Polymorphic Higher-Order Termination

Łukasz Czajka
Faculty of Informatics, TU Dortmund, Germany
http://www.mimuw.edu.pl/~lukaszcz/
lukaszcz@mimuw.edu.pl

Cynthia Kop
Institute of Computer Science, Radboud University Nijmegen, The Netherlands
https://www.cs.ru.nl/~cynthiakop/
c.kop@cs.ru.nl

Abstract

We generalise the termination method of higher-order polynomial interpretations to a setting with impredicative polymorphism. Instead of using weakly monotonic functionals, we interpret terms in a suitable extension of System $\mathtt{F}_\omega$. This enables a direct interpretation of rewrite rules which make essential use of impredicative polymorphism. In addition, our generalisation eases the applicability of the method in the non-polymorphic setting by allowing for the encoding of inductive data types. As an illustration of the potential of our method, we prove termination of a substantial fragment of full intuitionistic second-order propositional logic with permutative conversions.

2012 ACM Subject Classification Theory of computation → Rewrite systems; Theory of computation → Equational logic and rewriting; Theory of computation → Type theory

Keywords and phrases termination, polymorphism, higher-order rewriting, permutative conversions

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.12

Related Version Complete proofs for the results in this paper are available in an online appendix at https://arxiv.org/pdf/1904.09859.pdf.

1 Introduction

Termination of higher-order term rewriting systems [21, Chapter 11] has been an active area of research for several decades. One powerful method, introduced by v.d. Pol [23, 15], interprets terms into *weakly monotonic algebras*. In later work [6, 12], these algebra interpretations are specialised into *higher-order polynomial interpretations*, a generalisation of the popular – and highly automatable – technique of polynomial interpretations for first-order term rewriting.

The methods of weakly monotonic algebras and polynomial interpretation are both limited to *monomorphic* systems. In this paper, we will further generalise polynomial interpretations to a higher-order formalism with full impredicative polymorphism. This goes beyond shallow (rank-1, weak) polymorphism, where type quantifiers are effectively allowed only at the top of a type: it would be relatively easy to extend the methods to a system with shallow polymorphism since shallowly polymorphic rules can be seen as defining an infinite set of monomorphic rules. While shallow polymorphism often suffices in functional programming practice, there do exist interesting examples of rewrite systems which require higher-rank impredicative polymorphism.

For instance, in recent extensions of Haskell one may define a type of heterogeneous lists.

$$\begin{array}{ll} \mathtt{List} : * & \mathtt{foldl}_\sigma(f, a, \mathtt{nil}) \longrightarrow a \\ \mathtt{nil} : \mathtt{List} & \mathtt{foldl}_\sigma(f, a, \mathtt{cons}_\tau(x, l)) \longrightarrow \mathtt{foldl}_\sigma(f, f\tau a x, l) \\ \mathtt{cons} : \forall\alpha.\alpha \to \mathtt{List} \to \mathtt{List} & \\ \mathtt{foldl} : \forall\beta.(\forall\alpha.\beta \to \alpha \to \beta) \to \beta \to \mathtt{List} \to \beta & \end{array}$$

The above states that `List` is a type ($*$), gives the types of its two constructors `nil` and `cons`, and defines the corresponding fold-left function `foldl`. Each element of a heterogeneous list may have a different type. In practice, one would constrain the type variable α with a type class to guarantee the existence of some operations on list elements. The function argument of `foldl` receives the element together with its type. The $\forall$-quantifier binds type variables: a term of type $\forall\alpha.\tau$ takes a type ρ as an argument and the result is a term of type $\tau[\alpha := \rho]$.

Impredicativity of polymorphism means that the type itself may be substituted for its own type variable, e.g., if $\mathtt{f} : \forall\alpha.\tau$ then $f(\forall\alpha.\tau) : \tau[\alpha := \forall\alpha.\tau]$. Negative occurrences of impredicative type quantifiers prevent a translation into an infinite set of simply typed rules by instantiating the type variables. The above example is not directly reducible to shallow polymorphism as used in the ML programming language.

Related work. The term rewriting literature has various examples of higher-order term rewriting systems with some forms of polymorphism. To start, there are several studies that consider shallow polymorphic rewriting (e.g., [9, 11, 25]), where (as in ML-like languages) systems like `foldl` above cannot be handled. Other works consider extensions of the $\lambda\Pi$-calculus [2, 4] or the calculus of constructions [1, 26] with rewriting rules; only the latter includes full impredicative polymorphism. The termination techniques presented for these systems are mostly syntactic (e.g., a recursive path ordering [11, 26], or general schema [1]), as opposed to our more semantic method based on interpretations. An exception is [4], which defines interpretations into Π-algebras; this technique bears some similarity to ours, although the methodologies are quite different. A categorical definition for a general polymorphic rewriting framework is presented in [5], but no termination methods are considered for it.

Our approach. The technique we develop in this paper operates on *Polymorphic Functional Systems (PFSs)*, a form of higher-order term rewriting systems with full impredicative polymorphism (Section 3), that various systems of interest can be encoded into (including the example of heterogeneous fold above). Then, our methodology follows a standard procedure:

- we define a well-ordered set $(\mathcal{I}, \succ, \succeq)$ (Section 4);
- we provide a general methodology to map each PFS term s to a natural number $[\![s]\!]$, parameterised by a core interpretation for each function symbol (Section 5);
- we present a number of lemmas to make it easy to prove that $s \succ t$ or $s \succeq t$ whenever s reduces to t (Section 6).

Due to the additional complications of full polymorphism, we have elected to only generalise higher-order polynomial interpretations, and not v.d. Pol's weakly monotonic algebras. That is, terms of base type are always interpreted to natural numbers and all functions are interpreted to combinations of addition and multiplication.

We will use the system of heterogeneous fold above as a running example to demonstrate our method. However, termination of this system can be shown in other ways (e.g., an encoding in System `F`). Hence, we will also study a more complex example in Section 7: termination of a substantial fragment of IPC2, i.e., full intuitionistic second-order propositional logic with permutative conversions. Permutative conversions [22, Chapter 6] are used in proof theory to obtain "good" normal forms of natural deduction proofs, which satisfy e.g. the subformula property. Termination proofs for systems with permutative conversions are notoriously tedious and difficult, with some incorrect claims in the literature and no uniform methodology. It is our goal to make such termination proofs substantially easier in the future.

2 Preliminaries

In this section we recall the definition of System $\mathtt{F}_\omega$ (see e.g. [17, Section 11.7]), which will form a basis both of our interpretations and of a general syntactic framework for the investigated systems. In comparison to System F, System $\mathtt{F}_\omega$ includes type constructors which results in a more uniform treatment. We assume familiarity with core notions of lambda calculi such as substitution and α-conversion.

▶ **Definition 2.1.** Kinds *are defined inductively:* $*$ *is a kind, and if* κ_1, κ_2 *are kinds then so is* $\kappa_1 \Rightarrow \kappa_2$*. We assume an infinite set* $\mathcal{V}_\kappa$ *of* type constructor variables *of each kind* κ*. Variables of kind* $*$ *are* type variables*. We assume a fixed set* Σ^T_κ *of* type constructor symbols *paired with a kind* κ*, denoted* $c : \kappa$*. We define the set* $\mathcal{T}_\kappa$ *of* type constructors *of kind* κ *by the following grammar. Type constructors of kind* $*$ *are* types*.*

$$\begin{array}{rcl} \mathcal{T}_* & ::= & \mathcal{V}_* \mid \Sigma^T_* \mid \mathcal{T}_{\kappa\Rightarrow *}\mathcal{T}_\kappa \mid \forall \mathcal{V}_\kappa \mathcal{T}_* \mid \mathcal{T}_* \to \mathcal{T}_* \\ \mathcal{T}_{\kappa_1\Rightarrow\kappa_2} & ::= & \mathcal{V}_{\kappa_1\Rightarrow\kappa_2} \mid \Sigma^T_{\kappa_1\Rightarrow\kappa_2} \mid \mathcal{T}_{\kappa\Rightarrow(\kappa_1\Rightarrow\kappa_2)}\mathcal{T}_\kappa \mid \lambda\mathcal{V}_{\kappa_1}\mathcal{T}_{\kappa_2} \end{array}$$

We use the standard notations $\forall\alpha.\tau$ *and* $\lambda\alpha.\tau$*. When* α *is of kind* κ *then we use the notation* $\forall\alpha : \kappa.\tau$*. If not indicated otherwise, we assume* α *to be a type variable. We treat type constructors up to* α*-conversion.*

▶ **Example 2.2.** If $\Sigma^T_* = \{\mathtt{List}\}$ and $\Sigma^T_{*\Rightarrow *\Rightarrow *} = \{\mathtt{Pair}\}$, types are for instance $\mathtt{List}$ and $\forall\alpha.\mathtt{Pair}\,\alpha\,\mathtt{List}$. The expression $\mathtt{Pair}\,\mathtt{List}$ is a type constructor, but not a type. If $\Sigma^T_{(*\Rightarrow *)\Rightarrow *} = \{\exists\}$ and $\sigma \in \mathcal{T}_{*\Rightarrow *}$, then both $\exists(\sigma)$ and $\exists(\lambda\alpha.\sigma\alpha)$ are types.

The compatible closure of the rule $(\lambda\alpha.\varphi)\psi \to \varphi[\alpha := \psi]$ *defines* β*-reduction on type constructors. As type constructors are (essentially) simply-typed lambda-terms, their* β*-reduction terminates and is confluent; hence every type constructor* τ *has a unique* β*-normal form* $\mathrm{nf}_\beta(\tau)$*. A* type atom *is a type in* β*-normal form which is neither an arrow* $\tau_1 \to \tau_2$ *nor a quantification* $\forall\alpha.\tau$*.*

We define $\mathrm{FTV}(\varphi)$ *– the set of free type constructor variables of the type constructor* φ *– in an obvious way by induction on* φ*. A type constructor* φ *is* closed *if* $\mathrm{FTV}(\varphi) = \emptyset$*.*

We assume a fixed type symbol $\chi_* \in \Sigma^T_*$*. For* $\kappa = \kappa_1 \Rightarrow \kappa_2$ *we define* $\chi_\kappa = \lambda\alpha : \kappa_1.\chi_{\kappa_2}$*.*

▶ **Definition 2.3.** *We assume given an infinite set* $\mathcal{V}$ *of variables, each paired with a type, denoted* $x : \tau$*. We assume given a fixed set* Σ *of* function symbols*, each paired with a closed type, denoted* $\mathtt{f} : \tau$*. Every variable* x *and every function symbol* $\mathtt{f}$ *occurs only with one type declaration.*

The set of preterms *consists of all expressions* s *such that* $s : \sigma$ *can be inferred for some type* σ *by the following clauses:*

- $x : \sigma$ *for* $(x : \sigma) \in \mathcal{V}$*.*
- $\mathtt{f} : \sigma$ *for all* $(\mathtt{f} : \sigma) \in \Sigma$*.*
- $\lambda x : \sigma.s : \sigma \to \tau$ *if* $(x : \sigma) \in \mathcal{V}$ *and* $s : \tau$*.*
- $(\Lambda\alpha : \kappa.s) : (\forall\alpha : \kappa.\sigma)$ *if* $s : \sigma$ *and* α *does not occur free in the type of a free variable of* s*.*
- $s \cdot t : \tau$ *if* $s : \sigma \to \tau$ *and* $t : \sigma$
- $s * \tau : \sigma[\alpha := \tau]$ *if* $s : \forall\alpha : \kappa.\sigma$ *and* τ *is a type constructor of kind* κ*,*
- $s : \tau$ *if* $s : \tau'$ *and* $\tau =_\beta \tau'$*.*

The set of free variables of a preterm t*, denoted* $\mathrm{FV}(t)$*, is defined in the expected way. Analogously, we define the set* $\mathrm{FTV}(t)$ *of type constructor variables occurring free in* t*. If* α *is a type then we use the notation* $\Lambda\alpha.t$*. We denote an occurrence of a variable* x *of type* τ *by* x^τ*, e.g.* $\lambda x : \tau \to \sigma.x^{\tau\to\sigma}y^\tau$*. When clear or irrelevant, we omit the type annotations,*

denoting the above term by $\lambda x.xy$. Type substitution is defined in the expected way except that it needs to change the types of variables. Formally, a type substitution changes the types associated to variables in $\mathcal{V}$. We define the equivalence relation $\equiv$ by: $s \equiv t$ iff s and t are identical modulo β-conversion in types.

Note that we present terms in orthodox Church-style, i.e., instead of using contexts each variable has a globally fixed type associated to it.

▶ **Lemma 2.4.** *If $s : \tau$ and $s \equiv t$ then $t : \tau$.*

Proof. Induction on s. ◀

▶ **Definition 2.5.** *The set of* terms *is the set of the equivalence classes of $\equiv$.*

Because β-reduction on types is confluent and terminating, every term has a canonical preterm representative – the one with all types occurring in it β-normalised. We define $\mathrm{FTV}(t)$ as the value of FTV on the canonical representative of t. We say that t is *closed* if both $\mathrm{FTV}(t) = \emptyset$ and $\mathrm{FV}(t) = \emptyset$. Because typing and term formation operations (abstraction, application, ...) are invariant under $\equiv$, we may denote terms by their (canonical) representatives and informally treat them interchangeably.

We will often abuse notation to omit $\cdot$ and $*$. Thus, st can refer to both $s \cdot t$ and $s * t$. This is not ambiguous due to typing. When writing $\sigma[\alpha := \tau]$ we implicitly assume that α and τ have the same kind. Analogously with $t[x := s]$.

▶ **Lemma 2.6** (Substitution lemma)**.**
1. *If $s : \tau$ and $x : \sigma$ and $t : \sigma$ then $s[x := t] : \tau$.*
2. *If $t : \sigma$ then $t[\alpha := \tau] : \sigma[\alpha := \tau]$.*

Proof. Induction on the typing derivation. ◀

▶ **Lemma 2.7** (Generation lemma)**.** *If $t : \sigma$ then there is a type σ' such that $\sigma' =_\beta \sigma$ and $\mathrm{FTV}(\sigma') \subseteq \mathrm{FTV}(t)$ and one of the following holds.*
- *$t \equiv x$ is a variable with $(x : \sigma') \in \mathcal{V}$.*
- *$t \equiv \mathtt{f}$ is a function symbol with $\mathtt{f} : \sigma'$ in Σ.*
- *$t \equiv \lambda x : \tau_1.s$ and $\sigma' = \tau_1 \to \tau_2$ and $s : \tau_2$.*
- *$t \equiv \Lambda\alpha : \kappa.s$ and $\sigma' = \forall\alpha : \kappa.\tau$ and $s : \tau$ and α does not occur free in the type of a free variable of s.*
- *$t \equiv t_1 \cdot t_2$ and $t_1 : \tau \to \sigma'$ and $t_2 : \tau$ and $\mathrm{FTV}(\tau) \subseteq \mathrm{FTV}(t)$.*
- *$t \equiv s * \tau$ and $\sigma' = \rho[\alpha := \tau]$ and $s : \forall(\alpha : \kappa).\rho$ and τ is a type constructor of kind κ.*

Proof. By analysing the derivation $t : \sigma$. To ensure $\mathrm{FTV}(\sigma') \subseteq \mathrm{FTV}(t)$, note that if $\alpha \notin \mathrm{FTV}(t)$ is of kind κ and $t : \sigma'$, then $t : \sigma'[\alpha := \chi_\kappa]$ by the substitution lemma (thus we can eliminate α). ◀

3 Polymorphic Functional Systems

In this section, we present a form of higher-order term rewriting systems based on $\mathtt{F}_\omega$: *Polymorphic Functional Systems (PFSs)*. Systems of interest, such as logic systems like ICP2 and higher-order TRSs with shallow or full polymorphism can be encoded into PFSs, and then proved terminating with the technique we will develop in Sections 4 – 6.

▶ **Definition 3.1.** Kinds, type constructors *and* types *are defined like in Definition 2.1, parameterised by a fixed set* $\Sigma^T = \bigcup_\kappa \Sigma^T_\kappa$ *of type constructor symbols.*

Let Σ *be a set of function symbols such that for* $\mathtt{f} : \sigma \in \Sigma$*:*

$$\sigma = \forall(\alpha_1 : \kappa_1) \ldots \forall(\alpha_n : \kappa_n).\sigma_1 \to \ldots \to \sigma_k \to \tau \qquad (\text{with } \tau \text{ a type atom})$$

We define PFS terms *as in Definition 2.5 (based on Definition 2.3), parameterised by* Σ*, with the restriction that for any subterm* $s \cdot u$ *of a term* t*, we have* $s = \mathtt{f}\rho_1 \ldots \rho_n u_1 \ldots u_m$ *where:*

$$\mathtt{f} : \forall(\alpha_1 : \kappa_1) \ldots \forall(\alpha_n : \kappa_n).\sigma_1 \to \ldots \to \sigma_k \to \tau \qquad (\text{with } \tau \text{ a type atom and } k > m)$$

This definition does not allow for a variable or abstraction to occur at the head of an application, nor can we have terms of the form $s \cdot t * \tau \cdot q$ (although terms of the form $s \cdot t * \tau$, or $x * \tau$ with x a variable, *are* allowed to occur). To stress this restriction, we will use the notation $\mathtt{f}_{\rho_1,\ldots,\rho_n}(s_1, \ldots, s_m)$ as an alternative way to denote $\mathtt{f}\rho_1 \ldots \rho_n s_1 \ldots s_m$ when $\mathtt{f} : \forall(\alpha_1 : \kappa_1) \ldots \forall(\alpha_n : \kappa_n).\sigma_1 \to \ldots \to \sigma_k \to \tau$ is a function symbol in Σ with τ a type atom and $m \leq k$. This allows us to represent terms in a "functional" way, where application does not explicitly occur (only implicitly in the construction of $\mathtt{f}_{\rho_1,\ldots,\rho_n}(s_1, \ldots, s_m)$).

The following result follows easily by induction on term structure:

▶ **Lemma 3.2.** *If* t, s *are PFS terms then so is* $t[x := s]$*.*

PFS terms will be rewritten through a reduction relation $\longrightarrow_\mathcal{R}$ based on a (usually infinite) set of rewrite rules. To define this relation, we need two additional notions.

▶ **Definition 3.3.** *A* replacement *is a function* $\delta = \gamma \circ \omega$ *satisfying:*

1. ω *is a type constructor substitution,*
2. γ *is a term substitution such that* $\gamma(\omega(x)) : \omega(\tau)$ *for every* $(x : \tau) \in \mathcal{V}$*.*

For τ *a type constructor, we use* $\delta(\tau)$ *to denote* $\omega(\tau)$*. We use the notation* $\delta[x := t] = \gamma[x := t] \circ \omega$*. Note that if* $t : \tau$ *then* $\delta(t) : \delta(\tau)$*.*

▶ **Definition 3.4.** *A* σ-context C_σ *is a PFS term with a fresh function symbol* $\Box_\sigma \notin \Sigma$ *of type* σ *occurring exactly once. By* $C_\sigma[t]$ *we denote a PFS term obtained from* C_σ *by substituting* t *for* $\Box_\sigma$*. We drop the* σ *subscripts when clear or irrelevant.*

Now, the rewrite rules are simply a set of term pairs, whose monotonic closure generates the rewrite relation.

▶ **Definition 3.5.** *A set* $\mathcal{R}$ *of term pairs* (ℓ, r) *is a set of* rewrite rules *if: (a)* $\mathrm{FV}(r) \subseteq \mathrm{FV}(\ell)$*; (b)* ℓ *and* r *have the same type; and (c) if* $(\ell, r) \in \mathcal{R}$ *then* $(\delta(\ell), \delta(r)) \in \mathcal{R}$ *for any replacement* δ*. The reduction relation* $\longrightarrow_\mathcal{R}$ *on PFS terms is defined by:*

$$t \longrightarrow_\mathcal{R} s \text{ iff } t = C[\ell] \text{ and } s = C[r] \text{ for some } (\ell, r) \in \mathcal{R} \text{ and context } C.$$

▶ **Definition 3.6.** *A* Polymorphic Functional System (PFS) *is a triple* $(\Sigma^T, \Sigma, \mathcal{R})$ *where* Σ^T *is a set of type constructor symbols,* Σ *a set of function symbols (restricted as in Def. 3.1), and* $\mathcal{R}$ *is a set of rules as in Definition 3.5. A term of a PFS* A *is referred to as an* A*-term.*

While PFS-terms are a restriction from the general terms of system $\mathtt{F}_\omega$, the reduction relation allows us to actually encode, e.g., system $\mathtt{F}$ as a PFS: we can do so by including the symbol $@ : \forall\alpha\forall\beta.(\alpha \to \beta) \to \alpha \to \beta$ in Σ and adding all rules of the form $@_{\sigma,\tau}(\lambda x.s, t) \longrightarrow s[x := t]$. Similarly, β-reduction of type abstraction can be modelled by including a symbol

$\mathtt{A} : \forall\alpha : * \Rightarrow *.\forall\beta.(\forall\gamma.\alpha\gamma) \to \alpha\beta$ and rules $\mathtt{A}_{\lambda\gamma.\sigma,\tau}(\Lambda\gamma.s) \longrightarrow s[\gamma := \tau]$.[1] We can also use rules $(\Lambda\alpha.s) * \tau \longrightarrow s[\alpha := \tau]$ without the extra symbol, but to apply our method it may be convenient to use the extra symbol, as it creates more liberty in choosing an interpretation.

▶ **Example 3.7** (Fold on heterogenous lists)**.** The example from the introduction may be represented as a PFS with one type symbol $\mathtt{List} : *$, the following function symbols:

$$\begin{array}{rcl} @ & : & \forall\alpha\forall\beta.(\alpha \to \beta) \to \alpha \to \beta \\ \mathtt{A} & : & \forall\alpha : * \Rightarrow *.\forall\beta.(\forall\gamma.\alpha\gamma) \to \alpha\beta \\ \mathtt{nil} & : & \mathtt{List} \\ \mathtt{cons} & : & \forall\alpha.\alpha \to \mathtt{List} \to \mathtt{List} \\ \mathtt{foldl} & : & \forall\beta.(\forall\alpha.\beta \to \alpha \to \beta) \to \beta \to \mathtt{List} \to \beta \end{array}$$

and the following rules (which formally represents an infinite set of rules: one rule for each choice of types σ, τ and PFS terms s, t, etc.):

$$\begin{array}{rcl} @_{\sigma,\tau}(\lambda x : \sigma.s, t) & \longrightarrow & s[x := t] \\ \mathtt{A}_{\lambda\alpha.\sigma,\tau}(\Lambda\alpha.s) & \longrightarrow & s[\alpha := \tau] \\ \mathtt{foldl}_\sigma(f, s, \mathtt{nil}) & \longrightarrow & s \\ \mathtt{foldl}_\sigma(f, s, \mathtt{cons}_\tau(h, t)) & \longrightarrow & \mathtt{foldl}_\sigma(f, @_{\tau,\sigma}(@_{\sigma,\tau\to\sigma}(\mathtt{A}_{\lambda\alpha.\sigma\to\alpha\to\sigma,\tau}(f), s), h), t) \end{array}$$

4 A well-ordered set of interpretation terms

In polynomial interpretations of first-order term rewriting [21, Chapter 6.2], each term s is mapped to a natural number $[\![s]\!]$, such that $[\![s]\!] > [\![t]\!]$ whenever $s \longrightarrow_{\mathcal{R}} t$. In higher-order rewriting, this is not practical; instead, following [15], terms are mapped to weakly monotonic functionals according to their type (i.e., terms with a 0-order type are mapped to natural numbers, terms with a 1-order type to weakly monotonic functions over natural numbers, terms with a 2-order type to weakly monotonic functionals taking weakly monotonic functions as arguments, and so on). In this paper, to account for full polymorphism, we will interpret PFS terms to a set $\mathcal{I}$ of *interpretation terms* in a specific extension of System $\mathtt{F}_\omega$. This set is defined in Section 4.1; we provide a well-founded partial ordering $\succ$ on $\mathcal{I}$ in Section 4.2.

Although our world of interpretation terms is quite different from the weakly monotonic functionals of [15], there are many similarities. Most pertinently, every interpretation term $\lambda x.s$ essentially defines a weakly monotonic function from $\mathcal{I}$ to $\mathcal{I}$. This, and the use of both addition and multiplication in the definition of $\mathcal{I}$, makes it possible to lift higher-order polynomial interpretations [6] to our setting. We prove weak monotonicity in Section 4.3.

4.1 Interpretation terms

▶ **Definition 4.1.** *The set $\mathcal{Y}$ of* interpretation types *is the set of types as in Definition 2.1 with $\Sigma^T = \{\mathtt{nat} : *\}$, i.e., there is a single type constant* nat*. Then $\chi_* = \mathtt{nat}$.*

The set $\mathcal{I}$ of interpretation terms *is the set of terms from Definition 2.5 (see also Definition 2.3) where as types we take the interpretation types and for the set Σ of function symbols we take $\Sigma = \{n : \mathtt{nat} \mid n \in \mathbb{N}\} \cup \Sigma_f$, where $\Sigma_f = \{\oplus : \forall\alpha.\alpha \to \alpha \to \alpha, \otimes : \forall\alpha.\alpha \to \alpha \to \alpha, \mathtt{flatten} : \forall\alpha.\alpha \to \mathtt{nat}, \mathtt{lift} : \forall\alpha.\mathtt{nat} \to \alpha\}$.*

[1] The use of a type constructor variable α of kind $* \Rightarrow *$ makes it possible to do type substitution as part of a rule. An application $s * \tau$ with $s : \forall\gamma.\sigma$ is encoded as $\mathtt{A}_{\lambda\gamma.\sigma,\tau}(s)$, so α is substituted with $\lambda\gamma.\tau$. This is well-typed because $(\lambda\gamma.\sigma)\gamma =_\beta \sigma$ and $(\lambda\gamma.\sigma)\tau =_\beta \sigma[\gamma := \tau]$.

For easier presentation, we write $\oplus_\tau$, $\otimes_\tau$, etc., instead of $\oplus * \tau$, $\otimes * \tau$, etc. We will also use $\oplus$ and $\otimes$ in *infix, left-associative* notation, and omit the type denotation where it is clear from context. Thus, $s \oplus t \oplus u$ should be read as $\oplus_\sigma (\oplus_\sigma s\, t)\, u$ if s has type σ. Thus, our interpretation terms include natural numbers with the operations of addition and multiplication. It would not cause any fundamental problems to add more monotonic operations, e.g., exponentiation, but we refrain from doing so for the sake of simplicity.

Normalising interpretation terms

The set $\mathcal{I}$ of interpretation terms can be reduced through a relation $\rightsquigarrow$, that we will define below. This relation will be a powerful aid in defining the partial ordering $\succ$ in Section 4.2.

▶ **Definition 4.2.** *We define the relation $\rightsquigarrow$ on interpretation terms as the smallest relation on $\mathcal{I}$ for which the following properties are satisfied:*

1. *if $s \rightsquigarrow t$ then both $\lambda x.s \rightsquigarrow \lambda x.t$ and $\Lambda\alpha.s \rightsquigarrow \Lambda\alpha.t$*
2. *if $s \rightsquigarrow t$ then $u \cdot s \rightsquigarrow u \cdot t$*
3. *if $s \rightsquigarrow t$ then both $s \cdot u \rightsquigarrow t \cdot u$ and $s * \sigma \rightsquigarrow t * \sigma$*
4. *$(\lambda x : \sigma.s) \cdot t \rightsquigarrow s[x := t]$ and $(\Lambda\alpha.s) * \sigma \rightsquigarrow s[\alpha := \sigma]$ (β-reduction)*
5. *$\oplus_{\mathtt{nat}} \cdot n \cdot m \rightsquigarrow n + m$ and $\otimes_{\mathtt{nat}} \cdot n \cdot m \rightsquigarrow n \times m$*
6. *$\circ_{\sigma\to\tau} \cdot s \cdot t \rightsquigarrow \lambda x : \sigma.\, \circ_\tau \cdot (s \cdot x) \cdot (t \cdot x)$ for $\circ \in \{\oplus, \otimes\}$*
7. *$\circ_{\forall\alpha.\sigma} \cdot s \cdot t \rightsquigarrow \Lambda\alpha.\, \circ_\sigma \cdot (s * \alpha) \cdot (t * \alpha)$ for $\circ \in \{\oplus, \otimes\}$*
8. *$\mathtt{flatten}_{\mathtt{nat}} \cdot s \rightsquigarrow s$*
9. *$\mathtt{flatten}_{\sigma\to\tau} \cdot s \rightsquigarrow \mathtt{flatten}_\tau \cdot (s \cdot (\mathtt{lift}_\sigma \cdot 0))$*
10. *$\mathtt{flatten}_{\forall\alpha:\kappa.\sigma} \cdot s \rightsquigarrow \mathtt{flatten}_{\sigma[\alpha:=\chi_\kappa]} \cdot (s * \chi_\kappa)$*
11. *$\mathtt{lift}_{\mathtt{nat}} \cdot s \rightsquigarrow s$*
12. *$\mathtt{lift}_{\sigma\to\tau} \cdot s \rightsquigarrow \lambda x : \sigma.\mathtt{lift}_\tau \cdot s$*
13. *$\mathtt{lift}_{\forall\alpha.\sigma} \cdot s \rightsquigarrow \Lambda\alpha.\mathtt{lift}_\sigma \cdot s$*

Recall Definition 2.5 and Definition 4.1 of the set of interpretation terms $\mathcal{I}$ as the set of the equivalence classes of $\equiv$. So, for instance, $\mathtt{lift}_{\mathtt{nat}}$ above denotes the equivalence class of all preterms $\mathtt{lift}_\sigma$ with $\sigma =_\beta \mathtt{nat}$. Hence, the above rules are invariant under $\equiv$ (by confluence of β-reduction on types), and they correctly define a relation on interpretation terms. We say that s is a redex *if s reduces by one of the rules 4–13. A* final interpretation term *is an interpretation term $s \in \mathcal{I}$ such that (a) s is closed, and (b) s is in normal form with respect to $\rightsquigarrow$. We let $\mathcal{I}^f$ be the set of all final interpretation terms. By $\mathcal{I}_\tau$ ($\mathcal{I}^f_\tau$) we denote the set of all (final) interpretation terms of interpretation type τ.*

An important difference with System $\mathtt{F}_\omega$ and related ones is that the rules for $\oplus_\tau$, $\otimes_\tau$, $\mathtt{flatten}_\tau$ and $\mathtt{lift}_\tau$ depend on the type τ. In particular, type substitution in terms may create redexes. For instance, if α is a type variable then $\oplus_\alpha t_1 t_2$ is not a redex, but $\oplus_{\sigma\to\tau} t_1 t_2$ is. This makes the question of termination subtle. Indeed, System $\mathtt{F}_\omega$ is extremely sensitive to modifications which are not of a logical nature. For instance, adding a constant $\mathtt{J} : \forall\alpha\beta.\alpha \to \beta$ with a reduction rule $\mathtt{J}\tau\tau \rightsquigarrow \lambda x : \tau.x$ makes the system non-terminating [8]. This rule breaks parametricity by making it possible to compare two arbitrary types. Our rules do not allow such a definition. Moreover, the natural number constants cannot be distinguished "inside" the system. In other words, we could replace all natural number constants with 0 and this would not change the reduction behaviour of terms. So for the purposes of termination, the type $\mathtt{nat}$ is essentially a singleton. This implies that, while we have polymorphic functions between an arbitrary type α and $\mathtt{nat}$ which are not constant when seen "from outside" the system, they are constant for the purposes of reduction "inside" the system (as they would have to be in a parametric $\mathtt{F}_\omega$-like system). Intuitively, these properties of our system ensure that it stays "close enough" to $\mathtt{F}_\omega$ so that the standard termination proof still generalises.

Now we state some properties of $\rightsquigarrow$, including strong normalisation. Because of space limitations, most (complete) proofs are delegated to [3, Appendix A.1].

▶ **Lemma 4.3** (Subject reduction). *If $t : \tau$ and $t \rightsquigarrow t'$ then $t' : \tau$.*

Proof. By induction on the definition of $t \rightsquigarrow t'$, using Lemmas 2.6 and 2.7. ◀

▶ **Theorem 4.4.** *If $t : \sigma$ then t is terminating with respect to $\rightsquigarrow$.*

Proof. By an adaptation of the Tait-Girard computability method. The proof is an adaptation of chapters 6 and 14 from the book [7], and chapters 10 and 11 from the book [17]. Details are available in [3, Appendix A.1]. ◀

▶ **Lemma 4.5.** *Every term $s \in \mathcal{I}$ has a unique normal form $s{\downarrow}$. If s is closed then so is $s{\downarrow}$.*

Proof. One easily checks that $\rightsquigarrow$ is locally confluent. Since the relation is terminating by Theorem 4.4, it is confluent by Newman's lemma. ◀

▶ **Lemma 4.6.** *The only final interpretation terms of type* $\mathtt{nat}$ *are the natural numbers.*

▶ **Example 4.7.** Let $s \in \mathcal{I}_{\mathtt{nat}\to\mathtt{nat}}$ and $t \in \mathcal{I}_{\mathtt{nat}}$. Then we can reduce $(s \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1)) \cdot t \rightsquigarrow (\lambda x.sx \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1)x) \cdot t \rightsquigarrow st \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1)t \rightsquigarrow st \oplus (\lambda y.\mathtt{lift}_{\mathtt{nat}}(1))t \rightsquigarrow st \oplus \mathtt{lift}_{\mathtt{nat}}(1) \rightsquigarrow st \oplus 1$. If s and t are variables, this term is in normal form.

4.2 The ordering pair $(\succeq, \succ)$

With these ingredients, we are ready to define the well-founded partial ordering $\succ$ on $\mathcal{I}$. In fact, we will do more: rather than a single partial ordering, we will define an *ordering pair*: a pair of a quasi-ordering $\succeq$ and a compatible well-founded ordering $\succ$. The quasi-ordering $\succeq$ often makes it easier to prove $s \succ t$, since it suffices to show that $s \succeq s' \succ t' \succeq t$ for some interpretation terms s', t'. Having $\succeq$ will also allow us to use rule removal (Theorem 6.1).

▶ **Definition 4.8.** *Let $R \in \{\succ^0, \succeq^0\}$. For* closed *$s, t \in \mathcal{I}_\sigma$ and closed σ in β-normal form, the relation $s\ R_\sigma\ t$ is defined coinductively by the following rules.*

$$\frac{s{\downarrow}\ R\ t{\downarrow} \text{ in } \mathbb{N}}{s\ R_{\mathtt{nat}}\ t} \qquad \frac{s \cdot q\ R_\tau\ t \cdot q \text{ for all } q \in \mathcal{I}^f_\sigma}{s\ R_{\sigma\to\tau}\ t} \qquad \frac{s * \tau\ R_{\mathrm{nf}_\beta(\sigma[\alpha:=\tau])}\ t * \tau \text{ for all closed } \tau \in \mathcal{T}_\kappa}{s\ R_{\forall(\alpha:\kappa).\sigma}\ t}$$

We define $s \approx^0_\sigma t$ if both $s \succeq^0_\sigma t$ and $t \succeq^0_\sigma s$. We drop the type subscripts when clear or irrelevant.

Note that in the case for $\mathtt{nat}$ the terms $s{\downarrow}$, $t{\downarrow}$ are natural numbers by Lemma 4.6 ($s{\downarrow}, t{\downarrow}$ are closed and in normal form, so they are final interpretation terms).

Intuitively, the above definition means that e.g. $s \succ^0 t$ iff there exists a possibly infinite derivation tree using the above rules. In such a derivation tree all leaves must witness $s{\downarrow} > t{\downarrow}$ in natural numbers. However, this also allows for infinite branches, which solves the problem of repeating types due to impredicative polymorphism. If e.g. $s \succ^0_{\forall\alpha.\alpha} t$ then $s * \forall\alpha.\alpha \succ^0_{\forall\alpha.\alpha} t * \forall\alpha.\alpha$, which forces an infinite branch in the derivation tree. According to our definition, any infinite branch may essentially be ignored.

Formally, the above coinductive definition of e.g. $\succ^0_\sigma$ may be interpreted as defining the largest relation such that if $s \succ^0_\sigma t$ then:

- $\sigma = \mathtt{nat}$ and $s{\downarrow} > t{\downarrow}$ in $\mathbb{N}$, or
- $\sigma = \tau_1 \to \tau_2$ and $s \cdot q \succ^0_{\tau_2} t \cdot q$ for all $q \in \mathcal{I}^f_{\tau_1}$, or
- $\sigma = \forall(\alpha : \kappa).\rho$ and $s * \tau \succ^0_{\mathrm{nf}_\beta(\rho[\alpha:=\tau])} t * \tau$ for all closed $\tau \in \mathcal{T}_\kappa$.

For more background on coinduction see e.g. [14, 16, 10]. In this paper we use a few simple coinductive proofs to establish the basic properties of $\succ$ and $\succeq$. Later, we just use these properties and the details of the definition do not matter.

▶ **Definition 4.9.** *A* closure $\mathcal{C} = \gamma \circ \omega$ *is a replacement such that* $\omega(\alpha)$ *is closed for each type constructor variable* α*, and* $\gamma(x)$ *is closed for each term variable* x*. For arbitrary types* σ *and arbitrary terms* $s, t \in \mathcal{I}$ *we define* $s \succ_\sigma t$ *if for every closure* $\mathcal{C}$ *we can obtain* $\mathcal{C}(s) \succ^{c}_{\mathrm{nf}_\beta(\mathcal{C}(\sigma))} \mathcal{C}(t)$ *coinductively with the above rules. The relations* $\succeq_\sigma$ *and* $\approx_\sigma$ *are defined analogously.*

Note that for closed s, t and closed σ in β-normal form, $s \succ_\sigma t$ iff $s \succ^0_\sigma t$ (and analogously for $\succeq, \approx$). In this case we shall often omit the superscript 0.

The definition of $\succ$ and $\succeq$ may be reformulated as follows.

▶ **Lemma 4.10.** $t \succeq s$ *if and only if for every closure* $\mathcal{C}$ *and every sequence* $u_1, \ldots, u_n$ *of closed terms and closed type constructors such that* $\mathcal{C}(t)u_1 \ldots u_n : \mathtt{nat}$ *we have* $(\mathcal{C}(t)u_1 \ldots u_n)\!\downarrow \geq (\mathcal{C}(s)u_1 \ldots u_n)\!\downarrow$ *in natural numbers. An analogous result holds with* $\succ$ *or* $\approx$ *instead of* $\succeq$*.*

Proof. The direction from left to right follows by induction on n; the other by coinduction. ◀

In what follows, all proofs by coinduction could be reformulated to instead use the lemma above. However, this would arguably make the proofs less perspicuous. Moreover, a coinductive definition is better suited for a formalisation – the coinductive proofs here could be written in Coq almost verbatim.

Our next task is to show that $\succeq$ and $\succ$ have the desired properties of an ordering pair; e.g., transitivity and compatibility. We first state a simple lemma that will be used implicitly.

▶ **Lemma 4.11.** *If* $\tau \in \mathcal{Y}$ *is closed and* β*-normal, then* $\tau = \mathtt{nat}$ *or* $\tau = \tau_1 \to \tau_2$ *or* $\tau = \forall \alpha \sigma$*.*

▶ **Lemma 4.12.** $\succ$ *is well-founded.*

Proof. It suffices to show this for closed terms and closed types in β-normal form, because any infinite sequence $t_1 \succ_\tau t_2 \succ_\tau t_3 \succ_\tau \ldots$ induces an infinite sequence $\mathcal{C}(t_1) \succ_{\mathrm{nf}_\beta(\mathcal{C}(\tau))} \mathcal{C}(t_2) \succ_{\mathrm{nf}_\beta(\mathcal{C}(\tau))} \mathcal{C}(t_3) \succ_{\mathrm{nf}_\beta(\mathcal{C}(\tau))} \ldots$ for any closure $\mathcal{C}$. By induction on the size of a β-normal type τ (with size measured as the number of occurrences of $\forall$ and $\to$) one proves that there does not exist an infinite sequence $t_1 \succ_\tau t_2 \succ_\tau t_3 \succ_\tau \ldots$ For instance, if α has kind κ and $t_1 \succ_{\forall\alpha\tau} t_2 \succ_{\forall\alpha\tau} t_3 \succ_{\forall\alpha\tau} \ldots$ then $t_1 * \chi_\kappa \succ_{\tau'} t_2 * \chi_\kappa \succ_{\tau'} t_3 * \chi_\kappa \succ_{\tau'} \ldots$, where $\tau' = \mathrm{nf}_\beta(\tau[\alpha := \chi_\kappa])$. Because τ is in β-normal form, all redexes in $\tau[\alpha := \chi_\kappa]$ are created by the substitution and must have the form $\chi_\kappa u$. Hence, by the definition of χ_κ (see Definition 2.1) the type τ' is smaller than τ. This contradicts the inductive hypothesis. ◀

▶ **Lemma 4.13.** *Both* $\succ$ *and* $\succeq$ *are transitive.*

Proof. We show this for $\succ$, the proof for $\succeq$ being analogous. Again, it suffices to prove this for closed terms and closed types in β-normal form. We proceed by coinduction.

If $t_1 \succ_{\mathtt{nat}} t_2 \succ_{\mathtt{nat}} t_3$ then $t_1\!\downarrow > t_2\!\downarrow > t_3\!\downarrow$, so $t_1\!\downarrow > t_3\!\downarrow$. Thus $t_1 \succ_{\mathtt{nat}} t_3$.

If $t_1 \succ_{\sigma\to\tau} t_2 \succ_{\sigma\to\tau} t_3$ then $t_1 \cdot q \succ_\tau t_2 \cdot q \succ_\tau t_3 \cdot q$ for $q \in \mathcal{I}^f_\sigma$. Hence $t_1 \cdot q \succ_\tau t_3 \cdot q$ for $q \in \mathcal{I}^f_\sigma$ by the coinductive hypothesis. Thus $t_1 \succ_{\sigma\to\tau} t_3$.

If $t_1 \succ_{\forall(\alpha:\kappa)\sigma} t_2 \succ_{\forall(\alpha:\kappa)\sigma} t_3$ then $t_1 * \tau \succ_{\sigma'} t_2 * \tau \succ_{\sigma'} t_3 * \tau$ for any closed τ of kind κ, where $\sigma' = \mathrm{nf}_\beta(\sigma[\alpha := \tau])$. By the coinductive hypothesis $t_1 * \tau \succ_{\sigma'} t_3 * \tau$; thus $t_1 \succ_{\forall\alpha\sigma} t_3$. ◀

▶ **Lemma 4.14.** $\succeq$ *is reflexive.*

Proof. By coinduction one shows that $\succeq_\sigma$ is reflexive on closed terms for closed β-normal σ. The case of $\succeq$ is then immediate from definitions. ◀

▶ **Lemma 4.15.** *The relations $\succeq$ and $\succ$ are compatible, i.e., $\succ \cdot \succeq \subseteq \succ$ and $\succeq \cdot \succ \subseteq \succ$.*

Proof. By coinduction, analogous to the transitivity proof. ◀

▶ **Lemma 4.16.** *If $t \succ s$ then $t \succeq s$.*

Proof. By coinduction. ◀

▶ **Lemma 4.17.** *If $t \rightsquigarrow s$ then $t \approx s$.*

Proof. Follows from Lemma 4.10, noting that $t \rightsquigarrow s$ implies $\mathcal{C}(t) \rightsquigarrow \mathcal{C}(s)$ for all closures $\mathcal{C}$. ◀

▶ **Lemma 4.18.** *Assume $t \succ s$ (resp. $t \succeq s$). If $t \rightsquigarrow t'$ or $t' \rightsquigarrow t$ then $t' \succ s$ (resp. $t' \succeq s$). If $s \rightsquigarrow s'$ or $s' \rightsquigarrow s$ then $t \succ s'$ (resp. $t \succeq s'$).*

Proof. Follows from Lemma 4.17, transitivity and compatibility. ◀

▶ **Corollary 4.19.** *For $R \in \{\succ, \succeq, \approx\}$: $s\ R\ t$ if and only if $s\!\downarrow\ R\ t\!\downarrow$.*

▶ **Example 4.20.** We can prove that $x \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1) \succ x$: by definition, this holds if $s \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1) \succ s$ for all closed s, so if $(s \oplus \mathtt{lift}_{\mathtt{nat}\to\mathtt{nat}}(1))u \succ su$ for all closed s, u. Following Example 4.7 and Lemma 4.18, this holds if $su \oplus 1 \succ su$. By definition, this is the case if $(su \oplus 1)\!\downarrow > (su)\!\downarrow$ in the natural numbers, which clearly holds for any s, u.

4.3 Weak monotonicity

We will now show that $s \succeq s'$ implies $t[x := s] \succeq t[x := s']$ (weak monotonicity). For this purpose, we prove a few lemmas, many of which also apply to $\succ$, stating the preservation of $\succeq$ under term formation operations. We will need these results in the next section.

▶ **Lemma 4.21.** *For $R \in \{\succeq, \succ\}$: if $t\ R\ s$ then $tu\ R\ su$ with u a term or type constructor.*

Proof. Follows from definitions. ◀

▶ **Lemma 4.22.** *For $R \in \{\succeq, \succ\}$: if $n\ R\ m$ then $\mathtt{lift}_\sigma n\ R\ \mathtt{lift}_\sigma m$ for all types σ.*

Proof. Without loss of generality we may assume σ closed and in β-normal form. By coinduction we show $\mathtt{lift}(n)u_1 \dots u_k \succeq \mathtt{lift}(m)u_1 \dots u_k$ for closed $u_1, \dots, u_k$. First note that $(\mathtt{lift}\, t)u_1 \dots u_k \rightsquigarrow^* \mathtt{lift}(t)$ (with a different type subscript in $\mathtt{lift}$ on the right side, omitted for conciseness). If $\sigma = \mathtt{nat}$ then $(\mathtt{lift}(n)u_1 \dots u_k)\!\downarrow = n \geq m = (\mathtt{lift}(m)u_1 \dots u_k)\!\downarrow$. If $\sigma = \tau_1 \to \tau_2$ then by the coinductive hypothesis $\mathtt{lift}(n)u_1 \dots u_k q \succeq_{\tau_2} \mathtt{lift}(m)u_1 \dots u_k q$ for any $q \in \mathcal{I}^f_{\tau_1}$, so $\mathtt{lift}(n)u_1 \dots u_k \succeq_\sigma \mathtt{lift}(m)u_1 \dots u_k$ by definition. If $\sigma = \forall(\alpha : \kappa)\tau$ then by the coinductive hypothesis $\mathtt{lift}(n)u_1 \dots u_k\xi \succeq_{\sigma'} \mathtt{lift}(m)u_1 \dots u_k\xi$ for any closed $\xi \in \mathcal{T}_\kappa$, where $\sigma' = \tau[\alpha := \xi]$. Hence $\mathtt{lift}(n)u_1 \dots u_k \succeq_\sigma \mathtt{lift}(m)u_1 \dots u_k$ by definition. ◀

▶ **Lemma 4.23.** *For $R \in \{\succeq, \succ\}$: if $t\ R_\sigma\ s$ then $\mathtt{flatten}_\sigma t\ R_{\mathtt{nat}}\ \mathtt{flatten}_\sigma s$ for all types σ.*

Proof. Without loss of generality we may assume σ is closed and in β-normal form. Using Lemma 4.18, the lemma follows by induction on σ. ◀

▶ **Lemma 4.24.** *For $R \in \{\succeq, \succ\}$: if $t\ R\ s$ then $\lambda x.t\ R\ \lambda x.s$ and $\Lambda\alpha.t\ R\ \Lambda\alpha.s$.*

Proof. Assume $t \succeq_\tau s$ and $x : \sigma$. Let $\mathcal{C}$ be a closure. We need to show $\mathcal{C}(\lambda x.t) \succeq_{\mathcal{C}(\sigma\to\tau)} \mathcal{C}(\lambda x.s)$. Let $u \in \mathcal{I}^f_{\mathcal{C}(\sigma)}$. Then $\mathcal{C}' = \mathcal{C}[x := u]$ is a closure and $\mathcal{C}'(t) \succeq_{\mathcal{C}(\tau)} \mathcal{C}'(s)$. Hence $\mathcal{C}(t)[x := u] \succeq_{\mathcal{C}(\tau)} \mathcal{C}(s)[x := u]$. By Lemma 4.18 this implies $\mathcal{C}(\lambda x.t)u \succeq_{\mathcal{C}(\tau)} \mathcal{C}(\lambda x.s)u$. Therefore $\mathcal{C}(\lambda x.t) \succeq_{\mathcal{C}(\sigma\to\tau)} \mathcal{C}(\lambda x.s)$. The proof for $\succ$ is analogous. ◀

▶ **Lemma 4.25.** *Let s, t, u be terms of type σ.*

1. *If $s \succeq t$ then $s \oplus_\sigma u \succeq t \oplus_\sigma u$, $u \oplus_\sigma s \succeq u \oplus_\sigma t$, $s \otimes_\sigma u \succeq t \otimes_\sigma u$, and $u \otimes_\sigma s \succeq u \otimes_\sigma t$.*
2. *If $s \succ t$ then $s \oplus_\sigma u \succ t \oplus_\sigma u$ and $u \oplus_\sigma s \succ u \oplus_\sigma t$. Moreover, if additionally $u \succeq \mathtt{lift}_\sigma(1)$ then also $s \otimes_\sigma u \succ t \otimes_\sigma u$ and $u \otimes_\sigma s \succ u \otimes_\sigma t$.*

Proof. It suffices to prove this for closed s, t, u and closed σ in β-normal form. The proof is similar to the proof of Lemma 4.22. For instance, we show by coinduction that for closed $w_1, \dots, w_n$ (denoted $\vec{w}$): if $s\vec{w} \succ t\vec{w}$ and $u\vec{w} \succeq \mathtt{lift}(1)\vec{w}$ then $(s \otimes u)\vec{w} \succ (t \otimes u)\vec{w}$. ◀

The following lemma depends on the lemmas above. The full proof may be found in [3, Appendix A.2]. The proof is actually quite complex, and uses a method similar to Girard's method of candidates for the termination proof.

▶ **Lemma 4.26** (Weak monotonicity)**.** *If $s \succeq s'$ then $t[x := s] \succeq t[x := s']$.*

▶ **Corollary 4.27.** *If $s \succeq s'$ then $ts \succeq ts'$.*

5 A reduction pair for PFS terms

Recall that our goal is to prove termination of reduction in a PFS. To do so, in this section we will define a systematic way to generate *reduction pairs*. We fix a PFS A, and define:

▶ **Definition 5.1.** *A binary relation R on A-terms is* monotonic *if $R(s, t)$ implies $R(C[s], C[t])$ for every context C (we assume s, t have the same type σ).*

A reduction pair *is a pair $(\succeq^A, \succ^A)$ of a quasi-order $\succeq^A$ on A-terms and a well-founded ordering $\succ^A$ on A-terms such that: (a) $\succeq^A$ and $\succ^A$ are compatible, i.e., $\succ^A \cdot \succeq^A \subseteq \succ^A$ and $\succeq^A \cdot \succ^A \subseteq \succ^A$, and (b) $\succeq^A$ and $\succ^A$ are both monotonic.*

If we can generate such a pair with $\ell \succ^A r$ for each rule $(\ell, r) \in \mathcal{R}$, then we easily see that the PFS A is terminating. (If we merely have $\ell \succ^A r$ for *some* rules and $\ell \succeq^A r$ for the rest, we can still progress with the termination proof, as we will discuss in Section 6.) To generate this pair, we will define the notion of an *interpretation* from the set of A-terms to the set $\mathcal{I}$ of interpretation terms, and thus lift the ordering pair $(\succeq, \succ)$ to A. In the next section, we will show how this reduction pair can be used in practice to prove termination of PFSs.

One of the core ingredients of our interpretation function is a mapping to translate types:

▶ **Definition 5.2.** *A* type constructor mapping *is a function $\mathcal{TM}$ which maps each type constructor symbol to a closed interpretation type constructor of the same kind. A fixed type constructor mapping $\mathcal{TM}$ is extended inductively to a function from type constructors to closed interpretation type constructors in the expected way. We denote the extended* interpretation (type) mapping *by $[\![\sigma]\!]$. Thus, e.g. $[\![\forall\alpha.\sigma]\!] = \forall\alpha.[\![\sigma]\!]$ and $[\![\sigma \to \tau]\!] = [\![\sigma]\!] \to [\![\tau]\!]$.*

▶ **Lemma 5.3.** $[\![\sigma]\!][\alpha := [\![\tau]\!]] = [\![\sigma[\alpha := \tau]]\!]$

Proof. Induction on σ. ◀

Similarly, we employ a *symbol mapping* as the key ingredient to interpret PFS terms.

▶ **Definition 5.4.** *Given a fixed type constructor mapping $\mathcal{TM}$, a* symbol mapping *is a function $\mathcal{J}$ which assigns to each function symbol $\mathtt{f} : \rho$ a closed interpretation term $\mathcal{J}(\mathtt{f})$ of type $[\![\rho]\!]$. For a fixed symbol mapping $\mathcal{J}$, we define the* interpretation mapping $[\![s]\!]$ *inductively:*

$$\begin{array}{lcllcllcl} [\![x]\!] & = & x & [\![\Lambda\alpha.s]\!] & = & \Lambda\alpha.[\![s]\!] & [\![t_1 \cdot t_2]\!] & = & [\![t_1]\!] \cdot [\![t_2]\!] \\ [\![\mathtt{f}]\!] & = & \mathcal{J}(\mathtt{f}) & [\![\lambda x : \sigma.s]\!] & = & \lambda x : [\![\sigma]\!].[\![s]\!] & [\![t * \tau]\!] & = & [\![t]\!] * [\![\tau]\!] \end{array}$$

Note that $[\![\sigma]\!], [\![\tau]\!]$ above depend on $\mathcal{TM}$. Essentially, $[\![\cdot]\!]$ substitutes $\mathcal{TM}(\mathtt{c})$ for type constructor symbols $\mathtt{c}$, and $\mathcal{J}(\mathtt{f})$ for function symbols $\mathtt{f}$, thus mapping A-terms to interpretation terms. This translation preserves typing:

▶ **Lemma 5.5.** *If $s : \sigma$ then $[\![s]\!] : [\![\sigma]\!]$.*

Proof. By induction on the form of s, using Lemma 5.3. ◀

▶ **Lemma 5.6.** *For all s, t, x, α, τ: $[\![s]\!][\alpha := [\![\tau]\!]] = [\![s[\alpha := \tau]]\!]$ and $[\![s]\!][x := [\![t]\!]] = [\![s[x := t]]\!]$.*

Proof. Induction on s. ◀

▶ **Definition 5.7.** *For a fixed type constructor mapping $\mathcal{TM}$ and symbol mapping $\mathcal{J}$, the* interpretation pair $(\succeq^{\mathcal{J}}, \succ^{\mathcal{J}})$ *is defined as follows: $s \succeq^{\mathcal{J}} t$ if $[\![s]\!] \succeq [\![t]\!]$, and $s \succ^{\mathcal{J}} t$ if $[\![s]\!] \succ [\![t]\!]$.*

▶ Remark 5.8. The polymorphic lambda-calculus has a much greater expressive power than the simply-typed lambda-calculus. Inductive data types may be encoded, along with their constructors and recursors with appropriate derived reduction rules. This makes our interpretation method easier to apply, even in the non-polymorphic setting, thanks to more sophisticated "programming" in the interpretations. The reader is advised to consult e.g. [7, Chapter 11] for more background and explanations. We demonstrate the idea by presenting an encoding for the recursive type `List` and its fold-left function (see also Ex. 5.14).

▶ **Example 5.9.** Towards a termination proof of Example 3.7, we set $\mathcal{TM}(\mathtt{List}) = \forall\beta.(\forall\alpha.\beta \to \alpha \to \beta) \to \beta \to \beta$ and $\mathcal{J}(\mathtt{nil}) = \Lambda\beta.\lambda f : \forall\alpha.\beta \to \alpha \to \beta.\lambda x : \beta.x$. If we additionally choose $\mathcal{J}(\mathtt{foldl}) = \Lambda\beta.\lambda f.\lambda x.\lambda l.l\beta f x \oplus \mathtt{lift}_\beta(1)$, we have $[\![\mathtt{foldl}_\sigma(f, s, \mathtt{nil})]\!] = (\Lambda\beta.\lambda f.\lambda x.\lambda l.l\beta f x \oplus \mathtt{lift}_\beta(1))[\![\sigma]\!] f s (\Lambda\beta.\lambda f.\lambda x.x) \rightsquigarrow^* s \oplus \mathtt{lift}_{[\![\sigma]\!]}(1)$ by β-reduction steps. An extension of the proof from Example 4.20 shows that this term $\succ [\![s]\!]$.

It is easy to see that $\succeq^{\mathcal{J}}$ and $\succ^{\mathcal{J}}$ have desirable properties such as transitivity, reflexivity (for $\succeq^{\mathcal{J}}$) and well-foundedness (for $\succ^{\mathcal{J}}$). However, $\succ^{\mathcal{J}}$ is not necessarily monotonic. Using the interpretation from Example 5.9, $[\![\mathtt{foldl}_\sigma(\lambda x.s, t, \mathtt{nil})]\!] = [\![\mathtt{fold}_\sigma(\lambda x.w, t, \mathtt{nil})]\!]$ regardless of s and w, so a reduction in s would not cause a decrease in $\succ^{\mathcal{J}}$. To obtain a reduction pair, we must impose certain conditions on $\mathcal{J}$; in particular, we will require that $\mathcal{J}$ is *safe*.

▶ **Definition 5.10.** *If $s_1 \succ s_2$ implies $t[x := s_1] \succ t[x := s_2]$, then the interpretation term t is* safe for x. *A symbol mapping $\mathcal{J}$ is* safe *if for all $\mathtt{f} : \forall(\alpha_1 : \kappa_1) \ldots \forall(\alpha_n : \kappa_n).\sigma_1 \to \ldots \to \sigma_k \to \tau$ with τ a type atom we have: $\mathcal{J}(\mathtt{f}) = \Lambda\alpha_1 \ldots \alpha_n.\lambda x_1 \ldots x_k.t$ with t safe for each x_i.*

▶ **Lemma 5.11.**

1. *$x u_1 \ldots u_m$ is safe for x.*
2. *If t is safe for x then so are $\mathtt{lift}(t)$ and $\mathtt{flatten}(t)$.*
3. *If s_1 is safe for x or s_2 is safe for x then $s_1 \oplus s_2$ is safe for x.*
4. *If either (a) s_1 is safe for x and $s_2 \succeq \mathtt{lift}(1)$, or (b) s_2 is safe for x and $s_1 \succeq \mathtt{lift}(1)$, then $s_1 \otimes s_2$ is safe for x.*
5. *If t is safe for x then so is $\Lambda\alpha.t$ and $\lambda y.t$ $(y \neq x)$.*

Proof. Each point follows from one of the lemmas proven before, Lemma 4.16, Lemma 4.26, Lemma 4.15 and the transitivity of $\succeq$. For instance, for the first, assume $s_1 \succ s_2$ and let $u_i^j = u_i[x := s_j]$. Then $(xu_1 \dots u_m)[x := s_1] = s_1u_1^1 \dots u_m^1$. By Lemma 4.21 we have $s_1u_1^1 \dots u_m^1 \succ s_2u_1^1 \dots u_m^1$. By Lemma 4.16 and Lemma 4.26 we have $u_i^1 \succeq u_i^2$. By Corollary 4.27 and the transitivity of $\succeq$ we obtain $s_2u_1^1 \dots u_m^1 \succeq s_2u_1^2 \dots u_m^2$. By Lemma 4.15 finally $(xu_1 \dots u_m)[x := s_1] = s_1u_1^1 \dots u_m^1 \succ s_2u_1^2 \dots u_m^2 = (xu_1 \dots u_m)[x := s_2]$. ◀

▶ **Lemma 5.12.** *If $\mathcal{J}$ is safe then $\succ^{\mathcal{J}}$ is monotonic.*

Proof. Assume $s_1 \succ^{\mathcal{J}} s_2$. By induction on a context C we show $C[s_1] \succ^{\mathcal{J}} C[s_2]$. If $C = \Box$ then this is obvious. If $C = \lambda x.C'$ or $C = \Lambda\alpha.C'$ then $C'[s_1] \succ^{\mathcal{J}} C'[s_2]$ by the inductive hypothesis, and thus $C[s_1] \succ^{\mathcal{J}} C[s_2]$ follows from Lemma 4.24 and definitions. If $C = C't$ then $C'[s_1] \succ^{\mathcal{J}} C'[s_2]$ by the inductive hypothesis, so $C[s_1] \succ^{\mathcal{J}} C[s_2]$ follows from definitions.

Finally, assume $C = t \cdot C'$. Then $t = \mathtt{f}\rho_1 \dots \rho_n t_1 \dots t_m$ where $\mathtt{f} : \forall(\alpha_1 : \kappa_1) \dots \forall(\alpha_n : \kappa_n).\sigma_1 \to \dots \to \sigma_k \to \tau$ with τ a type atom, $m < k$, and $\mathcal{J}(\mathtt{f}) = \Lambda\alpha_1 \dots \alpha_n.\lambda x_1 \dots x_k.u$ with u safe for each x_i. Without loss of generality assume $m = k - 1$. Then $[\![C[s_i]]\!] \rightsquigarrow u'[x_k := [\![C'[s_i]]\!]]$ where $u' = u[\alpha_1 := [\![\rho_1]\!]] \dots [\alpha_n := [\![\rho_n]\!]][x_1 := [\![t_1]\!]] \dots [x_{k-1} := [\![t_{k-1}]\!]]$. By the inductive hypothesis $[\![C'[s_1]]\!] \succ [\![C'[s_2]]\!]$. Hence $u'[x_k := [\![C'[s_1]]\!]] \succ u'[x_k := [\![C'[s_2]]\!]]$, because u is safe for x_k. Thus $[\![C[s_1]]\!] \succ [\![C[s_2]]\!]$ by Lemma 4.18. ◀

▶ **Theorem 5.13.** *If $\mathcal{J}$ is safe then $(\succeq^{\mathcal{J}}, \succ^{\mathcal{J}})$ is a reduction pair.*

Proof. By Lemmas 4.13 and 4.14, $\succeq^{\mathcal{J}}$ is a quasi-order. Lemmas 4.12 and 4.13 imply that $\succ^{\mathcal{J}}$ is a well-founded ordering. Compatibility follows from Lemma 4.15. Monotonicity of $\succeq^{\mathcal{J}}$ follows from Lemma 4.26. Monotonicity of $\succ^{\mathcal{J}}$ follows from Lemma 5.12. ◀

▶ **Example 5.14.** The following is a safe interpretation for the PFS from Example 3.7:

$$
\begin{array}{rcll}
\mathcal{TM}(\mathtt{List}) & = & \forall\beta.(\forall\alpha.\beta \to \alpha \to \beta) \to \beta \to \beta & \\
\mathcal{J}(@) & = & \Lambda\alpha.\Lambda\beta.\lambda f.\lambda x. & f \cdot x \oplus \mathtt{lift}_\beta(\mathtt{flatten}_\alpha(x)) \\
\mathcal{J}(\mathtt{A}) & = & \Lambda\alpha.\Lambda\beta.\lambda x. & x * \beta \\
\mathcal{J}(\mathtt{nil}) & = & & \Lambda\beta.\lambda f.\lambda x.x \\
\mathcal{J}(\mathtt{cons}) & = & \Lambda\alpha.\lambda h.\lambda t. & \Lambda\beta.\lambda f.\lambda x.t\beta f(f\alpha xh \oplus \mathtt{lift}_\beta(\mathtt{flatten}_\beta(x) \oplus \\
 & & & \qquad \mathtt{flatten}_\alpha(h))) \oplus \\
 & & & \quad \mathtt{lift}_\beta(\mathtt{flatten}_\beta(f\alpha xh) \oplus \mathtt{flatten}_\alpha(h) \oplus 1) \\
\mathcal{J}(\mathtt{foldl}) & = & \Lambda\beta.\lambda f.\lambda x.\lambda l. & l\beta fx \oplus \mathtt{lift}_\beta(\mathtt{flatten}_{\forall\alpha.\beta\to\alpha\to\beta}(f) \oplus \\
 & & & \quad \mathtt{flatten}_\beta(x) \oplus 1)
\end{array}
$$

Note that $\mathcal{J}(\mathtt{cons})$ is *not* required to be safe for x, since x is not an argument of `cons`: following its declaration, `cons` takes one type and two terms as arguments. The variable x is only part of the *interpretation*. Note also that the current interpretation is a mostly straightforward extension of Example 5.9: we retain the same *core* interpretations (which, intuitively, encode @ and `A` as forms of application and encode a list as the function that executes a fold over the list's contents), but we add a clause $\oplus\mathtt{lift}(\mathtt{flatten}(x))$ for each argument x that the initial interpretation is not safe for. The only further change is that, in $\mathcal{J}(\mathtt{cons})$, the part between brackets has to be extended. This was necessitated by the change to $\mathcal{J}(\mathtt{foldl})$, in order for the rules to still be oriented (as we will do in Example 6.6).

6 Proving termination with rule removal

A PFS A is certainly terminating if its reduction relation $\longrightarrow_{\mathcal{R}}$ is contained in a well-founded relation, which holds if $\ell \succ^{\mathcal{J}} r$ for all its rules (ℓ, r). However, sometimes it is cumbersome to find an interpretation that orients all rules strictly. To illustrate, the interpretation of Example 5.14 gives $\ell \succ^{\mathcal{J}} r$ for two of the rules and $\ell \succeq^{\mathcal{J}} r$ for the others (as we will see in Example 6.6). In such cases, proof progress is still achieved through *rule removal*.

▶ **Theorem 6.1.** *Let $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$, and suppose that $\mathcal{R}_1 \subseteq \succ^{\mathcal{R}}$ and $\mathcal{R}_2 \subseteq \succeq^{\mathcal{R}}$ for a reduction pair $(\succeq^{\mathcal{R}}, \succ^{\mathcal{R}})$. Then $\longrightarrow_{\mathcal{R}}$ is terminating if and only if $\longrightarrow_{\mathcal{R}_2}$ is (so certainly if $\mathcal{R}_2 = \emptyset$).*

Proof. Monotonicity of $\succeq^{\mathcal{R}}$ and $\succ^{\mathcal{R}}$ implies that $\longrightarrow_{\mathcal{R}_1} \subseteq \succ^{\mathcal{R}}$ and $\longrightarrow_{\mathcal{R}_2} \subseteq \succeq^{\mathcal{R}}$.

By well-foundedness of $\succ^{\mathcal{R}}$, compatibility of $\succeq^{\mathcal{R}}$ and $\succ^{\mathcal{R}}$, and transitivity of $\succeq^{\mathcal{R}}$, every infinite $\longrightarrow_{\mathcal{R}}$ sequence can contain only finitely many $\longrightarrow_{\mathcal{R}_1}$ steps. ◀

The above theorem gives rise to the following *rule removal* algorithm:

1. While $\mathcal{R}$ is non-empty:
   a. Construct a reduction pair $(\succeq^{\mathcal{R}}, \succ^{\mathcal{R}})$ such that all rules in $\mathcal{R}$ are oriented by $\succeq^{\mathcal{R}}$ or $\succ^{\mathcal{R}}$, and at least one of them is oriented using $\succ^{\mathcal{R}}$.
   b. Remove all rules ordered by $\succ^{\mathcal{R}}$ from $\mathcal{R}$.

If this algorithm succeeds, we have proven termination.

To use this algorithm with the pair $(\succeq^{\mathcal{J}}, \succ^{\mathcal{J}})$ from Section 5, we should identify an interpretation $(\mathcal{TM}, \mathcal{J})$ such that (a) $\mathcal{J}$ is safe, (b) all rules can be oriented with $\succeq^{\mathcal{J}}$ or $\succ^{\mathcal{J}}$, and (c) at least one rule is oriented with $\succ^{\mathcal{J}}$. The first requirement guarantees that $(\succeq^{\mathcal{J}}, \succ^{\mathcal{J}})$ is a reduction pair (by Theorem 5.13). Lemma 5.11 provides some sufficient safety criteria. The second and third requirements have to be verified for each individual rule.

▶ **Example 6.2.** We continue with our example of fold on heterogeneous lists. We prove termination by rule removal, using the symbol mapping from Example 5.14. We will show:

$$\begin{aligned}
@_{\sigma,\tau}(\lambda x : \sigma.s, t) &\succeq^{\mathcal{J}} s[x := t] \\
\mathtt{A}_{\lambda\alpha.\sigma,\tau}(\Lambda\alpha.s) &\succeq^{\mathcal{J}} s[\alpha := \tau] \\
\mathtt{foldl}_{\sigma}(f, s, \mathtt{nil}) &\succ^{\mathcal{J}} s \\
\mathtt{foldl}_{\sigma}(f, s, \mathtt{cons}_{\tau}(h, t)) &\succ^{\mathcal{J}} \mathtt{foldl}_{\sigma}(f, @_{\tau,\sigma}(@_{\sigma,\tau\to\sigma}(\mathtt{A}_{\lambda\alpha.\sigma\to\alpha\to\sigma,\tau}(f), s), h), t)
\end{aligned}$$

Consider the first inequality; by definition it holds if $[\![@_{\sigma,\tau}(\lambda x : \sigma.s, t)]\!] \succeq [\![s[x := t]]\!]$. Since $[\![@_{\sigma,\tau}(\lambda x : \sigma.s, t)]\!] \rightsquigarrow^* [\![s]\!][x := [\![t]\!]] \oplus \mathtt{lift}_{[\![\tau]\!]}(\mathtt{flatten}_{[\![\sigma]\!]}([\![t]\!]))$, and $[\![s]\!][x := [\![t]\!]] = [\![s[x := t]]\!]$ (by Lemma 5.6), it suffices by Lemma 4.17 if $[\![s[x := t]]\!] \oplus \mathtt{lift}_{[\![\tau]\!]}(\mathtt{flatten}_{[\![\sigma]\!]}([\![t]\!])) \succeq [\![s[x := t]]\!]$. This is an instance of the general rule $u \oplus w \succeq u$ that we will obtain below.

To prove inequalities $s \succ t$ and $s \succeq t$, we will often use that $\succ$ and $\succeq$ are transitive and compatible with each other (Lem. 4.13 and 4.15), that $\rightsquigarrow \subseteq \approx$ (Lem. 4.17), that $\succeq$ is monotonic (Lem. 4.26), that both $\succ$ and $\succeq$ are monotonic over $\mathtt{lift}$ and $\mathtt{flatten}$ (Lem. 4.22 and 4.23) and that interpretations respect substitution (Lem. 5.6). We will also use Lemma 4.25 which states (among other things) that $s \succ t$ implies $s \oplus u \succ t \oplus u$. In addition, we can use the calculation rules below. The proofs may be found in [3, Appendix A.3].

▶ **Lemma 6.3.** *For all types σ and all terms s, t, u of type σ, we have:*

1. $s \oplus_\sigma t \approx t \oplus_\sigma s$ *and* $s \otimes_\sigma t \approx t \otimes_\sigma s$;
2. $s \oplus_\sigma (t \oplus_\sigma u) \approx (s \oplus_\sigma t) \oplus_\sigma u$ *and* $s \otimes_\sigma (t \otimes_\sigma u) \approx (s \otimes_\sigma t) \otimes_\sigma u$;
3. $s \otimes_\sigma (t \oplus_\sigma u) \approx (s \otimes_\sigma t) \oplus_\sigma (s \otimes_\sigma u)$;
4. $(\mathtt{lift}_\sigma 0) \oplus_\sigma s \approx s$ *and* $(\mathtt{lift}_\sigma 1) \otimes_\sigma s \approx s$.

▶ **Lemma 6.4.**
1. $\mathtt{lift}_\sigma(n+m) \approx_\sigma (\mathtt{lift}_\sigma n) \oplus_\sigma (\mathtt{lift}_\sigma m)$;
2. $\mathtt{lift}_\sigma(nm) \approx_\sigma (\mathtt{lift}_\sigma n) \otimes_\sigma (\mathtt{lift}_\sigma m)$;
3. $\mathtt{flatten}_\sigma(\mathtt{lift}_\sigma(n)) \approx n$.

▶ **Lemma 6.5.** *For all types σ, terms s,t of type σ and natural numbers $n > 0$:*
1. $s \oplus_\sigma t \succeq s$ *and* $s \oplus_\sigma t \succeq t$;
2. $s \oplus_\sigma (\mathtt{lift}_\sigma n) \succ s$ *and* $(\mathtt{lift}_\sigma n) \oplus_\sigma t \succ t$.

Note that these calculation rules immediately give the inequality $x \oplus \mathtt{lift}_{nat\to\mathtt{nat}}(1) \succ x$ from Example 4.20, and also that $\mathtt{lift}_\sigma(n) \succ \mathtt{lift}_\sigma(m)$ whenever $n > m$. By Lemmas 4.25 and 6.5 we can use *absolute positiveness*: the property that (a) $s \succeq t$ if we can write $s \approx s_1 \oplus \cdots \oplus s_n$ and $t \approx t_1 \oplus \cdots \oplus t_k$ with $k \le n$ and $s_i \succeq t_i$ for all $i \le k$, and (b) if moreover $s_1 \succ t_1$ then $s \succ t$. This property is typically very useful to dispense the obligations obtained in a termination proof with polynomial interpretations.

▶ **Example 6.6.** We now have the tools to finish the example of heterogeneous lists (still using the interpretation from Example 5.14). The proof obligation from Example 6.2, that $[\![@_{\sigma,\tau}(\lambda x:\sigma.s,t)]\!] \succeq [\![s[x:=t]]\!]$, is completed by Lemma 6.5(1). We have $[\![\mathtt{A}_{\lambda\alpha.\sigma,\tau}(\Lambda\alpha.s)]\!] \approx [\![\Lambda\alpha.s]\!] * [\![\tau]\!] \approx [\![s[\alpha:=\tau]]\!]$ by Lemma 5.6, and $[\![\mathtt{foldl}_\sigma(f,s,\mathtt{nil})]\!] = [\![\mathtt{nil}]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot [\![s]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\langle\text{something}\rangle \oplus 1) \approx [\![s]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\langle\text{something}\rangle \oplus 1) \succ [\![s]\!]$ by Lemmas 6.4(1) and 6.5(1). For the last rule note that (using only Lemmas 4.17 and 6.4(1)):

$$
\begin{array}{l}
[\![\mathtt{foldl}_\sigma(f,s,\mathtt{cons}_\tau(h,t))]\!] \approx \\
[\![\mathtt{cons}_\tau(h,t))]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot [\![s]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!]) \oplus \mathtt{flatten}([\![s]\!]) \oplus 1) \approx \\
(\ [\![t]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot ([\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]))) \oplus \\
\quad \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!]) \oplus \mathtt{flatten}([\![h]\!]) \oplus 1)\ ) \oplus \\
\ \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!]) \oplus \mathtt{flatten}([\![s]\!]) \oplus 1) \approx \\
[\![t]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot (\ [\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]))\ ) \oplus \\
\ \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!]) \oplus \mathtt{flatten}([\![h]\!]) \oplus \mathtt{flatten}([\![f]\!]) \oplus \mathtt{flatten}([\![s]\!]) \oplus 2)
\end{array}
$$

On the right-hand side of the inequality, noting that $\mathtt{lift}_{\sigma\to\tau}(u) \cdot w \rightsquigarrow^* \mathtt{lift}_\tau(u)$, we have:

$$
\begin{array}{l}
[\![\mathtt{foldl}_\sigma(f,@_{\tau,\sigma}(@_{\sigma,\tau\to\sigma}(\mathtt{A}_{\lambda\alpha.\sigma\to\alpha\to\sigma,\tau}(f),s),h),t)]\!] \approx \\
\mathcal{J}(\mathtt{foldl})_\sigma([\![f]\!],\ [\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!])),\ [\![t]\!]) \approx \\
[\![t]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot (\ [\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]))\ ) \oplus \\
\ \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!]) \oplus \mathtt{flatten}([\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \\
\qquad\qquad \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]))) \oplus 1) \approx \\
[\![t]\!] * [\![\sigma]\!] \cdot [\![f]\!] \cdot (\ [\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!] \oplus \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]))\ ) \oplus \\
\ \mathtt{lift}_{[\![\sigma]\!]}(\mathtt{flatten}([\![f]\!]) \oplus \mathtt{flatten}([\![f]\!] * [\![\tau]\!] \cdot [\![s]\!] \cdot [\![h]\!]) \oplus \mathtt{flatten}([\![s]\!]) \oplus \mathtt{flatten}([\![h]\!]) \oplus 1)
\end{array}
$$

Now the right-hand side is the left-hand side $\oplus\ \mathtt{lift}(1)$. Clearly, the rule is oriented with $\succ$. Thus, we may remove the last two rules, and continue the rule removal algorithm with only the first two, which together define β-reduction. This is trivial, for instance with an interpretation $\mathcal{J}(@) = \Lambda\alpha.\Lambda\beta.\lambda f.\lambda x.(f \cdot x) \oplus \mathtt{lift}_\beta(\mathtt{flatten}_\alpha(x) \oplus 1)$ and $\mathcal{J}(\mathtt{A}) = \Lambda\alpha.\Lambda\beta.\lambda x.x * \beta \oplus \mathtt{lift}_{\alpha\beta}(1)$.

7 A larger example

System F is System $\mathtt{F}_\omega$ where no higher kinds are allowed, i.e., there are no type constructors except types. By the Curry-Howard isomorphism F corresponds to the universal-implicational fragment of intuitionistic second-order propositional logic, with the types corresponding to formulas and terms to natural deduction proofs. The remaining connectives may be encoded in F, but the permutative conversion rules do not hold [7].

In this section we show termination of the system IPC2 (see [18]) of intuitionistic second-order propositional logic with all connectives and permutative conversions, minus a few of the permutative conversion rules for the existential quantifier. The paper [18] depends on termination of IPC2, citing a proof from [27], which, however, later turned out to be incorrect. Termination of Curry-style IPC2 without $\bot$ as primitive was shown in [20]. To our knowledge, termination of the full system IPC2 remains an open problem, strictly speaking.

▶ Remark 7.1. Our method builds on the work of van de Pol and Schwichtenberg, who used higher-order polynomial interpretations to prove termination of a fragment of intuitionistic first-order logic with permutative conversions [24], in the hope of providing a more perspicuous proof of this well-known result. Notably, they did not treat disjunction, as we will do. More fundamentally, their method cannot handle impredicative polymorphism necessary for second-order logic.

The system IPC2 can be seen as a PFS with type constructors:

$$\Sigma_\kappa^T = \{ \quad \bot : *, \quad \mathtt{or} : * \Rightarrow * \Rightarrow *, \quad \mathtt{and} : * \Rightarrow * \Rightarrow *, \quad \exists : (* \Rightarrow *) \Rightarrow * \}$$

We have the following function symbols:

$$\begin{array}{rcl@{\qquad}rcl}
@ & : & \forall\alpha\forall\beta.(\alpha \to \beta) \to \alpha \to \beta & \epsilon & : & \forall\alpha.\bot \to \alpha \\
\mathtt{tapp} & : & \forall\alpha : * \Rightarrow *.\forall\beta.(\forall\beta[\alpha\beta]) \to \alpha\beta & \mathtt{pr}^1 & : & \forall\alpha\forall\beta.\mathtt{and}\,\alpha\,\beta \to \alpha \\
\mathtt{pair} & : & \forall\alpha\forall\beta.\alpha \to \beta \to \mathtt{and}\,\alpha\,\beta & \mathtt{pr}^2 & : & \forall\alpha\forall\beta.\mathtt{and}\,\alpha\,\beta \to \beta \\
\mathtt{case} & : & \forall\alpha\forall\beta\forall\gamma.\mathtt{or}\,\alpha\,\beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma & \mathtt{in}^1 & : & \forall\alpha\forall\beta.\alpha \to \mathtt{or}\,\alpha\,\beta \\
\mathtt{let} & : & \forall\alpha : * \Rightarrow *.\forall\beta.(\exists(\alpha)) \to (\forall\gamma.\alpha\gamma \to \beta) \to \beta & \mathtt{in}^2 & : & \forall\alpha\forall\beta.\beta \to \mathtt{or}\,\alpha\,\beta \\
\mathtt{ext} & : & \forall\alpha : * \Rightarrow *.\forall\beta.\alpha\beta \to \exists(\alpha) & & &
\end{array}$$

The types represent formulas in intuitionistic second-order propositional logic, and the terms represent proofs. For example, a term $\mathtt{case}_{\sigma,\tau,\rho}\ s\ u\ v$ is a proof term of the formula ρ, built from a proof s of $\mathtt{or}\ \sigma\ \tau$, a proof u that σ implies ρ and a proof v that τ implies ρ. Proof terms can be simplified using 28 reduction rules, including the following (the full set of rules is available in [3, Appendix B]):

$$\begin{array}{rcl@{\qquad}rcl}
@_{\sigma,\tau}(\lambda x.s,t) & \longrightarrow & s[x := t] & & & \\
\mathtt{tapp}_{\lambda\alpha.\sigma,\tau}(\Lambda\alpha.s) & \longrightarrow & s[\alpha := \tau] & \mathtt{let}_{\varphi,\rho}(\mathtt{ext}_{\varphi,\tau}(s), \Lambda\alpha.\lambda x.t) & \longrightarrow & t[\alpha := \tau][x := s] \\
\mathtt{pr}^1_{\sigma,\tau}(\mathtt{pair}_{\sigma,\tau}(s,t)) & \longrightarrow & s & \mathtt{case}_{\sigma,\tau,\rho}(\mathtt{in}^1_{\sigma,\tau}(u), \lambda x.s, \lambda y.t) & \longrightarrow & s[x := u] \\
\mathtt{pr}^2_{\sigma,\tau}(\mathtt{pair}_{\sigma,\tau}(s,t)) & \longrightarrow & t & \mathtt{case}_{\sigma,\tau,\rho}(\mathtt{in}^2_{\sigma,\tau}(u), \lambda x.s, \lambda y.t) & \longrightarrow & t[x := u]
\end{array}$$

$$\begin{array}{ll}
 & @_{\sigma,\tau}(\epsilon_{\sigma\to\tau}(s), t) \longrightarrow \epsilon_\tau(s) \\
 & \mathtt{case}_{\sigma,\tau,\rho}(\epsilon_{\mathtt{or}\,\sigma\,\tau}(u), \lambda x.s, \lambda y.t) \longrightarrow \epsilon_\rho(u) \\
 & \epsilon_\rho(\mathtt{case}_{\sigma,\tau,\bot}(u, \lambda x.s, \lambda y.t)) \longrightarrow \mathtt{case}_{\sigma,\tau,\rho}(u, \lambda x.\epsilon_\rho(s), \lambda y.\epsilon_\rho(t)) \\
 & \mathtt{pr}^2_{\rho,\pi}(\mathtt{case}_{\sigma,\tau,\mathtt{and}\,\rho,\pi}(u, \lambda x.s, \lambda y.t)) \longrightarrow \mathtt{case}_{\sigma,\tau,\pi}(u, \lambda x.\mathtt{pr}^2_{\rho,\pi}(s), \lambda y.\mathtt{pr}^2_{\rho,\pi}(t)) \\
 & \mathtt{case}_{\rho,\pi,\xi}(\mathtt{case}_{\sigma,\tau,\mathtt{or}\,\rho\,\pi}(u, \lambda x.s, \lambda y.t), \lambda z.v, \lambda a.w) \longrightarrow \\
 & \quad \mathtt{case}_{\sigma,\tau,\xi}(u, \lambda x.\mathtt{case}_{\rho,\pi,\xi}(s, \lambda z.v, \lambda a.w), \lambda y.\mathtt{case}_{\rho,\pi,\xi}(t, \lambda z.v, \lambda a.w)) \\
 & \mathtt{let}_{\varphi,\rho}(\mathtt{case}_{\sigma,\tau,\exists\varphi}(u, \lambda x.s, \lambda y.t), v) \longrightarrow \mathtt{case}_{\sigma,\tau,\rho}(u, \lambda x.\mathtt{let}_{\varphi,\rho}(s, v), \lambda y.\mathtt{let}_{\varphi,\rho}(t, v)) \\
(*) & \mathtt{let}_{\psi,\rho}(\mathtt{let}_{\varphi,\exists\psi}(s, \Lambda\alpha.\lambda x : \varphi\alpha.t), u) \longrightarrow \mathtt{let}_{\varphi,\rho}(s, \Lambda\alpha.\lambda x : \varphi\alpha.\mathtt{let}_{\psi,\rho}(t, u))
\end{array}$$

To define an interpretation for IPC2, we will use the standard encoding of product and existential types (see [7, Chapter 11] for more details).

$$\begin{array}{rcl@{\qquad}rcl}
\sigma \times \tau & = & \forall p.(\sigma \to \tau \to p) \to p & \pi^1_{\sigma,\tau}(t) & = & t\sigma(\lambda x : \sigma.\lambda y : \tau.x) \\
\langle t_1, t_2\rangle_{\sigma,\tau} & = & \Lambda p.\lambda x : \sigma \to \tau \to p.x t_1 t_2 & \pi^2_{\sigma,\tau}(t) & = & t\tau(\lambda x : \sigma.\lambda y : \tau.y) \\
\Sigma\alpha.\sigma & = & \forall p.(\forall\alpha.\sigma \to p) \to p & [\tau, t]_{\Sigma\alpha.\sigma} & = & \Lambda p.\lambda x : \forall\alpha.\sigma \to p.x\tau t \\
 & & & \mathtt{let}_\rho\, t\, \mathtt{be}\, [\alpha, x : \sigma]\, \mathtt{in}\, s & = & t\rho(\Lambda\alpha.\lambda x : \sigma.s)
\end{array}$$

We do not currently have an algorithmic method to find a suitable interpretation. Instead, we used the following manual process. We start by noting the minimal requirements given by the first set of rules (e.g., that $\mathtt{pr}^1_{\sigma,\tau}(\mathtt{pair}_{\sigma,\tau}(s,t)) \succeq s$); to orient these inequalities, it would be good to for instance have $[\![\mathtt{pair}_{\sigma,\tau}(s,t)]\!] \succeq \langle [\![s]\!], [\![t]\!] \rangle_{[\![\sigma]\!],[\![\tau]\!]}$ and $[\![\mathtt{pr}^i_{\sigma,\tau}(s)]\!] = \pi^i_{[\![\sigma]\!],[\![\tau]\!]}([\![s]\!])$. To make the interpretation safe, we additionally include clauses $\mathtt{lift}(\mathtt{flatten}(x))$ for any unsafe arguments x; to make the rules *strictly* oriented, we include clauses $\mathtt{lift}(1)$. Unfortunately, this approach does not suffice to orient the rules where some terms are duplicated, such as the second- and third-last rules. To handle these rules, we *multiply* the first argument of several symbols with the second (and possibly third). Some further tweaking gives the following safe interpretation, which orients most of the rules:

$$\begin{array}{rcll}
\mathcal{TM}(\bot) & = & \mathtt{nat} & \mathcal{TM}(\mathtt{and}) = \lambda\alpha_1\lambda\alpha_2.\alpha_1 \times \alpha_2 \\
\mathcal{TM}(\exists) & = & \lambda(\alpha : * \Rightarrow *).\Sigma\gamma.\alpha\gamma & \mathcal{TM}(\mathtt{or}) = \lambda\alpha_1\lambda\alpha_2.\alpha_1 \times \alpha_2 \\
\\
\mathcal{J}(\epsilon) & = & \Lambda\alpha : *.\lambda x : \mathtt{nat}. & \mathtt{lift}_\alpha(2 \otimes x \oplus 1) \\
\mathcal{J}(@) & = & \Lambda\alpha\Lambda\beta\lambda x : \alpha \to \beta.\lambda y : \alpha. & \mathtt{lift}_\beta(2) \otimes (x \cdot y) \oplus \mathtt{lift}_\beta(\mathtt{flatten}_\alpha(y) \oplus \\
 & & & \quad \mathtt{flatten}_{\alpha\to\beta}(x) \otimes \mathtt{flatten}_\beta(y) \oplus 1) \\
\mathcal{J}(\mathtt{tapp}) & = & \Lambda\alpha : * \Rightarrow *.\Lambda\beta.\lambda x : \forall\gamma.\alpha\gamma. & \mathtt{lift}_{\alpha\beta}(2) \otimes (x * \beta) \oplus \mathtt{lift}_{\alpha\beta}(1) \\
\mathcal{J}(\mathtt{ext}) & = & \Lambda\alpha : * \Rightarrow *.\Lambda\beta : *.\lambda x : \alpha\beta. & [\beta, x] \oplus \mathtt{lift}_{\Sigma\gamma.\beta\gamma}(\mathtt{flatten}_{\alpha\gamma}(x)) \\
\mathcal{J}(\mathtt{pair}) & = & \Lambda\alpha\Lambda\beta\lambda x : \alpha, y : \beta. & \langle x, y\rangle \oplus \mathtt{lift}_{\alpha\times\beta}(\mathtt{flatten}_\alpha(x) \oplus \mathtt{flatten}_\beta(y)) \\
\mathcal{J}(\mathtt{pr}^1) & = & \Lambda\alpha\Lambda\beta\lambda x : \alpha \times \beta. & \mathtt{lift}_\alpha(2) \otimes \pi^1(x) \oplus \mathtt{lift}_\alpha(1) \\
\mathcal{J}(\mathtt{pr}^2) & = & \Lambda\alpha\Lambda\beta\lambda x : \alpha \times \beta. & \mathtt{lift}_\beta(2) \otimes \pi^2(x) \oplus \mathtt{lift}_\beta(1) \\
\mathcal{J}(\mathtt{in}^1) & = & \Lambda\alpha\Lambda\beta\lambda x : \alpha. & \langle x, \mathtt{lift}_\beta(1)\rangle \oplus \mathtt{lift}_{\alpha\times\beta}(\mathtt{flatten}_\alpha(x)) \\
\mathcal{J}(\mathtt{in}^2) & = & \Lambda\alpha\Lambda\beta\lambda x : \beta. & \langle \mathtt{lift}_\alpha(1), x\rangle \oplus \mathtt{lift}_{\alpha\times\beta}(\mathtt{flatten}_\beta(x)) \\
\\
\mathcal{J}(\mathtt{let}) & = & \Lambda\alpha : * \Rightarrow *.\Lambda\beta : *.\lambda x : \Sigma\xi.\alpha\xi, y : \forall\xi.\alpha\xi \to \beta. & \\
 & & \mathtt{lift}_\beta(1) \oplus \mathtt{lift}_\beta(2) \otimes (\mathtt{let}_\beta\, x\, \mathtt{be}\, [\xi, z]\, \mathtt{in}\, y\xi z) \oplus & \\
 & & \mathtt{lift}_\beta(\mathtt{flatten}_{\Sigma\gamma.\alpha\gamma}(x) \oplus 1) \otimes (y * \mathtt{nat} \cdot \mathtt{lift}_{\alpha\mathtt{nat}}(0)) & \\
\mathcal{J}(\mathtt{case}) & = & \Lambda\alpha, \beta, \xi.\lambda x : \alpha \times \beta, y : (\alpha \to \xi), z : (\beta \to \xi). & \\
 & & \quad \mathtt{lift}_\xi(2) \oplus \mathtt{lift}_\xi(3 \otimes \mathtt{flatten}_{\alpha\times\beta}(x)) \oplus & \\
 & & \qquad \mathtt{lift}_\xi(\mathtt{flatten}_{\alpha\times\beta}(x) \oplus 1) \otimes (y \cdot \pi^1(x) \oplus z \cdot \pi^2(x)) &
\end{array}$$

Above, $\otimes$ binds stronger than $\oplus$. The derivations to orient rules with these interpretations are also given in [3, Appendix B].

The only rules that are not oriented with this interpretation – not with $\succeq$ either – are the ones of the form $f(\mathtt{let}(s,t),\dots) \longrightarrow \mathtt{let}(s, f(t,\dots))$, like the rule marked (*) above. Nonetheless, this is already a significant step towards a systematic, extensible methodology of termination proofs for IPC2 and similar systems of higher-order logic. Verifying the orientations is still tedious, but our method raises hope for at least partial automation, as was done with polynomial interpretations for non-polymorphic higher-order rewriting [6].

8 Conclusions and future work

We introduced a powerful and systematic methodology to prove termination of higher-order rewriting with full impredicative polymorphism. To use the method one just needs to invent safe interpretations and verify the orientation of the rules with the calculation rules.

As the method is tedious to apply manually for larger systems, a natural direction for future work is to look into automation: both for automatic verification that a given interpretation suffices and – building on existing termination provers for first- and higher-order term rewriting – for automatically finding a suitable interpretation.

In addition, it would be worth exploring improvements of the method that would allow us to handle the remaining rules of IPC2, or extending other techniques for higher-order termination such as orderings (see, e.g., [11]) or dependency pairs (e.g., [13, 19]).

References

1 F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *MSCS*, 15(1):37–92, 2005.
2 D. Cousineau and G. Dowek. Embedding Pure Type Systemsin the lambda-Pi-calculus modulo. In *TLCA*, pages 102–117, 2017.
3 Ł. Czajka and C. Kop. Polymorphic Higher-Order Termination (extended version), 2019. `arXiv:1904.09859`.
4 G. Dowek. Models and termination of proof reduction in the λΠ-calculus modulo theory. In *ICALP*, pages 109:1–109:14, 2017.
5 M. Fiore and M. Hamana. Multiversal Polymorphic Algebraic Theories: syntacs, semantics, translations and equational logic. In *LICS*, pages 520–520, 2013.
6 C. Fuhs and C. Kop. Polynomial Interpretations for Higher-Order Rewriting. In *RTA*, pages 176–192, 2012.
7 J.-V. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.
8 J.-Y. Girard. Une Extension De l'Interpretation De Gödel a l'Analyse, Et Son Application a l'Elimination Des Coupures Dans l'Analyse Et La Theorie Des Types. In *SLS*, pages 63–92. Elsevier, 1971.
9 M. Hamana. Polymorphic Rewrite Rules: Confluence, Type Inference, and Instance Validation. In *FLOPS*, pages 99–115, 2018.
10 B. Jacobs and J. Rutten. An introduction to (co)algebras and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. Cambridge University Press, 2011.
11 J. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *JACM*, 54(1):1–48, 2007.
12 C. Kop. *Higher Order Termination*. PhD thesis, VU University Amsterdam, 2012.
13 C. Kop and F. van Raamsdonk. Dynamic Dependency Pairs for Algebraic Functional Systems. *LMCS*, 8(2):10:1–10:51, 2012.
14 D. Kozen and A. Silva. Practical coinduction. *Mathematical Structures in Computer Science*, 27(7):1132–1152, 2017.
15 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.
16 D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.
17 M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.
18 M.H. Sørensen and P. Urzyczyn. A Syntactic Embedding of Predicate Logic into Second-Order Propositional Logic. *Notre Dame Journal of Formal Logic*, 51(4):457–473, 2010.
19 S. Suzuki, K. Kusakari, and F. Blanqui. Argument Filterings and Usable Rules in Higher-Order Rewrite Systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011.
20 M. Tatsuta. Simple Saturated Sets for Disjunction and Second-Order Existential Quantification. In *TLCA 2007*, pages 366–380, 2007.
21 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.
22 A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.
23 J.C. van de Pol. Termination Proofs for Higher-order Rewrite Systems. In *HOA*, pages 305–325, 1993.
24 J.C. van de Pol and H. Schwichtenberg. Strict Functionals for Termination Proofs. In *TLCA 95*, pages 350–364, 1995.
25 D. Wahlstedt. *Type Theory with First-Order Data Types and Size-Change Termination*. PhD thesis, Göteborg University, 2004.
26 D. Walukiewicz-Chrząszcz. Termination of rewriting in the Calculus of Constructions. *JFP*, 13(2):339–414, 2003.
27 A. Wojdyga. Short Proofs of Strong Normalization. In *MFCS*, pages 613–623, 2008.