof for this case is the same as the original proof, we omit the proof and the definition of $\succeq$, see [Bla00] for details.

Lemma A.4. *Assume that $f(\bar{l}) \in M_{\Sigma^{\mathsf{lab}}}Z$ is a second-order pattern, a computable substitution θ for metavariables and $\theta^{\sharp}_{-}(\bar{l})$ is computable. Assume also that if $(f, \theta^{\sharp}_{-}(\bar{l})) \succeq (h, \overline{w})$ holds for $h \in \Sigma^{\mathsf{lab}}$ and computable $\overline{w}$, then $h(\overline{w})$ is computable. If $\mathsf{CC}_f(\bar{l}) \ni r$, then $\theta^{\sharp}_{-}(r\rho)$ is computable for every computable substitution $\rho : \mathsf{FV}(r) \to T_{\Sigma^{\mathsf{lab}}}\mathrm{V}$, where θ's codomain does not involve variables in $\mathsf{FV}(r)$ as free.*

Proof. First we note that without loss of generality, we assume that bound variables in binders are always fresh. The condition "θ's codomain does not involve variables in $\mathsf{FV}(t)$ as free" is satisfied in this case. We prove by induction on the definition of $\mathsf{CC}_f(\bar{l})$.

(i) Case $r = x$ of type b is SN. Then $\theta^{\sharp}_{-}(\rho(x)) = \rho(x)$ is computable by assumption.

(ii) Case $r = \overline{x^a}.t$ of type $\overline{a} \to b$. Now $\theta^{\sharp}_{-}(r) \bullet \rho = \theta^{\sharp}_{-}(r\rho)$ holds. Take $\overline{s} \in [\![\overline{a}]\!]$. Then

$$@(\overline{x}.\theta^{\sharp}_{-}(t\rho)\ , \overline{s}) \ \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} \theta^{\sharp}_{-}(t\rho) \bullet \{\overline{x} \mapsto \overline{s}\} = \theta^{\sharp}_{-}(t) \bullet (\{\overline{x} \mapsto \overline{s}\} \uplus \rho) \ \in \ [\![b]\!] \subseteq \mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$$

where "$\in$" is by I.H. Again by I.H., $\theta^{\sharp}_{-}(t\rho)$ is computable, hence in $\mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$. $\overline{s}$ are also computable, hence in $\mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$. Thus $@(\overline{x}.\theta^{\sharp}_{-}(t\rho)\ , \overline{s})$ is in $\mathbf{SN}(\Rightarrow_{\mathcal{C}^{\mathsf{lab}}})$, so $@(\overline{x}.\theta^{\sharp}_{-}(t\rho)\ , \overline{s}) \in [\![b]\!]$. Therefore, $\overline{x}.\theta^{\sharp}_{-}(t\rho) = \theta^{\sharp}_{-}(\overline{x}.t\rho) \in [\![\overline{a} \to b]\!]$.

(iii) Case $r = M[\overline{s}]$ with $M : \overline{a} \to b$. Now $\theta^{\sharp}_{-}(r) \bullet \rho = \theta^{\sharp}_{-}(r\rho)$ holds. We assume $\theta : M \mapsto \overline{x}.w$. Then

$$\theta^{\sharp}_{-}(M[\overline{s}]) \bullet \rho = \theta^{\sharp}_{-}(M[\overline{s\rho}]) = w \bullet \{\overline{x} \mapsto \theta^{\sharp}_{-}(\overline{s\rho})\}$$

By I.H., $\theta^{\sharp}_{-}(\overline{s\rho})$ is computable. By assumption, $\overline{x}.w$ is computable. So

$$[\![b]\!] \ni @(\overline{x}.w\ , \theta^{\sharp}_{-}(\overline{s\rho})) \ \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} w \bullet \{\overline{x} \mapsto \theta^{\sharp}_{-}(\overline{s\rho})\}$$

which is in $[\![b]\!]$ by Lemma A.3.

(iv) Case $r = h(\overline{w})$. Same as in [Bla00, Lemma 13]. □

A.6. **SN by GS.** Taking ρ to be the identity substitution in Lemma A.4 under the same assumptions, we have that if $\mathsf{CC}_f(\bar{l}) \ni r$, then $\theta^{\sharp}_{-}(r)$ is computable. Then the main result follows from it and [Bla00, Lemma 14].

Proposition 3.9 (repeated) Let $(\Sigma^{\mathsf{lab}}, \mathcal{C}^{\mathsf{lab}})$ be a labelled computation system. Suppose that $\leq_{\mathcal{T}}$ and $\leq_{\Sigma^{\mathsf{lab}}}$ are well-founded. If for all $f(\overline{t}) \Rightarrow r \in \mathcal{C}^{\mathsf{lab}}$, $\mathsf{CC}_f(\overline{t}) \ni r$, then $\mathcal{C}^{\mathsf{lab}}$ with $\Rightarrow_{\mathcal{C}^{\mathsf{lab}}}$ is strongly normalising. Note that f may be labelled.

Appendix B. A Higher-Order Semantic Labelling

We prove the postponed proposition.

Proposition 3.10 (repeated) If $\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})$ is SN, then $\mathcal{C}$ is SN.

We prove that every infinite $\mathcal{C}$-reduction sequence is strictly simulated by a labelled reduction sequence. We actually prove an infinite reduction-plus-subterm-step sequence on minimal non-terminating terms is strictly simulated by a labelled reduction sequence because without loss of generality, we can assume that the infinite $\mathcal{C}$-reduction sequence is a minimal one. A minimal non-terminating term is a non-terminating term whose strict subterms are SN [Gra94]. Let **SN** be the set of all strongly normalising terms by the computation system $\mathcal{C}$. We define the contextual set of minimal non-terminating terms by $T_\Sigma^\infty \mathrm{V} \triangleq \{t \in T_\Sigma \mathrm{V} \mid t \notin \mathbf{SN} \;\;\&\;\; (\forall s \in T_\Sigma \mathrm{V}.\ t \rhd s \text{ implies } s \in \mathbf{SN})\}$. We denote by $\overset{\geq\varepsilon}{\Rightarrow}{}^*_{\mathcal{C}}$ a many-step computation step below the root position, by $\overset{\varepsilon}{\Rightarrow}_{\mathcal{C}}$ a root computation step (i.e., rewriting the root position). The following is well-known.

Lemma B.1. *For every $u \in T_\Sigma^\infty \mathrm{V}$, there exist*

- *a rule $Z \rhd l \Rightarrow r : b \in \mathcal{C}$,*
- *an assignment $\theta : Z \to \mathbf{SN}$,*
- *a position p of $\theta^\sharp(r)$,*

such that $u \;\overset{\geq\varepsilon}{\Rightarrow}{}^*_{\mathcal{C}}\; \theta^\sharp(l) \;\overset{\varepsilon}{\Rightarrow}_{\mathcal{C}}\; \theta^\sharp(r) \trianglerighteq (\theta^\sharp(r))_{|p} \in T_\Sigma^\infty \mathrm{V}$, *and* $\theta^\sharp(l) \in T_\Sigma^\infty \mathrm{V}$.

Proof. Let $u \in T_\Sigma^\infty \mathrm{V}$ and Q an infinite reduction sequence

$$u = u_0 \Rightarrow_{\mathcal{C}} u_1 \Rightarrow_{\mathcal{C}} \cdots$$

Since all strict subterms of u are SN, Q must contain a root computation step. Suppose the first root computation step appears as (1) in $u \;\overset{(2)}{\Rightarrow}{}^*_{\mathcal{C}}\; \theta^\sharp(l) \;\overset{(1)}{\Rightarrow}_{\mathcal{C}}\; \theta^\sharp(r)$ where $f(\overline{x_1}.l_1, \ldots, \overline{x_n}.l_n) = l$. Since (2) is below the root step computation, it must be

$$u = f(\overline{x_1}.t_1, \ldots, \overline{x_n}.t_n) \Rightarrow^*_{\mathcal{C}} \theta^\sharp(f(\overline{x_1}.l_1, \ldots, \overline{x_n}.l_n)) = \theta^\sharp(l)$$

and for each t_i $(i = 1, \ldots, n)$,

$$\mathbf{SN} \ni \;\; t_i \Rightarrow^*_{\mathcal{C}} \theta^\sharp(l_i) \;\; \in \mathbf{SN}$$

by the assumption that all strict subterms of u are SN. Since l_i is a second-order pattern, we see $\theta(M) \in \mathbf{SN}$ for every metavariable M in l. Since $\theta^\sharp(r)$ is not SN, there exists a position p such that $\theta^\sharp(r) \trianglerighteq (\theta^\sharp(r))_{|p} \in T_\Sigma^\infty \mathrm{V}$. □

Lemma B.2. *Let $\theta : Z \longrightarrow T_\Sigma \mathrm{V}$, $t \in M_\Sigma Z$, and t is a second-order pattern. If $\mathsf{Fun}(t) \subseteq \Sigma_{\mathbf{A}} \uplus \Theta$, $\mathsf{tr}(\theta^\sharp(t)) = (\mathsf{tr} \circ \theta)^\sharp(t)$.*

Proof. We prove by induction on the structure of the meta-term t.

(i) If $t = x$, both sides are x.

(ii) Case $t = M[\overline{x}]$, where $\overline{x}$ are distinct bound variables because t is a second-order pattern. Suppose $\theta : M \mapsto \overline{y}.u$.

$$lhs = \mathsf{tr}(\theta^\sharp(M[\overline{x}])) = \mathsf{tr}(u\{\overline{y} \mapsto \overline{x}\})$$
$$rhs = (\mathsf{tr} \circ \theta)^\sharp(M[\overline{x}]) = \mathsf{tr}(u)\{\overline{y} \mapsto \overline{x}\} = \mathsf{tr}(u\{\overline{y} \mapsto \overline{x}\}) = lhs$$

(iii) Case $t = f(x.s)$. The general case $t = f(\overline{x}_1.s_1, \dots, \overline{x}_n.s_n)$ is proved similarly, hence omitted. Since $f \in \Sigma_{\mathbf{A}} \uplus \Theta$, we have by I.H.

$$lhs = f(x.\mathsf{tr}(\theta^\sharp(s))) = f(x.(\mathsf{tr} \circ \theta)^\sharp(s)) = rhs \qquad \square$$

The assumption that t is a second-order pattern is essential in the above lemma. Without it, the case (ii) does not hold because tr and substitution for metavariables do not commute for non-patterns.

The labelling labtr commutes with substitution for metavariables in the following sense.

Lemma B.3. *Let $\theta : Z \to \mathbf{SN}$ be an assignment. If a meta-term r satisfies the $\Sigma_{\mathbf{A}}$-layer condition, then* $\mathsf{labtr}(\theta^\sharp(r)) \;=\; \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(r))$.

Proof. By induction on the structure of the meta-term r.

(i) Case $r = x$, both sides are x.

(ii) Case $r = M[\overline{s}]$. Suppose $\theta : M \mapsto \overline{y}.u$.

$$\begin{aligned}
lhs &= \mathsf{labtr}(\theta^\sharp(M[\overline{s}])) = \mathsf{labtr}(u\{\overline{y} \mapsto \overline{\theta^\sharp(s)}\})\\
rhs &= \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(M[\overline{s}]))\\
&= \theta^{\underline{\sharp}}(M[\overline{\mathsf{lab}_{(\mathsf{tr}\circ\theta)}\ (s)}]) = \mathsf{labtr}(\ \| u\{\overline{y} \mapsto \overline{\theta^{\underline{\sharp}}\mathsf{lab}_{(\mathsf{tr}\circ\theta)}\ (s)}\}\|\ )\\
&= \mathsf{labtr}(\ \| u\{\overline{y} \mapsto \overline{\theta^{\underline{\sharp}}\mathsf{lab}_{(\mathsf{tr}\circ\theta)}\ (s)}\}\|\ )\\
&= \mathsf{labtr}(u\{\overline{y} \mapsto \overline{\theta^\sharp(s)}\}) = lhs
\end{aligned}$$

(iii) Case $r = f(x.s)$. Case $f(\overline{\overline{x}.s})$ is similar.

(a) Case $f \notin \Sigma_{\mathbf{A}}$.

$$\begin{aligned}
lhs &= \mathsf{labtr}(f(x.\theta^\sharp(s))) = f(x.\mathsf{labtr}(\theta^\sharp(s)))\\
&= f(x.\theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(s))) \quad \text{by I.H.}\\
&= \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(f(x.s))) = rhs
\end{aligned}$$

(b) Case $f \in \Sigma_{\mathbf{A}}$. Since r satisfies the $\Sigma_{\mathbf{A}}$-layer condition, $\mathsf{Fun}(s) \subseteq \Sigma_{\mathbf{A}} \uplus \Theta$ and s is a second-order pattern.

$$\begin{aligned}
lhs &= \mathsf{labtr}(f(x.\theta^\sharp(s))) = f_v(x.\mathsf{labtr}(\theta^\sharp(s))) \qquad \text{where } v = f(x.\mathsf{tr}(\theta^\sharp(s)))\\
rhs &= \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(f(x.s)))\\
&= \theta^{\underline{\sharp}}(f_w(x.\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(s))) \quad \text{where } w = f(x.(\mathsf{tr}\circ\theta)^\sharp(s))\\
&= f_w(x.\theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(s))) = f_v(x.\mathsf{labtr}(\theta^\sharp(s))) = lhs
\end{aligned}$$

By Lemma B.2, $\mathsf{tr}(\theta^\sharp(s)) = (\mathsf{tr}\circ\theta)^\sharp(s)$, hence $w = v$, The final decreasing step is by I.H. $\square$

The tr with $\mathbf{A}_{\mathsf{Proj}}$-reduction simulates a $\mathcal{C}$-reduction as follows.

Lemma B.4. *If* $\mathbf{SN} \ni s \Rightarrow_{\mathcal{C}} t$ *then* $\mathsf{tr}(s) \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \mathsf{tr}(t)$.

Proof. By induction on the proof of $\Rightarrow_{\mathcal{C}}$.

- **(Rule)** If $\theta^\sharp(l) \Rightarrow_{\mathcal{C}} \theta^\sharp(r)$, for $\theta : Z \longrightarrow T_\Sigma \mathrm{V}$ is derived from $Z \;\triangleright\; l \Rightarrow r : b \in \mathcal{C}$ with $l = f(\overline{\overline{x}.t})$. If $\theta^\sharp(l), \theta^\sharp(r) \notin \mathbf{SN}$, then it is vacuously true. Case $f \in \Theta$ is impossible.

(i) Case $f \in \Sigma_{\mathbf{A}}$. Then $l = f(\overline{x.t}) \Rightarrow r \in \mathbf{A}$ and $\mathsf{Fun}(l) \cup \mathsf{Fun}(r) \subseteq \Sigma_{\mathbf{A}} \uplus \Theta$.

$$\begin{aligned} lhs = \mathsf{tr}(\theta^\sharp(l)) = (\mathsf{tr} \circ \theta)^\sharp(l) \qquad & \text{by Lemma B.2} \\ \Rightarrow_{\mathbf{A}_{\mathsf{Proj}}} (\mathsf{tr} \circ \theta)^\sharp(r) \qquad & \text{since } (-)^\sharp \text{ is an instantiation of an } \mathbf{A}\text{-rule} \\ = \mathsf{tr}(\theta^\sharp(r)) \qquad & \text{by Lemma B.2} \\ = rhs & \end{aligned}$$

(ii) Case $f \in \Sigma_{\mathbf{B}}$. Then $f(\overline{x.t}) \Rightarrow r \in \mathbf{B}$.

$$\begin{aligned} lhs = \mathsf{tr}(\theta^\sharp(l)) &= \mathsf{list}\{\mathsf{tr}(u) \mid \theta^\sharp(l) \Rightarrow_{\mathcal{C}} u\} \\ &= \mathsf{list}\{\mathsf{tr}(u) \mid \theta^\sharp(l) \Rightarrow_{\mathcal{C}} u \text{ or } \theta^\sharp(l) \Rightarrow_{\mathcal{C}} \theta^\sharp(r) = u\} \\ &\Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \mathsf{tr}(\theta^\sharp(r)) = rhs \end{aligned}$$

- **(Fun)** $f(x.s) \Rightarrow_{\mathcal{C}} f(x.t)$ is derived from $s \Rightarrow_{\mathcal{C}} t$. (The n-ary case is similar.)
  (i) Case $f \notin \Sigma_{\mathbf{B}}$. Using I.H.,

$$lhs = \mathsf{tr}(f(x.s)) = f(x.\mathsf{tr}(s)) \Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} f(x.\mathsf{tr}(t)) = \mathsf{tr}(f(x.t)) = rhs$$

  (ii) Case $f \in \Sigma_{\mathbf{B}}$.

$$\begin{aligned} lhs = \mathsf{tr}(f(x.s)) &= \mathsf{list}\{\mathsf{tr}(u) \mid f(x.s) \Rightarrow_{\mathcal{C}} u\} \\ &= \mathsf{list}\{\mathsf{tr}(u) \mid f(x.s) \Rightarrow_{\mathcal{C}} u \ \text{ or } \ f(x.s) \Rightarrow_{\mathcal{C}} f(x.t) = u\} \\ &\Rightarrow^*_{\mathbf{A}_{\mathsf{Proj}}} \mathsf{tr}(f(x.t)) = rhs \end{aligned}$$

□

Then one-step $\mathcal{C}$-reduction can be transformed to a many-step trace labeled reduction possibly using decreasing steps on labels.

Lemma B.5. *If* $\mathbf{SN} \cup T^\infty_\Sigma\mathrm{V} \ni s \Rightarrow_{\mathcal{C}} t \in \mathbf{SN} \cup T^\infty_\Sigma\mathrm{V}$ *then*

$$\mathsf{labtr}(s) \Rightarrow^+_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} \mathsf{labtr}(t).$$

Proof. By induction on the proof of $\Rightarrow_{\mathcal{C}}$.

- **(Rule)** Suppose that $\theta^\sharp(l) \Rightarrow_{\mathcal{C}} \theta^\sharp(r)$, for $\theta : Z \longrightarrow T_\Sigma\mathrm{V}$ is derived from $Z \rhd l \Rightarrow r : b \in \mathcal{C}$.
  If $\theta^\sharp(l), \theta^\sharp(r) \notin \mathbf{SN} \cup T^\infty_\Sigma\mathrm{V}$, then it is vacuously true.
  Case $\theta^\sharp(l), \theta^\sharp(r) \in \mathbf{SN} \cup T^\infty_\Sigma\mathrm{V}$. We have a labelled rule $\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(l) \Rightarrow \mathsf{lab}_{(\mathsf{tr}\circ\theta)}(r) \in \mathcal{C}^{\mathsf{lab}}$. By Lemma B.3,

$$\mathsf{labtr}(\theta^\sharp(l)) = \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(l)) \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr}\circ\theta)}(r)) \ = \ \mathsf{labtr}(\theta^\sharp(r)).$$

- **(Fun)** $f(x.s) \Rightarrow_{\mathcal{C}} f(x.t)$ is derived from $s \Rightarrow_{\mathcal{C}} t$.
  (The n-ary case: $f(\ldots, \overline{x}.s, \ldots) \Rightarrow_{\mathcal{C}} f(\ldots, x.t, \ldots)$ is derived from $s \Rightarrow_{\mathcal{C}} t$, is similar).
  (i) Case $f \notin \Sigma_{\mathbf{A}}$. Using I.H., we have

$$\mathsf{labtr}(f(x.s)) = f(x.\mathsf{labtr}(s)) \Rightarrow^+_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} f(x.\mathsf{labtr}(t)) = \mathsf{labtr}(f(x.t))$$

(ii) Case $f \in \Sigma_{\mathbf{A}}$.

$$\begin{aligned}
\mathsf{labtr}(f(x.s)) = \quad & f_{f(x.\mathsf{tr}(s))}(x.\mathsf{labtr}(s)) \\
& \Rightarrow_{\mathsf{Decr}(\mathbf{A})} f_{f(x.\mathsf{tr}(t))}(x.\mathsf{labtr}(s)) && \text{by Lemma B.4} \\
& \Rightarrow^{+}_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} f_{f(x.\mathsf{tr}(t))}(x.\mathsf{labtr}(t)) && \text{by I.H.} \\
& = \mathsf{labtr}(f(x.t))
\end{aligned}$$

□

This establishes a variation of the higher-order semantic labelling method.

Proposition 3.10 (repeated) If $\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})$ is SN, then $\mathcal{C}$ is SN.

Proof. Let $t \in T^{\infty}_{\Sigma}\mathrm{V}$. By Lemma B.1, there exist a rule $Z \triangleright l \Rightarrow r : b \in \mathcal{C}$, an assignment $\theta : Z \to \mathbf{SN}$, and a position p of $\theta^{\sharp}(r)$ such that

$$T^{\infty}_{\Sigma}\mathrm{V} \ni t \overset{>\varepsilon}{\Rightarrow}{}^{*}_{\mathcal{C}} \theta^{\sharp}(l) \overset{\varepsilon}{\Rightarrow}_{\mathcal{C}} \theta^{\sharp}(r) \trianglerighteq (\theta^{\sharp}(r))_{|p} \in T^{\infty}_{\Sigma}\mathrm{V}$$

By Lemma B.5, $\mathsf{labtr}(t) \Rightarrow^{*}_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} \mathsf{labtr}(\theta^{\sharp}(l))$. By Lemma B.3, we have

$$\mathsf{labtr}(\theta^{\sharp}(l)) = \theta^{\underline{\sharp}}(\mathsf{lab}_{(\mathsf{tr} \circ \theta)}(l)).$$

Since $\mathsf{lab}_{(\mathsf{tr} \circ \theta)}(l) \Rightarrow \mathsf{lab}_{(\mathsf{tr} \circ \theta)}(r) \in \mathcal{C}^{\mathsf{lab}}$,

$$\theta^{\underline{\sharp}}\, \mathsf{lab}_{(\mathsf{tr} \circ \theta)}(l) \Rightarrow_{\mathcal{C}^{\mathsf{lab}}} \theta^{\underline{\sharp}}\, \mathsf{lab}_{(\mathsf{tr} \circ \theta)}(r).$$

By Assumption 3.1(v), r satisfies the $\Sigma_{\mathbf{A}}$-layer condition. Therefore, by Lemma B.3, $\theta^{\underline{\sharp}}\, \mathsf{lab}_{\mathsf{tr} \circ \theta}(r) = \mathsf{labtr}(\theta^{\sharp}(r))$. Hence

$$\begin{aligned}
\mathsf{labtr}(t) \Rightarrow^{+}_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} \theta^{\underline{\sharp}}\, \mathsf{lab}_{(\mathsf{tr} \circ \theta)}(r) = \mathsf{labtr}(\theta^{\sharp}(r)) & \trianglerighteq \mathsf{labtr}(\theta^{\sharp}\, r)_{|p} \\
& = \mathsf{labtr}((\theta^{\sharp}\, r)_{|p}).
\end{aligned}$$

Note that since labtr just attaches labels, it does not affect the position structure of r.

Now we show the contrapositive of the proposition. Suppose $\mathcal{C}$ is not SN. Then, there is an infinite reduction sequence

$$T^{\infty}_{\Sigma}\mathrm{V} \ni t_1 \overset{>\varepsilon}{\Rightarrow}{}^{*}_{\mathcal{C}} \circ \overset{\varepsilon}{\Rightarrow}_{\mathcal{C}} \circ \trianglerighteq t_2 \overset{>\varepsilon}{\Rightarrow}{}^{*}_{\mathcal{C}} \circ \overset{\varepsilon}{\Rightarrow}_{\mathcal{C}} \circ \trianglerighteq \cdots$$

By the above argument, we have

$$\mathsf{labtr}(t_1) \Rightarrow^{+}_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} \circ \trianglerighteq \mathsf{labtr}(t_2) \Rightarrow^{+}_{\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})} \circ \trianglerighteq \cdots$$

which is infinite, hence $\mathcal{C}^{\mathsf{lab}} \cup \mathsf{Decr}(\mathbf{A})$ is not SN. □