

# Representing Nonterminating Reductions in $F_2^\mu$

Peng Fu<sup>1</sup>

1 Dalhousie University  
peng-fu@uiowa.edu

---

## Abstract

We specify a second-order type system  $F_2^\mu$  that is tailored for representing nonterminations. The nonterminating trace of a term  $t$  in a rewrite system  $\mathcal{R}$  corresponds to a productive inhabitant  $e$  such that  $\Gamma_{\mathcal{R}} \vdash e : t$  in  $F_2^\mu$ , where  $\Gamma_{\mathcal{R}}$  is the environment representing the rewrite system. We prove that the productivity checking in  $F_2^\mu$  is decidable via a mapping to the  $\lambda$ -Y calculus. We develop a type checking algorithm for  $F_2^\mu$  based on second-order matching. We implement the type checking algorithm in a proof-of-concept type checker.

**Keywords and phrases** Nonterminating Rewriting, Typed Lambda Calculus, Hereditary Head Normalization, Corecursion, Second-order Type Checking

## 1 Introduction

Nontermination has been an active research area in the term rewriting community. Early studies includes classifying nonterminations based on the concept of looping reduction [6], i.e. a reduction of the shape  $t \rightarrow^+ C[\sigma t]$  for some substitution  $\sigma$ . More recently, many nontermination detection techniques are proposed and implemented. Emmes et. al. [8] considered a generalized notion of looping reduction, e.g.  $\sigma_2 \sigma_1^n t \rightarrow^+ C[\sigma_3 \sigma_2 \sigma_1^{f(n)} t]$  for some substitutions  $\sigma_1, \sigma_2, \sigma_3$  and some ascending linear function  $f$ . Endrullis and Zantema [9] used a SAT solver to search for a non-empty regular language of terms such that it is closed under reduction and does not contain normal forms.

The nonterminating reductions are usually described using mathematical notations and abbreviations. In this paper, we consider a novel representation using a relatively simple type system. In particular, a nonterminating reduction of a term will be encoded as a proof evidence in a type system called  $F_2^\mu$ . Representing nonterminating reduction is closely related to proving nontermination, but they have some subtle differences. For proving nontermination, it is enough to exhibit a nonterminating reduction for a term, while a term can admit multiple nonterminating reduction traces, with each trace exhibits a different kind of reduction pattern.

► **Example 1.** Consider the following two string rewriting rules:  $A \rightarrow_a AB, B \rightarrow_b A$ . It is nonterminating by the observation that it contains the rule  $A \rightarrow_a AB$ , which means there is a nonterminating reduction of the form  $A \rightarrow_a AB \rightarrow_a ABB \rightarrow_a ABAB \rightarrow_a \dots$ . We can also use a L-system<sup>1</sup> like parallel reduction strategy to reduce  $A$ , this gives rise to the nonterminating reduction:  $A \Rightarrow AB \Rightarrow ABA \Rightarrow ABAAB \Rightarrow ABAABABA \Rightarrow ABAABABAABAAB \Rightarrow \dots$ . Note that all the redexes at each step are reduced simultaneously and each word in the sequence is a concatenation of the previous two. The aforementioned two reduction sequences are fundamentally different. The first one exhibits a regular property, i.e. each string at each step can be described by the regular expression  $AB^*$ . In the second reduction sequence, each string is called a Fibonacci word, and the set of

---

<sup>1</sup> See <https://en.wikipedia.org/wiki/L-system>.



all such words is known to be *context-free free*, i.e. any infinite subset can not be described by a context-free language [25]. We will show how to represent the second reduction sequence in Section 6.

The main contributions of the paper are the following ones.

- Inspired by Leibniz equality, we represent a rewrite rule  $l \rightarrow r$  as a typing environment  $\kappa : \forall p. \forall x. p \ r \Rightarrow p \ l$ , where the type variable  $p$  of kind  $* \Rightarrow *$  represents a reduction context,  $\kappa$  is a fresh constant evidence and  $x$  denotes the set of variables in  $l$ . A specialized kind system is used to ensure the type variable of kind  $* \Rightarrow *$  represents a reduction context. We call this representation of rewrite rule *Leibniz representation* in Section 3.
- Nonterminating reductions would result in infinite proof evidence, we use the fixed point typing rule to represent the reductions finitely. Thus a nonterminating reduction of  $t$  in  $\mathcal{R}$  can be represented as  $\Gamma_{\mathcal{R}} \vdash e : t$ , where  $e$  is an evidence containing a fixed point and  $\Gamma_{\mathcal{R}}$  is the Leibniz representation of  $\mathcal{R}$ . We called the resulting type system  $\mathbf{F}_2^\mu$  (Section 3).
- We prove that if  $\Gamma_{\mathcal{R}} \vdash e : t$  and  $e$  is *hereditary head normalizing* (HHN), then we can recover from the evidence  $e$  a nonterminating reduction of  $t$  (Section 4). We also prove that the hereditary head normalization is decidable in  $\mathbf{F}_2^\mu$ . The decidability result is obtained via a mapping from  $\mathbf{F}_2^\mu$  to  $\lambda$ -Y calculus, for which HHN is decidable.
- It is more convenient to write the unannotated proof evidence and let the type checker fill in the annotations. For this purpose we develop a second-order type checking algorithm in Section 5 and Section 6. It simplifies the process of representing nonterminations in  $\mathbf{F}_2^\mu$ . We implement a prototype type checker<sup>2</sup> based on this algorithm and give some nontrivial examples in the Appendix.

All the examples and the missing proofs in this paper may be found in the Appendix.

## 2 The Main Idea

First, let us consider how to represent a rewrite system in a type system. We could model the rewrite rule  $l \rightarrow r$  as a typing environment  $\kappa : l \Rightarrow r$ , like many proof systems for rewriting ([22], [20]). However, modeling the rewrite rule  $l \rightarrow r$  as an implication type  $l \Rightarrow r$  will make it difficult to observe the proof evidence. For example, suppose we have a set of ground rewrite rules  $A_i \rightarrow A_{i+1}$  modelled by  $\kappa_i : A_i \Rightarrow A_{i+1}$  for  $0 \leq i \leq n$  for some  $n$ , where  $\kappa_i$  is a constant. Then the evidence for the reduction  $A_0 \rightarrow^* A_n$  would be  $\lambda\alpha. (\kappa_n \dots (\kappa_0 \ \alpha) \dots) : A_0 \Rightarrow A_n$ . Informally, we can see that the evidence  $\lambda\alpha. (\kappa_n \dots (\kappa_0 \ \alpha) \dots)$  grows outward as the number  $n$  gets larger. When the reduction is nonterminating, it would be difficult to observe the very first step of the reduction ( $\kappa_0$ ). Fortunately, this difficulty can be overcome by representing  $l \rightarrow r$  as  $r \Rightarrow l$ . Thus we have the evidence  $\lambda\alpha. (\kappa_0 \dots (\kappa_n \ \alpha) \dots) : A_n \Rightarrow A_0$ , with  $\kappa_i : A_{i+1} \Rightarrow A_i$  for all  $0 \leq i \leq n$ . So we can easily observe the first step of the reduction  $\kappa_0$  at the outermost position.

Next, we need to model the reduction context in rewriting. Given a rewrite rule  $l \rightarrow r$ , we have a one-step reduction  $C[l] \rightarrow C[r]$  for any first-order term context  $C$ . Inspired by *Leibniz equality*, we use the type  $\forall p. p \ r \Rightarrow p \ l$  to model the rewrite rule  $l \rightarrow r$ . The intended reading for this type is that  $l$  can be replaced by  $r$  under *any* first-order term context  $p$ . Note that  $p$  is a second-order type variable of kind  $* \Rightarrow *$ . So we can obtain  $C[r] \Rightarrow C[l]$  by instantiating  $p$  with  $\lambda x. C[x]$  in  $\forall p. p \ r \Rightarrow p \ l$ . This motivates our definition of *Leibniz*

<sup>2</sup> The prototype type checker is available at <https://github.com/Fermat/FCR>

representation for the rewrite rules in Section 3 and the use of the type system  $\mathbf{F}_2^\mu$ , as its kind system enforces that one can only instantiate type variable of kind  $* \Rightarrow *$  with a type that represents a term context.

Last but not least, we need a mechanism to handle the nonterminating reductions. Consider the cycling rewrite rules:  $A \rightarrow B$  and  $B \rightarrow A$ , which are represented as two axioms  $\Gamma = \kappa_A : B \Rightarrow A, \kappa_B : A \Rightarrow B$ . There is a cyclic reduction for  $A$ :  $A \rightarrow B \rightarrow A \rightarrow B \rightarrow \dots$ . Using the Leibniz representation, the corresponding proof evidence for this reduction would be an infinite proof evidence  $\kappa_A (\kappa_B (\kappa_A (\kappa_B \dots)))$ . But we want to use a finite evidence  $e$  to represent this nonterminating reduction. The solution here is to use a fixpoint operator. We can represent the infinite proof evidence finitely as  $\mu\alpha.\kappa_A (\kappa_B \alpha)$ , where the  $\mu$  is a fixpoint binder with the operational meaning of  $\mu\alpha.e \rightsquigarrow [\mu\alpha.e/\alpha]e$ . This motivates the following fixed point typing rule for  $\mathbf{F}_2^\mu$ .

$$\frac{\Gamma, \alpha : T \vdash e : T}{\Gamma \vdash \mu\alpha.e : T} \text{ Mu}$$

So  $\Gamma \vdash \mu\alpha.\kappa_A (\kappa_B \alpha) : A$  represents a nonterminating reduction of the shape  $A \rightarrow B \rightarrow A \rightarrow B \rightarrow \dots$ , since the unfolding of the evidence  $\mu\alpha.\kappa_A (\kappa_B \alpha)$  gives the sequence of rules that we are going to apply. Note that not all evidence of type  $A$  are representing nonterminating reductions. For example, according to the typing rule *Mu*, we have  $\Gamma \vdash \mu\alpha.\alpha : A$ , but  $\mu\alpha.\alpha$  does not give any information to reconstruct a nonterminating reduction. We show in Section 4 that only the *hereditary head normalizing* evidence are representing the nonterminating reductions.

We conclude this section by recasting our idea in the following example.

► **Example 2.** Consider the following rewrite rule.

$$F x \rightarrow G (F (G x))$$

The term  $F x$  admits a reduction sequence  $F x \rightarrow G (F (G x)) \rightarrow G^2 (F (G^2 x)) \rightarrow G^3 (F (G^3 x)) \rightarrow \dots$ , where  $G^i x$  is a shorthand for  $\underbrace{G (G \dots (G x))}_i$  for any  $i > 1$ . Using the

Leibniz representation, the rewrite system is represented by the following  $\mathbf{F}_2^\mu$  environments:

$$\begin{aligned} \Delta &= F : * \Rightarrow *, G : * \Rightarrow * \\ \Gamma &= \kappa : \forall p. \forall x. p (G (F (G x))) \Rightarrow p (F x) \end{aligned}$$

Note that  $\kappa : \forall p. \forall x. p (G (F (G x))) \Rightarrow p (F x)$  corresponds to the rewrite rule  $F x \rightarrow G (F (G x))$ , where  $p$  of kind  $* \Rightarrow *$  corresponds to a reduction context.

We will first construct a hereditary head normalizing (productive) evidence  $e$  such that  $\Gamma \vdash e : F x$ . Then we will show how to check whether such  $e$  is indeed representing the nonterminating reduction above. It is enough to derive  $\Gamma \vdash e' : \forall p. \forall x. p (F x)$  for some  $e'$ . Consider the following judgement.

$$(1) \Gamma, \alpha : \forall p. \forall x. p (F x) \vdash \lambda p. \lambda x. \alpha (\lambda y. p (G y)) (G x) : \forall p. \forall x. p (G (F (G x)))$$

In (1), we instantiate the type of  $\alpha$  as follows:  $p$  is instantiated by  $\lambda y. p (G y)$  and  $x$  is instantiated by  $G x$ . Since we know that  $(\lambda y. p (G y)) (F (G x)) = p (G (F (G x)))$ , thus  $\alpha (\lambda y. p (G y)) (G x)$  has the type  $p (G (F (G x)))$ . The lambda-abstractions  $\lambda p. \lambda x.$  is used to quantify over  $p$  and  $x$  in the type of  $\alpha (\lambda y. p (G y)) (G x)$ .

From  $\forall p. \forall x. p (G (F (G x))) \Rightarrow p (F x)$  and  $\forall p. \forall x. p (G (F (G x)))$ , we can deduce the following.

$$(2) \Gamma, \alpha : \forall p. \forall x. p (F x) \vdash \lambda p. \lambda x. \kappa p x (\alpha (\lambda y. p (G y)) (G x)) : \forall p. \forall x. p (F x)$$

We now can apply *Mu* rule to (2) and obtain the following:

$$(3) \Gamma \vdash e' \equiv \mu \alpha. \lambda p. \lambda x. \kappa p x (\alpha (\lambda y. p (G y)) (G x)) : \forall p. \forall x. p (F x)$$

Thus by instantiation we have  $\Gamma \vdash e' (\lambda y. y) x : F x$ . Observe the following unfolding of  $e' (\lambda y. y) x$  (we use beta-reduction and  $\mu \alpha. e \rightsquigarrow [\mu \alpha. e / \alpha] e$  to perform reduction):

$$\begin{aligned} e' (\lambda y. y) x &\rightsquigarrow^* \kappa (\lambda y. y) x (e' (\lambda y. G y) (G x)) \rightsquigarrow^* \\ \kappa (\lambda y. y) x &(\kappa (\lambda y. G y) (G x) (e' (\lambda y. G y) (G x))) \rightsquigarrow^* \dots \end{aligned}$$

As  $\kappa$  takes a reduction context and an instantiation as its first two arguments, the gray subterms  $\kappa (\lambda y. y) x$  and  $\kappa (\lambda y. G y) (G x)$  can be read as: the first step of the reduction for  $F x$  is under the empty context  $\bullet$  using  $\kappa$  with the instantiation  $[x/x]$ . The second step is also using the  $\kappa$  rule, reducing the redex under the term context  $G \bullet$ , with the instantiation  $[G x/x]$ . As  $e' (\lambda y. y) x$  is hereditary head normalizing (productive), the exact reduction information for  $F x$  can be obtained from the unfolding.

With the help of the prototype type checker for  $F_2^\mu$ , the construction of the fully annotated evidence  $e' (\lambda y. y) x$  can be semi-automated. For this example, the user will need to provide the following.

```
K : forall p x . p (G (F (G x))) => p (F x)
h : forall p x . p (F x)
h = K h
e : F x
e = h
```

The corecursive equation  $h = K h$  can be viewed as a proof sketch for  $\text{forall } p x . p (F x)$ , it reflects the observation that the rule  $K$  is repeatedly applied in the reduction for  $F x$ . The declaration  $e : F x = h$  means that in this case we are providing an evidence for the nonterminating reduction of the term  $F x$  under the empty term context. The type checker will try to fill in the exact term contexts and instantiations using the type checking algorithm we developed. It gives the following output (no existing first-order type checking algorithm can type check the above code).

```
e : F x = h (\ x1' . x1') x
h : forall p x . p (F x) =
  \ p0' x1' . K (\ m1' . p0' m1') x1'
              (h (\ m1' . p0' (G m1')) (G x1'))
```

### 3 Modeling First-order Term Rewriting System in $F_2^\mu$

To model term rewriting, we define the type system  $F_2^\mu$ , which restricts the type abstraction of  $F_\omega$  [11] to second-order. We define Leibniz representation of rewrite rules (Definition 16) and show how it can model rewriting via Theorem 17.

► **Definition 3** (Syntax of  $F_2^\mu$ ).

<i>Evidence</i>	$e$	$::=$	$\alpha \mid \kappa \mid \lambda \alpha. e \mid e e' \mid \mu \alpha. e \mid e T \mid \lambda x. e$
<i>Term Kinds</i>	$K$	$::=$	$* \mid * \Rightarrow K$
<i>Kinds</i>	$k$	$::=$	$o \mid K$
<i>Types</i>	$T$	$::=$	$F \mid x \mid \lambda x. T \mid \forall x : K. T \mid T T' \mid T \Rightarrow T'$
<i>Environment</i>	$\Gamma$	$::=$	$\cdot \mid \alpha : T, \Gamma \mid \kappa : T, \Gamma$
<i>Type Environment</i>	$\Delta$	$::=$	$\cdot \mid x : K, \Delta \mid F : K, \Delta$

Note that  $\kappa$  denotes an evidence constant and is used to label rewrite rules (see Definition 16). The letters such as  $F, G$  is used to denote constant types. We use letters such as  $\alpha, \beta$  to denote evidence variables, and  $x, y$  to denote type variables. We use  $\lambda x.e$  to denote type-abstraction on the evidence. Fixed point abstraction  $\mu$  in  $\mu\alpha.e$  binds the variable  $\alpha$  in  $e$ . Operationally,  $\mu\alpha.e$  behaves in the same way as the lambda term  $\mathbf{Y} (\lambda\alpha.e)$ , where  $\mathbf{Y}$  is a fixpoint combinator. In our paper  $\mu f.\lambda\alpha_1.\dots\lambda\alpha_n.e$  is also represented by the corecursive equation  $f \alpha_1 \dots \alpha_n = e$ . We use  $\forall \underline{x}.T$  as a shorthand for  $\forall x_1.\dots\forall x_n.T$ , and  $e \underline{e}'$  for  $e e'_1 \dots e'_n$ , where the number  $n$  is not important.

We distinguish two notions of kinds: kind  $o$  is intended to classify types that are of formula nature, while kind  $K$  is intended to classify types that are of first-order term nature. Observe that we only allow quantification over the variables of kind  $K$  for a type. We use  $*^n \Rightarrow *$  as a shorthand for  $\underbrace{* \Rightarrow \dots \Rightarrow *}_n \Rightarrow *$ .

Comparing to  $\mathbf{F}_\omega$ , the following kinding rules of  $\mathbf{F}_2^\mu$  restrict the level of type abstraction to second-order, and stratify the types into two kinds.

► **Definition 4** (Kinding Rules).  $\boxed{\Delta \vdash T : k}$

$$\begin{array}{c} \frac{(x|F : K) \in \Delta}{\Delta \vdash x|F : K} \quad \frac{\Delta \vdash T_1 : * \quad \Delta \vdash T_2 : * \Rightarrow K}{\Delta \vdash T_2 T_1 : K} \quad \frac{\Delta, x : * \vdash T : K \quad x \in \text{FV}(T)}{\Delta \vdash \lambda x.T : * \Rightarrow K} \\[10pt] \frac{\Delta, x : K \vdash T : o|*}{\Delta \vdash \forall x : K.T : o} \quad \frac{\Delta \vdash T : o|* \quad \Delta \vdash T' : o|*}{\Delta \vdash T \Rightarrow T' : o} \end{array}$$

We use  $(x|F : K) \in \Delta$  to abbreviate  $x : K \in \Delta$  or  $F : K \in \Delta$ . And  $\Delta \vdash T : o|*$  means  $\Delta \vdash T : o$  or  $\Delta \vdash T : *$ . The kinding rule for  $\lambda x.T$  is *relevant*, i.e. the lambda bound variable  $x$  must be used in  $T$ . We have this requirement is because we want types of kind  $* \Rightarrow *$  to represent a first-order term context with at least a hole, as the proof of Theorem 25 needs this. Given an environment  $\Delta$ , it is decidable whether a type  $T$  is well-kinded. Given a type  $T$ , it is also decidable to check if there is a  $\Delta$  such that  $\Delta \vdash T : k$  for some kind  $k$ . We use  $\forall x.T$  instead of  $\forall x : K.T$  in our examples. The kind system allows us to separate two different kinds of types in  $\mathbf{F}_2^\mu$ : types that will be used to represent first-order terms and types that allow variable instantiation and modus ponens.

► **Definition 5.** We define a reduction relation  $T \rightarrow_o T'$  on types, it is the compatible closure of type level beta reduction  $(\lambda x.T) T' \rightarrow_o [T'/x]T$ .

► **Proposition 1.** If  $\Delta \vdash T : k$ , then  $T$  is strongly normalizing with respect to  $\rightarrow_o$ , and  $\rightarrow_o$  is confluent.

Let  $\text{FV}(T)$  denote the set of free variables occurring in  $T$ . The following theorem shows that the kind system satisfies the subject reduction property and the set of free type variables is invariant under the  $\rightarrow_o$ -reduction.

► **Theorem 6** (Subject Reduction for Kinding). *If  $\Delta \vdash T : k$  and  $T \rightarrow_o T'$ , then  $\text{FV}(T) = \text{FV}(T')$  and  $\Delta \vdash T' : k$ .*

► **Definition 7** (Second-order Types). A type  $T$  is *flat* iff it is one of the following forms: (1)  $T \equiv x$  or  $T \equiv F$ . (2)  $T \equiv T_1 T_2$ , where  $T_1, T_2$  are flat. We say a type  $T$  is *second-order* if  $T$  is flat or  $T \equiv \lambda x_1.\dots\lambda x_n.T'$ , where  $T'$  is flat and  $x_i \in \text{FV}(T')$  for all  $x_i \in \{x_1, \dots, x_n\}$ .

Note that types such as  $\lambda x.F x x$ ,  $\lambda x.\lambda y.F x y$ ,  $\lambda x.x$  are second-order, but  $\lambda x.\lambda y.F x \Rightarrow F y$  are not second-order. We use second-order types to model both first-order term contexts and terms. The following theorem shows that the kind system stratifies types into two kinds.

► **Theorem 8** (Properties of Kinding).

1. If  $\Delta \vdash T : o$ , then  $T$  is of the form  $\forall x.T'$  or  $T_1 \Rightarrow T_2$ .
2. If  $\Delta \vdash T : *^n \Rightarrow *$ , then the  $\rightarrow_o$ -normal form of  $T$  is second-order.

We define reduction rules for the evidence in the following.

► **Definition 9** (Evidence Reduction).

Head reduction context  $\mathcal{H} ::= \bullet \mid \mathcal{H} \ e \mid \lambda \alpha. \mathcal{H} \mid \lambda x. \mathcal{H}$

General reduction context  $\mathcal{C} ::= \bullet \mid \mathcal{C} \ e \mid \mathcal{C} \ T \mid \lambda \alpha. \mathcal{C} \mid \lambda x. \mathcal{C} \mid e \ \mathcal{C} \mid \mu \alpha. \mathcal{C}$

$$\mathcal{H}[\mu \alpha. e] \rightsquigarrow_h \mathcal{H}[[\mu \alpha. e / \alpha] e] \quad \mathcal{H}[(\lambda \alpha. e) \ e'] \rightsquigarrow_h \mathcal{H}[[e' / \alpha] e] \quad \mathcal{C}[(\lambda x. e) \ T] \rightsquigarrow_\tau \mathcal{C}[[T / x] e]$$

$$\mathcal{C}[\mu \alpha. e] \rightsquigarrow_\mu \mathcal{C}[[\mu \alpha. e / \alpha] e] \quad \mathcal{C}[(\lambda \alpha. e) \ e'] \rightsquigarrow_\beta \mathcal{C}[[e' / \alpha] e] \quad \mathcal{C}[T] \rightsquigarrow_o \mathcal{C}[T'] \text{ if } T \rightarrow_o T'$$

We call the one step reduction  $\rightsquigarrow_h \cup \rightsquigarrow_\tau \cup \rightsquigarrow_o$  a one step *head reduction*<sup>3</sup>, denoted by  $\rightsquigarrow_{h\tau o}$ . The head reduction is lazy, i.e.,  $\mu \alpha. \kappa \ \alpha$  is normalizing with head reduction. We call an evidence a *head normal form* if it can not be one step reduced by  $\rightsquigarrow_{h\tau o}$ .

► **Theorem 10.**  $\rightsquigarrow_{\beta\mu\tau o}$  and  $\rightsquigarrow_{h\tau o}$  are confluent, and  $\rightsquigarrow_\tau$  is strongly normalizing.

We specify the typing rules for  $\mathbf{F}_2^\mu$  in the following.

► **Definition 11** (Typing of  $\mathbf{F}_2^\mu$ ).

$$\begin{array}{c} \frac{(\alpha \mid \kappa : T) \in \Gamma}{\Gamma \vdash \alpha \mid \kappa : T} \quad \frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T' \Rightarrow T}{\Gamma \vdash e_2 \ e_1 : T} \text{ (APP)} \quad \frac{\Gamma, \alpha : T' \vdash e : T}{\Gamma \vdash \lambda \alpha. e : T' \Rightarrow T} \text{ (LAM)} \\[10pt] \frac{\Gamma, \alpha : T \vdash e : T}{\Gamma \vdash \mu \alpha. e : T} \text{ (MU)} \quad \frac{\Gamma \vdash e : \forall x : K. T}{\Gamma \vdash e \ T' : [T' / x] T} \text{ (INST)} \quad \frac{\Gamma \vdash e : T \quad x \notin \text{FV}(\Gamma)}{\Gamma \vdash \lambda x. e : \forall x : K. T} \text{ (ABS)} \\[10pt] \frac{\Gamma \vdash e : T \quad T \leftrightarrow_o^* T'}{\Gamma \vdash e : T'} \text{ (CONV)} \end{array}$$

In the ABS rule, only the types of kind  $K$  are quantified. We use  $\text{FV}(\Gamma)$  to denote the set of free type variables occurs in  $\Gamma$ . We require that all the types are well-kinded. Since  $\rightarrow_o$  is strongly normalizing and confluent, we will work with types in  $\rightarrow_o$ -normal form in this paper. The rule CONV is used implicitly.

The followings theorems shows that the type system  $\mathbf{F}_2^\mu$  has the usual inversion and subject reduction properties.

► **Theorem 12** (Selected Inversion Theorems).

1. If  $\Gamma \vdash e \ e' : T$ , then  $\Gamma \vdash e : T_1 \Rightarrow T_2$ ,  $\Gamma \vdash e' : T_1$  and  $T_2 \leftrightarrow_o^* T$ .
2. If  $\Gamma \vdash e \ T_1 : T$ , then  $\Gamma \vdash e : \forall x : K. T'$  and  $[T_1 / x] T' \leftrightarrow_o^* T$ .

► **Theorem 13** (Subject Reduction). If  $\Gamma \vdash e : T$  and  $e \rightsquigarrow_{h\tau o} e'$ , then  $\Gamma \vdash e' : T$ .

Due to MU rule,  $\mathbf{F}_2^\mu$  allows diverging evidence with respect to  $\rightsquigarrow_{\beta\mu}$ . We will focus on the *hereditary head normalizing* evidence (Definition 19), which will be discussed in Section 4.

► **Definition 14** (Terms and Contexts).

First-order term  $t, l, r ::= x \mid F^n \ t_1 \ \dots \ t_n$

Term context  $C ::= \bullet \mid x \mid F^n \ C_1 \ \dots \ C_n$

<sup>3</sup> This definition is following Barendregt [3], Page 173.

Note that the term context can contains multiple  $\bullet$  and we use the notation  $C[t_1, \dots, t_n]$  to denote the result of replacing  $\bullet$  from left to right in  $C$  by  $t_1, \dots, t_n$ . A special case is  $C[t]$ , it means there is exactly one  $\bullet$  in  $C$ , which is replaced by  $t$ . The function symbol  $F$  of arity  $n$  is denoted by  $F^n$ . We work with applicative first-order terms in this paper, and we assume all function symbols are fully applied, thus we often write  $F t_1 \dots t_n$  instead of  $F^n t_1 \dots t_n$ . We reuse  $\text{FV}(t)$  to mean the set of free variables in  $t$ .

► **Definition 15** (Rewrite Rules). Suppose  $l$  and  $r$  are first-order terms, where  $l$  is not a variable and  $\text{FV}(r) \subseteq \text{FV}(l)$ , then  $l \rightarrow r$  is a first-order rewrite rule. A rewriting system is a set  $\mathcal{R}$  of rewrite rules. We write  $C[t] \rightarrow C[t']$  if there exists  $l \rightarrow r \in \mathcal{R}$  such that  $\sigma l \equiv t$  and  $\sigma r \equiv t'$  for some substitution  $\sigma$ .

**Important Notation Convention.** We use the notation  $t$  to denote a first-order type in  $\mathbf{F}_2^\mu$  that represents the first-order term  $t$ . The term context  $C$  containing one  $\bullet$  can be represented as  $\lambda x. C[x]$ , a second-order type of kind  $* \Rightarrow *$  in  $\mathbf{F}_2^\mu$ . We use letters  $F, G, D, S, Z$  to denote type constants as well as function symbols. Note that for any first-order term  $t$ , it is always a well-kinded first-order type, since for any function symbol  $F^n$  in  $t$ , we can assign the kind  $*^n \Rightarrow *$  for  $F$  and first-order term variable is of kind  $*$ . The following definition illustrates our use of this notation convention.

► **Definition 16** (Leibniz representation). Given a set of rewrite rules  $\mathcal{R}$ , we define the *Leibniz representation* of  $\mathcal{R}$  as  $\mathbf{F}_2^\mu$ -environments  $\Gamma_{\mathcal{R}}, \Delta_{\mathcal{R}}$ , as follows:

- $\kappa : \forall p. \forall \underline{x}. p r \Rightarrow p l \in \Gamma_{\mathcal{R}}$  whenever  $l \rightarrow r \in \mathcal{R}$ , and where  $\kappa$  is a fresh evidence constant and  $\underline{x}$  are the free variables in  $l$ .
- $F : *^n \Rightarrow * \in \Delta_{\mathcal{R}}$  if  $F^n$  is a function symbol in  $\mathcal{R}$ .

Leibniz representation allows us to represent a first-order term rewriting system as a typing environment in  $\mathbf{F}_2^\mu$ , together with the typing rules, finite reductions can be represented by a typing judgement in  $\mathbf{F}_2^\mu$ .

► **Theorem 17.** Let  $\mathcal{R}$  be a set of rewrite rules.

1. If  $C[t] \rightarrow C[t']$  by  $l \rightarrow r \in \mathcal{R}$ , then  $\Gamma_{\mathcal{R}} \vdash e : C[t'] \Rightarrow C[t]$  for some  $e$ .
2. If  $t_1 \rightarrow t_2 \rightarrow t_3$  is a reduction using  $\mathcal{R}$ , then  $\Gamma_{\mathcal{R}} \vdash e : t_3 \Rightarrow t_1$  for some  $e$ .

**Proof.** 1. By Definition 16, we have  $\kappa : \forall p. \forall \underline{x}. p r \Rightarrow p l \in \Gamma_{\mathcal{R}}$ . We instantiate  $p$  with  $\lambda y. C[y]$ , by rule CONV, we get  $\Gamma_{\mathcal{R}} \vdash \kappa (\lambda y. C[y]) : \forall \underline{x}. C[r] \Rightarrow C[l]$ . Since  $\sigma l \equiv t, \sigma r \equiv t'$ , let  $\underline{t}$  be the codomain of  $\sigma$ , we have  $\Gamma_{\mathcal{R}} \vdash \kappa (\lambda y. C[y]) \underline{t} : C[t'] \Rightarrow C[t]$ .

2. By (1), we have  $\Gamma_{\mathcal{R}} \vdash e_1 : t_2 \Rightarrow t_1$  and  $\Gamma_{\mathcal{R}} \vdash e_2 : t_3 \Rightarrow t_2$ , so  $\Gamma_{\mathcal{R}} \vdash \lambda \alpha. e_1 (e_2 \alpha) : t_3 \Rightarrow t_1$ . ◀

## 4 Hereditary Head Normalization and Faithfulness

In this section we define the *hereditary head normalization* for an evidence (Definition 19). The role of hereditary head normalization is similar to *productivity*, i.e. a hereditary head normalizing evidence can be associated with a computational tree (Böhm tree without bottom [3]). In  $\mathbf{F}_2^\mu$ , hereditary head normalization implies *faithfulness*. Informally, an evidence is *faithful* if we can recover a nonterminating reduction from it.

To define hereditary head normalization, we first define an erasure that maps  $\mathbf{F}_2^\mu$ -evidence to pure lambda term with fixed point operator.

► **Definition 18** (Erasure). We define erasure  $|\cdot|$  on evidence as follows.

$$|\alpha| = \alpha \quad |\kappa| = \kappa \quad |\lambda \alpha. e| = \lambda \alpha. |e| \quad |\mu \alpha. e| = \mu \alpha. |e| \quad |e e'| = |e| |e'| \quad |\lambda x. e| = |e| \quad |e T| = |e|$$



We call the erased evidence  $|e|$  *Curry-style evidence*. The following definition follows the same formulation by Raffalli [17] and Tatsuta [21].

► **Definition 19** (Hereditary Head Normalization). Let  $\Lambda$  be the set of Curry-style evidence. We say  $e$  is *hereditary head normalizing* (denoted by  $e \in \text{HHN}$ ) iff  $|e| \in \text{HN}_n$  for all  $n \geq 0$ . We define  $\text{HN}_n$  as follows.

- $e \in \text{HN}_0$  iff  $e \in \Lambda$ .
- $e \in \text{HN}_{n+1}$  iff  $e \rightsquigarrow_{\beta\mu}^* \lambda \underline{\alpha}. e' \ e_1 \ \dots \ e_m$ , where  $e'$  is a variable or a constant and  $e_i \in \text{HN}_n$  for all  $i$ .

We are going to show in Theorem 25 that if  $\Gamma_{\mathcal{R}} \vdash e : t$  in  $\mathbf{F}_2^\mu$  and  $e$  is hereditary head normalizing, then we can reconstruct a nonterminating reduction of  $t$  by following the unfolding of  $e$ . First we define the notion of *trace*. The position of a trace is described as follows: Let  $o$  denote the origin of a trace and  $s \cdot m$  denote the next position after  $m$ . For a trace  $\mathcal{T}$ , we use  $\mathcal{T}_m$  to refer to the node at position  $m$  in the trace. The following formalization of *evidence trace* is a degenerate case of Böhm tree ([4], [3, §10]).

► **Definition 20** (Evidence Trace). Suppose  $e \rightsquigarrow_{h\tau o}^* \kappa \ T_1 \dots T_n \ e'$ , with  $T_1, \dots, T_n$  in  $\rightarrow_o$ -normal form. The evidence trace of  $e$ , denoted by  $[e]$ , is defined as:

- $[e]_o = \kappa \ T_1 \dots T_n$ .
- $[e]_{s \cdot m} = [e']_m$ .

In the above definition, since  $\kappa \ T_1 \dots T_n \ e'$  is a head normal form, by the confluence of  $\rightsquigarrow_{h\tau o}$  (Theorem 10), we know that  $[e]$  is referring to at most one trace. When  $e \not\rightsquigarrow_{h\tau o}^* \kappa \ T_1 \dots T_n \ e'$ , we say  $[e]$  is undefined. For an example of finite evidence trace, consider  $e \equiv \kappa \ (\lambda y. y) \ (\kappa' \ (\lambda y. y))$ , in this case  $[e]_o = \kappa \ (\lambda y. y)$ ,  $[e]_{s \cdot o} = \kappa' \ (\lambda y. y)$ . For an example of an infinite evidence trace, consider  $e \equiv \mu \alpha. \kappa \ (\lambda y. y) \ \alpha$ , we have  $[e]_m = \kappa \ (\lambda y. y)$  for any position  $m$ .

Intuitively, an evidence trace can be viewed as a sequence of instructions (in the form of evidence constants) that we are going to follow in order to rewrite a term. The following definitions of *action* and *faithful action* on a first-order term reflects this intuition. Suppose  $C[\sigma l, \dots, \sigma l] \rightarrow^* C[\sigma r, \dots, \sigma r]$  by  $l \rightarrow r \in \mathcal{R}$ . We record the term context and the instantiation information along the reduction, i.e.  $C[\sigma l, \dots, \sigma l] \rightarrow_{(\kappa, C, \sigma)}^* C[\sigma r, \dots, \sigma r]$ .

► **Definition 21** (Action on First-Order Term). Suppose  $[e]_m = \kappa \ (\lambda x. C[x, \dots, x]) \ t_1 \dots t_n$  for some position  $m$  and  $\kappa : \forall p. \forall \underline{x}. p \ r \Rightarrow p \ l$ . An *action* of  $[e]_m$  on the first-order term  $t$  (denoted by  $[e]_m(t)$ ) is defined as follows.

- $[e]_m(t) = t'$  if  $t \rightarrow_{(\kappa, C, \sigma)}^* t'$ , where  $\sigma = [t_1/x_1, \dots, t_n/x_n]$ .
- otherwise  $[e]_m(t)$  is undefined.

Note that we write  $t \rightarrow^* [e]_m(t)$  when  $[e]_m(t)$  is defined. The following definition of *faithful action* shows how one follows a potentially infinite evidence trace to reduce a term.

► **Definition 22** (Faithful Action). The evidence trace  $[e]$  acts on  $t$  *faithfully*, if we have a reduction sequence  $t \rightarrow^* [e]_o(t) \rightarrow^* [e]_{s \cdot o}([e]_o(t)) \rightarrow^* [e]_{s \cdot s \cdot o}([e]_{s \cdot o}([e]_o(t))) \rightarrow^* \dots \rightarrow^* [e]_m(\dots [e]_o(t) \dots)$  for any position  $m$ .

► **Example 23.** To illustrate the intuition behind Definitions 20, 21, 22, let us consider the one rule rewriting system:  $F \ x \rightarrow G \ (F \ (G \ x))$  in Example 2. The Leibniz representation is  $\Delta = F : * \Rightarrow *, G : * \Rightarrow *, \Gamma = \kappa : \forall p. \forall x. p \ (G \ (F \ (G \ x))) \Rightarrow p \ (F \ x)$ . Recall that we had the following judgement.

$$(1) \ \Gamma \vdash e' \equiv \mu \alpha. \lambda p. \lambda x. \kappa \ p \ x \ (\alpha \ (\lambda y. p \ (G \ y)) \ (G \ x)) : \forall p. \forall x. p \ (F \ x)$$

$$(2) \ \Gamma \vdash e' \ (\lambda y. y) \ x : F \ x$$



We observed the following unfolding of  $e' (\lambda y.y) x$  (below  $\mathcal{C} \equiv \kappa (\lambda y.y) x (\kappa (\lambda y.G y) (G x) \bullet)$ ):

$$\begin{aligned} e' (\lambda y.y) x &\rightsquigarrow_{\beta\mu\tau o}^* \kappa (\lambda y.y) x (e' (\lambda y.G y) (G x)) \rightsquigarrow_{\beta\mu\tau o}^* \\ \kappa (\lambda y.y) x (\kappa (\lambda y.G y) (G x) (e' (\lambda y.G (G y)) (G (G x)))) &\rightsquigarrow_{\beta\mu\tau o}^* \\ \mathcal{C}[(\kappa (\lambda y.G (G y)) (G (G x))) (e' (\lambda y.G (G (G y)))) (G (G (G x))))] &\rightsquigarrow_{\beta\mu\tau o}^* \dots \end{aligned}$$

It gives rise to the following evidence trace:  $[e]_o = \kappa (\lambda y.y) x$ ,  $[e]_{s.o} = \kappa (\lambda y.G y) (G x)$ ,  $[e]_{s.s.o} = \kappa (\lambda y.G (G y)) (G (G x))$ , etc. Moreover  $[e]$  acts faithfully on  $F x$  (by Theorem 25). For example, we observe that  $F x \rightarrow [e]_o(F x) \rightarrow [e]_{s.o}([e]_o(F x)) \rightarrow [e]_{s.s.o}([e]_{s.o}([e]_o(F x)))$ , which is the following reduction trace.

$$\begin{aligned} F x &\rightarrow_{(\kappa, \bullet, [x/x])} G (F (G x)) \rightarrow_{(\kappa, G \bullet, [(G x)/x])} \\ G (G (F (G (G x)))) &\rightarrow_{(\kappa, G (G \bullet), [(G (G x))/x])} G (G (G (F (G (G (G x))))) \end{aligned}$$

► **Lemma 24.** Suppose  $\Gamma_{\mathcal{R}} \vdash e : t$  for some first-order term  $t$  and  $e$  is head normalizing. We have  $e \rightsquigarrow_{h\tau o}^* \kappa (\lambda x.C[x, \dots, x]) t_1 \dots t_n e'$  for some  $\kappa : \forall p. \forall x.p \Rightarrow p \mid \in \Gamma_{\mathcal{R}}$ . Furthermore, we have  $\Gamma_{\mathcal{R}} \vdash e' : C[\sigma r, \dots, \sigma r]$  and  $C[\sigma l, \dots, \sigma l] = t$ , where  $\text{codom}(\sigma) = \{t_1, \dots, t_n\}$  and  $\text{dom}(\sigma) = \text{FV}(l)$ .

► **Theorem 25 (Faithfulness of Corecursive Evidence).** Suppose  $\Gamma_{\mathcal{R}} \vdash e : t$  in  $\mathbf{F}_2^\mu$  and  $e \in \text{HHN}$ . We have  $t \rightarrow^* [e]_o(t) \rightarrow^* \dots \rightarrow^* [e]_m(\dots [e]_o(t) \dots)$  for any position  $m$ , i.e.  $e$  acts faithfully on  $t$ .

**Proof.** By Lemma 24, we know that  $e \rightsquigarrow_{h\tau o}^* \kappa (\lambda x.C[x, \dots, x]) t_1 \dots t_n e'$  for some  $\kappa : \forall p. \forall x.p \Rightarrow p \mid \in \Gamma_{\mathcal{R}} \vdash e' : C[\sigma r, \dots, \sigma r]$ ,  $C[\sigma l, \dots, \sigma l] = t$ , where  $\text{codom}(\sigma) = \{t_1, \dots, t_n\}$  and  $\text{dom}(\sigma) = \text{FV}(l)$ . Thus  $t = C[\sigma l, \dots, \sigma l] \rightarrow_{(\kappa, C, \sigma)}^* C[\sigma r, \dots, \sigma r]$ . We prove the theorem by induction on  $m$ .

- $m = o$ . We have  $[e]_o = \kappa (\lambda x.C[x, \dots, x]) t_1 \dots t_n$ , since  $t \rightarrow_{(\kappa, C, \sigma)}^* C[\sigma r, \dots, \sigma r]$ , so  $t \rightarrow^* [e]_o(t)$ .
- $m = s \cdot m'$ . We need to show  $t \rightarrow^* [e]_o(t) \rightarrow^* \dots \rightarrow^* [e]_{s.m'}(\dots [e]_o(t) \dots)$ . Since  $\Gamma_{\mathcal{R}} \vdash e' : C[\sigma r, \dots, \sigma r]$  and  $e' \in \text{HHN}$ , by IH, we have  $C[\sigma r, \dots, \sigma r] \rightarrow^* [e']_o(C[\sigma r, \dots, \sigma r]) \rightarrow^* \dots \rightarrow^* [e']_{m'}(\dots [e']_o(C[\sigma r, \dots, \sigma r]) \dots)$ . Thus  $t \rightarrow^* [e]_o(t) = C[\sigma r, \dots, \sigma r] \rightarrow^* [e']_o([e]_o(t)) \rightarrow^* \dots \rightarrow^* [e']_{m'}(\dots [e']_o([e]_o(t)) \dots)$ . Since  $[e']_a = [e]_{s.a}$  for any position  $a$ , we have  $t \rightarrow^* [e]_o(t) \rightarrow^* [e]_{s.o}([e]_o(t)) \rightarrow^* \dots \rightarrow^* [e]_{s.m'}(\dots [e]_{s.o}([e]_o(t)) \dots)$ . ◀

Now we are going to show the hereditary head normalization for  $\mathbf{F}_2^\mu$  is decidable by mapping a typable evidence in  $\mathbf{F}_2^\mu$  to a typable evidence in  $\lambda$ -Y calculus (simply typed lambda calculus with fixpoint typing rule [19])<sup>4</sup>.

► **Definition 26.** We define a function  $\theta$  that maps  $\mathbf{F}_2^\mu$  types to  $\lambda$ -Y types.

$$\theta(x|F) = B \quad \theta(\lambda x.T) = \theta(T) \quad \theta(T \ T') = \theta(T) \quad \theta(T \Rightarrow T') = \theta(T) \Rightarrow \theta(T') \quad \theta(\forall x.T) = \theta(T)$$

We write  $\theta(\Gamma)$  to mean applying the function  $\theta$  to all the types in  $\Gamma$ . Type  $B$  is the based type in  $\lambda$ -Y.

► **Theorem 27.** If  $\Gamma \vdash e : T$  and  $\Delta \vdash T : *|o$  in  $\mathbf{F}_2^\mu$ , then  $\theta(\Gamma) \vdash |e| : \theta(T)$  in  $\lambda$ -Y.

Theorem 27 implies that the hereditary head normalization for  $\mathbf{F}_2^\mu$  is decidable, since it is well-known that hereditary head normalization for  $\lambda$ -Y is decidable ([5], [18], [13]).

<sup>4</sup> Please see Appendix F for full details.

## 5 Type Checking $F_2^\mu$ Based on Resolution with Second-order Matching

Modeling first-order term contexts is one of the reasons we use second-order types. Quantification over second-order type variables also enables us to represent some *nonlooping* nonterminations in  $F_2^\mu$ .

► **Example 28.** Consider the following rewrite rules [10].

$$\begin{aligned} D (S x) y &\rightarrow_a D x (S y) \\ D Z y &\rightarrow_b D (S y) Z \end{aligned}$$

The term  $D Z Z$  will give rise to the following nonlooping nonterminating reduction, where no cycle or loop can be observed:

$$\begin{aligned} D Z Z &\rightarrow_b D (S Z) Z \rightarrow_a D Z (S Z) \rightarrow_b D (S (S Z)) Z \rightarrow_a D (S Z) (S Z) \rightarrow_a D Z (S (S Z)) \rightarrow_b \\ &D (S (S (S Z))) Z \rightarrow_a D (S (S Z)) (S Z) \rightarrow_a D (S Z) (S (S Z)) \rightarrow_a D Z (S (S (S Z))) \rightarrow \dots \end{aligned}$$

The rule sequence for this reduction exhibits the pattern: “ $ba, baa, baaa, \dots$ ”, which can be represented by the corecursive function  $f \alpha \beta = (\beta \cdot \alpha) (f \alpha (\beta \cdot \alpha))$  (here  $\cdot$  denotes functional composition), as  $f a b$  would give rise to the following reduction (we omit the compositional symbols):

$$f a b \rightsquigarrow (ba)(f a (ba)) \rightsquigarrow (baba)(f a (baa)) \rightsquigarrow (babaabaaa)(f a (baaa))$$

Let the Leibniz representation of the rewriting system be as follows:

$$\begin{aligned} \Delta &= D : *^2 \Rightarrow *, Z : *, S : * \Rightarrow * \\ \Gamma &= \kappa_a : \forall p. \forall x. \forall y. p (D x (S y)) \Rightarrow p (D (S x) y), \kappa_b : \forall p. \forall y. p (D (S y) Z) \Rightarrow p (D Z y) \end{aligned}$$

We would like to provide a type annotation for  $f$  such that  $\Gamma \vdash f \kappa_a \kappa_b : D Z Z$ . But it is not obvious as we cannot type check  $f \kappa_a \kappa_b$  with  $D Z Z$  using any first-order type checking algorithm (e.g. the one in Haskell). We will show how to type check  $f$  using the type checking algorithm we introduce in this section.

By *type checking*, we mean the following problem: given an environment  $\Gamma$ , a Curry-style evidence  $e$  and a type  $T$ , construct a fully annotated evidence  $e'$  such that  $\Gamma \vdash e' : T$  and  $|e'| = e$ . We use the terminology *proof checking* to mean the following: given an environment  $\Gamma$ , a fully annotated evidence  $e$  and a type  $T$ , check if  $\Gamma \vdash e : T$ . The type checking problem for Curry-style System  $F$  and  $F_\omega$  are well-known to be undecidable ([24], [23]). The type system  $F_2^\mu$  appears to be a much weaker system compared to System  $F$  and  $F_\omega$  (HHN is decidable in  $F_2^\mu$ ), we will show a type checking algorithm for  $F_2^\mu$  inspired by SLD-resolution [16]. We will work on types that are kindable by our decidable kind system (Definition 4). Moreover, we will consider the following reformulation of type  $T$  from Definition 3:

$$T ::= A \mid \forall \underline{x}. T \Rightarrow \dots \Rightarrow T \Rightarrow A$$

Here  $A$  is of kind  $*$ . We use  $T_1, \dots, T_n \Rightarrow A$  as a shorthand for  $T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow A$  and we call  $A$  the *head* of  $T_1, \dots, T_n \Rightarrow A$ . These types are a generalized version of Horn formulas, called *hereditary Harrop formula* in the literature [15].

In this section we use  $A, B$  to denote a type of kind  $*$ , and we use  $a, b$  to denote a type variable or a type constant. The following definition of second-order matching follows Dowek’s treatment [7] of Huet’s algorithm [14].

► **Definition 29** (Second-order Matching). Let  $E$  be a set of second-order matching problems  $\{A_1 \mapsto B_1, \dots, A_n \mapsto B_n\}$ . The following rules (intended to apply top-down) show how to transform  $E$ .

$$\begin{array}{c} \frac{\{F A_1 \dots A_n \mapsto G B_1 \dots B_m, E\}}{\perp} \\[10pt] \frac{E' \equiv \{y A_1 \dots A_n \mapsto a B_1 \dots B_m, E\}}{[(\lambda x_1. \dots \lambda x_n. x_i / y) E']} \text{ PROJ} \end{array} \quad \begin{array}{c} \frac{\{a A_1 \dots A_n \mapsto a B_1 \dots B_n, E\}}{\{A_1 \mapsto B_1, \dots, A_n \mapsto B_n, E\}} \\[10pt] \frac{E' \equiv \{y A_1 \dots A_n \mapsto a B_1 \dots B_m, E\}}{[(\lambda x_1. \dots \lambda x_n. a (y_1 \ x) \dots (y_m \ x)) / y] E'} \text{ IMI} \end{array}$$

Note that  $\perp$  denotes a failure in matching. In the IMI rule, the variables  $y_1, \dots, y_m$  are fresh type variables. The PROJ and IMI rules introduce nondeterminism, so there may be multiple matchers for a matching problem  $A \mapsto B$ . We write  $A \mapsto_\sigma B$  to mean there is a derivation from  $A \mapsto B$  to  $\emptyset$  using rules in the above definition with a second-order matcher  $\sigma$ . The second-order matching is decidable (all derivations are finite using Definition 29) and all the resulted matchers are finite, but second-order unification is not decidable [12].

The standard second-order matching algorithm usually generates many vacuous substitutions, we can exclude them by kinding, as we work with kindable types. For example, when we match  $d Z Z$  against  $D Z (S Z)$ , the second-order matching algorithm would generate matchers such as  $[\lambda x. \lambda y. D Z (S Z) / d]$  and  $[\lambda x. \lambda y. D y (S y) / d]$ , which are not kindable.

Let  $T = \forall x_1. \dots \forall x_m. T_1, \dots, T_n \Rightarrow A$ , the set of variables  $\{x_i \mid x_i \notin \text{FV}(A), 1 \leq i \leq m\} \cup \text{FV}(T)$  are called *existential variables*. In this section, we work with types that do not have any existential variables, we will show how to handle existential variables in the next section. We use  $\Phi$  to denote a set of tuples of the form  $(\Gamma, e, T)$ . We define *resolution by second-order matching* as a transition system from  $\Phi$  to  $\Phi'$  as follows:

► **Definition 30** (Resolution by Second-order Matching (RSM)).  $\boxed{\Phi \longrightarrow \Phi'}$

1.  $\{(\Gamma, (\kappa|\alpha) e_1 \dots e_n, A), \Phi\} \longrightarrow_a \{(\Gamma, e_1, \sigma T_1), \dots, (\Gamma, e_n, \sigma T_n), \Phi\}$  if  $\kappa|\alpha : \forall \underline{x}. T_1, \dots, T_n \Rightarrow B \in \Gamma$  with  $B \mapsto_\sigma A$ .
2.  $\{(\Gamma, \lambda \alpha_1. \dots \lambda \alpha_n. e, T_1, \dots, T_n \Rightarrow A), \Phi\} \longrightarrow_i \{([\Gamma, \alpha_1 : T_1, \dots, \alpha_n : T_n], e, A), \Phi\}$ .
3.  $\{(\Gamma, e, \forall x_1. \dots \forall x_n. T), \Phi\} \longrightarrow_\forall \{(\Gamma, e, T), \Phi\}$ .
4.  $\{(\Gamma, \mu \alpha. e, T), \Phi\} \longrightarrow_c \{([\Gamma, \alpha : T], e, T), \Phi\}$ .

As before,  $\kappa|\alpha$  means “ $\kappa$  or  $\alpha$ ”. The rule (1) allow the the size of  $\{e_1, \dots, e_n\}$  to be zero. We require the sizes of  $\{\alpha_1, \dots, \alpha_n\}$  and  $\{x_1, \dots, x_n\}$  both to be nonzero for rules (2) and (3). Rule (3) also introduces fresh *eigenvariables*  $\{x_1, \dots, x_n\}$  for  $T$ , they behave the same as constants during RSM. In rule (1), when perform matching  $B \mapsto_\sigma A$ , we rename the bound variables  $\underline{x}$  in  $T_1, \dots, T_n, B$  to fresh variables. The  $T$  in the tuple  $(\Gamma, e, T)$  intuitively corresponds to the current goal for the resolution and  $e$  is a Curry-style evidence that can be understood as a list of instructions for the resolution algorithm. The resolution is defined by case analysis on the Curry-style evidence and the current goal  $T$  and it is terminating. If it terminates with the empty set, then we say the resolution succeeds, otherwise it fails. The following theorem shows that if the resolution succeeds, then the type checking succeeds, i.e. we can obtain the corresponding fully annotated evidence.

► **Theorem 31** (Soundness of RSM). *If  $\{(\Gamma, e, T)\} \longrightarrow^* \emptyset$ , then there exists an evidence  $e'$  such that  $\Gamma \vdash e' : T$  in  $\mathbf{F}_2^\mu$  and  $|e'| = e$ .*

The proof of Theorem 31 gives us an algorithm to compute the annotated evidence  $e'$ . This algorithm is implemented in our prototype.

► **Example 32.** Continuing the Example 28, let us illustrate how to use RSM to type check the function  $f$ . Consider the long form of  $f$ , namely,  $f = \mu f. \lambda \alpha. \lambda \beta. \beta(\alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha'))))$  and the Leibniz representation:

$$\Delta = D : *^2 \Rightarrow *, Z : *, S : * \Rightarrow * \\ \Gamma = \kappa_a : \forall p. \forall x. \forall y. p (D x (S y)) \Rightarrow p (D (S x) y), \kappa_b : \forall p. \forall y. p (D (S y) Z) \Rightarrow p (D Z y).$$

As we want  $\Gamma \vdash f \kappa_a \kappa_b : D Z Z$ , the most intuitive type that we can assign to  $f$  is the following.

$$T \equiv (\forall p. \forall x. \forall y. p (D x (S y)) \Rightarrow p (D (S x) y)) \Rightarrow (\forall p. \forall y. p (D (S y) Z) \Rightarrow p (D Z y)) \Rightarrow D Z Z$$

But  $f$  can not be type checked with  $T$  by RSM. The solution is abstracting  $D$  to a second-order variable  $d$  and assigning the following type to  $f$ :

$$T' \equiv \forall d. \underbrace{(\forall p. \forall x. \forall y. p (d x (S y)) \Rightarrow p (d (S x) y)) \Rightarrow (\forall p. \forall y. p (d (S y) Z) \Rightarrow p (d Z y)) \Rightarrow d Z Z}_{T''}$$

This change yields the following successful RSM resolution trace.

$$\begin{aligned} & \{(\Gamma, \mu f. \lambda \alpha. \lambda \beta. \beta(\alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha')))), T')\} \longrightarrow_c \\ & \{([\Gamma, f : T'], \lambda \alpha. \lambda \beta. \beta(\alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha')))), T')\} \longrightarrow_\forall \\ & \{([\Gamma, f : T'], \lambda \alpha. \lambda \beta. \beta(\alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha')))), [d_1/d]T')\} \longrightarrow_i \\ & \{(\Gamma'', \beta(\alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha')))), d_1 Z Z)\} \longrightarrow_a \\ & \{(\Gamma'', \alpha (f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha'))), d_1 (S Z) Z)\} \longrightarrow_a \\ & \{(\Gamma'', f (\lambda \alpha'. \alpha \alpha') (\lambda \alpha'. \beta(\alpha \alpha')), d_1 Z (S Z))\} \longrightarrow_a \\ & \{(\Gamma'', \lambda \alpha'. \alpha \alpha', \forall p. \forall x. \forall y. p (d_1 x (S (S y))) \Rightarrow p (d_1 (S x) (S y))), \Phi_1 \equiv \\ & (\Gamma'', \lambda \alpha'. \beta(\alpha \alpha'), \forall p. \forall y. p (d_1 (S y) (S Z)) \Rightarrow p (d_1 Z (S y)))\} \longrightarrow_\forall \\ & \{(\Gamma'', \lambda \alpha'. \alpha \alpha', p_1 (d_1 x_1 (S (S y_1))) \Rightarrow p_1 (d_1 (S x_1) (S y_1))), \Phi_1\} \longrightarrow_i \\ & \{([\Gamma'', \alpha' : p_1 (d_1 x_1 (S (S y_1)))]], \alpha \alpha', p_1 (d_1 (S x_1) (S y_1))), \Phi_1\} \longrightarrow_a \\ & \{([\Gamma'', \alpha' : p_1 (d_1 x_1 (S (S y_1)))]], \alpha', p_1 (d_1 x_1 (S (S y_1))), \Phi_1\} \longrightarrow_a \\ & \{(\Gamma'', \lambda \alpha'. \beta(\alpha \alpha'), \forall p. \forall y. p (d_1 (S y) (S Z)) \Rightarrow p (d_1 Z (S y)))\} \longrightarrow_\forall \\ & \{(\Gamma'', \lambda \alpha'. \beta(\alpha \alpha'), p_2 (d_1 (S y_2) (S Z)) \Rightarrow p_2 (d_1 Z (S y_2)))\} \longrightarrow_i \\ & \{([\Gamma'', \alpha' : p_2 (d_1 (S y_2) (S Z))], \beta(\alpha \alpha'), p_2 (d_1 Z (S y_2))\} \longrightarrow_a \\ & \{([\Gamma'', \alpha' : p_2 (d_1 (S y_2) (S Z))], \alpha \alpha', p_2 (d_1 (S (S y_2)) Z)\} \longrightarrow_a \\ & \{([\Gamma'', \alpha' : p_2 (d_1 (S y_2) (S Z))], \alpha', p_2 (d_1 (S y_2) (S Z))\} \longrightarrow_a \emptyset \end{aligned}$$

Note that  $\Gamma'' = \Gamma, f : T', \alpha : \forall p. \forall x. \forall y. p (d_1 x (S y)) \Rightarrow p (d_1 (S x) y), \beta : \forall p. \forall y. p (d_1 (S y) Z) \Rightarrow p (d_1 Z y)$ . At the third  $\longrightarrow_a$ -step, by second-order matching, we instantiate the  $d$  in the type of  $f$  to  $\lambda x. \lambda y. d_1 x (S y)$ . Now that  $f$  is typable with  $T'$ , we have  $\Gamma \vdash f D \kappa_a \kappa_b : D Z Z$ . Since the rewriting system is non-overlapping and  $f$  is hereditary head normalizing, by Theorem 25 we know  $f D \kappa_a \kappa_b$  represents the nonterminating reduction of  $D Z Z$ .

Representing nonterminations in general follows the same method as the above example: one first writes down a corecursive function that represents the rule sequence in a nonterminating reduction, and then provides the proper type signature for such function. Once the function is type checked, a finite representation can be obtained. We illustrate how the prototype works for this example and some other challenging examples in the Appendix H, J.

## 6 RSM Algorithm with Existential Variables

The RSM algorithm in Definition 30 fails to type check some judgements in presence of existential variables. In this section, we extend RSM to cope with existential variables. As a result, the nontermination reduction in the Example 1 can also be type checked.

We consider the following sequential reduction that simulates the parallel reduction sequence in the Example 1. At each reduction step, we underline the chosen redex.

$$\begin{aligned} \underline{A} &\rightarrow_a \underline{AB} \rightarrow_a \underline{ABB} \rightarrow_b \underline{ABA} \rightarrow_a \underline{ABBA} \rightarrow_b \underline{ABAA} \rightarrow_a \underline{ABAAB} \rightarrow_a \underline{ABBAAB} \rightarrow_b \\ &\underline{ABAAAB} \rightarrow_a \underline{ABAABAB} \rightarrow_a \underline{ABAABABB} \rightarrow_b \underline{ABAABABA} \rightarrow_a \underline{ABBAABABA} \rightarrow_b \\ &\underline{ABAAABABA} \rightarrow_a \underline{ABAABABABA} \rightarrow_a \underline{ABAABABBABA} \rightarrow_b \underline{ABAABABAABA} \rightarrow_a \\ &\underline{ABAABABAABBA} \rightarrow_b \underline{ABAABABAABA} \rightarrow_a \underline{ABAABABAABAAB} \rightarrow \dots \end{aligned}$$

Observe that the length of the gray strings grows according to the Fibonacci sequence, and each gray string is a result of concatenation of the previous two.

The rule sequence in the above reduction is “ $a, ab, aba, abaab, abaababa$ ” (each word in the rule sequence is a concatenation of the previous two). We can use the corecursive function  $f\alpha\beta = \alpha(f(\alpha\beta)\alpha)$  to generate such sequences.

$$f\ a\ b \rightsquigarrow a(f(ab)\ a) \rightsquigarrow (aab)(f(aba)(ab)) \rightsquigarrow (aababa)(f(abaab)(aba))$$

We can use a standard method [22] to represent string rewriting systems as first-order term rewriting systems. In this case, the corresponding rules would be  $A\ x \rightarrow_a A\ (B\ x)$  and  $B\ x \rightarrow_b A\ x$ . The reduction would begin with  $A\ x$ . The Leibniz representation for this rewrite system is the following:

$$\begin{aligned} \Delta &= A : * \Rightarrow *, B : * \Rightarrow * \\ \Gamma &= \kappa_a : \forall p. \forall x. p\ (A\ (B\ x)) \Rightarrow p\ (A\ x), \kappa_b : \forall p. \forall x. p\ (A\ x) \Rightarrow p\ (B\ x) \end{aligned}$$

To represent the rewriting sequence above, we need to give a type to the function  $f$  such that  $\Gamma \vdash f\ \kappa_a\ \kappa_b : A\ x$ . The most intuitive type we can assign to the corecursive function  $f\alpha\beta = \alpha(f(\alpha\beta)\alpha)$  is the following:

$$(I) \ \forall x. (\forall p_2. \forall y_2. p_2\ (A\ (B\ y_2)) \Rightarrow p_2\ (A\ y_2)) \Rightarrow (\forall p_1. \forall y_1. p_1\ (A\ y_1) \Rightarrow p_1\ (B\ y_1)) \Rightarrow A\ x$$

Then we would have  $\Gamma \vdash f\ x\ \kappa_a\ \kappa_b : A\ x$ . Unfortunately this will not be type checked by RSM (the resolution will fail). We need to perform abstraction on type (I), here we abstract the function symbol  $B$  to a functional variable  $b : * \Rightarrow *$ , and  $A$  to a functional variable  $a : * \Rightarrow *$ , obtaining the following type for  $f$ .

$$(II) \ T \equiv \forall a. \forall b. \forall x. \underbrace{(\forall p. \forall y. p\ (a\ (b\ y)) \Rightarrow p\ (a\ y)) \Rightarrow (\forall p. \forall y. p\ (a\ y) \Rightarrow p\ (b\ y))}_{T'} \Rightarrow a\ x$$

Note that the quantified variable  $b$  in (II) is an existential variable. If  $f$  is typable with (II), then we know that  $\Gamma \vdash f\ A\ B\ x\ \kappa_a\ \kappa_b : A\ x$ , which encodes the nonterminating reduction starting from  $A\ x$ . But RSM will fail again in this case, due to the appearance of the existential variable  $b$ .

Ideally, the best way to deal with existential variables is by unification, we would need to replace rule (1) in RSM with the following:

$$\begin{aligned} \{(\Gamma, (\kappa|\alpha)\ e_1 \dots e_n, A), \Phi\} &\longrightarrow_a \{(\sigma\Gamma, e_1, \sigma T_1), \dots, (\sigma\Gamma, e_1, \sigma T_n), \sigma\Phi\} \text{ if} \\ &\kappa|\alpha : \forall \underline{x}. T_1, \dots, T_n \Rightarrow B \in \Gamma \text{ with } B \sim_\sigma A \end{aligned}$$

Here  $B \sim_\sigma A$  means  $A$  and  $B$  are second-orderly unifiable by  $\sigma$ . And  $\sigma\Gamma, \sigma\Phi$  means applying the substitution  $\sigma$  to all the types in  $\Gamma, \Phi$ . But second-order unification is not decidable and we need a finite set of unifiers. Thus we replace  $B \sim_\sigma A$  with  $B \mapsto_\sigma A$ .

► **Definition 33** (Existential RSM (ERSM)).

We replace (1) in Definition 30 to the following (Keeping rules (2), (3), (4) unchanged):

$$(1') \{(\Gamma, (\kappa|\alpha) e_1 \dots e_n, A), \Phi\} \longrightarrow_a \{(\sigma\Gamma, e_1, \sigma T_1), \dots, (\sigma\Gamma, e_n, \sigma T_n), \sigma\Phi\}$$

if  $\kappa|\alpha : \forall x_1 \dots \forall x_m. T_1, \dots, T_n \Rightarrow B \in \Gamma$  with  $B \mapsto_\sigma A$ .

Note that the formula  $\forall x_1 \dots \forall x_m. T_1, \dots, T_n \Rightarrow B$  in rule (1') may contain existential variables. The idea of this change is that by reordering the  $(\Gamma, e, T)$  pairs, we give priority to resolve the pair  $(\Gamma, e, T)$  where the head of  $T$  does not contain any existential variables. If the  $A$  in (1') does not contain existential variables, we can use rule (1') to eliminate the existential variables in  $\forall x_1 \dots \forall x_m. T_1, \dots, T_n \Rightarrow B$ . This extension allows us to avoid using the undecidable second-order unification, and it is good enough to handle all of our examples involving existential variables<sup>5</sup>.

With the Definition 33, we can obtain the following successful ERSM reduction, where  $\mu f. \lambda \alpha. \lambda \beta. \alpha (f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha')$  is the long form of  $f \alpha \beta = \alpha (f(\alpha \cdot \beta) \alpha)$ .

$$\begin{aligned} & \{(\Gamma, \mu f. \lambda \alpha. \lambda \beta. \alpha (f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha')), T\} \longrightarrow_c \\ & \{([\Gamma, f : T], \lambda \alpha. \lambda \beta. \alpha (f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha')), T\} \longrightarrow_\forall \\ & \{([\Gamma, f : T], \lambda \alpha. \lambda \beta. \alpha (f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha')), [a_1/a, b_1/b, x_1/x]T'\} \longrightarrow_i \\ & \{(\Gamma', \alpha (f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha')), a_1 x_1\} \longrightarrow_a \\ & \{(\Gamma', f (\lambda \alpha'. (\alpha (\beta \alpha')))) (\lambda \alpha'. \alpha \alpha'), a_1 (b_1 x_1)\} \longrightarrow_a \\ & \{(\Gamma', \lambda \alpha'. \alpha (\beta \alpha'), \forall p. \forall y. p (a_1 (b_1 (b_2 y))) \Rightarrow p (a_1 (b_1 y))), \Phi \equiv \\ & (\Gamma', \lambda \alpha'. \alpha \alpha', (\forall p. \forall y. p (a_1 (b_1 y)) \Rightarrow p (b_2 y)))\} \longrightarrow_\forall \\ & \{(\Gamma', \lambda \alpha'. \alpha (\beta \alpha'), p_2 (a_1 (b_1 (b_2 y_2))) \Rightarrow p_2 (a_1 (b_1 y_2))), \Phi\} \longrightarrow_i \\ & \{([\Gamma', \alpha' : p_2 (a_1 (b_1 (b_2 y_2)))]], \alpha (\beta \alpha'), p_2 (a_1 (b_1 y_2))), \Phi\} \longrightarrow_a \\ & \{([\Gamma', \alpha' : p_2 (a_1 (b_1 (b_2 y_2)))]], \beta \alpha', p_2 (a_1 (b_1 (b_1 y_2)))), \Phi\} \longrightarrow_a \\ & \{([\Gamma', \alpha' : p_2 (a_1 (b_1 (b_2 y_2)))]], \alpha', p_2 (a_1 (b_1 (a_1 y_2)))), \Phi\} \longrightarrow_a \\ & [(\lambda y. a_1 y) / b_2] \Phi \equiv \{(\Gamma', \lambda \alpha'. \alpha \alpha', \forall p. \forall y. p (a_1 (b_1 y)) \Rightarrow p (a_1 y))\} \longrightarrow_\forall \\ & \{(\Gamma', \lambda \alpha'. \alpha \alpha', p_3 (a_1 (b_1 y_3)) \Rightarrow p_3 (a_1 y_3))\} \longrightarrow_i \\ & \{([\Gamma', \alpha' : p_3 (a_1 (b_1 y_3))], \alpha \alpha', p_3 (a_1 y_3))\} \longrightarrow_a \\ & \{([\Gamma', \alpha' : p_3 (a_1 (b_1 y_3))], \alpha', p_3 (a_1 (b_1 y_3)))\} \longrightarrow_a \emptyset \end{aligned}$$

Note that  $\Gamma' = \Gamma, f : T, \alpha : \forall p. \forall y. p (a_1 (b_1 y)) \Rightarrow p (a_1 y), \beta : \forall p. \forall y. p (a_1 y) \Rightarrow p (b_1 y)$ . At the second  $\longrightarrow_a$ -step, by second-order matching, variable  $a$  is instantiated with  $\lambda y. a_1 (b_1 y)$  for the type of  $f$  and the existential variable  $b$  is instantiated with fresh variable  $b_2$ . At the fifth  $\longrightarrow_a$ -step, the existential variable  $b_2$  is instantiated with  $\lambda y. a_1 y$ , and there is a substitution for  $b_2$  applying to  $\Phi$ . But RSM will not perform this substitution, as a result, RSM cannot resolve  $\Phi$  to  $\emptyset$ .

## 7 Conclusion and Future Work

We present a novel method to represent nonterminating reductions in  $\mathbf{F}_2^\mu$ , where the rewrite rules and first-order terms are modeled by types, and the nonterminations are modeled by

<sup>5</sup> There is a well-known scope problem [7, Section 5], we show how to solve it for ERSM and prove the soundness of ERSM in Appendix I.

the hereditary head normalizing evidence. We prove that the hereditary head normalizing evidence for a first-order term is faithful, i.e. it represents a nonterminating reduction. We also prove the hereditary head normalization property for  $\mathbf{F}_2^\mu$  is decidable. To ease the representation process, we develop a type checking algorithm based on second-order matching, where fully annotated evidence can be generated from Curry-style evidence with only top-level type annotations.

**Future work.** We would like to investigate the nonterminating reductions that are currently outside the scope of  $\mathbf{F}_2^\mu$  and study the expressivity of  $\mathbf{F}_2^\mu$  in terms of representing nonterminations. The RSM/ERSM type checking algorithm is not very flexible. For example the Curry style evidence currently has to be in long form. We plan to relax this restriction.

## Acknowledgement

I would like to thank Tom Schrijvers for coming up with Example 28 and showing me a solution in Haskell using type family (See Fu et. al. [10]), at a time when I thought this whole thing is impossible. I also like to thank Ekaterina Komendantskaya for many helpful discussions, which leads me to consider the automation aspect, eventually I discover that quantification over higher-order variables leads to another solution for Example 28 without using type family, hence this paper. Reviewer 1 from FSCD 2016 discovered an error in an earlier version of the paper, which leads to a more rigid formulation of  $\mathbf{F}_2^\mu$ . Reviewer A from POPL 2017 suggests a possible simplification of productivity checking by mapping  $\mathbf{F}_2^\mu$  to  $\lambda$ -Y, which I carried out in this paper, and it greatly simplifies and strengthens the paper. Leibniz representation in this paper is inspired by Stump and Schürmann [20]’s treatment on rewriting and Girard’s recent criticism about Leibniz equality<sup>6</sup>. I would also like to thank the School of Computing at University of Dundee, and my mother Chen Xingzhen for generously providing a working space for me when I was in transitions between Postdocs.

---

## References

- 1 Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and computation*, 139(2):154–233, 1997.
- 2 F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.
- 3 H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
- 4 C. Böhm. Alcune proprietà delle forme  $\beta$ - $\eta$ -normali nel  $\lambda$ -K-calcolo. *IAC Pubbl*, 1968.
- 5 C. Broadbent, A. Carayol, L. Ong, and O. Serre. Recursion schemes and logical reflection. In *Twenty-Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 120–129, 2010.
- 6 N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 1987.
- 7 G. Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, 2:1009–1062, 2001.
- 8 F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Automated Reasoning*, pages 225–240. Springer, 2012.
- 9 J. Endrullis and H. Zantema. Proving non-termination by finite automata. In *26th International Conference on Rewriting Techniques and Applications, RTA*, 2015.
- 10 P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond. Proof relevant corecursive resolution. In *Functional and Logic Programming*. Springer, 2016.

---

<sup>6</sup> J.Y. Girard, *Transcendental syntax III: equality*



- 11 J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- 12 W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.
- 13 C. Grellois. *Semantics of linear logic and higher-order model-checking*. PhD thesis, Université Denis Diderot Paris 7, 2016.
- 14 G. Huet. *Resolution d'équations dans des langages d'ordre 1,2,..., omega*. PhD thesis, Université de Paris VII, 1976.
- 15 D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1):125–157, 1991.
- 16 U. Nilsson and J. Małuszyński. *Logic, programming and Prolog*. Wiley Chichester, 1990.
- 17 C. Raffalli. Data types, infinity and equality in system AF2. In *Computer Science Logic*, pages 280–294. Springer, 1994.
- 18 O. Serre. Playing with trees and logic. *Mémoire d'Habilitation*, 2015.
- 19 R. Statman. On the  $\lambda Y$  calculus. *Ann. Pure Appl. Logic*, 130(1-3):325–337, 2004.
- 20 A. Stump and C. Schürmann. Logical semantics for the rewriting calculus. *Electronic Notes in Theoretical Computer Science*, 125(2):149–164, 2005.
- 21 M. Tatsuta. Types for hereditary head normalizing terms. In *Functional and Logic Programming*, pages 195–209. Springer, 2008.
- 22 Terese. *Term rewriting systems*. Cambridge University Press, 2003.
- 23 P. Urzyczyn. Type reconstruction in fomega. *Mathematical Structures in Computer Science*, 7(4):329–358, 1997.
- 24 J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156, 1999.
- 25 S.S. Yu and Y.-K. Zhao. Properties of fibonacci languages. *Discrete Mathematics*, 224(1):215–223, 2000.
- 26 H. Zantema and A. Geser. *Non-looping rewriting*. Universiteit Utrecht, Faculty of Mathematics & Computer Science, 1996.

## A

 Proof of Theorem 6

► **Theorem 34.** *If  $\Delta \vdash T : k$  and  $T \rightarrow_o T'$ , then  $\text{FV}(T) = \text{FV}(T')$  and  $\Delta \vdash T' : k$*

**Proof.** By induction on the derivation of  $\Delta \vdash T : k$ .

**Case.**

$$\frac{(x|F : K) \in \Delta}{\Delta \vdash x|F : K}$$

Obvious.

**Case.**

$$\frac{\Delta, x : * \vdash T : K \quad x \in \text{FV}(T)}{\Delta \vdash \lambda x.T : * \Rightarrow K}$$

We have  $T \rightarrow_o T'$ . By IH, we have  $\Delta, x : * \vdash T' : K$  and  $\text{FV}(T) = \text{FV}(T')$ . Thus  $x \in \text{FV}(T')$ . So  $\Delta \vdash \lambda x.T' : * \Rightarrow K$ .

**Case.**

$$\frac{\Delta \vdash T_1 : * \quad \Delta \vdash \lambda x.T_2 : * \Rightarrow K}{\Delta \vdash (\lambda x.T_2) T_1 : K}$$

We have  $(\lambda x.T_2) T_1 \rightarrow_o [T_1/x]T_2$ . Since  $\Delta \vdash \lambda x.T_2 : * \Rightarrow K$ , by inversion we know that  $\Delta, x : * \vdash T_2 : K$  and  $x \in \text{FV}(T_2)$ . So  $\text{FV}((\lambda x.T_2) T_1) = \text{FV}([T_1/x]T_2)$  and  $\Delta \vdash [T_1/x]T_2 : K$ .

**Case.**

$$\frac{\Delta, x : K \vdash T : o|*}{\Delta \vdash \forall x.T : o}$$

Suppose  $\forall x.T \rightarrow_o \forall x.T'$  by  $T \rightarrow_o T'$ . By IH,  $\Delta, x : K \vdash T' : o|*$  and  $\text{FV}(T) = \text{FV}(T')$ . Thus  $\Delta \vdash \forall x.T' : o$  and  $\text{FV}(\forall x.T) = \text{FV}(\forall x.T')$ .

All the other cases are similar. ◀

## B

 Proof of Theorem 8

► **Theorem 35.**

1. *If  $\Delta \vdash T : o$ , then  $T$  is of the form  $\forall x.T'$  or  $T_1 \Rightarrow T_2$ .*
2. *If  $\Delta \vdash T : *^n \Rightarrow *$ , then the normal form of  $T$  is second-order.*

**Proof.** (1) Obvious.

(2). By induction on the derivation of  $\Delta \vdash T : *^n \Rightarrow *$ .

**Case.**

$$\frac{(x|F : K) \in \Delta}{\Delta \vdash x|F : K}$$

Obvious.

**Case.**

$$\frac{\Delta \vdash T_1 : * \quad \Delta \vdash T_2 : * \Rightarrow K}{\Delta \vdash T_2 T_1 : K}$$

We need to show the normal form of  $T_2 T_1$  is second-order. By IH, we know the normal form of  $T_1, T_2$  are second-order, moreover,  $T_1$  is flat since  $\Delta \vdash T_1 : *$ . Suppose  $T_2 \equiv F$  or  $T_2 \equiv x$ , then by definition we know  $T_2 T_1$  is second-order. Suppose  $T_2 \equiv \lambda x.T'$ , where  $x \in \text{FV}(T')$  and  $T'$  is second-order. Then  $(\lambda x.T') T_1 \rightarrow_o [T_1/x]T'$  and  $[T_1/x]T'$  is second-order.

**Case.**

$$\frac{\Delta, x : * \vdash T : K \quad x \in \text{FV}(T)}{\Delta \vdash \lambda x.T : * \Rightarrow K}$$

Let  $[T]$  be the normal form of  $T$ . By IH, we know that  $[T]$  is second-order. By Theorem 6, we know that  $x \in \text{FV}([T])$ . Thus  $\lambda x.[T]$  is second-order.  $\blacktriangleleft$

## C Proof of Theorem 10

► **Theorem 36.**  $\rightsquigarrow_{\beta\mu\tau o}$  and  $\rightsquigarrow_{h\tau o}$  are confluent, and  $\rightsquigarrow_\tau$  is strongly normalizing.

**Proof.** Note that  $\rightsquigarrow_\tau$  commutes with  $\rightsquigarrow_o$ ,  $\rightsquigarrow_h$  and  $\rightsquigarrow_{\beta\mu}$ . Also  $\rightsquigarrow_o$  commutes with  $\rightsquigarrow_h$  and  $\rightsquigarrow_{\beta\mu}$ . Thus it is enough to show that  $\rightsquigarrow_h$  and  $\rightsquigarrow_{\beta\mu}$  are confluent. For  $\rightsquigarrow_h$ , we just need to check  $\mathcal{H}_1[(\lambda x.(\mathcal{H}_2[\mu\alpha.e'])) e]$ , as it is the only critical pair. We know that:

$$\begin{aligned} \mathcal{H}_1[(\lambda x.(\mathcal{H}_2[\mu\beta.e'])) e] &\rightsquigarrow_h \mathcal{H}_1[(e/\alpha)\mathcal{H}_2][\mu\beta.[e/\alpha]e'] \rightsquigarrow_h \mathcal{H}_1[(e/\alpha)\mathcal{H}_2][[\mu\beta.[e/\alpha]e']/\beta][e/\alpha]e'] \\ \mathcal{H}_1[(\lambda x.(\mathcal{H}_2[\mu\beta.e'])) e] &\rightsquigarrow_h \mathcal{H}_1[(\lambda x.\mathcal{H}_2[[\mu\beta.e']/\beta]e')] e \rightsquigarrow_h \mathcal{H}_1[(e/\alpha)\mathcal{H}_2][[\mu\beta.[e/\alpha]e']/\beta][e/\alpha]e'] \end{aligned}$$

Thus  $\rightsquigarrow_h$  is confluent. For the confluence of  $\rightsquigarrow_{\beta\mu}$ , we refer to the existing literature (e.g. [1, §7.1]). Finally,  $\rightsquigarrow_\tau$  is strongly normalizing because the number of  $\rightsquigarrow_\tau$ -redex is strictly decreasing.  $\blacktriangleleft$

## D Proof of Theorem 13

► **Theorem 37 (Inversion).**

1. If  $\Gamma \vdash x : T$ , then there exists  $(x : T') \in \Gamma$  and  $T \leftrightarrow_o^* T'$ .
2. If  $\Gamma \vdash \kappa : T$ , then there exists  $(\kappa : T') \in \Gamma$  and  $T \leftrightarrow_o^* T'$ .
3. If  $\Gamma \vdash \lambda\alpha.e : T$ , then  $\Gamma, \alpha : T_1 \vdash e : T_2$  and  $T_1 \Rightarrow T_2 \leftrightarrow_o^* T$ .
4. If  $\Gamma \vdash e e' : T$ , then  $\Gamma \vdash e : T_1 \Rightarrow T_2$ ,  $\Gamma \vdash e' : T_1$  and  $T_2 \leftrightarrow_o^* T$ .
5. If  $\Gamma \vdash \lambda x.e : T$ , then  $\Gamma \vdash e : T'$ ,  $x \notin \text{FV}(\Gamma)$  and  $\forall x.T' \leftrightarrow_o^* T$ .
6. If  $\Gamma \vdash e T_1 : T$ , then  $\Gamma \vdash e : \forall x.T'$  and  $[T_1/x]T' \leftrightarrow_o^* T$ .
7. If  $\Gamma \vdash \mu\alpha.e : T$ , then  $\Gamma, \alpha : T' \vdash e : T'$  and  $T' \leftrightarrow_o^* T$ .

**Proof.** By induction on derivation.  $\blacktriangleleft$

► **Lemma 38.**

1.  $\Gamma, \alpha : T \vdash e : T'$  and  $\Gamma \vdash e' : T$ , then  $\Gamma \vdash [e'/\alpha]e : T'$ .
2.  $\Gamma \vdash e : T'$ , then  $[T_1/x]\Gamma \vdash [T_1/x]e : [T_1/x]T'$ .

**Proof.** By induction on the derivation. ◀

► **Theorem 39.** *If  $\Gamma \vdash e : T$  and  $e \rightsquigarrow_{h\tau o} e'$ , then  $\Gamma \vdash e' : T$ .*

**Proof.** By induction on the derivation of  $\Gamma \vdash e : T$ .

**Case.**

$$\frac{\Gamma, \alpha : T \vdash e : T}{\Gamma \vdash \mu\alpha.e : T} \text{ (MU)}$$

We know  $\mu\alpha.e \rightsquigarrow_h [\mu\alpha.e/\alpha]e$ . By lemma 38 (1), we know that  $\Gamma \vdash [\mu\alpha.e/\alpha]e : T$ .

**Case.**

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash \lambda\alpha.e : T' \Rightarrow T}{\Gamma \vdash (\lambda\alpha.e) e_1 : T} \text{ (APP)}$$

Suppose  $(\lambda\alpha.e) e_1 \rightsquigarrow_h [e_1/\alpha]e$ . By Theorem 37 (4), we have  $\Gamma, \alpha : T_1 \vdash e : T_2$  and  $T_1 \Rightarrow T_2 \leftrightarrow_o^* T' \Rightarrow T$ . Since  $\rightarrow_o$  is confluent, we have  $T_1 \leftrightarrow_o^* T'$  and  $T_2 \leftrightarrow_o^* T$ . Thus  $\Gamma \vdash e_1 : T_1$ . By Lemma 38 (1), we know  $\Gamma \vdash [e_1/\alpha]e : T_2$ . Thus  $\Gamma \vdash [e_1/\alpha]e : T$ .

**Case.**

$$\frac{\Gamma \vdash \lambda x.e : \forall x : K.T}{\Gamma \vdash (\lambda x.e) T' : [T'/x]T} \text{ (INST)}$$

Suppose that  $(\lambda x.e) T' \rightsquigarrow_\tau [T'/x]e$ . By Theorem 37 (5), we have  $\Gamma \vdash e : T_1$ ,  $x \notin \text{FV}(\Gamma)$  and  $\forall x.T_1 \leftrightarrow_o^* \forall x.T$ . By Lemma 38 (2), we have  $\Gamma \vdash [T'/x]e : [T'/x]T_1$ . Since  $\forall x.T_1 \leftrightarrow_o^* \forall x.T$  implies  $[T'/x]T_1 \leftrightarrow_o^* [T'/x]T$ , we have  $\Gamma \vdash [T'/x]e : [T'/x]T$ .

Suppose that  $(\lambda x.e) T' \rightsquigarrow_o (\lambda x.e) T''$  with  $T' \rightarrow_o T''$ . So by APP rule, we have  $\Gamma \vdash (\lambda x.e) T'' : [T''/x]T$ . By CONV rule, we have  $\Gamma \vdash (\lambda x.e) T'' : [T'/x]T$ .

For all the other cases are easy. ◀

## E Proof of Theorem 24

► **Lemma 40.** *Suppose  $\Gamma_{\mathcal{R}} \vdash e : t$  for some first-order term  $t$  and  $e$  is head normalizing. We have  $e \rightsquigarrow_{h\tau}^* \kappa (\lambda x.C[x, \dots, x]) t_1 \dots t_n e'$  for some  $\kappa : \forall p. \forall \underline{x}. p \ r \Rightarrow p \ l \in \Gamma_{\mathcal{R}}$ . Furthermore, we have  $\Gamma_{\mathcal{R}} \vdash e' : C[\sigma r, \dots, \sigma r]$  and  $C[\sigma l, \dots, \sigma l] = t$ , where  $\text{codom}(\sigma) = \{t_1, \dots, t_n\}$  and  $\text{dom}(\sigma) = \text{FV}(l)$ .*

**Proof.** Since  $e$  is head normalizing and  $\Gamma_{\mathcal{R}} \vdash e : t$ , its head normal form must be of the  $\kappa T T_1 \dots T_n e'$  for some  $\kappa : \forall p. \forall \underline{x}. p \ r \Rightarrow p \ l \in \Gamma_{\mathcal{R}}$ . By subject reduction (Theorem 6, Theorem 13), we have  $\Gamma_{\mathcal{R}} \vdash \kappa T T_1 \dots T_n e' : t$ . By inversion Theorem 12 (1) on  $\Gamma_{\mathcal{R}} \vdash \kappa T T_1 \dots T_n e' : t$ , we know that  $\Gamma_{\mathcal{R}} \vdash \kappa T T_1 \dots T_n : T'_1 \Rightarrow T'_2$ ,  $\Gamma_{\mathcal{R}} \vdash e' : T'_1$  and  $T'_2 \leftrightarrow_o t$ . By inversion Theorem 12 (2) on  $\Gamma_{\mathcal{R}} \vdash \kappa T T_1 \dots T_n : T'_1 \Rightarrow T'_2$ , we have  $\sigma(p \ r) \Rightarrow \sigma(p \ l) \leftrightarrow_o^* T'_1 \Rightarrow T'_2$ , where  $\sigma = [T/p, T_1/x_1, \dots, T_n/x_n]$ . Since we are working with well-kinded types, we know that  $\Gamma_{\mathcal{R}} \vdash T : * \Rightarrow *$  and  $\Gamma_{\mathcal{R}} \vdash T_i : *$  for all  $i$ . By Theorem 8, we know  $T = \lambda x.C[x, \dots, x]$  and  $T_i$  is flat for all  $i$ . By confluence of  $\leftrightarrow_o$ , we have  $\sigma(p \ r) \leftrightarrow_o^* T'_1$  and  $\sigma(p \ l) \leftrightarrow_o^* T'_2 \leftrightarrow_o^* t$ . Thus  $\sigma(p \ l) \equiv [T/p, T_1/x_1, \dots, T_n/x_n](p \ l) \equiv (\lambda x.C[x, \dots, x]) (\sigma l) \rightarrow_o^* t$ . So  $C[\sigma l, \dots, \sigma l] = t$ . Since  $\sigma(p \ r) \leftrightarrow_o^* T'_1$ , we have  $\Gamma_{\mathcal{R}} \vdash e' : C[\sigma r, \dots, \sigma r]$ . ◀

## 

 F Mapping  $\mathbf{F}_2^\mu$  to  $\lambda\text{-Y}$ 

► **Definition 41** ( $\lambda\text{-Y}$  calculus).

$\lambda\text{-Y terms } e ::= \alpha \mid \kappa \mid \lambda\alpha.e \mid e e' \mid \mu\alpha.e$

$\lambda\text{-Y types } T ::= B \mid T \Rightarrow T'$

$\lambda\text{-Y environment } \Gamma ::= \cdot \mid \alpha : T, \Gamma \mid \kappa : T$

Note that  $B$  denotes a constant type in  $\lambda\text{-Y}$ .

► **Definition 42** (Typing of  $\lambda\text{-Y}$ ).

$$\frac{(\alpha|\kappa : T) \in \Gamma}{\Gamma \vdash \alpha|\kappa : T} \quad \frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T' \Rightarrow T}{\Gamma \vdash e_2 e_1 : T} \text{ (APP)}$$

$$\frac{\Gamma, \alpha : T' \vdash e : T}{\Gamma \vdash \lambda\alpha.e : T' \Rightarrow T} \text{ (LAM)} \quad \frac{\Gamma, \alpha : T \vdash e : T}{\Gamma \vdash \mu\alpha.e : T} \text{ (MU)}$$

► **Definition 43.** We define a function  $\theta$  that maps  $\mathbf{F}_2^\mu$  types to  $\lambda\text{-Y}$  types.

$$\theta(F) = B$$

$$\theta(x) = B$$

$$\theta(\lambda x.T) = \theta(T)$$

$$\theta(T T') = \theta(T)$$

$$\theta(T \Rightarrow T') = \theta(T) \Rightarrow \theta(T')$$

$$\theta(\forall x.T) = \theta(T)$$

► **Lemma 44.** If  $\Delta \vdash T : K$  in  $\mathbf{F}_2^\mu$ , then  $\theta(T) = B$ .

**Proof.** By induction on the derivation of  $\Delta \vdash T : K$ . ◀

► **Lemma 45.** If  $\Delta \vdash T' : K$  in  $\mathbf{F}_2^\mu$ , then  $\theta([T'/x]T) \equiv \theta(T)$  for any  $T$  in  $\mathbf{F}_2^\mu$ .

**Proof.** Using Lemma 44 and induction on the structure of  $T$ . ◀

► **Lemma 46.** If  $T_1 \leftrightarrow_o^* T_2$  and  $\Delta \vdash T_1|T_2 : k$  in  $\mathbf{F}_2^\mu$ , then  $\theta(T_1) \equiv \theta(T_2)$ .

**Proof.** By induction on the derivation of  $T_1 \leftrightarrow_o^* T_2$ . ◀

► **Definition 47.**

$$\theta(\cdot) = \cdot$$

$$\theta(\Gamma, \alpha : T) = \theta(\Gamma), \alpha : \theta(T)$$

$$\theta(\Gamma, \kappa : T) = \theta(\Gamma), \kappa : \theta(T)$$

► **Theorem 48.** If  $\Gamma \vdash e : T$  and  $\Delta \vdash T : *|_o$  in  $\mathbf{F}_2^\mu$ , then  $\theta(\Gamma) \vdash |e| : \theta(T)$  in  $\lambda\text{-Y}$ .

**Proof.** By induction on derivaton of  $\Gamma \vdash e : T$  in  $\mathbf{F}_2^\mu$ .

■ Case:

$$\frac{(\alpha|\kappa : T) \in \Gamma}{\Gamma \vdash \alpha|\kappa : T}$$

We just need to show  $\theta(\Gamma) \vdash \alpha|\kappa : \theta(T)$  in  $\lambda\text{-Y}$ , which we know is the case by definition of  $\theta(\Gamma)$ .

■ Case:

$$\frac{\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T' \Rightarrow T}{\Gamma \vdash e_2 e_1 : T} \text{ (APP)}$$

We need to show  $\theta(\Gamma) \vdash |e_2 e_1| : \theta(T)$  in  $\lambda\text{-Y}$ . By induction, we know that  $\theta(\Gamma) \vdash |e_1| : \theta(T')$  and  $\theta(\Gamma) \vdash |e_2| : \theta(T') \Rightarrow \theta(T)$  in  $\lambda\text{-Y}$ . Thus we have  $\theta(\Gamma) \vdash |e_2| |e_1| : \theta(T)$ .

■ Case:

$$\frac{\Gamma, \alpha : T' \vdash e : T}{\Gamma \vdash \lambda \alpha. e : T' \Rightarrow T} \text{ (LAM)}$$

We need to show  $\theta(\Gamma) \vdash \lambda \alpha. |e| : \theta(T') \Rightarrow \theta(T)$  in  $\lambda$ -Y. By induction, we know that  $\theta(\Gamma), \alpha : \theta(T') \vdash |e| : \theta(T)$  in  $\lambda$ -Y.

■ Case:

$$\frac{\Gamma, \alpha : T \vdash e : T}{\Gamma \vdash \mu \alpha. e : T} \text{ (MU)}$$

We need to show  $\theta(\Gamma) \vdash \mu \alpha. |e| : \theta(T)$  in  $\lambda$ -Y. By induction, we know that  $\theta(\Gamma), \alpha : \theta(T) \vdash |e| : \theta(T)$  in  $\lambda$ -Y.

■ Case:

$$\frac{\Gamma \vdash e : T \quad x \notin \text{FV}(\Gamma)}{\Gamma \vdash \lambda x. e : \forall x : K. T} \text{ (ABS)}$$

We need to show  $\theta(\Gamma) \vdash |e| : \theta(T)$  in  $\lambda$ -Y, which is the case by induction.

■ Case:

$$\frac{\Gamma \vdash e : \forall x : K. T}{\Gamma \vdash e : [T'/x]T} \text{ (INST)}$$

We need to show  $\theta(\Gamma) \vdash |e| : \theta([T'/x]T)$  in  $\lambda$ -Y. By induction, we know that  $\theta(\Gamma) \vdash |e| : \theta(T)$ . By Lemma 45, we know that  $\theta([T'/x]T) \equiv \theta(T)$ .

■ Case:

$$\frac{\Gamma \vdash e : T \quad T \leftrightarrow_o^* T'}{\Gamma \vdash e : T'} \text{ (CONV)}$$

We need to show  $\theta(\Gamma) \vdash |e| : \theta(T')$  in  $\lambda$ -Y. By induction, we know that  $\theta(\Gamma) \vdash |e| : \theta(T)$ . By Lemma 46, we know that  $\theta(T') \equiv \theta(T)$ .

◀

## G Proof of Theorem 31

► **Lemma 49.** *If  $\{(\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow^* \emptyset$ , then there exists an evidence  $e'_1, \dots, e'_n$  such that  $\Gamma_i \vdash e'_i : T_i$  and  $|e'_i| = e_i$  for all  $i$ .*

**Proof.** By induction on the length of  $\{(\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow^* \emptyset$ .

■ Case  $\{(\Gamma, \alpha|\kappa, A)\} \longrightarrow_a \emptyset$ .

In this case  $\alpha|\kappa : \forall \underline{x}. B \in \Gamma$  and  $B \mapsto_\sigma A$ . Since  $\forall \underline{x}. B$  does not contain existential variables, by INST, we have  $\Gamma \vdash (\alpha|\kappa) \underline{T} : A$ , where  $\{\underline{T}\} = \text{codom}(\sigma)$  and  $|(\alpha|\kappa) \underline{T}| = \alpha|\kappa$ .

■ Case

$\{(\Gamma, (\alpha|\kappa) e''_1 \dots e''_m, A), (\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow_a$   
 $\{(\Gamma, e''_1, \sigma T'_1), \dots, (\Gamma, e''_m, \sigma T'_m), (\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow^* \emptyset$ , where  $\kappa|\alpha : \forall \underline{x}. T'_1, \dots, T'_n \Rightarrow$   
 $B \in \Gamma$  with  $B \mapsto_\sigma A$ .

By IH, we know that  $\Gamma \vdash e'''_1 : \sigma T'_1, \dots, \Gamma \vdash e''_m : \sigma T'_m, \Gamma_1 \vdash e'_1 : T_1, \dots, \Gamma_n \vdash e'_n : T_n$  and  $|e'''_1| = e''_1, \dots, |e'''_m| = e''_m, |e'_1| = e_1, \dots, |e'_n| = e_n$ . Let  $\text{codom}(\sigma) = \underline{T}$ , since  $\forall \underline{x}. T'_1, \dots, T'_n \Rightarrow B$  does not contain existential variables, we have  $\Gamma \vdash (\alpha|\kappa) \underline{T} : \sigma T'_1, \dots, \sigma T'_n \Rightarrow \sigma B$ . Thus

- $\Gamma \vdash (\alpha|\kappa) \underline{T} e_1''' \dots e_m''' : \sigma B$ . By CONV, we have  $\Gamma \vdash (\alpha|\kappa) \underline{T} e_1''' \dots e_m''' : A$ . Moreover,  $|(\alpha|\kappa) \underline{T} e_1''' \dots e_m'''| = (\alpha|\kappa) |e_1'''| \dots |e_m'''| = (\alpha|\kappa) e_1'' \dots e_m''$ .
- Case  $\{(\Gamma, \lambda\alpha_1 \dots \lambda\alpha_n.e, T_1, \dots, T_n \Rightarrow A), (\Gamma_1, e_1, T_1), \dots, (\Gamma_l, e_l, T_l)\} \longrightarrow_i \{([\Gamma, \alpha_1 : T_1, \dots, \alpha_n : T_n], e, A), (\Gamma_1, e_1, T_1), \dots, (\Gamma_l, e_l, T_l)\} \longrightarrow^* \emptyset$   
By IH, we have  $\Gamma, \alpha_1 : T_1, \dots, \alpha_n : T_n \vdash e' : A, \Gamma_1 \vdash e'_1 : T_1, \dots, \Gamma_l \vdash e'_l : T_l$  with  $|e'| = e, |e'_1| = e_1, \dots, |e'_l| = e_l$ . Thus by LAM rule, we have  $\Gamma \vdash \lambda\alpha_1 \dots \lambda\alpha_n.e' : T_1, \dots, T_n \Rightarrow A$  and  $|\lambda\alpha_1 \dots \lambda\alpha_n.e'| = \lambda\alpha_1 \dots \lambda\alpha_n.e$ .
  - Case  $\{(\Gamma, e, \forall x_1 \dots \forall x_m.T), (\Gamma_1, e_1, T_1), \dots, (\Gamma_l, e_l, T_l)\} \longrightarrow_\forall \{(\Gamma, e, T), (\Gamma_1, e_1, T_1), \dots, (\Gamma_l, e_l, T_l)\} \longrightarrow^* \emptyset$   
By IH, we have  $\Gamma \vdash e' : A, \Gamma_1 \vdash e'_1 : T_1, \dots, \Gamma_l \vdash e'_l : T_l$  with  $|e'| = e, |e'_1| = e_1, \dots, |e'_l| = e_l$ . Since  $\{x_1, \dots, x_m\} \cap \text{FV}(\Gamma) = \emptyset$ , by ABS rules, we have  $\Gamma \vdash \lambda x_1 \dots \lambda x_m.e' : \forall x_1 \dots \forall x_m.T$  and  $|\lambda x_1 \dots \lambda x_m.e'| = e$ .
  - Case  $\{(\Gamma, \mu\alpha.e, T), (\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow_c \{([\Gamma, \alpha : T], e, T), (\Gamma_1, e_1, T_1), \dots, (\Gamma_n, e_n, T_n)\} \longrightarrow^* \emptyset$   
By IH, we know that  $\Gamma, \alpha : T \vdash e' : T, \Gamma_1 \vdash e'_1 : T_1, \dots, \Gamma_n \vdash e'_n : T_n$  and  $|e'_i| = e_i$  for all  $i$ . By MU rule, we have  $\Gamma \vdash \mu\alpha.e' : T$ . Thus  $|\mu\alpha.e'| = \mu\alpha.e$ .

◀

## H Examples in the Paper

In this section we show how to represent nonterminations for all the examples in the paper using the prototype FCR (for Functional Certification of Rewriting), the prototype is available at <https://github.com/Fermat/FCR>. It tries to generate typable  $F_2^\mu$  evidence from the corecursive equations and the type declarations.

### H.1 Example in Section 5

The following is the input file for FCR.

```
A : forall p x y . p (D x (S y)) => p (D (S x) y)
B : forall p y . p (D (S y) Z) => p (D Z y)

g : forall d .
  (forall p x y . p (d x (S y)) => p (d (S x) y)) =>
  (forall p y . p (d (S y) Z) => p (d Z y)) =>
  d Z Z

g a1 a2 = a2 (a1 (g (\ v . a1 v) (\ v . a2 (a1 v))))

e : D Z Z
e = g (\ v . A v) (\ v . B v)
```

The capitalized words for FCR are intended to denote both type and evidence constant, unc capitalized words are intended to denote both type and evidence variables. In the definition of corecursive function `g`, “`\`” denotes the  $\lambda$  binder, its type declaration is discussed in the paper. FCR currently uses long normal form to make variable instantiation, so we have to use (I) instead of (II).

$$\begin{aligned} \text{(I)} \quad & g \ a1 \ a2 = a2 \ (a1 \ (g \ (\backslash v . \ a1 \ v) \ (\backslash v . \ a2 \ (a1 \ v)))) \\ \text{(II)} \quad & g \ a1 \ a2 = (a2 \ . \ a1) \ (g \ a1 \ (a2 \ . \ a1)) \end{aligned}$$



Evidence such as  $\mu f.\lambda a.e$  is represented as equation  $f\ a = e$ , so there is no explicit  $\mu$  binder in the input file. The corecursive evidence for  $D\ Z\ Z$  is  $e$ . The following is the output by the type checker.

```

rewrite rules
kinds
D : * => * => *
S : * => *
Z : *
axioms
A : forall p x y . p (D x (S y)) => p (D (S x) y)
B : forall p y . p (D (S y) Z) => p (D Z y)
proof declarations
g : forall d .
  (forall p x y . p (d x (S y)) => p (d (S x) y))
  =>
  (forall p y . p (d (S y) Z) => p (d Z y)) => d Z Z =
\ a1 a2 . a2 (a1 (g (\ v . a1 v) (\ v . a2 (a1 v))))
e : D Z Z =
g (\ v . A v) (\ v . B v)
lemmas
e : D Z Z =
  g (\ m1' m2' . D m1' m2')
    (\ p1' x2' y3' (v : p1' (D x2' (S y3')))) .
      A (\ m1' . p1' m1') x2' y3' v)
    (\ p7' y8' (v : p7' (D (S y8') Z)) . B (\ m1' . p7' m1') y8' v)
g : forall d .
  (forall p x y . p (d x (S y)) => p (d (S x) y))
  =>
  (forall p y . p (d (S y) Z) => p (d Z y)) => d Z Z =
\ d0'
  (a1 : forall p x y . p (d0' x (S y)) => p (d0' (S x) y))
  (a2 : forall p y . p (d0' (S y) Z) => p (d0' Z y)) .
  a2 (\ x1' . x1') Z
  (a1 (\ x1' . x1') Z Z
    (g (\ m1' m2' . d0' m1' (S m2'))
      (\ p7' x8' y9' (v : p7' (d0' x8' (S (S y9')))) .
        a1 (\ m1' . p7' m1') x8' (S y9') v)
      (\ p13' y14' (v : p13' (d0' (S y14') (S Z))) .
        a2 (\ m1' . p13' m1') (S y14')
          (a1 (\ m1' . p13' m1') (S y14') Z v))))))
steps
automated proof reconstruction success!

```

The `lemmas` section contains the annotated evidence. All variables generated by FCR are variables end with “'”. All lambda-bound evidence variables are annotated with the type information. This is needed for decidable proof checking, we do not need to annotate lambda-bound type variables. The annotated evidence generated by our type checker is checked by a separate  $F_2^\mu$  proof checker.

We can translated the input file into the following Haskell code, but it will not pass Haskell's type checker.

```
data D :: * -> * -> *
data S :: * -> *
data Z :: *
a :: forall p x y . p (D x (S y)) -> p (D (S x) y)
a = undefined
b :: forall p y . p (D (S y) Z) -> p (D Z y)
b = undefined
g :: forall d .
  (forall p x y . p (d x (S y)) -> p (d (S x) y)) ->
  (forall p y . p (d (S y) Z) -> p (d Z y)) ->
  d Z Z
g a1 a2 = a2 (a1 (g (\ v -> a1 v) (\ v -> a2 (a1 v))))

e :: D Z Z
e = g (\ v -> a v) (\ v -> b v)
```

## H.2 Example in Section 6

The following is the input file for FCR.

```
Ka : A x <= A (B x)
Kb : B x <= A x

g : forall a b x .
  (forall p y . p (a (b y)) => p (a y)) =>
  (forall p y . p (a y) => p (b y)) => a x

g a b = a (g (\ v . a (b v)) (\ v . a v))

h : A x
h = g (\ v . Ka v) Kb

step h 20
```

We use the alternative notation  $A\ x \leq A\ (B\ x)$  to represent the rewrite rule from  $A\ x$  to  $A\ (B\ x)$ , it will be translated to its Leibniz representation by FCR. And `step h 20` is a command telling FCR to output the 20th first-order term in the reduction `h` began with term `A x`. The following is the output information.

```
rewrite rules
Ka : A x <= A (B x)
Kb : B x <= A x
kinds
A : * => *
B : * => *
axioms
Ka : forall p x . p (A (B x)) => p (A x)
```

```

Kb : forall p x . p (A x) => p (B x)
proof declarations
g : forall a b x .
  (forall p y . p (a (b y)) => p (a y))
=>
  (forall p y . p (a y) => p (b y)) => a x =
\ a b . a (g (\ v . a (b v)) (\ v . a v))
h : A x =
g (\ v . Ka v) Kb
lemmas
h : A x =
  g (\ m1' . A m1') (\ m1' . B m1') x
  (\ p3' y4' (v : p3' (A (B y4')))) . Ka (\ m1' . p3' m1') y4' v)
  Kb
g : forall a b x .
  (forall p y . p (a (b y)) => p (a y))
=>
  (forall p y . p (a y) => p (b y)) => a x =
\ a0'
  b1'
  x2'
  (a : forall p y . p (a0' (b1' y)) => p (a0' y))
  (b : forall p y . p (a0' y) => p (b1' y)) .
  a (\ x1' . x1') x2'
  (g (\ m1' . a0' (b1' m1')) (\ m1' . a0' m1') x2'
    (\ p8' y9' (v : p8' (a0' (b1' (a0' y9')))) .
      a (\ m1' . p8' m1') (b1' y9')
        (b (\ m1' . p8' (a0' (b1' m1')) y9' v))
        (\ p14' y15' (v : p14' (a0' (b1' y15')) .
          a (\ m1' . p14' m1') y15' v))
    )
steps
step h 20
automated proof reconstruction success!
steps results
A (B (A (A (B (A (B (A (A (B (A (A (B x))))))))))))))

```

We can check that the term  $A (B (A (A (B (A (B (A (A (B (A (A (B x))))))))))))$  represents the string we obtain in the very end of the string reduction trace in Section 6. Note that this term is obtained directly from the unfolding of the reduction trace without invoking any term rewriting reduction.

## I Solving the Scope Problem in ERSM and the Soundness of ERSM

Due to lack of space, we did not explain nor discuss the soundness of ERSM in Section 6. In fact, the ERSM is not sound in its current form due to a subtle scope problem. We will show how to solve this soundness problem in this section. To explain the scope problem, let us consider the following two formulas.

(I)  $\text{forall } p \ x \ y . \ p \ (G \ (F \ Z \ x \ (S \ y)) \ (F \ x \ y \ (S \ (S \ Z)))) \Rightarrow p \ (F \ Z \ (S \ x) \ y)$

(II)  $\text{forall } p \ x \ y . \ p \ (qa \ (F \ Z \ x \ (S \ y))) \Rightarrow p \ (F \ Z \ (S \ x) \ y)$

It may appear that these two formulas are second-orderly unifiable if we instantiate  $qa$  in (II) to  $\lambda m . \ G \ m \ (F \ x \ y \ (S \ (S \ Z)))$ . But this instantiation assumes the variable  $x, y$  in  $\lambda m . \ G \ m \ (F \ x \ y \ (S \ (S \ Z)))$  can be automatically captured by the `forall` binder in (II), this is not a correct assumption. In fact (I) and (II) are not unifiable, this kind of problem is called *scope problem* by Dowek [7, Section 5].

The solution of the scope problem is conceptually simple, i.e. we just need to prevent the instantiation of the existential variables when there is such a scope problem. However, to implement this solution within the ERSM framework requires some efforts.

We works with *idempotent* substitution, i.e. for a substitution  $\sigma$ , we require that  $\sigma \cdot \sigma = \sigma$ . Idempotency is easy to check, due to the following property [2]:  $\sigma$  is idempotent iff  $\text{dom}(\sigma) \cap \text{FV}(\text{codom}(\sigma)) = \emptyset$ . This requirement is needed in order to prove the soundness theorem.

► **Definition 50.** Let  $L$  denote a list of variables. We define  $y \sqsubset_L x$  if  $L = L_1, y, L_2, x, L_3$  for some  $L_1, L_2, L_3$ . We define  $\text{scope}(L, \sigma)$  to be the conjunction of the following two predicates: (1)  $\forall x \in \text{dom}(\sigma) \cap L, \forall y \in \text{FV}(\sigma x), y \sqsubset_L x$ . (2)  $\forall x \in \text{dom}(\sigma) - L, \text{FV}(\sigma x) \cap L = \emptyset$ .

Let  $\Phi$  denotes a set of tuple  $(L, \Gamma, e, T)$ . We use  $\sigma L$  to denote  $L - \text{dom}(\sigma)$  and we use  $L + L'$  to mean appending  $L, L'$ .

► **Definition 51.**  $\boxed{\sigma\Gamma, \sigma\Phi}$

$$\begin{aligned} \sigma \cdot &= \cdot \\ \sigma[\alpha : T, \Gamma] &= \alpha : \sigma T, \sigma\Gamma \\ \sigma[\kappa : T, \Gamma] &= \kappa : \sigma T, \sigma\Gamma \\ \sigma\{\} &= \{\} \\ \sigma \{(L, \Gamma, e, T), \Phi\} &= \{(\sigma L, \sigma\Gamma, e, \sigma T), \sigma\Phi\}, \text{ where } \text{scope}(L, \sigma). \end{aligned}$$

Let  $S$  be a set of variables, we write  $\sigma/S = [t/x \mid x \in (\text{dom}(\sigma) - S)]$ .

► **Definition 52 (ERSM with Scope Check).**  $\boxed{(\Phi, \sigma) \longrightarrow (\Phi', \sigma')}$

1.  $(\{(L, \Gamma, (\kappa|\alpha) e_1 \dots e_n, A), \Phi\}, \sigma) \longrightarrow_a (\{(L', \sigma''\Gamma, e_1, \sigma'T_1), \dots, (L', \sigma''\Gamma, e_n, \sigma'T_n), \sigma''\Phi\}, \sigma'' \cdot \sigma)$  if  $\kappa|\alpha : \forall x_1 \dots \forall x_m. T_1, \dots, T_n \Rightarrow B \in \Gamma$  with  $B \mapsto_{\sigma'} A$ . Moreover,  $\sigma'' = \sigma' / \{x_1, \dots, x_m\}$ ,  $\text{scope}(L, \sigma'')$  and  $L' = \sigma''L + [x_i \mid x_i \notin \text{FV}(B), 1 \leq i \leq m]$ .
2.  $(\{(L, \Gamma, \lambda\alpha_1 \dots \lambda\alpha_n. e, T_1, \dots, T_n \Rightarrow A), \Phi\}, \sigma) \longrightarrow_i (\{(L, [\Gamma, \alpha_1 : T_1, \dots, \alpha_n : T_n], e, A), \Phi\}, \sigma)$ .
3.  $(\{(L, \Gamma, e, \forall x_1 \dots \forall x_n. T), \Phi\}, \sigma) \longrightarrow_v (\{([L, x_1, \dots, x_n], \Gamma, e, T), \Phi\}, \sigma)$ .
4.  $(\{(L, \Gamma, \mu\alpha. e, T), \Phi\}, \sigma) \longrightarrow_c (\{(L, [\Gamma, \alpha : T], e, T), \Phi\}, \sigma)$ .

We can see if we eliminate  $L$  and  $\text{scope}(L, \sigma)$ , we can obtain ERSM described in the paper.

► **Lemma 53.** If  $\Gamma \vdash e : T$ , then  $\sigma\Gamma \vdash \sigma e : \sigma T$ .

If  $S$  is a set of variables, we define  $\sigma S := \{\sigma x \mid x \in S\}$ . Moreover, we extend FV function to obtain all the free variables of a set of terms. Note that all the substitutions are *idempotent* and *disjoint*, i.e.  $\text{FV}(\text{codom}(\sigma)) \cap \text{dom}(\sigma) = \emptyset$  for any  $\sigma$  and  $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset$ , for any  $\sigma_1, \sigma_2$ .

► **Lemma 54 (Scope Check Composition).** Suppose  $\text{FV}(\text{codom}(\sigma_2)) \cap \text{dom}(\sigma_1) = \emptyset$ . If  $\text{Scope}(L, \sigma_1)$  and  $\text{Scope}(\sigma_1 L + L', \sigma_2)$  for some fresh  $L'$ , then  $\text{Scope}(L, \sigma_2 \cdot \sigma_1)$ .

**Proof.** ■ Case  $y \in \text{dom}(\sigma_2 \cdot \sigma_1) - L$ .

We need to show  $\text{FV}(\sigma_2 \sigma_1 y) \cap L = \emptyset$ , i.e.  $\text{FV}(\sigma_2(\text{FV}(\sigma_1 y))) \cap L = \emptyset$ . We know that  $\text{dom}(\sigma_2 \cdot \sigma_1) = \text{dom}(\sigma_2) \uplus \text{dom}(\sigma_1)$ . Suppose  $y \in \text{dom}(\sigma_1)$ , we know that  $\text{FV}(\sigma_1 y) \cap L = \emptyset$ . For any  $z \in \text{FV}(\sigma_1 y) \cap \text{dom}(\sigma_2)$ , we have  $\text{FV}(\sigma_2 z) \cap (\sigma_1 L + L') = \emptyset$ , which implies  $\text{FV}(\sigma_2 z) \cap L = \emptyset$ . For any  $z \in \text{FV}(\sigma_1 y) - \text{dom}(\sigma_2)$ , we have  $\text{FV}(\sigma_2 z) = \{z\}$ ,  $\{z\} \cap L = \emptyset$ . Suppose  $y \in \text{dom}(\sigma_2)$ , we need to show  $\text{FV}(\sigma_2 y) \cap L = \emptyset$ , this is the case since  $\text{FV}(\sigma_2 y) \cap (\sigma_1 L + L') = \emptyset$  and  $\text{FV}(\text{codom}(\sigma_2)) \cap \text{dom}(\sigma_1) = \emptyset$ .

■ Case.  $y \in \text{dom}(\sigma_2 \cdot \sigma_1) \cap L$ .

We need to show for any  $z \in \text{FV}(\sigma_2(\text{FV}(\sigma_1 y))) \cap L$ ,  $z \sqsubset_L y$ . Let  $x \in \text{FV}(\sigma_1 y)$ , we just need to show for any  $z \in \text{FV}(\sigma_2 x) \cap L$ ,  $z \sqsubset_L y$ . Suppose  $x \notin \text{dom}(\sigma_2)$ . Then  $\text{FV}(\sigma_2 x) = \{x\}$ . So  $x \sqsubset_L y$  if  $x \in L$ . Suppose  $x \in \text{dom}(\sigma_2) \cap L$ , we know that  $(\text{FV}(\sigma_2 x) \cap (\sigma_1 L + L')) \sqsubset_{\sigma_1 L + L'} x$ . Since  $z \in \text{FV}(\sigma_2 x) \cap L$  implies  $z \in \text{FV}(\sigma_2 x) \cap (\sigma_1 L + L')$ , we have  $z \sqsubset_{\sigma_1 L + L'} x \sqsubset_L y$ . Since  $x \notin L'$  and  $x \notin \text{dom}(\sigma_1)$ , we have  $z \sqsubset_L x \sqsubset_L y$ . Suppose  $x \in \text{dom}(\sigma_2) - L$ , then  $x \in \text{dom}(\sigma_2) - (\sigma_1 L + L')$ , thus  $\text{FV}(\sigma_2 x) \cap (\sigma_1 L + L') = \emptyset$ , which implies  $\text{FV}(\sigma_2 x) \cap L = \emptyset$ .

Suppose  $y \in \text{dom}(\sigma_2)$ , we just need to show for any  $z \in \text{FV}(\sigma_2 y) \cap L$ ,  $z \sqsubset_L y$ . Since  $z \notin \text{dom}(\sigma_1)$ , we have  $z \in \text{FV}(\sigma_2 y) \cap (\sigma_1 L + L')$ . Thus  $z \sqsubset_{\sigma_1 L + L'} y$ , which implies  $z \sqsubset_L y$ . ◀

► **Lemma 55** (Scope Invariant).

1. If  $(\{(L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow (\{(L'_1, \Gamma'_1, e'_1, T'_1), \dots, (L'_m, \Gamma'_m, e'_m, T'_m)\}, \sigma')$ , then  $\text{Scope}(L_i, \sigma')$  for all  $i$ .
2. If  $(\{(L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow^* (\{(L'_1, \Gamma'_1, e'_1, T'_1), \dots, (L'_m, \Gamma'_m, e'_m, T'_m)\}, \sigma')$ , then  $\text{Scope}(L_i, \sigma')$  for all  $i$ .

**Proof.** By Lemma 54 and induction. ◀

► **Lemma 56.** If  $(\{(L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow^* (\emptyset, \sigma' \cdot \sigma)$  for some  $\sigma'$ , then  $\sigma' \Gamma_i \vdash e'_i : \sigma' T_i$  and  $|e'_i| = e_i$  for all  $i$ .

**Proof.** By induction on the length of  $(\{(L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow^* (\sigma' \cdot \sigma, \emptyset)$ .

■ Case  $(\{(L, \Gamma, \alpha|\kappa, A)\}, \sigma) \longrightarrow_a (\emptyset, \sigma'' \cdot \sigma)$ .

In this case  $\alpha|\kappa : \forall \underline{x}. B \in \Gamma$ ,  $\sigma'' = \sigma' / \{\underline{x}\}$ ,  $\text{scope}(L, \sigma'')$  and  $B \mapsto_{\sigma'} A$ . By INST rule and the idempotentness of  $\sigma'$ , we have  $\sigma'' \Gamma \vdash (\alpha|\kappa) (\sigma' \underline{x}) : \sigma' B \equiv \sigma'' \sigma' B = \sigma'' A$ , where  $|(\alpha|\kappa) (\sigma' \underline{x})| = \alpha|\kappa$ .

■ Case  $(\{(L, \Gamma, (\alpha|\kappa) e''_1 \dots e''_m, A), (L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow_a (\{(L', \sigma' \Gamma, e'_1, \sigma_1 T'_1), \dots, (L', \sigma' \Gamma, e'_m, \sigma_1 T'_m), (\sigma' L_1, \sigma' \Gamma_1, e_1, \sigma' T_1), \dots, (\sigma' L_n, \sigma' \Gamma_n, e_n, \sigma' T_n)\}, \sigma') \longrightarrow^* (\emptyset, \sigma'' \cdot \sigma' \cdot \sigma)$ ,

where  $\kappa|\alpha : \forall \underline{x}. T'_1, \dots, T'_n \Rightarrow B \in \Gamma$  with  $B \mapsto_{\sigma_1} A$ ,  $\sigma' = \sigma_1 / \{\underline{x}\}$ ,  $\text{scope}(L, \sigma')$  and  $L' = \sigma' L + [x_i \mid x_i \notin \text{FV}(B), 1 \leq i \leq m]$ .

By IH, we know that  $\sigma'' \sigma' \Gamma' \vdash e'''_1 : \sigma'' \sigma_1 T'_1, \dots, \sigma'' \sigma' \Gamma' \vdash e'''_m : \sigma'' \sigma_1 T'_m, \sigma'' \sigma' \Gamma_1 \vdash e'_1 : \sigma'' \sigma' T_1, \dots, \sigma'' \sigma' \Gamma_n \vdash e'_n : \sigma'' \sigma' T_n$  and  $|e'''_1| = e'_1, \dots, |e'''_m| = e'_m, |e'_1| = e_1, \dots, |e'_n| = e_n$ . We have  $\sigma'' \sigma' \Gamma \vdash (\alpha|\kappa) (\sigma'' \sigma' \underline{x}) : \sigma'' \sigma_1 T'_1, \dots, \sigma'' \sigma_1 T'_n \Rightarrow \sigma'' \sigma_1 B$ . By CONV, APP and idempotentness, we have  $\sigma'' \sigma' \Gamma \vdash (\alpha|\kappa) (\sigma'' \sigma' \underline{x}) e'''_1 \dots e'''_m : \sigma'' \sigma_1 B = \sigma'' \sigma' A$ . Moreover,  $|(\alpha|\kappa) (\sigma'' \sigma' \underline{x}) e'''_1 \dots e'''_m| = (\alpha|\kappa) |e'''_1| \dots |e'''_m| = (\alpha|\kappa) e'_1 \dots e'_m$ .

■ Case  $(\{(L, \Gamma, \lambda \alpha_1 \dots \lambda \alpha_n. e, T_1, \dots, T_n \Rightarrow A), (L_1, \Gamma_1, e_1, T_1), \dots, (L_l, \Gamma_l, e_l, T_l)\}, \sigma) \longrightarrow_i (\{(L, [\Gamma, \alpha_1 : T_1, \dots, \alpha_n : T_n], e, A), (L_1, \Gamma_1, e_1, T_1), \dots, (L_l, \Gamma_l, e_l, T_l)\}, \sigma) \longrightarrow^* (\emptyset, \sigma' \cdot \sigma)$

- By IH, we have  $\sigma'\Gamma, \alpha_1 : \sigma'T_1, \dots, \alpha_n : \sigma'T_n \vdash e' : \sigma'A, \sigma'\Gamma_1 \vdash e'_1 : \sigma'T_1, \dots, \sigma'\Gamma_l \vdash e'_l : \sigma'T_l$  with  $|e'| = e, |e'_1| = e_1, \dots, |e'_l| = e_l$ . Thus by LAM rule, we have  $\sigma'\Gamma \vdash \lambda\alpha_1 \dots \lambda\alpha_n. e' : \sigma'T_1, \dots, \sigma'T_n \Rightarrow \sigma'A$  and  $|\lambda\alpha_1 \dots \lambda\alpha_n. e'| = \lambda\alpha_1 \dots \lambda\alpha_n. e$ .
- Case  $(\{(L, \Gamma, e, \forall x_1 \dots \forall x_m. T), (L_1, \Gamma_1, e_1, T_1), \dots, (L_l, \Gamma_l, e_l, T_l)\}, \sigma) \longrightarrow_{\forall} (\{([L, x_1, \dots, x_m], \Gamma, e, T), (L_1, \Gamma_1, e_1, T_1), \dots, (L_l, \Gamma_l, e_l, T_l)\}, \sigma) \longrightarrow^* (\emptyset, \sigma' \cdot \sigma)$   
By IH, we have  $\sigma'\Gamma \vdash e' : \sigma'T, \sigma'\Gamma_1 \vdash e'_1 : \sigma'T_1, \dots, \sigma'\Gamma_l \vdash e'_l : \sigma'T_l$  with  $|e'| = e, |e'_1| = e_1, \dots, |e'_l| = e_l$ . By Lemma 55 (2),  $\text{scope}([L, x_1, \dots, x_m], \sigma')$ . So  $\text{FV}(\text{codom}(\sigma')) \cap \{x_1, \dots, x_m\} = \emptyset$ . Thus by ABS rule, we have  $\sigma'\Gamma \vdash \lambda x_1 \dots \lambda x_m. e' : \forall x_1 \dots \forall x_m. \sigma'T = \sigma'(\forall x_1 \dots \forall x_m. T)$  and  $|\lambda x_1 \dots \lambda x_m. e'| = e$ .
  - Case  $(\{(L, \Gamma, \mu\alpha. e, T), (L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow_c (\{([L, \alpha : T], e, T), (L_1, \Gamma_1, e_1, T_1), \dots, (L_n, \Gamma_n, e_n, T_n)\}, \sigma) \longrightarrow^* (\emptyset, \sigma' \cdot \sigma)$   
By IH, we know that  $\sigma'\Gamma, \alpha : \sigma'T \vdash e' : \sigma'T, \sigma'\Gamma_1 \vdash e'_1 : \sigma'T_1, \dots, \sigma'\Gamma_n \vdash e'_n : \sigma'T_n$  and  $|e'_i| = e_i$  for all  $i$ . By MU rule, we have  $\sigma'\Gamma \vdash \mu\alpha. e' : \sigma'T$ . Thus  $|\mu\alpha. e'| = \mu\alpha. e$ .

► **Theorem 57** (Soundness of ERSM). *If  $(\{([], \Gamma, e, T)\}, \text{id}) \longrightarrow^* (\emptyset, \sigma)$  and  $\text{FV}(\Gamma) = \text{FV}(T) = \emptyset$ , then  $\Gamma \vdash e' : T$  and  $|e'| = e$ .*

**Proof.** By Lemma 56.

We now can understand the error message when we try to type check the following declarations in FCR.

```
K : forall p x y . p (G (F Z x (S y)) (F x y (S (S Z)))) => p (F Z (S x) y)

K2 : forall qa . (forall p x y . p (qa (F Z x (S y))) => p (F Z (S x) y)) => B

h : B
h = K2 (\ c . K c)
```

Note that type checking `h` will give a scope problem as (I) and (II) above does not unify. FCR will print out the following message.

```
scope error when matching [p1'] (qa0' (F Z [x2'] (S [y3'])))
against [p1'] (G (F Z [x2'] (S [y3']))) (F [x2'] [y3'] (S (S Z))))
when applying c : [p1'] (qa0' (F Z [x2'] (S [y3'])))
when applying substitution [ qa0' : \ m1' .
                                G m1' (F [x2'] [y3'] (S (S Z))) ]

current variables list:
  qa0' p1' x2' y3'
the current mixed proof term:
  K2 qa0'
    (\ p1' x2' y3' (c : [p1'] (qa0' (F Z [x2'] (S [y3'])))) .
      K (\ m1' . [p1'] m1') [x2'] [y3']
        ([p1'] (G (F Z [x2'] (S [y3']))) (F [x2'] [y3'] (S (S Z))))))
```

The eigenvariables are the variables surrounded by brackets, and the substitution  $[t/x]$  is represented as  $[x : t]$ . In this case the FCR will try to instantiate the existential variable `qa0'` with `\m1' . G m1' (F [x2'] [y3'] (S (S Z)))`. The `L` is the **current variables list** for the scope function, we can see the substitution will not pass the scope check. Moreover, we can inspect the mix proof term, we see that `qa0'` is not in the scope of `[x2']`, `[y3']`. Thus the function `h` gives a typing error.

## J Examples from Term Rewriting Literature

We demonstrate how to use the prototype FCR to represent some nontrivial nonterminations in this section. All of the examples in this section are from the existing term rewriting literature, and we will focus on representing nonlooping nonterminating reductions.

The general idea of representing a nonterminating reduction trace is the following: we need to see if the rule sequence can be generated by a corecursive function. Then we will try to assign a type for the corecursive function. Most of the efforts will be put on abstracting the right universal and existential type variables. Obtaining the right type for the corecursive function usually requires interactions with FCR and a good understanding of the type checking algorithm ERSM.

### J.1

The following string rewriting system is from Endrullis and Zantema [9], Example 29.

$$AL \rightarrow_1 LA \quad RA \rightarrow_2 AR \quad BL \rightarrow_3 BR \quad RB \rightarrow_4 LAB$$

Observe the following nonlooping nonterminating reduction:

$$\begin{aligned} BLB &\rightarrow_3 BRB \rightarrow_4 BLAB \rightarrow_3 BRAB \rightarrow_2 BARB \rightarrow_4 BALAB \rightarrow_1 BLAAB \rightarrow_3 \\ &BRAAB \rightarrow_2 BARAB \rightarrow_2 BAARB \rightarrow_4 BAALAB \rightarrow_1 BALAAB \rightarrow_1 BLAAAB \rightarrow_3 \dots \end{aligned}$$

Observe that all the strings in the reduction can be described by the regular expression  $BA^*(L|R)A^*B$ . We focus on the rule sequence:  $\underline{343241322411} \dots$ . The rule sequence can be generated by the following corecursive function:  $f \ a_1 \ a_2 \ a_3 \ a_4 = a_3 \cdot a_4 \cdot (f \ a_1 \ a_2 \ (a_3 \cdot a_2) \ (a_4 \cdot a_1))$ , i.e.  $f \ 1 \ 2 \ 3 \ 4$  gives the rule sequence.

The term rewriting system corresponds to the above string rewriting system is the following.

$$\begin{aligned} A(Lx) &\rightarrow_1 L(Ax) & R(Ax) &\rightarrow_2 A(Rx) & B(Lx) &\rightarrow_3 B(Rx) \\ & & R(Bx) &\rightarrow_4 L(A(Bx)) \end{aligned}$$

The following is the type assignment for the function  $f$ , where the variable  $\mathbf{r}$  is an existential variable and will be instantiated by  $(\backslash \mathbf{m1}' \ . \ A \ (\mathbf{r2}' \ \mathbf{m1}'))$  at the corecursive call of  $\mathbf{f}$ .

```
K1 : A (L x) <= L (A x)
K2 : R (A x) <= A (R x)
K3 : B (L x) <= B (R x)
K4 : R (B x) <= L (A (B x))
```

```
f : forall p l r y .
  (forall p x . p (l (A x)) => p (A (l x))) =>
  (forall p x . p (A (r x)) => p (r (A x))) =>
  (forall p x . p (B (r x)) => p (B (l x))) =>
  (forall p x . p (l (A (B x))) => p (r (B x))) =>
  p (B (l (B y)))
```

```
f a1 a2 a3 a4 = a3 (a4 (f (\ c . a1 c)
                          (\ c . a2 c))
```



$(\backslash \text{ c } . \text{ a3 } (\text{a2 c}))$   
 $(\backslash \text{ c } . \text{ a4 } (\text{a1 c})))$

$h : B (L (B y))$   
 $h = f \text{ K1 K2 } (\backslash \text{ c } . \text{ K3 c}) \text{ K4}$

## J.2

The following string rewriting system is from Endrullis and Zantema [9], Example 34.

$$ZL \rightarrow_1 LZ \quad RZ \rightarrow_2 ZR \quad ZLL \rightarrow_3 ZLR \quad RRZ \rightarrow_4 LZRZ$$

Observe the following nonlooping nonterminating reduction:

$$\begin{aligned}
\underline{ZLL}ZZRZ &\rightarrow_3 \underline{ZLR}ZZRZ \rightarrow_2 \underline{ZLZR}ZRZ \rightarrow_2 \underline{ZLZZRR}Z \rightarrow_4 \underline{ZLZZL}ZRZ \rightarrow_1 \\
\underline{ZLZL}ZZRZ &\rightarrow_1 \underline{ZLL}ZZZRZ \rightarrow_3 \underline{ZLR}ZZZRZ \rightarrow_2 \cdot \rightarrow_2 \cdot \rightarrow_2 \underline{ZLZZZRR}Z \rightarrow_4 \\
\underline{ZLZZZL}ZRZ &\rightarrow_1 \cdot \rightarrow_1 \cdot \rightarrow_1 \underline{ZLL}ZZZZRZ \rightarrow_3 \dots
\end{aligned}$$

Observe the rule sequence: 32241132224111..... This rule sequence can be generated by the following corecursive function:  $f \text{ a}_1 \text{ a}_2 \text{ a}_3 \text{ a}_4 = \text{a}_3 \cdot \text{a}_2 \cdot \text{a}_2 \cdot \text{a}_4 \cdot \text{a}_1 \cdot \text{a}_1 \cdot (f \text{ a}_1 \text{ a}_2 (\text{a}_3 \cdot \text{a}_2) (\text{a}_4 \cdot \text{a}_1))$ , i.e.  $f \text{ 1 2 3 4}$  gives the rule sequence.

The term rewriting system corresponds to the above string rewriting system is the following.

$$\begin{aligned}
Z (L x) &\rightarrow_1 L (Z x) \quad R (Z x) \rightarrow_2 Z (R x) \quad Z (L (L x)) \rightarrow_3 Z (L (R x)) \\
R (R (Z x)) &\rightarrow_4 L (Z (R (Z x)))
\end{aligned}$$

The following is the type that we assign to  $f$ . The existential variable  $r$  is instantiated by  $(\backslash m1' . Z (r2' m1'))$  at the corecursive call of  $f$ .

$K1 : Z (L x) \leq L (Z x)$   
 $K2 : R (Z x) \leq Z (R x)$   
 $K3 : Z (L (L x)) \leq Z (L (R x))$   
 $K4 : R (R (Z x)) \leq L (Z (R (Z x)))$

$f : \text{forall } p \text{ l r y } .$   
 $(\text{forall } p \text{ x } . p (l (Z x)) \Rightarrow p (Z (l x))) \Rightarrow$   
 $(\text{forall } p \text{ x } . p (Z (r x)) \Rightarrow p (r (Z x))) \Rightarrow$   
 $(\text{forall } p \text{ x } . p (Z (L (r x))) \Rightarrow p (Z (L (l x)))) \Rightarrow$   
 $(\text{forall } p \text{ x } . p (l (Z (R (Z x)))) \Rightarrow p (r (R (Z x)))) \Rightarrow$   
 $p (Z (L (l (Z (Z (R (Z y)))))))$

$f \text{ a1 a2 a3 a4 } = \text{a3 } (\text{a2 } (\text{a2 } (\text{a4 } (\text{a1 } (\text{a1 } (f (\backslash \text{ c } . \text{ a1 c})$   
 $(\backslash \text{ c } . \text{ a2 c})$   
 $(\backslash \text{ c } . \text{ a3 } (\text{a2 c}))$   
 $(\backslash \text{ c } . \text{ a4 } (\text{a1 c})))))))))$

$h : (Z (L (L (Z (Z (R (Z y)))))))$   
 $h = f \text{ K1 K2 } (\backslash \text{ c } . \text{ K3 c}) \text{ K4}$

### J.3

The following string rewriting system is from Endrullis and Zantema [9], Example 33.

$$AAL \rightarrow_1 LAA \quad RA \rightarrow_2 AR \quad BL \rightarrow_3 BR \quad RB \rightarrow_4 LAB \quad RB \rightarrow_5 ALB$$

Observe the following nonlooping nonterminating reduction:

$$\begin{aligned} & \underline{BRB} \rightarrow_4 \underline{BLAB} \rightarrow_3 \underline{BRAB} \rightarrow_2 \underline{BARB} \rightarrow_5 \underline{BAALB} \rightarrow_1 \underline{BLAAB} \rightarrow_3 \underline{BRAAB} \rightarrow_2 \cdot \rightarrow_2 \\ & \quad \underline{BAARB} \rightarrow_4 \underline{BAALAB} \rightarrow_1 \underline{BLAAAB} \rightarrow_3 \underline{BRAAAB} \rightarrow_2 \cdot \rightarrow_2 \cdot \rightarrow_2 \underline{BAAARB} \rightarrow_5 \\ & \quad \underline{BAAAALB} \rightarrow_1 \cdot \rightarrow_1 \underline{BLAAAAB} \rightarrow_3 \underline{BRAAAAAB} \rightarrow_2 \cdot \rightarrow_2 \cdot \rightarrow_2 \cdot \rightarrow_2 \underline{BAAAARB} \rightarrow_4 \dots \end{aligned}$$

Observe the rule sequence: 43251322, 41322251132222, 4113222225111322222.... This rule sequence can be generated by the following corecursive function:  $f \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 = a_4 \cdot a_3 \cdot a_2 \cdot a_5 \cdot a_1 \cdot a_3 \cdot a_2 \cdot a_2 \cdot (f \ a_1 \ a_2 \ (a_3 \cdot a_2 \cdot a_2) \ (a_4 \cdot a_1) \ (a_5 \cdot a_1))$ , i.e.  $f \ 1 \ 2 \ 3 \ 4 \ 5$  gives the rule sequence.

The term rewriting system corresponds to the above string rewriting system is the following.

$$\begin{aligned} & A(A(Lx)) \rightarrow_1 L(A(Ax)) \quad R(Ax) \rightarrow_2 A(Rx) \quad B(Lx) \rightarrow_3 B(Rx) \\ & R(Bx) \rightarrow_4 L(A(Bx)) \quad R(Bx) \rightarrow_5 A(L(Bx)) \end{aligned}$$

We assign a type for  $f$  in the following. The existential variable  $l$  is instantiated with  $(\backslash m1' \ . \ l1' \ (A \ (A \ m1')))$  at the corecursive call of  $f$ .

```

K1 : A (A (L x)) <= L (A (A x))
K2 : R (A x) <= A (R x)
K3 : B (L x) <= B (R x)
K4 : R (B x) <= L (A (B x))
K5 : R (B x) <= A (L (B x))

f : forall p l r y .
  (forall p x . p (l (A (A x))) => p (A (A (l x)))) =>
  (forall p x . p (A (r x)) => p (r (A x))) =>
  (forall p x . p (B (r x)) => p (B (l x))) =>
  (forall p x . p (l (A (B x))) => p (r (B x))) =>
  (forall p x . p (A (l (B x))) => p (r (B x))) =>
  p (B (r (B y)))

f a1 a2 a3 a4 a5 =
  a4 (a3 (a2 (a5 (a1 (a3 (a2 (a2 (f (\ c . a1 c)
                                     (\ c . a2 c)
                                     (\ c . a3 (a2 (a2 c)))
                                     (\ c . a4 (a1 c))
                                     (\ c . a5 (a1 c))))))))))

h : B (R (B y))
h = f K1 K2 K3 (\ c . K4 c) K5

```

## J.4

Consider the following rewriting system (from Zantema and Geser [26]) :

$$\begin{aligned} F Z (S x) y &\rightarrow_a F Z x (S y) \\ F Z (S x) y &\rightarrow_b F x y (S (S Z)) \end{aligned}$$

Observe the following nonlooping reduction trace.

$$\begin{aligned} F Z (S Z) (S Z) &\rightarrow_b F Z (S Z) (S (S Z)) \rightarrow_b F Z (S (S Z)) (S (S Z)) \rightarrow_a \\ &F Z (S Z) (S (S (S Z))) \rightarrow_b F Z (S (S (S Z))) (S (S Z)) \rightarrow_a \dots \end{aligned}$$

Note that the rule sequence for this reduction is: bbabaabaaab..... The nontermination can only be observed via the full reduction tree. The following partial reduction tree produced by FCR is an infinite binary tree structure with each branch finite (by issuing command `:full 6 (F Z (S Z) (S Z))` to FCR). Each node is a triple (e.g. `[] , B , F Z (S Z) (S (S Z))`), the first element denotes the redex position of the parent (which is a list of number, but all of them are at root position, hence `[]`), second element denotes the label of the rewrite rule applied, the third element denotes the contractum.

```

[] , _ , F Z (S Z) (S Z)
|
+- [] , B , F Z (S Z) (S (S Z))
| |
| +- [] , B , F Z (S (S Z)) (S (S Z))
| | |
| | +- [] , B , F (S Z) (S (S Z)) (S (S Z))
| | |
| | '- [] , A , F Z (S Z) (S (S (S Z)))
| | |
| | +- [] , B , F Z (S (S (S Z))) (S (S Z))
| | | |
| | | +- [] , B , F (S (S Z)) (S (S Z)) (S (S Z))
| | | |
| | | '- [] , A , F Z (S (S Z)) (S (S (S Z)))
| | | |
| | | +- [] , B , F (S Z) (S (S (S Z))) (S (S Z))
| | | |
| | | '- [] , A , F Z (S Z) (S (S (S (S Z))))
| | | |
| | | '- [] , A , F Z Z (S (S (S (S Z))))
| | |
| | '- [] , A , F Z Z (S (S (S Z)))
| |
| '- [] , A , F Z Z (S (S Z))
|
'- [] , A , F Z Z (S (S Z))

```

Note that the rule sequence can be described by the corecursive function  $f \ a_1 \ a_2 = a_2 \ (f \ (\lambda c. a_1 \ c) \ (\lambda c. a_2 \ (a_1 \ c)))$ . We assign a type for  $f$  in the following. The universal type variable  $f$  is instantiated by `\m1' m2' m3' . f1' m1' m2' (S m3')` at the corecursive call of function `f`. We observe `step h 7` gives `F Z (S Z) (S (S (S (S Z))))`, which is the reducible leaf at depth 6 in the reduction tree.

```

A : forall p x y . p (F Z x (S y)) => p (F Z (S x) y)
B : forall p x y . p (F x y (S (S Z))) => p (F Z (S x) y)

f : forall p f . (forall p x y . p (f Z x (S y)) => p (f Z (S x) y)) =>
  (forall p y . p (f Z y (S (S Z))) => p (f Z (S Z) y)) =>
    p (f Z (S Z) (S Z))
f a1 a2 = a2 (f (\ c . a1 c) (\ c . a2 (a1 c)))

h : F Z (S Z) (S Z)
h = f A (\ c . B c)
step h 7

```

## J.5



Consider the following one rule rewriting system (from Zantema and Geser [26]) :

$$F Z (S x) y \rightarrow_K G (F Z x (S y)) (F x y (S (S Z)))$$

Note that the rewrite system in Section J.4 is the *dummy eliminated* version of this rewriting system. Issuing command `:inner 6 (F Z (S Z) (S Z))` to FCR, we obtain the following reduction trace.

the execution trace is:

```

F Z (S Z) (S Z)
-K-> G (F Z Z (S (S Z))) (F Z (S Z) (S (S Z)))
-K-> G (F Z Z (S (S Z)))
  (G (F Z Z (S (S (S Z)))) (F Z (S (S Z)) (S (S Z))))
-K-> G (F Z Z (S (S Z)))
  (G (F Z Z (S (S (S Z))))
    (G (F Z (S Z) (S (S (S Z)))) (F (S Z) (S (S Z)) (S (S Z)))))
-K-> G (F Z Z (S (S Z)))
  (G (F Z Z (S (S (S Z))))
    (G (G (F Z Z (S (S (S (S Z))))) (F Z (S (S (S Z)) (S (S Z)))))
      (F (S Z) (S (S Z)) (S (S Z)))))
-K-> G (F Z Z (S (S Z)))
  (G (F Z Z (S (S (S Z))))
    (G (G (F Z Z (S (S (S (S Z)))))
      (G (F Z (S (S Z)) (S (S (S Z)))))
        (F (S (S Z)) (S (S Z)) (S (S Z)))))
      (F (S Z) (S (S Z)) (S (S Z)))))
-K-> G (F Z Z (S (S Z)))
  (G (F Z Z (S (S (S Z))))
    (G (G (F Z Z (S (S (S (S Z)))))
      (G (G (F Z (S Z) (S (S (S Z)))))
        (F (S Z) (S (S (S Z)) (S (S Z)))))
          (F (S (S Z)) (S (S Z)) (S (S Z)))))
      (F (S Z) (S (S Z)) (S (S Z)))))

```

In this case the rule sequence is pretty simple, so we cannot learn much from the rule sequence. But when we observe the redexes, the reduction appear to have the same patterns

as the one in Section J.4. The dummy elimination technique makes the reduction pattern explicit in the rule sequence, it inspires us to arrive at the following representation.

```
K : F Z (S x) y <= G (F Z x (S y)) (F x y (S (S Z)))

f : forall p qa qb f .
  (forall p x y . p (qa (f Z x (S y)) x y) => p (f Z (S x) y)) =>
  (forall p y . p (qb (f Z y (S (S Z))) y) => p (f Z (S Z) y)) =>
  p (f Z (S Z) (S Z))
f a1 a2 = a2 (f (\ c . a1 c) (\ c . (a2 (a1 c))))

h : F Z (S Z) (S Z)
h = f (\ c . K c) (\ c . K c)
```

step h 7

The function  $f$  follows the exact same pattern as in Section J.4, but its type reflect the two use case of the rule  $K$ , i.e. applying  $K$  to the left or right argument of  $G$ . For each case we use a existential variable to capture the resulting contexts. Note that the existential variable  $qa$  has arity 3 and the existential variable  $qb$  has arity 2. Let us observe the following fully annotated  $h$  and  $f$  from FCR. Notice that the third argument for  $f$  in the definition of  $h$  is  $\backslash m1' m2' . G (F Z Z (S m2')) m1'$  (the order of  $m1'$  and  $m2'$  is switched in the body). And the third argument is  $\backslash m1' m2' . qb2' (qa1' m1' m2' (S (S Z))) (S m2')$  at the corecursive call of  $f$  in the definition of  $f$  (the variable  $m2'$  is duplicated).

lemmas

```
h : F Z (S Z) (S Z) =
  f (\ x1' . x1') (\ m1' m2' m3' . G m1' (F m2' m3' (S (S Z))))
  (\ m1' m2' . G (F Z Z (S m2')) m1')
  (\ m1' m2' m3' . F m1' m2' m3')
  (\ p4'
    x5'
    y6'
    (c : p4' (G (F Z x5' (S y6')) (F x5' y6' (S (S Z))))) .
      K (\ m1' . p4' m1') x5' y6' c)
  (\ p10' y11' (c : p10' (G (F Z Z (S y11')) (F Z y11' (S (S Z))))) .
    K (\ m1' . p10' m1') Z y11' c)
f : forall p qa qb f .
  (forall p x y . p (qa (f Z x (S y)) x y) => p (f Z (S x) y))
  =>
  (forall p y . p (qb (f Z y (S (S Z))) y) => p (f Z (S Z) y))
  =>
  p (f Z (S Z) (S Z)) =
  \ p0'
  qa1'
  qb2'
  f3'
  (a1 : forall p x y .
    p (qa1' (f3' Z x (S y)) x y) => p (f3' Z (S x) y))
  (a2 : forall p y .
    p (qb2' (f3' Z y (S (S Z))) y) => p (f3' Z (S Z) y)) .
  a2 (\ m1' . p0' m1') (S Z)
  (f (\ m1' . p0' (qb2' m1' (S Z)))
```

```

(\ m1' m2' m3' . qa1' m1' m2' (S m3'))
(\ m1' m2' . qb2' (qa1' m1' m2' (S (S Z))) (S m2'))
(\ m1' m2' m3' . f3' m1' m2' (S m3'))
(\ p10'
  x11'
  y12'
  (c : p10' (qa1' (f3' Z x11' (S (S y12')))) x11' (S y12')) .
    a1 (\ m1' . p10' m1') x11' (S y12') c)
(\ p16'
  y17'
  (c : p16'
    (qb2' (qa1' (f3' Z y17' (S (S (S Z)))) y17' (S (S Z))) (S y17')) .
    a2 (\ m1' . p16' m1') (S y17')
    (a1 (\ m1' . p16' (qb2' m1' (S y17')) y17' (S (S Z)) c)))

```

## J.6



The following term rewriting system is adapted from a string rewriting system in [9] (Section 7), no current automated termination checker can detect the nontermination for this example.

$$\begin{aligned}
Bl (B x) &\rightarrow_1 B (Bl x) \\
Bl (Cl (Dl x)) &\rightarrow_2 B (Cl (D x)) \\
D (Dl x) &\rightarrow_3 Dl (D x) \\
Al (X x) &\rightarrow_4 Al (Bl (Bl x)) \\
B (X x) &\rightarrow_5 X (Cl (Y x)) \\
Bl (Cl (Dl x)) &\rightarrow_6 X (Cl (Y x)) \\
Y (D x) &\rightarrow_7 Dl (Y x) \\
Y (El x) &\rightarrow_8 Dl (Dl (El x))
\end{aligned}$$

Observe the following nonlooping reduction trace ( $\rightarrow_{a,b}$  is a shorthand for  $\rightarrow_a \cdot \rightarrow_b$ ):

$$\begin{aligned}
&Al (Bl (Bl (Cl (Dl (Dl (El x))))) \rightarrow_2 Al (Bl (B (Cl (D (Dl (El x))))) \rightarrow_{1,3} \\
&Al (B (Bl (Cl (Dl (Dl (El x))))) \rightarrow_6 Al (B (X (Cl (Y (D (El x))))) \rightarrow_{5,7} \\
&Al (X (Bl (Cl (Dl (Y (El x))))) \rightarrow_{4,8} Al (Bl (Bl (Bl (Cl (Dl (Dl (Dl (El x))))) \rightarrow_2 \\
&Al (Bl (Bl (B (Cl (D (Dl (Dl (El x))))) \rightarrow_{1,3,1,3} \\
&Al (B (Bl (Bl (Cl (Dl (Dl (Dl (El x))))) \rightarrow_2 \\
&Al (B (Bl (B (Cl (Dl (Dl (Dl (El x))))) \rightarrow_{1,3} \\
&Al (B (B (Bl (Cl (Dl (Dl (Dl (El x))))) \rightarrow_6 \\
&Al (B (B (X (Cl (X (Dl (Dl (El x))))) \rightarrow_{5,7,5,7} \\
&Al (X (Bl (Bl (Cl (Dl (Dl (Y (El x))))) \rightarrow_{4,8} \\
&Al (Bl (Bl (Bl (Bl (Cl (Dl (Dl (Dl (El x))))) \rightarrow \dots
\end{aligned}$$

The rewriting system admits reductions of the form:  $Al (Bl^n (Cl (Dl^n (El x)))) \rightarrow^* Al (Bl^{n+1} (Cl (Dl^{n+1} (El x))))$  for any for every  $n > 1$ . The rule sequence of the above reduction is the following: 213, 657, 48, 21313, 213, 65757, 48, 2131313, 21313, 213, 6575757, 48, .... We now represent this rule sequence by the following corecursive function:

$$\begin{aligned}
&f \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ b = \\
&(b \cdot a_6 \cdot a_5 \cdot a_7 \cdot a_4 \cdot a_8) (f \ a_1 \ (a_2 \cdot a_1 \cdot a_3) \ a_3 \ a_4 \ a_5 \ (a_6 \cdot a_5 \cdot a_7) \ a_7 \ a_8 \ (a_2 \cdot a_1 \cdot a_3 \cdot a_1 \cdot a_3 \cdot b))
\end{aligned}$$

Note that  $f \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ (2 \cdot 1 \cdot 3)$  generates the rule sequence above. The following is the type we assign for  $f$ .

```

K1 : B1 (B x) <= B (B1 x)
K2 : B1 (C1 (D1 x)) <= B (C1 (D x))
K3 : D (D1 x) <= D1 (D x)
K4 : A1 (X x) <= A1 (B1 (B1 x))
K5 : B (X x) <= X (B1 x)
K6 : B1 (C1 (D1 x)) <= X (C1 (Y x))
K7 : Y (D x) <= D1 (Y x)
K8 : Y (E1 x) <= D1 (D1 (E1 x))

f : forall p0 c b d y .
  (forall p x . p (B (B1 x)) => p (B1 (B x))) =>
  (forall p x . p (B ( c (D x))) => p (B1 ( c (D1 x)))) =>
  (forall p x . p (D1 (D x)) => p (D (D1 x))) =>
  (forall p x . p (A1 (B1 (B1 x))) => p (A1 (X x))) =>
  (forall p x . p (X (B1 x)) => p (B (X x))) =>
  (forall p x . p (X ( c (Y x))) => p ( b (C1 ( d x)))) =>
  (forall p x . p (D1 (Y x)) => p (Y (D x))) =>
  (forall p x . p (D1 (D1 (E1 x))) => p (Y (E1 x))) =>
  (forall p x . p (B ( b (C1 ( d (D x))))) => p (B1 (B1 ( c (D1 (D1 x)))))) =>
  p0 (A1 (B1 (B1 ( c (D1 (D1 (E1 y)))))))

f a1 a2 a3 a4 a5 a6 a7 a8 b =
  b (a6 (a5 (a7 (a4 (a8 (f a1
    (\ c1 . a2 (a1 (a3 c1)))
    a3
    a4
    a5
    (\ c1. a6 (a5 (a7 c1)))
    a7
    a8
    (\ c1 . a2 (a1 (a3 (a1 (a3 (b c1))))))))))))))

h : (A1 (B1 (B1 ( C1 (D1 (D1 (E1 y)))))))
h = f K1 K2 K3 K4 K5 K6 K7 K8 (\ c . K2 (K1 (K3 c)))

```

Note that the quantified variables  $b, d$  in the type of  $f$  are existential variables. In the corecursive call of  $f$ , the variable  $c$  will be instantiated with  $(\backslash m1' . B1 (c1' (D1 m1')))$ ,  $b$  will be instantiated with  $(\backslash m1' . B (b2' m1'))$  and  $d$  will be instantiated with  $(\backslash m1' . d3' (D m1'))$ .

## J.7



The following rewriting system is from Emmes et. al. [8], which according to them is outside the scope of the their nontermination detection techniques.

$$\begin{aligned}
 G \ T \ T \ x \ (S \ y) &\rightarrow_1 G \ (N \ x) \ (N \ y) \ (S \ x) \ (D \ (S \ y)) \\
 N \ Z &\rightarrow_2 T \\
 N \ (S \ x) &\rightarrow_3 N \ x
 \end{aligned}$$



$$\begin{aligned} D Z &\rightarrow_4 Z \\ D (S x) &\rightarrow_5 S (S (D x)) \end{aligned}$$

Observe the following nonlooping nonterminating reduction trace for  $G T T Z (S Z)$  (using left to right, inner-most reduction strategy).

$$\begin{aligned} G T T Z (S Z) &\rightarrow_1 G (N Z) (N Z) (S Z) (D (S Z)) \rightarrow_2 G T (N Z) (S Z) (D (S Z)) \rightarrow_2 \\ &G T T (S Z) (D (S Z)) \rightarrow_5 G T T (S Z) (S (S (D Z))) \rightarrow_4 G T T (S Z) (S (S Z)) \rightarrow_1 \\ &G (N (S Z)) (N (S Z)) (S (S Z)) (D (S (S Z))) \rightarrow_3 \\ G (N Z) (N (S Z)) (S (S Z)) (D (S (S Z))) &\rightarrow_2 G T (N (S Z)) (S (S Z)) (D (S (S Z))) \rightarrow_3 \\ &G T (N Z) (S (S Z)) (D (S (S Z))) \rightarrow_2 G T T (S (S Z)) (D (S (S Z))) \rightarrow_5 \\ &G T T (S (S Z)) (S (S (D (S Z)))) \rightarrow_5 G T T (S (S Z)) (S (S (S (D Z)))) \rightarrow_4 \\ &G T T (S (S Z)) (S (S (S (S Z)))) \dots \end{aligned}$$

The rule sequence is of the shape 1, 22, 54, 1, 3232, 554, 1, 3323332, 55554.... This rule sequence can be represented by the following corecursive equation.

$$f a_1 a_2 b_2 a_3 b_3 a_4 a_5 = a_1 a_2 b_2 a_5 a_4 (f a_1 (a_3 \cdot a_2) (b_3 \cdot b_2) a_3 (b_3 \cdot b_3) a_4 (a_5 \cdot a_5))$$

Note that  $f 1 2 2 3 3 4 5$  gives rise to the rule sequence. The following is the type that we assign to  $f$ .

```
K1 : forall p x y . p (G (N x) (N y) (S x) (D (S y))) => p (G T T x (S y))
K2 : forall p . p T => p (N Z)
K3 : forall p x . p (N x) => p (N (S x))
K4 : forall p . p Z => p (D Z)
K5 : forall p x . p (S (S (D x))) => p (D (S x))
f : forall p g n1 n2 s .
  (forall p x y . p (g (n1 x) (n2 y) (S x) (D (s y)))) => p (g T T x (s y))) =>
  (forall p . p T => p (n1 Z)) =>
  (forall p . p T => p (n2 Z)) =>
  (forall p x . p (n1 x) => p (n1 (S x))) =>
  (forall p x . p (n2 x) => p (n2 (s x))) =>
  (forall p . p Z => p (D Z)) =>
  (forall p x . p (s (s (D x))) => p (D (s x))) =>
  p (g T T Z (s Z))
```

```
f a1 a2 b2 a3 b3 a4 a5 =
  a1 (a2 (b2 (a5 (a4 (f (\ c . a1 c)
                        (\ c . a3 (a2 c))
                        (\ c . (b3 (b2 c)))
                        (\ c . a3 c)
                        (\ c . b3 (b3 c)))
                    a4
                    (\ c . a5 (a5 c)))))))
```

```
h : G T T Z (S Z)
h = f (\ c . K1 c) K2 K2 K3 K3 K4 K5
```

Note that  $n1$ ,  $n2$  in the type of  $f$  are existential variables. At the corecursive call of  $f$ , variable  $g$  is instantiated by  $(\backslash m1' m2' m3' m4' . g1' m1' m2' (S m3') m4')$ , variable

$n1$  is instantiated by  $(\backslash m1' \ . \ n12' \ (S \ m1'))$ , variable  $n2$  is instantiated by  $(\backslash m1' \ . \ n23' \ (s4' \ m1'))$ , variable  $s$  is instantiated by  $(\backslash m1' \ . \ s4' \ (s4' \ m1'))$ .