EPTCS 49

Proceedings of the

5th International Workshop on Higher-Order Rewriting

Edinburgh, UK, July 14, 2010

Edited by: Eduardo Bonelli

Published: 17th February 2011
DOI: 10.4204/EPTCS.49
ISSN: 2075-2180
Open Publishing Association

Preface

HOR 2010 is a forum to present work concerning all aspects of higher-order rewriting. The aim is to provide an informal and friendly setting to discuss recent work and work in progress. Previous editions of HOR were held in Copenhagen – Denmark (HOR 2002), Aachen – Germany (HOR 2004), Seattle – USA (HOR 2006) and Paris – France (HOR 2007).

In addition to an interesting set of submissions, this year we had the following invited speakers to whom I would like to give thanks:

- Maribel Fernández (King's College London) who talked about *Closed Nominal Rewriting: Properties and Applications* and
- Silvia Ghilezan (University of Novi Sad) who talked about *Computational Interpretations of Logic*.

My appreciation also to the members of the PC (Zena Ariola, Frédéric Blanqui, Mariangiola Dezani-Ciancaglini and Roel de Vrijer) for lending their time and expertise, to the referees, and to Delia Kesner and Femke van Raamsdonk for providing valuable support. Thanks also to GDR-IM which awarded funds to HOR'2010 that were used for supporting presentation of papers by students.

Finally, I would like to thank the organizers of FLoC 2010 and affiliated events for contributing towards such an exciting event.

Eduardo Bonelli (Universidad Nacional de Quilmes, Argentina)

Table of Contents

Swapping: a natural bridge between named and indexed explicit substitution calculi

Ariel Mendelzon
Depto. de Computación, FCEyN,
Universidad de Buenos Aires.
amendelzon@dc.uba.ar

Alejandro Ríos
Depto. de Computación, FCEyN,
Universidad de Buenos Aires.
rios@dc.uba.ar

Beta Ziliani
Depto. de Computación, FCEyN,
Universidad de Buenos Aires.
lziliani@dc.uba.ar

This article is devoted to the presentation of λrex, an explicit substitution calculus with *de Bruijn* indexes and a simple notation. By being isomorphic to λex – a recent formalism with variable names –, λrex accomplishes simulation of β-reduction (Sim), preservation of β-strong normalization (PSN) and meta-confluence (MC), among other desirable properties. Our calculus is based on a novel presentation of λ_{dB}, using a *swap* notion that was originally devised by de Bruijn. Besides λrex, two other indexed calculi isomorphic to λx and λxgc are presented, demonstrating the potential of our technique when applied to the design of indexed versions of known named calculi.

1 Introduction

This article is devoted to explicit substitutions (ES, for short), a formalism that has attracted attention since the appearance of $\lambda\sigma$ [1] and, later, of Melliès' counterexample [17], showing the lack of the preservation of β-strong normalization property (PSN, for short) in $\lambda\sigma$. One of the main motivations behind the field of ES is studying how substitution behaves when internalized in the language it serves (in the classic λ-calculus, substitution is a meta-level operation). Several calculi have been proposed since the counterexample of Melliès, and few have been shown to have a whole set of desirable properties: simulation of β-reduction, PSN, meta-confluence, full composition, etc. For a detailed introduction to the ES field, we refer the reader to e.g. [16, 15, 20].

In 2008, D. Kesner proposed λex [14, 15], a formalism with variable names that has the entire set of properties expected from an ES calculus. As Kesner points in [15], for implementation purposes a different approach to variable names should be taken, since bound variable renaming (*i.e.,* working modulo α-equivalence) is known to be error-prone and computationally expensive. Among others, one of the ways this problem is tackled is by using *de Bruijn* notation [5], which is a technique that simply avoids the need of working modulo α-equivalence. As far as we know, no ES calculus with *de Bruijn* indexes and the whole set of properties enjoyed by λex exists to date. The main target of this article is the introduction of λrex, an ES calculus with *de Bruijn* indexes that, by being isomorphic to λex, enjoys the same set of properties. λrex is based on λr, a novel *swapping*-based version of the classic λ_{dB} [5], that we also introduce here.

It is important to remark that the whole development was made on a staged basis: we first devised λr, and then made substitutions explicit orienting the definition for λr's meta-substitution. At that point, we got a calculus we called λre, which turned out to be isomorphic to λx [4, 3]. Encouraged by this result, we added *Garbage Collection* to λre, obtaining a calculus isomorphic to λxgc [4]: $\lambda \mathrm{re}_{\mathrm{gc}}$. Finally, we added composition of substitutions in the style of λex to $\lambda \mathrm{re}_{\mathrm{gc}}$, obtaining λrex. Thus, besides fulfilling our original aim, we introduced *swapping*, a technique that turns out to behave as a natural bridge between named and indexed formalisms. Furthermore, we didn't know any indexed isomorphic versions of λx nor λxgc.

The content of the article is as follows: in Section 2 we present λr, an alternative version of λ_{dB}. Next, in Section 3, we introduce λre, $\lambda\mathrm{re}_{\mathrm{gc}}$ and λrex, the three ES calculi derived from λr already mentioned. We show the isomorphism between these and the λx, λxgc and λex calculi in Section 4. Last, in sections 5 and 6, we point out related work and present the conclusions, respectively. We refer the interested reader to [18] for complete proofs over the whole development.

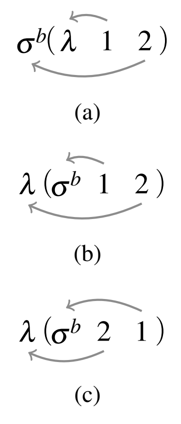
2 A new presentation for λ_{dB}: the λr-calculus

2.1 Intuition

The λ-calculus with *de Bruijn* indexes (λ_{dB}, for short) [5] accomplishes the elimination of α-equivalence, since α-equivalent λ-terms are syntactically identical under λ_{dB}. This greatly simplifies implementations, since caring about bound variable renaming is no longer necessary. One usually refers to a *de Bruijn* indexed calculus as a *nameless calculus*, for binding is positional – relative – instead of absolute (indexes are used in place of names for this purpose). We observe here that, even though this *nameless* notion makes sense in the classical λ_{dB}-calculus (because the substitution operator is located in the meta-level), it seems not to be the case in certain ES calculi derived from λ_{dB}, such as: λs [11], $\lambda\mathrm{s}_e$ [12] or λt [13]. These calculi have constructions of the form $a[i := b]$ to denote ES (notations vary). Here, even though i is not a name *per se*, it plays a similar role: i indicates which free variable should be substituted; then, these calculi are not purely *nameless*, *i.e.,* binding is *mixed*: positional (relative) for abstractions and named (absolute) for closures.

In general, we observe that not a single ES calculus with *de Bruijn* indexes to date is completely nameless. This assertion rests on the following observation: in each and every case, the (Lamb) rule is of the form $(\lambda a)[s] \to \lambda a[s']$. Thus, since the term a is not altered, an "absolute binding technique" *must* be implemented inside s in order to indicate which free variable is to be substituted. To further support this *not-completely-nameless* assertion, we note that even though there is a known isomorphism between the classic λ-calculus and the λ_{dB}-calculus, when substitutions are made explicit in both calculi, the isomorphism does not hold just by adding the new ES case (which would be reasonable to expect). The problem is that λ_{dB}'s classic definition is always – tacitly, at least – being used for the explicitation task, thus obtaining calculi with mixed binding approaches, as mentioned earlier. As shown throughout the rest of the paper, our (Lamb) rules will be of the form $(\lambda a)[s] \to \lambda a'[s']$, *i.e.,* altering *both a and s* to enforce a *completely nameless* approach.

In order to obtain a *completely nameless* notion for an explicit substitutions λ_{dB}, we start by eliminating the index i from the substitution operator. Then, we are left with terms of the form $a[b]$, and with a (Beta) reduction rule that changes from $(\lambda a)\, b \to a[1 := b]$ to $(\lambda a)\, b \to a[b]$. The semantics of $a[b]$ should be clear from the new (Beta) rule. The problem is, of course, how to define it. Two difficulties arise when a substitution crosses (goes into) an abstraction: first, the indexes of b should be incremented in order to reflect the new variable bindings; second – and the key to our technology –, some mechanism should be implemented in order to replace the need for indexes inside closures (since these should be incremented, too).


$\sigma^b(\lambda\ 1\ 2)$
(a)
$\lambda\,(\sigma^b\ 1\ 2)$
(b)
$\lambda\,(\sigma^b\ 2\ 1)$
(c)
Figure 1: Bindings

The first problem is solved easily: we just use an operator to progressively increment indexes with every abstraction crossing, in the style of λt [13]. The second issue is a bit harder. Figure 1 will help us clarify what we do when a substitution crosses an abstraction, momentarily using $\sigma^b a$ to denote $a[b]$ in order to emphasize the binding character of the substitution (by writing the substitution construction *before* the term and *annotating* it with the substituent – which does not actually affect binding –, it resembles the abstraction operation; thus, "reading" the term is much easier for those who are already familiar with *de Bruijn* notation). In this example we use the term $\sigma^b(\lambda 1 2)$ (which stands for $(\lambda 1 2)[b]$). Figure 1(a) shows the bindings in the original term; Figure 1(b) shows that bindings are inverted if we cross the abstraction and do not make any changes. Then, in order to get bindings "back on the road", we just *swap* indexes 1 and 2! (Figure 1(c)). With this operation we recover, intuitively, the original semantics of the term. Summarizing, all that is needed when abstractions are crossed is: *swap* indexes 1 and 2 and, also, increment the indexes of the term carried in the substitution. That is exactly what λr does, with substitutions in the meta-level.

In Section 2.2 we define both λ_{dB} and λr; in Section 2.3 we show that they are the same calculus.

2.2 Definitions

First of all, we define some operations on sets of naturals numbers.

Definition 1 (Operations on sets of natural numbers)**.** For every $N \subset \mathbb{N}$, $k \in \mathbb{N}$:

1. $N + k = \{n + k : n \in N\}$
2. $N - k = \{n - k : n \in N \wedge n > k\}$
3. $N_{\oplus k} = \{n : n \in N \wedge n \oplus k\}$, with $\oplus \in \{=, <, \leq, >, \geq\}$

Terms for λr are the same as those for λ_{dB}. That is:

Definition 2 (Terms for λ_{dB} and λr)**.** The set of terms for λ_{dB} and λr, denoted Λ_{dB}, is given in BNF by:

$$a ::= n \mid a\,a \mid \lambda a \qquad (n \in \mathbb{N}_{>0})$$

Definition 3 (Free variables)**.** The free variables of a term, $\text{FV} : \Lambda_{\text{dB}} \to \mathscr{P}(\mathbb{N}_{>0})$, is given by:

$$\text{FV}(n) = \{n\} \qquad \text{FV}(ab) = \text{FV}(a) \cup \text{FV}(b) \qquad \text{FV}(\lambda a) = \text{FV}(a) - 1$$

Classical definitions

We recall the classical definitions for λ_{dB} (see e.g. [11] for a more detailed introduction).

Definition 4 (Updating meta-operator for λ_{dB})**.** For every $k \in \mathbb{N}$, $i \in \mathbb{N}_{>0}$, $\text{U}^i_k : \Lambda_{\text{dB}} \to \Lambda_{\text{dB}}$ is given inductively by:

$$\text{U}^i_k(n) \;=\; \begin{cases} n & \text{if } n \leq k \\ n + i - 1 & \text{if } n > k \end{cases} \qquad \begin{array}{rcl} \text{U}^i_k(ab) & = & \text{U}^i_k(a)\,\text{U}^i_k(b) \\ \text{U}^i_k(\lambda a) & = & \lambda\,\text{U}^i_{k+1}(a) \end{array}$$

Definition 5 (Meta-substitution for λ_{dB})**.** For every $a, b, c \in \Lambda_{\text{dB}}$, $m, n \in \mathbb{N}_{>0}$, $\bullet\{\{\bullet \leftarrow \bullet\}\} : \Lambda_{\text{dB}} \times \mathbb{N}_{>0} \times \Lambda_{\text{dB}} \to \Lambda_{\text{dB}}$ is given inductively by:

$$m\{\{n \leftarrow c\}\} \;=\; \begin{cases} m & \text{if } m < n \\ \text{U}^n_0(c) & \text{if } m = n \\ m - 1 & \text{if } m > n \end{cases} \qquad \begin{array}{rcl} (ab)\{\{n \leftarrow c\}\} & = & a\{\{n \leftarrow c\}\}\,b\{\{n \leftarrow c\}\} \\ (\lambda a)\{\{n \leftarrow c\}\} & = & \lambda a\{\{n + 1 \leftarrow c\}\} \end{array}$$

Definition 6 (λ_{dB}-calculus)**.** The λ_{dB}-calculus is the reduction system $(\Lambda_{\text{dB}}, \beta_{\text{dB}})$, where:

$$(\forall a, b \in \Lambda_{\text{dB}})\left(a \to_{\beta_{\text{dB}}} b \iff (\exists\, C \text{ context; } c, d \in \Lambda_{\text{dB}})\,(a = \text{C}[(\lambda c)d] \wedge b = \text{C}[c\{\{1 \leftarrow d\}\}])\right)$$

New definitions

We now define the new meta-operators used to implement index increments and swaps.

Definition 7 (*Increment operator* – $\uparrow_i$)**.** For every $i \in \mathbb{N}$, $\uparrow_i : \Lambda_{\mathrm{dB}} \to \Lambda_{\mathrm{dB}}$ is given inductively by:

$$\uparrow_i(n) \;=\; \begin{cases} n & \text{if} \quad n \le i \\ n+1 & \text{if} \quad n > i \end{cases} \qquad \begin{array}{rcl} \uparrow_i(ab) & = & \uparrow_i(a)\uparrow_i(b) \\ \uparrow_i(\lambda a) & = & \lambda \uparrow_{i+1}(a) \end{array}$$

Definition 8 (*Swap operator* – $\updownarrow_i$)**.** For every $i \in \mathbb{N}_{>0}$, $\updownarrow_i : \Lambda_{\mathrm{dB}} \to \Lambda_{\mathrm{dB}}$ is given inductively by:

$$\updownarrow_i(n) \;=\; \begin{cases} n & \text{if} \quad n < i \vee n > i+1 \\ i+1 & \text{if} \quad n = i \\ i & \text{if} \quad n = i+1 \end{cases} \qquad \begin{array}{rcl} \updownarrow_i(ab) & = & \updownarrow_i(a)\updownarrow_i(b) \\ \updownarrow_i(\lambda a) & = & \lambda \updownarrow_{i+1}(a) \end{array}$$

Finally, we present the meta-level substitution definition for λr, and then the λr-calculus itself.

Definition 9 (Meta-substitution for λr)**.** For every $a,b,c \in \Lambda_{\mathrm{dB}}$, $n \in \mathbb{N}_{>0}$, $\bullet\{\bullet\} : \Lambda_{\mathrm{dB}} \times \Lambda_{\mathrm{dB}} \to \Lambda_{\mathrm{dB}}$ is given inductively by:

$$n\{c\} \;=\; \begin{cases} c & \text{if} \quad n = 1 \\ n-1 & \text{if} \quad n > 1 \end{cases} \qquad \begin{array}{rcl} (ab)\{c\} & = & a\{c\}b\{c\} \\ (\lambda a)\{c\} & = & \lambda \updownarrow_1(a)\{\uparrow_0(c)\} \end{array}$$

Definition 10 (λr-calculus)**.** The λr-calculus is the reduction system $(\Lambda_{\mathrm{dB}}, \beta_{\mathrm{r}})$, where:

$$(\forall a,b \in \Lambda_{\mathrm{dB}})\left(a \to_{\beta_{\mathrm{r}}} b \iff (\exists\, C \text{ context; } c,d \in \Lambda_{\mathrm{dB}})\,(a = \mathrm{C}[(\lambda c)d] \wedge b = \mathrm{C}[c\{d\}])\right)$$

2.3 λ_{dB} and λr are the same calculus

We want to prove that λr equals λ_{dB}. That is, we want to show that $a\{\{1 \leftarrow b\}\} = a\{b\}$. In order to do this, however, we should first prove the general case: $a\{\{n \leftarrow b\}\} = a'\{b'\}$, with a' and b' being the result of a series of *swaps* and *increments* over a and b, respectively. This comes from observing that, while λ_{dB} increments the index inside the substitution when going into an abstraction, λr performs a *swap* over the affected term, and an index increment over the term carried in the substitution. Thus, comparing what happens after the "crossing" of $n-1$ abstractions in $(\underbrace{\lambda \cdots \lambda}_{n-1} a)\{\{1 \leftarrow b\}\}$ and $(\underbrace{\lambda \cdots \lambda}_{n-1} a)\{b\}$, we get to:

$$\underbrace{\lambda \cdots \lambda}_{n-1} a\{\{n \leftarrow b\}\} \qquad \text{and} \qquad \underbrace{\lambda \cdots \lambda}_{n-1} \updownarrow_1(\cdots \updownarrow_{n-1}(a)\cdots)\{\underbrace{\uparrow_0(\cdots\uparrow_0(}_{n-1}b)\cdots)\}$$

Therefore, the idea for the proof is showing that the above terms are equal for every $n \in \mathbb{N}_{>0}$. We formalize this idea by introducing two additional definitions: stacked swaps and stacked increments.

Definition 11 (Stacked swap)**.** For every $i \in \mathbb{N}_{>0}$, $j \in \mathbb{N}$, $\Updownarrow_i^j : \Lambda_{\mathrm{dB}} \to \Lambda_{\mathrm{dB}}$ is given inductively by:

$$\Updownarrow_i^j(a) \;=\; \begin{cases} a & \text{if } j = 0 \\ \Updownarrow_i^{j-1}(\updownarrow_{i+j-1}(a)) & \text{if } j > 0 \end{cases}$$

The intuitive idea behind $\Updownarrow_i^j(a)$ is that of: $\underbrace{\updownarrow_i(\updownarrow_{i+1}(\cdots \updownarrow_{i+j-1}(}_{j\ swaps}a)\cdots))$

Definition 12 (Stacked increment)**.** For every $i \in \mathbb{N}$, $\Uparrow^i \colon \Lambda_{\text{dB}} \to \Lambda_{\text{dB}}$ is given inductively by:

$$\Uparrow^i(a) \quad = \quad \begin{cases} a & \text{if } i = 0 \\ \Uparrow^{i-1}(\uparrow_0(a)) & \text{if } i > 0 \end{cases}$$

The intuitive idea behind $\Uparrow^i(a)$ is that of: $\quad \underbrace{\uparrow_0(\cdots\uparrow_0}_{i \text{ increments}}(a)\cdots)$

Based on this last two definitions, the next theorem states the relationship between λr and λ_{dB} meta-substitution operators, having as an immediate corollary that λr and λ_{dB} are the same calculus.

Theorem 13 (Correspondence between λ_{dB} and λr meta-substitution)**.** For every $a, b \in \Lambda_{\text{dB}}$, $n \in \mathbb{N}_{>0}$:

$$a\{\{n \leftarrow b\}\} = \Updownarrow_1^{n-1}(a)\{\Uparrow^{n-1}(b)\}$$

Proof. See Appendix A. □

Corollary 14. For every $a, b \in \Lambda_{\text{dB}}$: $a\{\{1 \leftarrow b\}\} = a\{b\}$. Therefore, λ_{dB} and λr are the same calculus.

Proof. Use Theorem 13 with $n = 1$, and conclude the equality of both calculi by definition. This result was checked using the *Coq* theorem prover[1]. □

3 Devising the λre, λre$_{\text{gc}}$ and λrex calculi

In order to derive an ES calculus from λr, we first need to internalize substitutions in the language. Thus, we add the construction $a[b]$ to Λ_{dB}, and call the resulting set of terms Λre. The definition for the free variables of a term is extended to consider the ES case as follows: $\text{FV}(a[b]) = (\text{FV}(a) - 1) \cup \text{FV}(b)$. Also, and as a design decision, operators $\uparrow_i$ and $\updownarrow_i$ are left in the meta-level. Naturally, we must extend their definitions to the ES case, task that needs some lemmas over λr's meta-operators in order to ensure correctness. We use lemmas 26 and 27 in Appendix B for the extension of swap and increment meta-operators:

$$\uparrow_i(a[b]) = \uparrow_{i+1}(a)[\uparrow_i(b)] \qquad \text{and} \qquad \updownarrow_i(a[b]) = \updownarrow_{i+1}(a)[\updownarrow_i(b)]$$

Then, we just orient the equalities from the meta-substitution definition as expected and get a calculus we call λre (that turns out to be isomorphic to λx [4, 3], as we will later explain).

As a next step in our work, we add *Garbage Collection* to λre. The goal is removing useless substitutions, *i.e.,* when the index 1 does not appear free in the term. When removing a substitution, free indexes of the term must be updated, decreasing them by 1. To accomplish this, we introduce a new meta-operator: $\downarrow_i$. The operator is inspired in a similar one from [19]. We first define it for the set Λ_{dB}:

Definition 15 (*Decrement operator –* $\downarrow_i$)**.** For every $i \in \mathbb{N}_{>0}$, $\downarrow_i : \Lambda_{\text{dB}} \to \Lambda_{\text{dB}}$ is given inductively by:

$$\downarrow_i(n) \quad = \quad \begin{cases} n & \text{if } \; n < i \\ \text{undefined} & \text{if } \; n = i \\ n - 1 & \text{if } \; n > i \end{cases} \qquad \begin{aligned} \downarrow_i(ab) \quad &= \quad \downarrow_i(a)\ \downarrow_i(b) \\ \downarrow_i(\lambda a) \quad &= \quad \lambda\ \downarrow_{i+1}(a) \end{aligned}$$

Note. *Notice that* $\downarrow_i(a)$ *is well-defined iff* $i \notin FV(a)$.

[1]The proof can be downloaded from `http://www.mpi-sws.org/~beta/lambdar.v`

As for the $\updownarrow_i$ and $\uparrow_i$ meta-operators, we need a few lemmas to ensure a correct definition for the extension of the $\downarrow_i$ meta-operator to the ES case. Particularly, Lemma 28 (see Appendix B) is used for this purpose. The extension resembles those of the $\updownarrow_i$ and $\uparrow_i$ meta-operators:

$$\downarrow_i(a[b]) = \downarrow_{i+1}(a)[\downarrow_i(b)]$$

The *Garbage Collection* rule added to λre (GC) can be seen in Figure 2, and the resulting calculus is called $\lambda\text{re}_{\text{gc}}$ (which, as we will see, is isomorphic to λxgc [4]).

Finally, in order to mimic the behavior of λex [15], an analogue method for the composition of substitutions must be devised. In λex, composition is handled by one rule and one equation:

$$\begin{array}{llll} t[x:=u][y:=v] & \to_{(\text{Comp})} & t[y:=v][x:=u[y:=v]] & \text{if} \quad y \in \text{FV}(u) \\ t[x:=u][y:=v] & =_{\text{C}} & t[y:=v][x:=u] & \text{if} \quad y \notin \text{FV}(u) \wedge x \notin \text{FV}(v) \end{array}$$

The rule (Comp) is used when substitutions are *dependent*, and reasoning modulo C-equation is needed for *independent* substitutions. Since in λr-derived calculi there is no simple way of implementing an ordering of substitutions (remember: no indexes inside closures!), and thus no trivial path for the elimination of equation C exists, we need an analogue equation.

Let us start with the composition rule: in a term of the form $a[b][c]$, substitutions $[b]$ and $[c]$ are *dependent* iff $1 \in \text{FV}(b)$. In such a term, indexes 1 and 2 in a are being affected by $[b]$ and $[c]$, respectively. Consequently, if we were to reduce to a term of the form $a'[c'][b']$, a *swap* should be performed over a. Moreover, as substitution $[c]$ crosses the binder $[b]$, an index increment should also be done. Finally, since substitutions are dependent – that is, $[c]$ affects b –, b' should be $b[c]$. Then, we are left with the term $\updownarrow_1(a)[\uparrow_0(c)][b[c]]$.

For the equation, let us suppose we negate the composition condition (*i.e.,* $1 \notin \text{FV}(b)$). Using *Garbage Collection* in the last term, we have $\updownarrow_1(a)[\uparrow_0(c)][b[c]] \to_{(\text{GC})} \updownarrow_1(a)[\uparrow_0(c)][\downarrow_1(b)]$. It is important to notice that the condition in rule (Comp) is essential; that is: we cannot leave (Comp) unconditional and let (GC) do its magic: we would immediately generate infinite reductions, losing PSN. Thus, our composition rule and equation are:

$$\begin{array}{llll} a[b][c] & \to_{(\text{Comp})} & \updownarrow_1(a)[\uparrow_0(c)][b[c]] & \text{if} \quad 1 \in \text{FV}(b) \\ a[b][c] & =_{\text{D}} & \updownarrow_1(a)[\uparrow_0(c)][\downarrow_1(b)] & \text{if} \quad 1 \notin \text{FV}(b) \end{array}$$

Rules for the λrex-calculus can be seen in Figure 2. The relation rex_{p} is generated by the set of rules (App), (Lamb), (Var), (GC) and (Comp); $\lambda\text{rex}_{\text{p}}$ by (Beta) + rex_{p}. D-equivalence is the least equivalence and compatible relation generated by (EqD). Relations λrex (resp. rex) are obtained from $\lambda\text{rex}_{\text{p}}$ (resp. rex_{p}) modulo D-equivalence (thus specifying rewriting on D-equivalence classes). That is,

$$\forall a, a' \in \Lambda\text{re} : a \to_{(\lambda)\text{rex}} a' \iff \left(\exists b, b' \in \Lambda\text{re} : a =_{\text{D}} b \to_{(\lambda)\text{rex}_{\text{p}}} b' =_{\text{D}} a' \right)$$

We define λrex as the reduction system $(\Lambda\text{re}, \lambda\text{rex})$. We shall define λre and $\lambda\text{re}_{\text{gc}}$ next. Since the rule (VarR) does not belong to λrex, but only to λre and $\lambda\text{re}_{\text{gc}}$, we present it here:

$$\text{(VarR)} \qquad (n+1)[c] \to n$$

The relation re is generated by (App), (Lamb), (Var) and (VarR); λre by (Beta) + re; the relation re_{gc} by re + (GC); and $\lambda\text{re}_{\text{gc}}$ by (Beta) + re_{gc}. Finally, the λre and $\lambda\text{re}_{\text{gc}}$ calculi are the reduction systems $(\Lambda\text{re}, \lambda\text{re})$ and $(\Lambda\text{re}, \lambda\text{re}_{\text{gc}})$, respectively.

(EqD)	$a[b][c]$	$=$	$\updownarrow_1(a)[\uparrow_0(c)][\downarrow_1(b)]$	$(1 \notin \mathrm{FV}(b))$
(Beta)	$(\lambda a)\, b$	$\to$	$a[b]$	
(App)	$(a\, b)[c]$	$\to$	$a[c]\; b[c]$	
(Lamb)	$(\lambda a)[c]$	$\to$	$\lambda\, \updownarrow_1(a)[\uparrow_0(c)]$	
(Var)	$1\,[c]$	$\to$	c	
(GC)	$a[c]$	$\to$	$\downarrow_1(a)$	$(1 \notin \mathrm{FV}(a))$
(Comp)	$a[b][c]$	$\to$	$\updownarrow_1(a)[\uparrow_0(c)][b[c]]$	$(1 \in \mathrm{FV}(b))$

Figure 2: Equations and rules for the λrex-calculus

4 The isomorphisms

For the isomorphism between λex and λrex (and also between λx and λre; and between λxgc and $\lambda\mathrm{re}_{\mathrm{gc}}$), we must first give a translation from the set Λx (*i.e.,* the set of terms for λx, λxgc and λex; see e.g. [15] for the expected definition) to Λre, and vice versa. It is important to notice that our translations depend on a list of variables, which will determine the indexes of the free variables. All this work is inspired in a similar proof that shows the isomorphism between the λ and λ_{dB} calculi, found in [13].

Definition 16 (Translation from Λx to Λre)**.** For every $t \in \Lambda\mathrm{x}$, $n \in \mathbb{N}$, such that $\mathrm{FV}(t) \subseteq \{x_1, \ldots, x_n\}$, $\mathrm{w}_{[x_1,\ldots,x_n]} : \Lambda\mathrm{x} \to \Lambda\mathrm{re}$ is given inductively by:

$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{w}_{[x_1,\ldots,x_n]}(x) & = & \min\{j : x_j = x\} & \mathrm{w}_{[x_1,\ldots,x_n]}(\lambda x.t) & = & \lambda \mathrm{w}_{[x,x_1,\ldots,x_n]}(t) \\
\mathrm{w}_{[x_1,\ldots,x_n]}(t\, u) & = & \mathrm{w}_{[x_1,\ldots,x_n]}(t)\; \mathrm{w}_{[x_1,\ldots,x_n]}(u) & \mathrm{w}_{[x_1,\ldots,x_n]}(t[x := u]) & = & \mathrm{w}_{[x,x_1,\ldots,x_n]}(t)\left[\mathrm{w}_{[x_1,\ldots,x_n]}(u)\right]
\end{array}$$

Definition 17 (Translation from Λre to Λx)**.** For every $a \in \Lambda\mathrm{re}$, $n \in \mathbb{N}$, such that $\mathrm{FV}(a) \subseteq \{1, \ldots, n\}$, $\mathrm{u}_{[x_1,\ldots,x_n]} : \Lambda\mathrm{re} \to \Lambda\mathrm{x}$, with $\{x_1, \ldots, x_n\}$ different variables, is given inductively by:

$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{u}_{[x_1,\ldots,x_n]}(j) & = & x_j & \mathrm{u}_{[x_1,\ldots,x_n]}(\lambda a) & = & \lambda x.\mathrm{u}_{[x,x_1,\ldots,x_n]}(a) \\
\mathrm{u}_{[x_1,\ldots,x_n]}(a b) & = & \mathrm{u}_{[x_1,\ldots,x_n]}(a)\; \mathrm{u}_{[x_1,\ldots,x_n]}(b) & \mathrm{u}_{[x_1,\ldots,x_n]}(a[b]) & = & \mathrm{u}_{[x,x_1,\ldots,x_n]}(a)\,[x := \mathrm{u}_{[x_1,\ldots,x_n]}(b)]
\end{array}$$

with $x \notin \{x_1, \ldots, x_n\}$ in the cases of abstraction and closure.

Translations are correct w.r.t. α-equivalence. That is, α-equivalent Λx terms have the same image under $\mathrm{w}_{[x_1,\ldots,x_n]}$, and identical Λre terms have α-equivalent images under different choices of x for $\mathrm{u}_{[x_1,\ldots,x_n]}$. Besides, adding variables at the end of translation lists does not affect the result; thus, uniform translations w and u can be defined straightforwardly, depending only on a preset ordering of variables. See Appendix C for details.

We now state the isomorphisms:

Theorem 18 ($\lambda\mathrm{ex} \cong \lambda\mathrm{rex}$, $\lambda\mathrm{x} \cong \lambda\mathrm{re}$ and $\lambda\mathrm{xgc} \cong \lambda\mathrm{re}_{\mathrm{gc}}$)**.** The λex (resp. λx, λxgc) and λrex (resp. λre, $\lambda\mathrm{re}_{\mathrm{gc}}$) calculi are isomorphic. That is,

A. $\mathrm{w} \circ \mathrm{u} = \mathrm{Id}_{\Lambda\mathrm{re}} \wedge \mathrm{u} \circ \mathrm{w} = \mathrm{Id}_{\Lambda\mathrm{x}}$

B. $\forall t, u \in \Lambda\mathrm{x} : t \to_{\lambda\mathrm{ex}(\lambda\mathrm{x},\lambda\mathrm{xgc})} u \implies \mathrm{w}(t) \to_{\lambda\mathrm{rex}(\lambda\mathrm{re},\lambda\mathrm{re}_{\mathrm{gc}})} \mathrm{w}(u)$

C. $\forall a, b \in \Lambda\mathrm{re} : a \to_{\lambda\mathrm{rex}(\lambda\mathrm{re},\lambda\mathrm{re}_{\mathrm{gc}})} b \implies \mathrm{u}(a) \to_{\lambda\mathrm{ex}(\lambda\mathrm{x},\lambda\mathrm{xgc})} \mathrm{u}(b)$

Proof. This is actually a three-in-one theorem. Proofs require many auxiliary lemmas that assert the interaction between translations and meta-operators. See Appendix D for details. □

Finally, in order to show meta-confluence (MC) for λrex, meta-variables are added to the set of terms, and hence, functions and meta-operators are extended accordingly. Particularly, each metavariable is decorated with a set Δ of available free variables. This, in order to achieve an isomorphism with λex's corresponding extension (c.f. [15]). Extensions are as follows:

1. Set of terms $\Lambda\mathrm{re_{op}}$: $a ::= n \mid \mathrm{X}_\Delta \mid a\,a \mid \lambda a \mid a[a] \qquad (n \in \mathbb{N}_{>0}, \mathrm{X} \in \{X,Y,Z,\ldots\}, \Delta \in \mathscr{P}(\mathbb{N}_{>0}))$
2. Free variables of a metavariable: $\mathrm{FV}(\mathrm{X}_\Delta) = \Delta$
3. Swap over a metavariable: $\updownarrow_i(\mathrm{X}_\Delta) = \mathrm{X}_{\Delta'} \qquad$ with $\Delta' = \Delta_{<i} \cup \Delta_{>i+1} \cup (\Delta_{=i}+1) \cup (\Delta_{=i+1}-1)$
4. Increment over a metavariable: $\uparrow_i(\mathrm{X}_\Delta) = \mathrm{X}_{\Delta'} \qquad$ with $\Delta' = \Delta_{\leq i} \cup (\Delta_{>i}+1)$
5. Decrement over a metavariable: $\downarrow_i(\mathrm{X}_\Delta) = \begin{cases} \mathrm{X}_{\Delta'} \quad \text{with } \Delta' = \Delta_{<i} \cup (\Delta_{>i}-1) & \text{if } i \notin \Delta \\ \text{undefined} & \text{if } i \in \Delta \end{cases}$
6. Translation from $\Lambda\mathrm{x_{op}}$ to $\Lambda\mathrm{re_{op}}$: $\mathrm{w}_{[x_1,\ldots,x_n]}(\mathrm{X}_\Delta) = \mathrm{X}_{\Delta'} \qquad$ with $\Delta' = \{\mathrm{w}_{[x_1,\ldots,x_n]}(x) : x \in \Delta\}$
7. Translation from $\Lambda\mathrm{re_{op}}$ to $\Lambda\mathrm{x_{op}}$: $\mathrm{u}_{[x_1,\ldots,x_n]}(\mathrm{X}_\Delta) = \mathrm{X}_{\Delta'} \qquad$ with $\Delta' = \{\mathrm{u}_{[x_1,\ldots,x_n]}(j) : j \in \Delta\}$

Theorem 19. The λrex and λex calculi on open terms are isomorphic.

Proof. This is proved as an extension of the proof for Theorem 18, considering the new case. A few simple lemmas about how meta-operators alter the set of free variables are needed. We refer the reader to [18], chapter 6, section 3 for details (space constraints disallow further technicality here). □

As a direct consequence of theorems 18 and 19, we have:

Corollary 20 (Preservation of properties)**.** The λex (resp. λx, λxgc) and λrex (resp. λre, $\lambda\mathrm{re_{gc}}$) have the same properties. In particular, this implies λrex has, among other properties, Sim, PSN and MC.

Proof sketch for e.g. PSN in λrex. Assume PSN does not hold in λrex. Then, there exists $a \in \mathrm{SN}_{\lambda_{\mathrm{dB}}}$ s.t. $a \notin \mathrm{SN}_{\lambda\mathrm{rex}}$. Besides, $a \in \mathrm{SN}_{\lambda_{\mathrm{dB}}}$ implies $\mathrm{u}(a) \in \mathrm{SN}_\lambda$. Therefore, by PSN of λex [15], $\mathrm{u}(a) \in \mathrm{SN}_{\lambda\mathrm{ex}}$. Now, since $a \notin \mathrm{SN}_{\lambda\mathrm{rex}}$, there exists an infinite reduction $a \to_{\lambda\mathrm{rex}} a_1 \to_{\lambda\mathrm{rex}} a_2 \to_{\lambda\mathrm{rex}} \cdots$. Thus, by Theorem 18, we have $\mathrm{u}(a) \to_{\lambda\mathrm{ex}} \mathrm{u}(a_1) \to_{\lambda\mathrm{ex}} \mathrm{u}(a_2) \to_{\lambda\mathrm{ex}} \cdots$, contradicting the fact that $\mathrm{u}(a) \in \mathrm{SN}_{\lambda\mathrm{ex}}$. □

5 Related work

It is important to mention that, even though independently discovered, the swapping mechanism introduced in this article was first depicted by de Bruijn for his ES calculus $C\lambda\xi\phi$ [6], and, later, updated w.r.t. notation – $\lambda\xi\phi$ – and compared to $\lambda\upsilon$ in [2]. We will now briefly discuss the main differences between these calculi and our swapping-based approach.

Firstly, neither $C\lambda\xi\phi$ nor $\lambda\xi\phi$ have composition of substitutions nor Garbage Collection, two keys for the accomplishment of meta-confluence. In that sense, these two calculi only resemble closely our first λr-based ES calculus: λre. Thus, both $\lambda\mathrm{re_{gc}}$ and λrex represent a relevant innovation for swapping-based formalisms, specially considering the fact that, as far as we know, no direct successor of $C\lambda\xi\phi$ nor $\lambda\xi\phi$ was found to satisfy PSN and MC.

As a second fundamental difference, both $C\lambda\xi\phi$ and $\lambda\xi\phi$ are entirely explicit formalisms. In the end, internalizing meta-operations is desirable, both theoretically and practically; nevertheless, the presence of meta-operations in λre, $\lambda\mathrm{re_{gc}}$ and λrex are mandatory for the accomplishment of isomorphisms w.r.t. λx, λxgc and λex, respectively. Particularly, the isomorphism between λex and λrex represents a step forward in the explicit substitutions area. Moreover, these isomorphisms – impossible in the case of $C\lambda\xi\phi$ and $\lambda\xi\phi$ – allow simple and straightforward proofs for every single property enjoyed by the calculi.

Last but not least, in $C\lambda\xi\phi$ as well as in $\lambda\xi\phi$, swap and increment operations are implemented by means of a special sort of substitution that only operates on indexes (c.f. [2]). Even though undoubtedly a very clever setting for these operations – specially compared to ours, much more conservative –, the fact is that we still use meta-operations. With this in mind, it may be the case that de Bruijn's formulation for both the swap and increment operations, if taken to the meta-level, would lead to *exactly the same* functional relations between terms than those defined by our method. Consequently, this difference loses importance in the presence of meta-operations. Nevertheless, if swap and increment meta-operations were to be made explicit, a deep comparison between our approach and de Bruijn's should be carried out before deciding for the use of either.

6 Conclusions and further work

We have presented λrex, an ES calculus with *de Bruijn* indexes that is isomorphic to λex, a formalism with variable names that fulfills a whole set of interesting properties. As a consequence of the isomorphism, λrex inherits all of λex's properties. This, together with a simple notation makes it, as far as we know, the first calculus of its kind. Besides, the λre and $\lambda\mathrm{re}_{\mathrm{gc}}$ calculi (isomorphic to λx and λxgc, respectively) were also introduced. The development was based on a novel presentation of the classical λ_{dB}. Given the homogeneity of definitions and proofs, not only for λr and λrex, but also for λre and $\lambda\mathrm{re}_{\mathrm{gc}}$, we think we found a truly *natural* bridge between named and indexed formalisms. We believe this opens a new set of possibilities in the area: either by translating and studying existing calculi with good properties; or by rethinking old calculi from a different perspective (*i.e.,* with λr's concept in mind).

Work is yet to be done in order to get a more suitable theoretical tool for implementation purposes, for unary closures and equations still make such a task hard. In this direction, a mix of ideas from λrex and calculi with *n*-ary substitutions (*i.e.,* $\lambda\sigma$-styled calculi) may lead to the solution of both issues. Particularly, a swap-based $\lambda\sigma_{\Uparrow}$ [7] could be an option. This comes from the following observation: in $\lambda\sigma_{\Uparrow}$, the (Lamb) rule is:

$$\text{(Lamb)} \qquad (\lambda a)[s] \to \lambda a[\Uparrow(s)]$$

where the intuitive semantics of $\Uparrow(s)$ is: $1 \cdot (s \circ \uparrow)$. We observe here that *this is not nameless*! The reason is that, even though there are no explicit indexes inside closures, this lift operation resembles closely the classic definition of the λ_{dB} calculus (particularly, leaving lower indexes untouched). Thus, we propose replacing this rule by one of the form:

$$\text{(Lamb)} \qquad (\lambda a)[s] \to \lambda \Updownarrow(a)[\Uparrow(s)]$$

with the semantics of $\Uparrow(s)$ being $s \circ \uparrow$, and that of $\Updownarrow(a)$ being *swapping* a's indexes in concordance with the substitution s, therefore mimicking λr's behavior. This approach is still in its early days, but we feel it is quite promising.

In a different line of work, the explicitation of meta-operators may also come to mind: we think this is not a priority, because the main merit of λrex is evidencing the accessory nature of index updates.

From a different perspective, an attempt to use λrex in proof assistants or higher order unification [8] implementations may be taken into account. In such a case, a typed version of λrex should be developed as well. Also, adding an η rule to λrex should be fairly simple using the decrement meta-operator. Finally, studying the possible relation between these *swapping-based* formalisms and nominal logic or nominal rewriting (see e.g. [10, 9]) could be an interesting approach in gathering a deeper understanding of λr's underlying logic.

Acknowledgements: Special thanks to Delia Kesner for valuable discussions and insight on the subject; as well as to the anonymous referees for their very useful comments.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lévy (1991): *Explicit Substitutions*. *J. Funct. Prog.* 1, pp. 31–46.

[2] Z. Benaissa, D. Briaud, P. Lescanne & J. Rouyer-Degli (1996): *$\lambda\upsilon$, a Calculus of Explicit Substitutions which Preserves Strong Normalisation*. *J. Funct. Prog.* 6(5), pp. 699–722.

[3] R. Bloo & H. Geuvers (1999): *Explicit substitution on the edge of strong normalization*. *Theor. Comput. Sci.* 211(1-2), pp. 375–395.

[4] R. Bloo & K. H. Rose (1995): *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*. In: *CSN-95: Computing Science in the Netherlands*, pp. 62–72.

[5] N. G. de Bruijn (1972): *Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*. *Indagationes Mathematicae* 34, pp. 381–392.

[6] N. G. de Bruijn (1978): *A namefree λ calculus with facilities for internal definition of expressions and segments*. *Tech. Rep. TH-Report 78-WSK-03, Dept. of Mathematics, Technical University of Eindhoven* .

[7] P.-L. Curien, Th. Hardin & J.-J. Lévy (1996): *Confluence properties of weak and strong calculi of explicit substitutions*. *Journal of the ACM* 43(2), pp. 362–397.

[8] G. Dowek, Th. Hardin & C. Kirchner (2000): *Higher order unification via explicit substitutions*. *Inf. Comput.* 157(1-2), pp. 183–235.

[9] M. Fernández & M. J. Gabbay (2007): *Nominal rewriting*. *Inf. Comput.* 205(6), pp. 917–965.

[10] M. J. Gabbay & A. M. Pitts (2002): *A new approach to abstract syntax with variable binding*. *Formal Aspects of Computing* 13, pp. 341–363.

[11] F. Kamareddine & A. Ríos (1995): *A Lambda-Calculus à la de Bruijn with Explicit Substitutions*. In: *PLILP '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs, Lecture Notes in Computer Science 982*, pp. 45–62.

[12] F. Kamareddine & A. Ríos (1997): *Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms*. *J. Funct. Prog.* 7(4), pp. 395–420.

[13] F. Kamareddine & A. Ríos (1998): *Bridging de Bruijn Indices and Variable Names in Explicit Substitutions Calculi*. *Logic Journal of the IGPL* 6(6), pp. 843–874.

[14] D. Kesner (2008): *Perpetuality for Full and Safe Composition (in a Constructive Setting)*. In: *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, Springer-Verlag, Berlin, Heidelberg, pp. 311–322.

[15] D. Kesner (2009): *A Theory of Explicit Substitutions with Safe and Full Composition*. *Logical Methods in Computer Science* 5(3:1), pp. 1–29.

[16] P. Lescanne (1994): *From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions*. In: *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages*, ACM, New York, NY, USA, pp. 60–69.

[17] P.-A. Melliès (1995): *Typed lambda-calculi with explicit substitutions may not terminate*. In: *TLCA '95: Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science 902*, pp. 328–334.

[18] A. Mendelzon (2010). *Una curiosa versión de λ_{dB} basada en "swappings": aplicación a traducciones entre cálculos de sustituciones explícitas con nombres e índices*. Master's thesis, FCEyN, Univ. de Buenos Aires. Available at `http://publi.dc.uba.ar/publication/pdffile/128/tesis_amendelzon.pdf`.

[19] A. Ríos (1993): *Contributions à l'étude des Lambda-calculus avec Substitutions Explicites*. Ph.D. thesis, Université Paris 7.

[20] K. H. Rose, R. Bloo & F. Lang (2009): *On explicit substitution with names*. Technical Report, IBM. Available at `http://domino.research.ibm.com/library/cyberdig.nsf/papers/39D13836281BDD328525767F0056CE65`.

A Proofs for the $\lambda_{\mathbf{dB}} = \lambda\mathbf{r}$ assertion

We first show the auxiliary lemmas that allow us to prove the main theorem of Subsection 2.3.

Lemma 21. For every $a \in \Lambda_{\mathrm{dB}}$, $n \in \mathbb{N}_{>0}$, $\Uparrow^{n-1}(a) = \mathrm{U}_0^n(a)$

Proof. Easy induction on n, using that $l \le k < l+j \implies \mathrm{U}_k^i(\mathrm{U}_l^j(a)) = \mathrm{U}_l^{j+i-1}(a)$ (c.f. [12], lemma 6), and the fact that $\uparrow_0(a) = \mathrm{U}_0^2(a)$. □

Lemma 22. For every $m, i \in \mathbb{N}_{>0}$, $n \in \mathbb{N}$:

1. $m > n+i \implies \Updownarrow_i^n(m) = m$
2. $i \le m < n+i \implies \Updownarrow_i^n(m) = m+1$
3. $\Updownarrow_i^n(n+i) = i$

Proof. Easy inductions on n. □

Lemma 23. For every $a, b \in \Lambda_{\mathrm{dB}}$, $n \in \mathbb{N}$, $i \in \mathbb{N}_{>0}$:

1. $\Updownarrow_i^n(ab) = \Updownarrow_i^n(a)\ \Updownarrow_i^n(b)$
2. $\Updownarrow_i^n(\lambda a) = \lambda\ \Updownarrow_{i+1}^n(a)$
3. $(\lambda\ \Updownarrow_2^n(a))\{\Uparrow^n(b)\} = \lambda\ \Updownarrow_1^{n+1}(a)\{\Uparrow^{n+1}(b)\}$

Proof. Easy inductions on n. □

We now restate and prove the main theorem:

Theorem (13). For every $a, b \in \Lambda_{\mathrm{dB}}$, $n \in \mathbb{N}_{>0}$, we have that $a\{\{n \leftarrow b\}\} = \Updownarrow_1^{n-1}(a)\{\Uparrow^{n-1}(b)\}$.

Proof. Induction on a.

- $a = m \in \mathbb{N}_{>0}$. Then, $a\{\{n \leftarrow b\}\} = m\{\{n \leftarrow b\}\} = \begin{cases} m-1 & \text{if } m > n \\ \mathrm{U}_0^n(b) & \text{if } m = n \\ m & \text{if } m < n \end{cases}$

  We consider each case separately:

  1. $m > n \implies \Updownarrow_1^{n-1}(m)\{\Uparrow^{n-1}(b)\} \underset{\text{L.22.1}}{=} m\{\Uparrow^{n-1}(b)\} \underset{m>n\ge 1}{=} m-1$
  2. $m = n \implies \Updownarrow_1^{n-1}(n)\{\Uparrow^{n-1}(b)\} \underset{\text{L.22.3}}{=} 1\{\Uparrow^{n-1}(b)\} \underset{\text{def}}{=} \Uparrow^{n-1}(b) \underset{\text{L.21}}{=} \mathrm{U}_0^n(b)$
  3. $m < n \implies \Updownarrow_1^{n-1}(m)\{\Uparrow^{n-1}(b)\} \underset{\text{L.22.2}}{=} m+1\{\Uparrow^{n-1}(b)\} \underset{m+1>1}{=} m$

  Then,
  $$m\{\{n \leftarrow b\}\} = \Updownarrow_1^{n-1}(m)\{\Uparrow^{n-1}(b)\}$$

- $a = cd$, $c, d \in \Lambda_{\mathrm{dB}}$. Use inductive hypothesis and Lemma 23.1.

- $a = \lambda c$, $c \in \Lambda_{\mathrm{dB}}$. Then,

$$a\{\{n \leftarrow b\}\} = (\lambda c)\{\{n \leftarrow b\}\} \underset{\text{def}}{=} \lambda c\{\{n+1 \leftarrow b\}\} \underset{\text{HI}}{=} \lambda\ \Updownarrow_1^n(c)\{\Uparrow^n(b)\} \underset{\text{L.23.3}}{=}$$
$$(\lambda\ \Updownarrow_2^{n-1}(c))\{\Uparrow^{n-1}(b)\} \underset{\text{L.23.2}}{=} \Updownarrow_1^{n-1}(\lambda c)\{\Uparrow^{n-1}(b)\} = \Updownarrow_1^{n-1}(a)\{\Uparrow^{n-1}(b)\}$$

□

B Extension lemmas for the $\updownarrow_i$, $\uparrow_i$ and $\downarrow_i$ meta-operators

Lemma 24. For every $a \in \Lambda_{\mathrm{dB}}$, $i, j \in \mathbb{N}_{>0}$, $k \in \mathbb{N}$:

1. $k < i \implies \updownarrow_{i+1}(\uparrow_k(a)) = \uparrow_k(\updownarrow_i(a))$
2. $j \geq 2 \implies \updownarrow_{i+j}(\updownarrow_i(a)) = \updownarrow_i(\updownarrow_{i+j}(a))$

Proof. Easy induction on a. $\square$

Lemma 25. For every $a \in \Lambda_{\mathrm{dB}}$, $i \in \mathbb{N}_{>0}$, $j \in \mathbb{N}$:

1. $i \geq j+2 \wedge i-1 \notin \mathrm{FV}(a) \implies \downarrow_i(\uparrow_j(a)) = \uparrow_j(\downarrow_{i-1}(a))$
2. $j \geq 2 \wedge i+j \notin \mathrm{FV}(a) \implies \downarrow_{i+j}(\updownarrow_i(a)) = \updownarrow_i(\downarrow_{i+j}(a))$

Proof. Easy induction on a. $\square$

Lemma 26. For every $a, b \in \Lambda_{\mathrm{dB}}$, $i \in \mathbb{N}_{>0}$, $\updownarrow_i(a\{b\}) = \updownarrow_{i+1}(a)\{\updownarrow_i(b)\}$

Proof. Easy induction on a, using Lemma 24. $\square$

Lemma 27. For every $a, b \in \Lambda_{\mathrm{dB}}$, $i \in \mathbb{N}$, $\uparrow_i(a\{b\}) = \uparrow_{i+1}(a)\{\uparrow_i(b)\}$

Proof. Use that $\mathrm{U}^i_k(a\{\{1 \leftarrow b\}\}) = \mathrm{U}^i_{k+1}(a)\{\{1 \leftarrow \mathrm{U}^i_k(b)\}\}$ (c.f. [12], Lemma 10 with $n = 1$), the fact that $\uparrow_i(a) = \mathrm{U}^2_i(a)$ and Corollary 14. $\square$

Lemma 28. For every $a, b \in \Lambda_{\mathrm{dB}}$, $i \in \mathbb{N}_{>0}$, $i+1 \notin \mathrm{FV}(a) \wedge i \notin \mathrm{FV}(b)$, we have that $\downarrow_i(a\{b\}) = \downarrow_{i+1}(a)\{\downarrow_i(b)\}$.

Proof. Easy induction on a, using Lemma 25. $\square$

C Correction proofs for translations

We show the lemmas necessary to prove that the translations given (*i.e.,* $\mathrm{w}_{[x_1,\ldots,x_n]}$ and $\mathrm{u}_{[x_1,\ldots,x_n]}$) are correct w.r.t. α-equivalence.

Lemma 29. For every $t \in \Lambda\mathrm{x}$, $n \in \mathbb{N}$ such that $\mathrm{FV}(t) \subseteq \{x_1, \ldots, x_n\}$, we have that $\forall y \notin \{x_1, \ldots, x_n\}$, $z \in \{x_1, \ldots, x_n\}$, $\mathrm{w}_{[x_1,\ldots,x_n]}(t) = \mathrm{w}_{[x_1,\ldots,x_{k-1},y,x_{k+1},\ldots,x_n]}(t\{z := y\})$, with $k = \min\{j : x_j = z\}$.

Proof. Easy induction on t, but using the non-Barendregt-variable-convention definition for the meta-substitution operation (otherwise, we would be assuming that $t =_\alpha u \implies \mathrm{w}_{[x_1,\ldots,x_n]}(t) = \mathrm{w}_{[x_1,\ldots,x_n]}(u)$, which is what we ultimately want to prove). See e.g. [3] for an expected definition. $\square$

Lemma 30. For every $t, u \in \Lambda\mathrm{x}$, $n \in \mathbb{N}$ such that $\mathrm{FV}(t) \subseteq \{x_1, \ldots, x_n\}$, we have that $t =_\alpha u \implies \mathrm{w}_{[x_1,\ldots,x_n]}(t) = \mathrm{w}_{[x_1,\ldots,x_n]}(u)$. Notice that $\mathrm{w}_{[x_1,\ldots,x_n]}(u)$ is well-defined, since $t =_\alpha u \implies \mathrm{FV}(t) = \mathrm{FV}(u)$.

Proof. Easy induction on t, using Lemma 29. Once again, the non-Barendregt-variable-convention definition for the meta-substitution operation must be used here. $\square$

Lemma 31. For every $a \in \Lambda\mathrm{re}$, $n \in \mathbb{N}$, $\{x_1, \ldots, x_n\}$ distinct variables such that $\mathrm{FV}(a) \subseteq \{1, \ldots, n\}$, we have that $\forall y \notin \{x_1, \ldots, x_n\}$, $1 \leq k \leq n : \mathrm{u}_{[x_1,\ldots,x_n]}(a)\{x_k := y\} =_\alpha \mathrm{u}_{[x_1,\ldots,x_{k-1},y,x_{k+1},\ldots,x_n]}(a)$.

Proof. Easy induction on a. $\square$

Lemma 32. For every $a,b \in \Lambda\mathrm{re}$, $n \in \mathbb{N}$, $\{x_1,\dots,x_n\}$ distinct variables, $x,y \notin \{x_1,\dots,x_n\}$ such that $\mathrm{FV}(a) \subseteq \{1,\dots,n\}$, we have that:

1. $\lambda x.\mathrm{u}_{[x,x_1,\dots,x_n]}(a) =_\alpha \lambda y.\mathrm{u}_{[y,x_1,\dots,x_n]}(a)$
2. $\mathrm{u}_{[x,x_1,\dots,x_n]}(a)\,[x := \mathrm{u}_{[x_1,\dots,x_n]}(b)] =_\alpha \mathrm{u}_{[y,x_1,\dots,x_n]}(a)\,[y := \mathrm{u}_{[x_1,\dots,x_n]}(b)]$

Proof. Direct in both cases, using the α-equivalence definition and Lemma 31. □

Last, we show two lemmas that assert that adding variables at the end of translation lists does not affect the result of the translation and, thus, gives the possibility of defining uniform translations that depend only on a preset ordering of variables.

Lemma 33. For every $t \in \Lambda\mathrm{x}$ such that $\mathrm{FV}(t) \subseteq \{x_1,\dots,x_n\}$, and for every $\{y_1,\dots,y_m\} \subset \mathbb{V}$, we have that $\mathrm{w}_{[x_1,\dots,x_n]}(t) = \mathrm{w}_{[x_1,\dots,x_n,y_1,\dots,y_m]}(t)$.

Proof. Easy induction on t. □

Lemma 34. For every $a \in \Lambda\mathrm{re}$, $\{x_1,\dots,x_n\}$ distinct variables such that $\mathrm{FV}(a) \subseteq \{1,\dots,n\}$, and for every $\{y_1,\dots,y_m\}$ distinct variables such that $\{x_1,\dots,x_n\} \cap \{y_1,\dots,y_m\} = \emptyset$, we have that $\mathrm{u}_{[x_1,\dots,x_n]}(a) =_\alpha \mathrm{u}_{[x_1,\dots,x_n,y_1,\dots,y_m]}(a)$.

Proof. Easy induction on a. □

Last, we show the definitions for uniform translations.

Definition 35 (Uniform translation from $\Lambda\mathrm{x}$ to $\Lambda\mathrm{re}$)**.** Given an enumeration $[v_1,v_2,\dots]$ of $\mathbb{V}$, for every $t \in \Lambda\mathrm{x}$, $n \in \mathbb{N}$ such that $\mathrm{FV}(t) \subseteq \{v_1,\dots,v_n\}$, we define $\mathrm{w} : \Lambda\mathrm{x} \to \Lambda\mathrm{re}$ as: $\mathrm{w}(t) = \mathrm{w}_{[v_1,\dots,v_n]}(t)$.

Definition 36 (Uniform translation from $\Lambda\mathrm{re}$ to $\Lambda\mathrm{x}$)**.** Given an enumeration $[v_1,v_2,\dots]$ of $\mathbb{V}$, for every $a \in \Lambda\mathrm{re}$, $n \in \mathbb{N}$ such that $\mathrm{FV}(a) \subseteq \{1,\dots,n\}$, we define $\mathrm{u} : \Lambda\mathrm{re} \to \Lambda\mathrm{x}$ as: $\mathrm{u}(a) = \mathrm{u}_{[v_1,\dots,v_n]}(a)$.

D Isomorphisms proofs

In order to prove Theorem 18, we must show:

A. $\mathrm{w} \circ \mathrm{u} = \mathrm{Id}_{\Lambda\mathrm{re}} \wedge \mathrm{u} \circ \mathrm{w} = \mathrm{Id}_{\Lambda\mathrm{x}}$

B. $\forall t,u \in \Lambda\mathrm{x} : t \to_{\lambda\mathrm{ex}(\lambda\mathrm{x},\lambda\mathrm{xgc})} u \implies \mathrm{w}(t) \to_{\lambda\mathrm{rex}(\lambda\mathrm{re},\lambda\mathrm{re}_{\mathrm{gc}})} \mathrm{w}(u)$

C. $\forall a,b \in \Lambda\mathrm{re} : a \to_{\lambda\mathrm{rex}(\lambda\mathrm{re},\lambda\mathrm{re}_{\mathrm{gc}})} b \implies \mathrm{u}(a) \to_{\lambda\mathrm{ex}(\lambda\mathrm{x},\lambda\mathrm{xgc})} \mathrm{u}(b)$

For Part A, the following two lemmas are needed.

Lemma 37. For every $t \in \Lambda\mathrm{x}$, $a \in \Lambda\mathrm{re}$, $\{x_1,\dots,x_n\}$ variables, $\{y_1,\dots,y_m\}$ distinct variables:

1. $\mathrm{FV}(t) \subseteq \{x_1,\dots,x_n\} \implies \mathrm{FV}\big(\mathrm{w}_{[x_1,\dots,x_n]}(t)\big) \subseteq \{1,\dots,n\}$
2. $\mathrm{FV}(a) \subseteq \{1,\dots,m\} \implies \mathrm{FV}\big(\mathrm{u}_{[y_1,\dots,y_m]}(a)\big) \subseteq \{y_1,\dots,y_m\}$

Proof. Easy inductions on t and a, respectively. □

Lemma 38. For every $a \in \Lambda\mathrm{re}$, $t \in \Lambda\mathrm{x}$:

1. $\mathrm{w}(\mathrm{u}(a)) = a$
2. $\mathrm{u}(\mathrm{w}(t)) =_\alpha t$

Proof. Easy inductions on a and t, respectively, using Lemma 37. □

Next, to prove Part B of the theorem, we need several auxiliary lemmas, that we now state.

Lemma 39. For every $\{x_1,\dots,x_n\}$, $y \in \{x_1,\dots,x_n\}$, $x \notin \{x_1,\dots,x_n\}$, $\mathrm{w}_{[x,x_1,\dots,x_n]}(y) = \mathrm{w}_{[x_1,\dots,x_n]}(y)+1$.

Proof. Direct, using w's definition. □

Lemma 40. For every $t \in \Lambda\mathrm{x}$, $i \in \mathbb{N}_{>0}$ such that $\mathrm{FV}(t) \subseteq \{x_1,\dots,x_n\} \wedge i < n \wedge x_i \neq x_{i+1}$, $\updownarrow_i(\mathrm{w}_{[x_1,\dots,x_i,x_{i+1},\dots,x_n]}(t)) = \mathrm{w}_{[x_1,\dots,x_{i+1},x_i,\dots,x_n]}(t)$.

Proof. Easy induction on t. □

Lemma 41. For every $t \in \Lambda\mathrm{x}$, $m \in \mathbb{N}$, $x \in \mathbb{V}$ such that $\mathrm{FV}(t) \subseteq \{x_1,\dots,x_n\} \wedge m \leq n \wedge x \notin \{x_1,\dots,x_n\}$, $\mathrm{w}_{[x_1,\dots,x_m,x,x_{m+1},\dots,x_n]}(t) = \uparrow_m(\mathrm{w}_{[x_1,\dots,x_n]}(t))$.

Proof. Easy induction on t. □

Lemma 42. For every $t \in \Lambda\mathrm{x}$, $m \in \mathbb{N}$, $x \in \mathbb{V}$ such that $\mathrm{FV}(t) \subseteq \{x_1,\dots,x_n\} \wedge 1 \leq m \leq n+1$:

1. $x \notin \{x_1,\dots,x_n\} \implies m \notin \mathrm{FV}\big(\mathrm{w}_{[x_1,\dots,x_{m-1},x,x_m,\dots,x_n]}(t)\big)$
2. $x \notin \{x_1,\dots,x_{m-1}\} \wedge x \in \mathrm{FV}(t) \implies m \in \mathrm{FV}\big(\mathrm{w}_{[x_1,\dots,x_{m-1},x,x_m,\dots,x_n]}(t)\big)$
3. $x \notin \{x_1,\dots,x_n\} \implies \mathrm{w}_{[x_1,\dots,x_n]}(t) = \downarrow_m(\mathrm{w}_{[x_1,\dots,x_{m-1},x,x_m,\dots,x_n]}(t))$

Proof. Easy inductions on t. □

Given the auxiliary lemmas, we proceed to prove Part B of the isomorphism theorem. Item 1 of the next lemma is enough to prove the reduction preservation under translation w for the λre and $\lambda\mathrm{re}_{\mathrm{gc}}$ calculi. For λrex, Item 2 – showing the preservation of the equivalence relations under translation w – is also needed. Then, preservation for λrex follows immediately from the definition of reduction modulo an equivalence relation.

Lemma 43. For every $t,u \in \Lambda\mathrm{x}$:

1. $t \rightarrow_{\mathrm{Bx}(\lambda\mathrm{x},\lambda\mathrm{xgc})} u \implies \mathrm{w}(t) \rightarrow_{\lambda\mathrm{rex_p}(\lambda\mathrm{re},\lambda\mathrm{re_{gc}})} \mathrm{w}(u)$
2. $t =_{\mathrm{C}} u \implies \mathrm{w}(t) =_{\mathrm{D}} \mathrm{w}(u)$

Proof. **Part 1.** Induction on t. The only interesting cases are those of the explicit substitution when the reduction takes place at the root. The rest of the cases are either trivial or easily shown by using the inductive hypothesis. We will show the explicit substitution case in which reduction is done by using the (Comp) rule. The other two relevant cases, (Lamb) and (GC), omitted here for a matter of space, are proved in a similar fashion. Since we are working in the explicit substitution case, t is of the form $t_1[x:=t_2]$. Now, as the (Comp) rule is used, we have that:

$$t_1[x:=t_2] = t_3[y:=t_4][x:=t_2] \underset{\text{(Comp)}}{\rightarrow_{\mathrm{Bx}}} t_3[x:=t_2][y:=t_4[x:=t_2]] = u$$

with $x \in \mathrm{FV}(t_4)$. By the variable convention, we assume $x \neq y \wedge y \notin \{x_1,\dots,x_n\}$. Thus,

$$\mathrm{w}_{[x_1,\dots,x_n]}(t_3[y:=t_4][x:=t_2]) \underset{\text{def}}{=} \mathrm{w}_{[x,x_1,\dots,x_n]}(t_3[y:=t_4])\big[\mathrm{w}_{[x_1,\dots,x_n]}(t_2)\big] \underset{\text{def}}{=}$$

$$\mathrm{w}_{[y,x,x_1,\dots,x_n]}(t_3)\big[\mathrm{w}_{[x,x_1,\dots,x_n]}(t_4)\big]\big[\mathrm{w}_{[x_1,\dots,x_n]}(t_2)\big] \quad \underset{\text{L.42.2, } x\in\mathrm{FV}(t_4)\text{, (Comp)}}{\rightarrow_{\lambda\mathrm{rex_p}}}$$

$$\updownarrow_1(\mathrm{w}_{[y,x,x_1,\dots,x_n]}(t_3))\big[\uparrow_0(\mathrm{w}_{[x_1,\dots,x_n]}(t_2))\big]\big[\mathrm{w}_{[x,x_1,\dots,x_n]}(t_4)\big[\mathrm{w}_{[x_1,\dots,x_n]}(t_2)\big]\big] \underset{\text{L.40, } x\neq y}{=}$$

$$\mathrm{w}_{[x,y,x_1,\ldots,x_n]}(t_3)\left[\uparrow_0(\mathrm{w}_{[x_1,\ldots,x_n]}(t_2))\right]\left[\mathrm{w}_{[x,x_1,\ldots,x_n]}(t_4)\left[\mathrm{w}_{[x_1,\ldots,x_n]}(t_2)\right]\right] \underset{\text{L.41, } y\notin \mathrm{FV}(t_2)}{=}$$

$$\mathrm{w}_{[x,y,x_1,\ldots,x_n]}(t_3)\left[\mathrm{w}_{[y,x_1,\ldots,x_n]}(t_2)\right]\left[\mathrm{w}_{[x,x_1,\ldots,x_n]}(t_4)\left[\mathrm{w}_{[x_1,\ldots,x_n]}(t_2)\right]\right] \underset{\text{def}}{=}$$

$$\mathrm{w}_{[x,y,x_1,\ldots,x_n]}(t_3)\left[\mathrm{w}_{[y,x_1,\ldots,x_n]}(t_2)\right]\left[\mathrm{w}_{[x_1,\ldots,x_n]}(t_4[x:=t_2])\right] \underset{\text{def}}{=}$$

$$\mathrm{w}_{[y,x_1,\ldots,x_n]}(t_3[x:=t_2])\left[\mathrm{w}_{[x_1,\ldots,x_n]}(t_4[x:=t_2])\right] \underset{\text{def}}{=} \mathrm{w}_{[x_1,\ldots,x_n]}(t_3[x:=t_2][y:=t_4[x:=t_2]])$$

Part 2. Induction on the inference of $t =_{\mathrm{C}} u$. The only interesting case is when the actual equation is used. Then, $t = t_1[y:=t_2][x:=t_3] =_{\mathrm{C}} t_1[x:=t_3][y:=t_2] = u$, with $x \neq y \wedge x \notin \mathrm{FV}(t_2) \wedge y \notin \mathrm{FV}(t_3)$. By the variable convention, assume that $\{x,y\} \cap \{x_1,\ldots,x_n\} = \emptyset$. Proceed in a similar way than that of the proof of Part 1 in the (Comp) case. □

Finally, to prove Part C of the isomorphism theorem, we also need several auxiliary lemmas analogue to those used for Part B. We will now state them.

Lemma 44. For every $a \in \Lambda\mathrm{re}$, $i \in \mathbb{N}_{>0}$, $\{x_1,\ldots,x_n\}$ distinct variables such that $\mathrm{FV}(a) \subseteq \{1,\ldots,n\}$ $\wedge\, i < n$, we have that $\mathrm{u}_{[x_1,\ldots,x_i,x_{i+1},\ldots,x_n]}(a) =_\alpha \mathrm{u}_{[x_1,\ldots,x_{i+1},x_i,\ldots,x_n]}(\updownarrow_i(a))$.

Proof. Easy induction on a. □

Lemma 45. For every $a \in \Lambda\mathrm{re}$, $m \in \mathbb{N}$, $\{x_1,\ldots,x_n\}$ distinct variables, $x \notin \{x_1,\ldots,x_n\}$ such that $\mathrm{FV}(a) \subseteq \{1,\ldots,n\} \wedge m \leq n$, we have that $\mathrm{u}_{[x_1,\ldots,x_m,x,x_{m+1},\ldots,x_n]}(\uparrow_m(a)) =_\alpha \mathrm{u}_{[x_1,\ldots,x_n]}(a)$.

Proof. Easy induction on a. □

Lemma 46. For every $a \in \Lambda\mathrm{re}$, $m \in \mathbb{N}$, $x \in \mathbb{V}$, $\{x_1,\ldots,x_n\}$ distinct variables such that $\mathrm{FV}(a) \subseteq \{1,\ldots,n\} \wedge$ $1 \leq m \leq n+1 \wedge x \notin \{x_1,\ldots,x_n\}$:

1. $m \notin \mathrm{FV}(a) \implies x \notin \mathrm{FV}\big(\mathrm{u}_{[x_1,\ldots,x_{m-1},x,x_m,\ldots,x_n]}(a)\big)$
2. $m \in \mathrm{FV}(a) \implies x \in \mathrm{FV}\big(\mathrm{u}_{[x_1,\ldots,x_{m-1},x,x_m,\ldots,x_n]}(a)\big)$
3. $m \notin \mathrm{FV}(a) \implies \mathrm{u}_{[x_1,\ldots,x_{m-1},x,x_m,\ldots,x_n]}(a) =_\alpha \mathrm{u}_{[x_1,\ldots,x_n]}(\downarrow_m(a))$

Proof. Easy inductions on a. □

Given, once again, the auxiliary lemmas, we will now state Part C of the isomorphism theorem. As for Part B, Item 1 of Lemma 47 will be enough to prove preservation for the λx and λxgc calculi, whereas Item 2 will also be needed for the case of λex, concluding preservation by definition of reduction modulo an equation.

Lemma 47. For every $a, b \in \Lambda\mathrm{re}$:

1. $a \rightarrow_{\lambda\mathrm{rex_p}(\lambda\mathrm{re},\lambda\mathrm{re_{gc}})} b \implies \mathrm{u}(a) \rightarrow_{\mathrm{Bx}(\lambda\mathrm{x},\lambda\mathrm{xgc})} \mathrm{u}(b)$
2. $a =_{\mathrm{D}} b \implies \mathrm{u}(a) =_{\mathrm{C}} \mathrm{u}(b)$

Proof. For part 1, perform induction on a analogue to that of lemma 43.1. For part 2, perform induction on the inference of $a =_{\mathrm{D}} b$, analogue to that of lemma 43.2. In both cases, use auxiliary lemmas 44, 45 and 46. □

On the Implementation of Dynamic Patterns

Thibaut Balabonski
Laboratoire PPS, CNRS and Université Paris Diderot
thibaut.balabonski@pps.jussieu.fr

The evaluation mechanism of pattern matching with dynamic patterns is modelled in the *Pure Pattern Calculus* by one single meta-rule. This contribution presents a refinement which narrows the gap between the abstract calculus and its implementation. A calculus is designed to allow reasoning on matching algorithms. The new calculus is proved to be confluent, and to simulate the original *Pure Pattern Calculus*. A family of new, matching-driven, reduction strategies is proposed.

Introduction: Dynamic Patterns

Pattern matching is a basic mechanism used to deal with algebraic data structures in functional programming languages. It allows to define a function by reasoning on the shape of the arguments. For instance, define a binary tree to be either a single data or a node with two subtrees (code on the left, in ML-like syntax). Then a function on binary trees may be defined by reasoning on the shapes generated by these two possibilities (code on the right).

```
type 'a tree =
       | Data 'a
       | Node of 'a tree * 'a tree
```

```
let f t = match t with
        | Data d            ->  <code1>
        | Node (Data d) r   ->  <code2>
        | Node l r          ->  <code3>
```

An argument given to the function `f` is first compared to (or **matched** against) the shape `Data d` (called a **pattern**). In case of success, the occurrences of `d` in `<code1>` are replaced by the corresponding part of the argument, and `<code1>` is executed. In case of failure of this first matching (the argument is not a data) the argument is matched against the second pattern, and so on until a matching succeeds or there is no pattern left.

One limit of this approach is that patterns are fixed expressions mentioning explicitly the constructors to which they can apply, which restricts polymorphism and reusability of the code. This can be improved by allowing patterns to be parametrised: one single function can be specialised in various ways by instantiating the parameters of its patterns by different constructors or even by functions building patterns. For instance in the following code, the function `f` would take an additional parameter **p** which would then be used to define the first two patterns. In this case, instantiating **p** with the constructor `Data` would yield the same function as before, but any other function building a pattern can be used for **p**!

```
let f  p t = match t with
           |  p d           ->  <code1>
           | Node ( p d) r  ->  <code2>
           | Node l r       ->  <code3>
```

However, introducing parameters and functions inside patterns deeply modifies their nature: they become dynamic objects that have to be evaluated. This disrupts the matching algorithms and introduces new evaluation behaviours. This paper intends to give tools to study these extended evaluation possibilities.

The *Pure Pattern Calculus* (*PPC*) of B. Jay and D. Kesner [JK09, Jay09] models the behaviour of dynamic patterns by using a meta-level notion of pattern matching. The present contribution analyses the content of the meta pattern matching of *PPC* (reviewed in Section 1), and proposes an explicit pattern matching calculus (Section 2) which is confluent, which simulates *PPC*, and which allows the description of new reduction strategies (Section 3.1). An extension of the explicit calculus is then discussed (Section 3.2) before a conclusion is drawn.

1 The Pure Pattern Calculus

This section only reviews some key aspects of *PPC*. Please refer to [JK09] for a complete story with more examples. The syntax of *PPC* is close to the one of λ-calculus. The main difference is the replacement of the abstraction over a variable $\lambda x.b$ by an abstraction over a pattern (with a list of matching variables) written $[\theta]p \to b$. There is also a new distinction between **variable** occurrences x and **matchable** occurrences $\hat{x}$ of a name x. Variable occurrences are usual variables which may be substituted while matchable occurrences are immutable and used as matching variables or constructors.

$$t \quad ::= \quad x \mid \hat{x} \mid tt \mid [\theta]t \to t \qquad\qquad \textit{PPC}\text{ Terms}$$

where θ is a list of names. Letter a (resp. b, p) is used to indicate a term in position of **a**rgument (resp. function **b**ody, **p**attern).

As pictured below, in the abstraction $[\theta]p \to b$ the list of names θ binds matchable occurrences in the pattern p and variable occurrences in the body b. Substitution of free variables and α-conversion are deduced (see [JK09] for details on *PPC*, or Figures 1 and 2 for a formal definition in an extended setting).

$$[x]\ x\,\hat{x} \to x\,\hat{x} \quad =_\alpha \quad [y]\ x\,\hat{y} \to y\,\hat{x}$$

One of the features of *PPC* is the use of a single syntactic application for two different meanings: the term $t_1 t_2$ may represent either the usual **functional application** of a function t_1 to an argument t_2 or the construction of a data structure by **structural application** of a constructor to one or more arguments. The latter is invariant: any structural application is forever a data structure, whereas the functional application may be evaluated or instantiated someday (and then turn into anything else, including a structural application).

The simplest notion of pattern matching is syntactic: an argument a matches a pattern p if and only if there is a substitution σ such that $a = p^\sigma$. However, with arbitrary patterns, this solution generates non-confluent calculi [vOo90]. To recover confluence, syntactic matching can be used together with a restriction on patterns, as for instance the *rigid pattern condition* of the lambda-calculus with patterns [KvOdV08]. The alternative solution of *PPC* allows a priori any term to be a pattern, and checks the validity of patterns only a posteriori, when pattern matching is performed. In particular, the restriction on patterns applies only once the evaluation of the pattern is completed. This allows a greater freedom of evaluation and a greater polymorphism of patterns, and hence a greater expressivity.

This is done by a more subtle notion of matching, called **compound matching**, which tests whether patterns and arguments are in a so-called **matchable form**. A matchable form denotes a term which is understood as a value, or in other words a term whose current form is stable and then allows matching. Matchable forms are described in *PPC* at the meta-level by the following grammar:

$$\begin{array}{lcll} d & ::= & \hat{x} \mid dt & \qquad \textit{PPC}\text{ data structures} \\ m & ::= & d \mid [\theta]t \to t & \qquad \textit{PPC}\text{ matchable forms} \end{array}$$

Compound matching is then defined (still at the meta-level) by the following equations, taken in order.

$$\begin{array}{rcll} \{\!\{a /_\theta \hat{x}\}\!\} & := & \{x \mapsto a\} & \text{if } x \in \theta \\ \{\!\{\hat{x} /_\theta \hat{x}\}\!\} & := & \{\} & \text{if } x \notin \theta \\ \{\!\{a_1a_2 /_\theta p_1p_2\}\!\} & := & \{\!\{a_1 /_\theta p_1\}\!\} \uplus \{\!\{a_2 /_\theta p_2\}\!\} & \text{if } a_1a_2 \text{ and } p_1p_2 \text{ are matchable forms} \\ \{\!\{a /_\theta p\}\!\} & := & \bot & \text{if } a \text{ and } p \text{ are matchable forms, otherwise} \\ \{\!\{a /_\theta p\}\!\} & := & \texttt{wait} & \text{otherwise} \end{array}$$

Its result, called a **match** and denoted by ρ, may be a substitution (written σ), a matching failure (written $\bot$) or the special value `wait`. The latter case represents undefined cases of matching, when the pattern or the argument has still to be evaluated or instantiated before being matched.

Decomposition of compound patterns in the equations above is associated with an operation $\uplus$ of disjoint union which ensures linearity of patterns: no matching variable should be used twice in the same pattern, or confluence would be broken [Klo80]. Its formal definition is:

- $\uplus$ is commutative.
- $\bot \uplus \rho = \bot$ for any ρ (even `wait`).
- $\texttt{wait} \uplus \rho = \texttt{wait}$ for $\rho \neq \bot$.
- $\sigma_1 \uplus \sigma_2 = \bot$ if domains of σ_1 and σ_2 overlap.
- $\sigma_1 \uplus \sigma_2$ is the union of σ_1 and σ_2 otherwise.

Finally, *PPC* has to deal with a problem related to the dynamics of patterns: a matching variable may be erased from a pattern during its evaluation. In this case, no part of the argument would be bound to this matching variable and then no term would be substituted to the corresponding variable. Hence free variables would not be preserved, which would make reduction ill-defined (see Example 1). This is avoided in *PPC* by a last (meta-level) test, called *check*: the result $\{a /_\theta p\}$ of the matching of a against p is defined as follows.

- if $\{\!\{a /_\theta p\}\!\} = \bot$ then $\{a /_\theta p\} = \bot$.
- if $\{\!\{a /_\theta p\}\!\} = \sigma$ with $dom(\sigma) \neq \theta$ then $\{a /_\theta p\} = \bot$.
- if $\{\!\{a /_\theta p\}\!\} = \sigma$ with $dom(\sigma) = \theta$ then $\{a /_\theta p\} = \sigma$.

Remark that $\{a /_\theta p\}$ is not defined if $\{\!\{a /_\theta p\}\!\} = \texttt{wait}$.
Finally, the reduction $\longrightarrow_{PPC}$ of *PPC* is defined by a unique reduction rule (applied in any context):

$$([\theta]p \to b)a \quad \longrightarrow_{\beta_m} \quad b^{\{a /_\theta p\}}$$

where for any b and σ the expression b^σ denotes the application of the substitution σ to the term b, and $b^\bot$ denotes some fixed closed normal term $\bot$.

Example 1. *Let t be a* PPC *term. The redex $([x]\hat{c}\hat{x} \to x)\ (\hat{c}t)$ reduces to t: the constructor $\hat{c}$ matches itself and the matchable $\hat{x}$ is associated to t. On the other hand, $([x,y]\hat{c}\hat{x} \to xy)\ (\hat{c}t)$ reduces to $\bot$: whereas the compound matching is defined and successful, the check fails since there is no match for y and the result would be ty where y appears as a free variable. The redex $([x]\hat{c}\hat{x} \to x)\ (\hat{c})$ also reduces to $\bot$ since a constructor will never match a structural application. And last, $([x]y\hat{x} \to x)\ (\hat{c}t)$ is not a redex since the pattern $y\hat{x}$ has to be instantiated.*

2 Explicit Matching

This section defines the *Pure Pattern Calculus with Explicit Matching* (*PPC_{EM}*), a calculus which gives an account of all the steps of a pattern matching process of *PPC*. The first point discussed is the identification of structural application (Section 2.1). An explicit calculus is then fully detailed (Section 2.2) and some of its basic properties are proved (Section 2.3). Explicit formulations of simpler pattern calculi already appear in [CK04, For02, CFK04].

2.1 Explicit Data Structures

Firstly, a new syntactic construct is introduced to discriminate between functional and structural applications (as in [FMS06] for the rewriting calculus for instance). Any application is supposed functional *a priori*, and two reduction rules propagate structural information. The explicit structural application of t to u is written $t \bullet u$.

$$\begin{array}{rcll} t & ::= & x \mid \hat{x} \mid tt \mid t \bullet t \mid [\theta]t \to t & \qquad PPC_\bullet \text{ terms} \\ d & ::= & \hat{x} \mid t \bullet t & \qquad PPC_\bullet \text{ data structures} \end{array}$$

$$\begin{array}{rcl} \hat{x}\, t & \longrightarrow_\bullet & \hat{x} \bullet t \\ (t_1 \bullet t_2)\, t_3 & \longrightarrow_\bullet & (t_1 \bullet t_2) \bullet t_3 \end{array}$$

The identity morphism embeds *PPC* into $PPC_\bullet$. The subset of $PPC_\bullet$ defined by *PPC* is referred to as the set of pure terms. On the other hand, a “forgetful” morphism maps $PPC_\bullet$ terms back to *PPC* terms (or pure terms):

$$\begin{array}{rcl} [\![x]\!] & := & x \\ [\![\hat{x}]\!] & := & \hat{x} \\ [\![t_1 t_2]\!] & := & [\![t_1]\!][\![t_2]\!] \\ [\![t_1 \bullet t_2]\!] & := & [\![t_1]\!][\![t_2]\!] \\ [\![[\theta]p \to b]\!] & := & [\theta][\![p]\!] \to [\![b]\!] \end{array}$$

Some $PPC_\bullet$ data structures are not mapped to data structures of *PPC*, for instance $([\theta]p \to b) \bullet a$. However, for any pure term t, if $t \longrightarrow_\bullet^* t'$ and t' is a $PPC_\bullet$ data structure, then t is a *PPC* data structure (proof by induction on t). One can also observe that for every *PPC* data structure t, there exists a reduction $t \longrightarrow_\bullet^* t'$ with t' a $PPC_\bullet$ data structure. Call **well-formed** a term t such that $[\![t]\!] \longrightarrow_\bullet^* t$.

2.2 Explicit Pattern Matching

Another new syntactic object has to be introduced to represent an ongoing matching operation. The basic information contained in such an object are: the list of matching variables, a partial result recording what has already been computed, and a representation of what has still to be solved.

This new object is called **matching** and is written $\langle \theta | \mu | \Delta \rangle$ with θ a list of names, μ a **decided match** (that means, $\perp$ or a substitution), and Δ the collection of submatchings that have still to be solved (a multiset of pairs of terms). For now on, we will consider only decided matches, written μ (`wait` does not exist as such in PPC_{EM}).

The complete new grammar is:

$$\begin{array}{rcll} t & ::= & x \mid \hat{x} \mid tt \mid t \bullet t \mid [\theta]t \to t \mid t\,\langle \theta | \mu | \Delta \rangle & \qquad PPC_{EM} \text{ terms} \\ d & ::= & \hat{x} \mid t \bullet t & \qquad PPC_{EM} \text{ data structures} \\ m & ::= & d \mid [\theta]t \to t & \qquad PPC_{EM} \text{ matchable forms} \end{array}$$

The set of free names of a term t is $fn(t) = fv(t) \cup fm(t)$.

Free variables

$$
\begin{array}{rcl}
fv(x) & := & \{x\} \\
fv(\hat{x}) & := & \emptyset \\
fv(t_1 t_2) & := & fv(t_1) \cup fv(t_2) \\
fv(t_1 \bullet t_2) & := & fv(t_1) \cup fv(t_2) \\
fv([\theta]p \to b) & := & fv(p) \cup (fv(b) \setminus \theta) \\
fv(t\,\langle \theta | \mu | \Delta \rangle) & := & (fv(t) \setminus \theta) \cup fv(codom(\mu)) \cup fv(\Delta)
\end{array}
$$

Free matchables

$$
\begin{array}{rcl}
fm(x) & := & \emptyset \\
fm(\hat{x}) & := & \{x\} \\
fm(t_1 t_2) & := & fm(t_1) \cup fm(t_2) \\
fm(t_1 \bullet t_2) & := & fm(t_1) \cup fm(t_2) \\
fm([\theta]p \to b) & := & (fm(p) \setminus \theta) \cup fm(b) \\
fm(t\,\langle \theta | \mu | \Delta \rangle) & := & fm(t) \cup fm(codom(\mu)) \cup fm(\pi_1(\Delta)) \cup (fm(\pi_2(\Delta)) \setminus \theta)
\end{array}
$$

where if $\Delta = (a_1, p_1)...(a_n, p_n)$ then $fm(\pi_1(\Delta)) = \bigcup_i fm(a_i)$ and $fm(\pi_2(\Delta)) = \bigcup_i fm(p_i)$.

Figure 1: Free names of a PPC_{EM} term

$$
\begin{array}{rcll}
x^{\sigma} & := & \sigma_x & x \in dom(\sigma) \\
x^{\sigma} & := & x & x \notin dom(\sigma) \\
\hat{x}^{\sigma} & := & \hat{x} & \\
(tu)^{\sigma} & := & t^{\sigma} u^{\sigma} & \\
(t \bullet u)^{\sigma} & := & t^{\sigma} \bullet u^{\sigma} & \\
([\theta]p \to b)^{\sigma} & := & ([\theta]p^{\sigma} \to b^{\sigma}) & \theta \cap (dom(\sigma) \cup fn(\sigma)) = \emptyset \\
(t\,\langle \theta | \mu | \Delta \rangle)^{\sigma} & := & t^{\sigma}\,\langle \theta | \mu^{\sigma} | \Delta^{\sigma} \rangle & \theta \cap (dom(\sigma) \cup fn(\sigma)) = \emptyset
\end{array}
$$

where in Δ^{σ} (resp. μ^{σ}) the substitution propagates in all terms of Δ (resp. of the codomain of μ).

Figure 2: Substitution in PPC_{EM}

Initialisation

$$([\theta]p \to b)a \quad \longrightarrow_B \quad b\,\langle\theta|\emptyset|(a,p)\rangle$$

Structural application

$$\begin{array}{rcl} \hat{x}\,t & \longrightarrow_\bullet & \hat{x}\bullet t \\ (t_1 \bullet t_2)\,t_3 & \longrightarrow_\bullet & (t_1 \bullet t_2)\bullet t_3 \end{array}$$

Matching

Since Δ has been defined as a multiset of pairs of terms, its elements are not ordered. In the following rules $(a,p)\Delta$ denotes the (multiset) union of Δ with the singleton $\{(a,p)\}$.

The first three matching rules are for successful matching steps.

$$\begin{array}{rcll} b\,\langle\theta|\mu|(a,\hat{x})\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\mu \uplus \{x \mapsto a\}|\Delta\rangle & \text{if } x \in \theta \text{ and } fn(a) \cap \theta = \emptyset \\ b\,\langle\theta|\mu|(\hat{x},\hat{x})\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\mu|\Delta\rangle & \text{if } x \notin \theta \\ b\,\langle\theta|\mu|(a_1 \bullet a_2, p_1 \bullet p_2)\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\mu|(a_1,p_1)(a_2,p_2)\Delta\rangle & \end{array}$$

The last six matching rules are for failure, and could be summed up as "for any other matchable forms a and p, let $b\,\langle\theta|\mu|(a,p)\Delta\rangle$ reduce to $b\,\langle\theta|\bot|\Delta\rangle$".

$$\begin{array}{rcll} b\,\langle\theta|\mu|(\hat{y},\hat{x})\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \text{if } x \notin \theta \text{ and } x \neq y \\ b\,\langle\theta|\mu|(a_1 \bullet a_2,\hat{x})\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \text{if } x \notin \theta \\ b\,\langle\theta|\mu|([\theta_a]p_a \to b_a,\hat{x})\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \text{if } x \notin \theta \\ b\,\langle\theta|\mu|(\hat{x}, p_1 \bullet p_2)\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \\ b\,\langle\theta|\mu|([\theta_a]p_a \to b_a, p_1 \bullet p_2)\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \\ b\,\langle\theta|\mu|(a, [\theta_p]p_p \to b_p)\Delta\rangle & \longrightarrow_m & b\,\langle\theta|\bot|\Delta\rangle & \end{array}$$

Resolution

$$\begin{array}{rcll} b\,\langle\theta|\sigma|\emptyset\rangle & \longrightarrow_r & b^\sigma & \text{if } dom(\sigma) = \theta \quad (\textbf{substitution rule}) \\ b\,\langle\theta|\sigma|\emptyset\rangle & \longrightarrow_r & \bot & \text{if } dom(\sigma) \neq \theta \\ b\,\langle\theta|\bot|\Delta\rangle & \longrightarrow_r & \bot & \end{array}$$

Figure 3: Rules of PPC_{EM}

A pure term of PPC_{EM} is a term without any structural application or matching (that means a PPC term). As in PPC, the symbol $\bot$ used as a term denotes a fixed closed pure normal term.

Free variables and matchables are defined in Figure 1 as a natural extension of PPC mechanisms to explicit matching. Similarly, a notion of (meta-level) substitution is deduced from this definition (Figure 2). Finally, a notion of α-conversion is associated, and from now, on it is supposed that all bound names in a term are different, and disjoint from free names.

New rules for matching are of three kinds: an *initialisation rule* $\longrightarrow_B$ which triggers a new matching operation, several *matching rules* $\longrightarrow_m$ corresponding to all possible elementary matching steps and three *resolution rules* $\longrightarrow_r$ that apply the result of a completed matching. The complete set of rules of PPC_{EM} is given in Figure 3.

Reduction $\longrightarrow_{EM}$ of PPC_{EM} is defined by application of any rule of $\longrightarrow_B$, $\longrightarrow_\bullet$, $\longrightarrow_m$ or $\longrightarrow_r$ in any context. The subsystem $\longrightarrow_p = \longrightarrow_\bullet \cup \longrightarrow_m \cup \longrightarrow_r$ computes (when possible) already existing pattern matchings but does not create new ones.

2.3 Confluence and Simulation properties

This section states and proves four theorems on basic properties of PPC_{EM} and its links with PPC. The first one is a result on the normalization of already existing pattern matchings.

Theorem 1. *$\longrightarrow_p$ is confluent and strongly normalizing.*

Proof.

- We define two well-founded orders $\prec_{\mathscr{N}}$ and $\prec_{\mathscr{S}}$, whose lexicographic product contains ${}_p\!\longleftarrow$. This will enforce strong normalization.
  - $\prec_{\mathscr{N}}$ sorts terms with respect to the nesting of matchings. It is based on an over-approximation of the depth of potentially nested matchings (matchings that are syntactically nested or that may become such after some substitutions). For any lists of names θ_i, decided matches μ_i, and lists of pairs of terms Δ_i, the sequence $\langle\theta_1|\mu_1|\Delta_1\rangle;\ldots;\langle\theta_n|\mu_n|\Delta_n\rangle$ is called a potentially nested chain of length n if for each $i \in \{1\ldots n-1\}$ one of these conditions holds:
    - **Nesting:** $\langle\theta_{i+1}|\mu_{i+1}|\Delta_{i+1}\rangle$ appears in Δ_i or in the codomain of μ_i.
    - **Potential nesting:** a variable of θ_{i+1} appears in Δ_i or in the codomain of μ_i.

    The set of maximal chains of a term t is the set of all potentially nested chains that can be built using the matchings appearing in t and that can not be extended (neither by the left nor by the right) using other matchings of t. For this extraction, remember that all bound names in t are supposed to be different, and disjoint from free names. The depth of t is the multiset of the lengths of the maximal chains of t.

    Example 2. *Write $t = \hat{c}\,\langle\emptyset|\emptyset|(x,\hat{c})(x,\hat{c})\rangle\,\langle x|x\mapsto y\,\langle y|\emptyset|(\hat{c},\hat{y})\rangle\,|\emptyset\rangle$. The term t contains three matchings and has one maximal chain of length 3, which is*

    $$\langle\,\emptyset|\emptyset|(x,\hat{c})(x,\hat{c})\,\rangle\,;\,\langle\,x|x\mapsto y\,\langle y|\emptyset|(\hat{c},\hat{y})\rangle\,|\emptyset\,\rangle\,;\,\langle\,y|\emptyset|(\hat{c},\hat{y})\,\rangle$$

    The reduction $t \longrightarrow_r t' = \hat{c}\,\langle\,\emptyset|\emptyset|\,(y_1\,\langle y_1|\emptyset|(\hat{c},\hat{y_1})\rangle\,,\hat{c})\;(y_2\,\langle y_2|\emptyset|(\hat{c},\hat{y_2})\rangle\,,\hat{c})\,\rangle$ yields a new term t' which still contains three matchings (one was reduced and disappeared but another one was duplicated) and admits two maximal chains of length 2, namely

    $$\langle\,\emptyset|\emptyset|\,(y_1\,\langle y_1|\emptyset|(\hat{c},\hat{y_1})\rangle\,,\hat{c})\;(y_2\,\langle y_2|\emptyset|(\hat{c},\hat{y_2})\rangle\,,\hat{c})\,\rangle\,;\,\langle\,y_1|\emptyset|(\hat{c},\hat{y_1})\,\rangle$$

    $$\langle\,\emptyset|\emptyset|\,(y_1\,\langle y_1|\emptyset|(\hat{c},\hat{y_1})\rangle\,,\hat{c})\;(y_2\,\langle y_2|\emptyset|(\hat{c},\hat{y_2})\rangle\,,\hat{c})\,\rangle\,;\,\langle\,y_2|\emptyset|(\hat{c},\hat{y_2})\,\rangle$$

The usual order on natural integers gives a well-founded order on the lengths of potentially nested chains. $\prec_{\mathscr{N}}$ is defined as the multiset extension of this order, applied to the depths of terms. It strictly decreases for any reduction by the substitution rule, and is less or equal for any other reduction.

– $\prec_{\mathscr{S}}$ is the natural order on the size of terms, defined as follows:

$$\begin{array}{rcl}
\mathscr{S}(x) & := & 1 \\
\mathscr{S}(\hat{x}) & := & 1 \\
\mathscr{S}(t_1 t_2) & := & \mathscr{S}(t_1)+\mathscr{S}(t_2)+2 \\
\mathscr{S}(t_1 \bullet t_2) & := & \mathscr{S}(t_1)+\mathscr{S}(t_2)+1 \\
\mathscr{S}([\theta]p \to b) & := & \mathscr{S}(p)+\mathscr{S}(b) \\
\mathscr{S}(b\langle\theta|\mu|\Delta\rangle) & := & \mathscr{S}(b)+\mathscr{S}(\perp)+\sum_{x\in dom(\mu)}\mathscr{S}(\mu_x)+\sum_{(a,p)\in_k\Delta}k(\mathscr{S}(a)+\mathscr{S}(p))
\end{array}$$

where we write $e \in_k \Delta$ when the element e appears in the multiset Δ with multiplicity k. $\prec_{\mathscr{S}}$ strictly decreases for any reduction except by the substitution rule.

- Matching rules generate some critical pairs, most of which are trivially convergent. The most subtle case is the reduction of a non linear matching:

$$\langle\theta|\mu\uplus\{x\mapsto a_1\}|(a_2,\hat{x})\Delta\rangle \quad {}_p\!\longleftarrow \quad \langle\theta|\mu|(a_1,\hat{x})(a_2,\hat{x})\Delta\rangle \quad \longrightarrow_p \quad \langle\theta|\mu\uplus\{x\mapsto a_2\}|(a_1,\hat{x})\Delta\rangle$$

Since $\uplus$ is a disjoint union of substitutions, both sides can be reduced to $\langle\theta|\perp|\Delta\rangle$.
Finally, $\longrightarrow_p$ is weakly confluent, and then confluent by Newman's Lemma [Ter03].

□

The second theorem states the confluence of $\longrightarrow_{EM}$. Since the reduction of PPC_{EM} is defined by several rules, the result does not fall into the modular framework of [JK09]. It is proved here directly by the Tait and Martin-Löf's technique. The main construction of the proof is the definition (in Figure 4) of a parallel reduction relation $\Longrightarrow$ enjoying the diamond property (Lemma 3). The relation $\Longrightarrow$ is first linked to $\longrightarrow_{EM}$ in Lemma 1.

Lemma 1. $\longrightarrow_{EM} \quad\subseteq\quad \Longrightarrow \quad\subseteq\quad \longrightarrow_{EM}^*$

Proof.

- $\longrightarrow_{EM} \quad\subseteq\quad \Longrightarrow$ by induction on the definition of $\longrightarrow_{EM}$.
- $\Longrightarrow \quad\subseteq\quad \longrightarrow_{EM}^*$ by induction on the definition of $\Longrightarrow$.

□

Lemma 2. *If $t \Longrightarrow t'$ and $\sigma \Longrightarrow \sigma'$ then $t^\sigma \Longrightarrow t'^{\sigma'}$.*

Proof. By induction on the derivation of $t \Longrightarrow t'$. □

Lemma 3. $\Longleftarrow\!\!\Longrightarrow \quad\subseteq\quad \Longrightarrow\Longleftarrow$

Proof. Suppose $t_1 \Longleftarrow t \Longrightarrow t_2$. Induction on the derivations of $t \Longrightarrow t_1$ and $t \Longrightarrow t_2$:

- If one of the reductions is by "Id", the conclusion is immediate.
- If one reduction is by a "Cgr" rule, and the other by a "Cgr", "Init", "Struct", or "Match" rule, then the induction hypothesis applies straightforwardly.

Id.

$$t \Longrightarrow t$$

Cgr.

$$\frac{t_1 \Longrightarrow t_1' \qquad t_2 \Longrightarrow t_2'}{t_1 t_2 \Longrightarrow t_1' t_2'} \qquad \frac{t_1 \Longrightarrow t_1' \qquad t_2 \Longrightarrow t_2'}{t_1 \bullet t_2 \Longrightarrow t_1' \bullet t_2'} \qquad \frac{p \Longrightarrow p' \qquad b \Longrightarrow b'}{[\theta]p \to b \Longrightarrow [\theta]p' \to b'}$$

$$\frac{b \Longrightarrow b' \qquad \mu \Longrightarrow \mu' \qquad \Delta \Longrightarrow \Delta'}{b\,\langle\theta|\mu|\Delta\rangle \Longrightarrow b'\,\langle\theta|\mu'|\Delta'\rangle}$$

Init.

$$\frac{p \Longrightarrow p' \qquad b \Longrightarrow b' \qquad a \Longrightarrow a'}{([\theta]p \to b)a \Longrightarrow b'\,\langle\theta|\emptyset|(a',p')\rangle}$$

Struct.

$$\frac{t \Longrightarrow t'}{\hat{x}\,t \Longrightarrow \hat{x} \bullet t'} \qquad \frac{t_1 \Longrightarrow t_1' \qquad t_2 \Longrightarrow t_2' \qquad t_3 \Longrightarrow t_3'}{(t_1 \bullet t_2)\,t_3 \Longrightarrow (t_1' \bullet t_2') \bullet t_3'}$$

Match.

$$\frac{b \Longrightarrow b' \qquad \mu \Longrightarrow \mu' \qquad a \Longrightarrow a' \qquad \Delta \Longrightarrow \Delta'}{b\,\langle\theta|\mu|(a,\hat{x})\Delta\rangle \Longrightarrow b'\,\langle\theta|\mu' \uplus \{x \mapsto a'\}|\Delta'\rangle}\; x \in \theta, fn(a) \cap \theta = \emptyset$$

$$\frac{b \Longrightarrow b' \qquad \mu \Longrightarrow \mu' \qquad \Delta \Longrightarrow \Delta'}{b\,\langle\theta|\mu|(\hat{x},\hat{x})\Delta\rangle \Longrightarrow b'\,\langle\theta|\mu'|\Delta'\rangle}\; x \notin \theta$$

$$\frac{b \Longrightarrow b' \qquad \mu \Longrightarrow \mu' \qquad \Delta \Longrightarrow \Delta' \qquad a_i \Longrightarrow a_i' \qquad p_i \Longrightarrow p_i'}{b\,\langle\theta|\mu|(a_1 \bullet a_2, p_1 \bullet p_2)\Delta\rangle \Longrightarrow b'\,\langle\theta|\mu'|(a_1',p_1')(a_2',p_2')\Delta'\rangle}$$

$$\frac{b \Longrightarrow b' \qquad \Delta \Longrightarrow \Delta'}{b\,\langle\theta|\mu|(a,p)\Delta\rangle \Longrightarrow b'\,\langle\theta|\bot|\Delta'\rangle}\; a \text{ and } p \text{ other matchable forms}$$

Res.

$$\frac{b \Longrightarrow b' \qquad \sigma \Longrightarrow \sigma'}{b\,\langle\theta|\sigma|\emptyset\rangle \Longrightarrow (b')^{\sigma'}}\; dom(\sigma) = \theta \qquad \frac{dom(\sigma) \neq \theta}{b\,\langle\theta|\sigma|\emptyset\rangle \Longrightarrow \bot} \qquad b\,\langle\theta|\bot|\Delta\rangle \Longrightarrow \bot$$

As in Figure 3, the last "Match" rule could be explicited in six fail rules.
Parallel reduction is straightforwardly extended:

- to decided matches (μ) by applying $\Longrightarrow$ to all terms in the codomain of a substitution (with moreover $\bot \Longrightarrow \bot$).
- to multisets of pairs of terms (Δ) by applying $\Longrightarrow$ to all terms.

Figure 4: Definition of parallel reduction relation $\Longrightarrow$

- If one reduction is by a "Cgr" rule and the other by a "Res" rule, there is one non trivial case: suppose $t_1\langle\theta|\sigma_1|\emptyset\rangle \Longleftarrow t\langle\theta|\sigma|\emptyset\rangle \Longrightarrow t_2^{\sigma_2}$. By induction hypothesis there are t_3 and σ_3 such that $t_1 \Longrightarrow t_3 \Longleftarrow t_2$ and $\sigma_1 \Longrightarrow \sigma_3 \Longleftarrow \sigma_2$. Then we can derive $t_1\langle\theta|\sigma_1|\emptyset\rangle \Longrightarrow t_3^{\sigma_3}$. Finally, by Lemma 2 we conclude that $t_2^{\sigma_2} \Longrightarrow t_3^{\sigma_3}$.
- If both reductions are by a "Init" rule, then the induction hypotheses apply straightforwardly.
- Idem for two "Struct" or two "Match" rules.
- Case where both reductions are by a "Res" rule. Reductions to $\bot$ are straightforward. Then consider the following case: $t_1^{\sigma_1} \Longleftarrow t\langle\theta|\sigma|\emptyset\rangle \Longrightarrow t_2^{\sigma_2}$. By induction hypotheses $t_1 \Longrightarrow t_3 \Longleftarrow t_2$ and $\sigma_1 \Longrightarrow \sigma_3 \Longleftarrow \sigma_2$. By Lemma 2 $t_1^{\sigma_1} \Longrightarrow t_3^{\sigma_3} \Longleftarrow t_2^{\sigma_2}$.

□

Theorem 2. *PPC_{EM} is confluent.*

Proof. Since $\Longrightarrow$ has the diamond property (Lemma 3), its transitive closure $\Longrightarrow^*$ also enjoys the diamond property ([Ter03]). Moreover Lemma 1 implies $\longrightarrow_{EM}^* = \Longrightarrow^*$, and then $\longrightarrow_{EM}^*$ enjoys the diamond property. Finally, $\longrightarrow_{EM}$ is confluent. □

The last two theorems establish a link between the calculus with explicit matching PPC_{EM} and the original implicit PPC.

Lemma 4. *If $\{\!\{a /_\theta\, p\}\!\} = \mu$ with μ a decided match, then for any μ_0 and Δ there are μ' with $[\![\mu']\!] = \mu$ and a reduction*

$$\langle\theta|\mu_0|(a,p)\Delta\rangle \ (\longrightarrow_\bullet \cup \longrightarrow_m)^* \ \langle\theta|\mu_0 \uplus \mu'|\Delta\rangle$$

Proof. Induction on $\{\!\{a /_\theta\, p\}\!\}$.

- $\{\!\{a /_\theta\, \hat{x}\}\!\}$ with $x \in \theta$ or $\{\!\{\hat{x} /_\theta\, \hat{x}\}\!\}$ with $x \notin \theta$: immediate.
- $\{\!\{aa_0 /_\theta\, pp_0\}\!\}$ with aa_0 and pp_0 matchable forms. Hence $a = a_n ... a_1$ and $p = p_m ... p_1$ with a_n and p_m constructors. Then $a_n ... a_1 a_0 \longrightarrow_\bullet^* a_n \bullet ... \bullet a_1 \bullet a_0$ and $p_m ... p_1 p_0 \longrightarrow_\bullet^* p_m \bullet ... \bullet p_1 \bullet p_0$. Suppose $n \geq m$, then $\{\!\{aa_0 /_\theta\, pp_0\}\!\} = \{\!\{a_m ... a_n /_\theta\, p_n\}\!\} \uplus \{\!\{a_{n-1} /_\theta\, p_{n-1}\}\!\} \uplus ... \uplus \{\!\{a_0 /_\theta\, p_0\}\!\}$ and $\langle\theta|\mu_0|(a_n \bullet ... \bullet a_0, p_m \bullet ... \bullet p_0)\Delta\rangle \longrightarrow_m^* \langle\theta|\mu_0|(a_m \bullet ... \bullet a_n, p_n)(a_{n-1}, p_{n-1})...(a_0,p_0)\Delta\rangle$. Case on $p_n = \hat{x}$:
  - If $x \in \theta$ then the matching reduces to $\langle\theta|\mu_0 \uplus \{x \mapsto a_m \bullet ... \bullet a_n\}|(a_{n-1}, p_{n-1})...(a_0,p_0)\Delta\rangle$.
  - If $x \notin \theta$ then the matching reduces to $\langle\theta|\mu_0'|(a_{n-1}, p_{n-1})...(a_0,p_0)\Delta\rangle$ with $\mu_0' = \mu_0$ or $\mu_0' = \bot$.

  In any of these two cases, the induction hypothesis gives the conclusion. In the case where $m > n$, the same method allows to derive a reduction to $\bot$.
- Cases of matching failure: for instance $\{\!\{\hat{x} /_\theta\, \hat{y}t\}\!\}$. The following reduction gives the conclusion: $\langle\theta|\mu_0|(\hat{x}, \hat{y}t)\Delta\rangle \longrightarrow_\bullet \langle\theta|\mu_0|(\hat{x}, \hat{y} \bullet t)\Delta\rangle \longrightarrow_m \langle\theta|\bot|\Delta\rangle$.

□

Theorem 3. *For any terms t and t' of* PPC, *if* $t \longrightarrow_{PPC} t'$ *then* $t \longrightarrow_{EM}^* t'$.

Proof. Suppose $t \longrightarrow_{PPC} t'$. There is a context $C[]$ such that $t = C[([\theta]p \to b)a] \longrightarrow_{PPC} C[b'] = t'$ and $\{\!\{a /_\theta\, p\}\!\} = \mu$ with μ a decided match.
By Lemma 4 $([\theta]p \to b)a \longrightarrow_B b\langle\theta|\emptyset|(a,p)\rangle \ (\longrightarrow_\bullet \cup \longrightarrow_m)^* \ b\langle\theta|\mu|\emptyset\rangle$.

Case on μ:

- If $\mu = \bot$ then $b' = \bot$ and $b\,\langle\theta|\bot|\emptyset\rangle \longrightarrow_r \bot$.
- Else $\mu = \sigma$ and:
  - If $dom(\sigma) = \theta$ then $b' = b^{\sigma}$ and $b\,\langle\theta|\sigma|\emptyset\rangle \ \longrightarrow_r\ b^{\sigma}$.
  - Else $b' = \bot$ and $b\,\langle\theta|\bot|\emptyset\rangle \ \longrightarrow_r\ \bot$.

□

The map $[\![\cdot]\!]$ is naturally extended to any PPC_{EM} term, set of PPC_{EM} terms and decided match, as well as the notion of well-formedness. Then, for any μ and Δ not containing any explicit matching, define the semantics of the matching $\langle\theta|\mu|\Delta\rangle$ by:

$$[\![\theta|\mu|\Delta]\!] = [\![\mu]\!] \uplus \left(\biguplus_{(a,p)\in\Delta} \{\!\{ [\![a]\!] /_{\theta}\ [\![p]\!] \}\!\} \right)$$

Note that the semantics can be `wait`.

Lemma 5. *For any well-formed μ, μ', Δ and Δ' which do not contain any explicit matching, if $\langle\theta|\mu|\Delta\rangle \longrightarrow_m \langle\theta|\mu'|\Delta'\rangle$ or $\langle\theta|\mu|\Delta\rangle \longrightarrow_{\bullet} \langle\theta|\mu'|\Delta'\rangle$ then $[\![\theta|\mu|\Delta]\!] = [\![\theta|\mu'|\Delta']\!]$.*

Proof. Case on the reduction rules. □

Lemma 6 ([JK09])**.** *If $t \longrightarrow_{PPC} t'$, then $t^{\sigma} \longrightarrow_{PPC} t'^{\sigma}$.*

Let t be a PPC_{EM} term, and t' the unique normal form of t by $\longrightarrow_p$. Write $t\!\downarrow$ and call purification of t the term $[\![t']\!]$. Note that the purification may not be a pure term if there is an unsolvable matching in it.

Theorem 4. *For any well-formed terms t and t' of PPC_{EM}, if $t \longrightarrow_{EM} t'$ and $t\!\downarrow$ and $t'\!\downarrow$ are pure, then $t\!\downarrow = t'\!\downarrow$ or $t\!\downarrow \longrightarrow_{PPC} t'\!\downarrow$.*

Proof. Induction on $t \longrightarrow_{EM} t'$.

- Case $t = ([\theta]p \to b)a \ \longrightarrow_B\ b\,\langle\theta|\emptyset|(p,a)\rangle = t'$. The term $t'\!\downarrow$ is pure, then there is a sequence $b\!\downarrow\,\langle\theta|\emptyset|(p\!\downarrow, a\!\downarrow)\rangle\ (\longrightarrow_{\bullet} \cup \longrightarrow_m)^*\ b\!\downarrow\,\langle\theta|\mu|\Delta\rangle \longrightarrow_r t''$ where $[\![t'']\!] = t'\!\downarrow$ and where $\Delta = \emptyset$ or $\mu = \bot$. By Lemma 5, $[\![\mu]\!] = \{\!\{a\!\downarrow\ /_{\theta}\ p\!\downarrow\}\!\}$. Then, by case on matching resolution, $t\!\downarrow \longrightarrow_{PPC} [\![t'']\!] = t'\!\downarrow$.
- Other base cases: if $t \longrightarrow_p t'$, then $t\!\downarrow = t'\!\downarrow$.
- Case $t = b\,\langle\theta|\mu|\Delta\rangle \longrightarrow_{EM} b'\,\langle\theta|\mu|\Delta\rangle = t'$. The term $t\!\downarrow$ is pure. Then $\langle\theta|\mu|\Delta\rangle \longrightarrow_p^* \langle\theta|\mu'|\Delta'\rangle$ where $\Delta' = \emptyset$ or $\mu' = \bot$. If $\mu' = \bot$ or $dom(\mu') \neq \theta$, then $t\!\downarrow = t'\!\downarrow = \bot$. Suppose $\Delta' = \emptyset$ and $\mu' = \sigma$ with $dom(\sigma) = \theta$. Hence $t\!\downarrow = (b\!\downarrow)^{\sigma}$ and $t'\!\downarrow = (b'\!\downarrow)^{\sigma}$. By induction hypothesis $b\!\downarrow \longrightarrow_{PPC} b'\!\downarrow$, and then by Lemma 6 $t\!\downarrow \longrightarrow_{PPC} t'\!\downarrow$.
- Other inductive cases are straightforward.

□

This section introduced the new calculus PPC_{EM} for explicit matching with dynamic patterns, and proved its confluence. It also expressed a bidirectional simulation between PPC and PPC_{EM}: first any reduction of PPC is reflected in PPC_{EM} by a sequence. On the other hand, a reduction of PPC_{EM} can be mapped on zero or one step of PPC if and only if its source and its target are well-formed and can be purified. Next section discusses how this new calculus can be used.

3 Discussion

3.1 Reduction Strategies

Pattern matching raises at least two new issues concerning reduction strategies (*i.e.* the evaluation order of programs). One is related to the order in which pattern matching steps are performed, the other concerns the amount of evaluation of the pattern and of the argument performed before pattern matching is solved.

Some remarks about the order of pattern matching steps.
PPC_{EM} uses a multiset as the third component of a matching $\langle\theta|\mu|\Delta\rangle$ to represent all the remaining work. The calculus is thus able to cover all the possible orders of pattern matching steps. A particular strategy may be enforced by giving more structure to the multiset Δ and by adapting the matching reduction rules.

Example 3. *Suppose that Δ is now a* list *of pairs of terms, and $(a,p)\Delta$ denotes the usual "cons": it builds the list whose head is (a,p) and whose tail is Δ. Then the rules of Figure 3 implement a depth-first, left-to-right pattern matching algorithm.*

Example 4. *Now assume the list structure of Example 3 and replace the right member of the reduction rule $\langle\theta|\mu|(a_1 \bullet a_2, p_1 \bullet p_2)\Delta\rangle \longrightarrow_m \langle\theta|\mu|(a_1,p_1)(a_2,p_2)\Delta\rangle$ by $\langle\theta|\mu|\Delta(a_1,p_1)(a_2,p_2)\rangle$. Then pattern matching is done in a completely different order!*

More generally, if some permutations of the elements of Δ are allowed, lots of richer matching behaviours may be described in PPC_{EM}.

Pattern and argument evaluation: what is needed?
In PPC, a naive evaluation strategy for a term $([\theta]p \rightarrow b)a$ could be: evaluate the pattern p and the argument a, then solve the matching (atomically). As the usual call-by-value, this solution may perform unneeded evaluation of the argument, for instance in parts that are not reused in the body b of the function. The most basic solution to this problem, call-by-name, allows the substitution of non-evaluated arguments. But how can such a solution be described in a pattern calculus?

In the context of pattern matching, some evaluation of the argument has to be done before pattern matching is solved. However the exact amount of needed evaluation depends on the pattern. Hence pattern matching enforces some kind of call-by-value where the notion of value is context-sensitive. Moreover, even the evaluation of the pattern may depend on the argument!

This makes the description of a strategy performing a minimal evaluation of the dynamic pattern and the argument rather difficult. One may keep for the object-level a compact formalism like PPC by defining complex meta-level operations finely parametrised by terms. This is done in [KLR10] to describe standard reductions in a simpler pattern calculus. In contrast to this solution, we want to show here how the richer syntax of PPC_{EM} allows a simple description of such a reduction strategy.

Indeed PPC_{EM} allows to interleave pattern and argument reduction with pattern matching steps. This finer control allows for instance an easy definition of a "matching-driven" reduction, as pictured in Figure 5.

The idea here is to trigger pattern matchings as soon as possible. Then the pattern and the argument are evaluated until they become matchable, and one or more pattern matching steps are performed before the story goes on. A formal definition of a strategy implementing this picture is by restricting the reduction under a context to the only four rules given in Figure 6.

Moreover, it can be checked that the list structure of Example 3 associated with the rules of Figure 3 and the context rules of Figure 6 gives a deterministic reduction strategy for PPC_{EM} (which means that any term has at most one authorised redex).

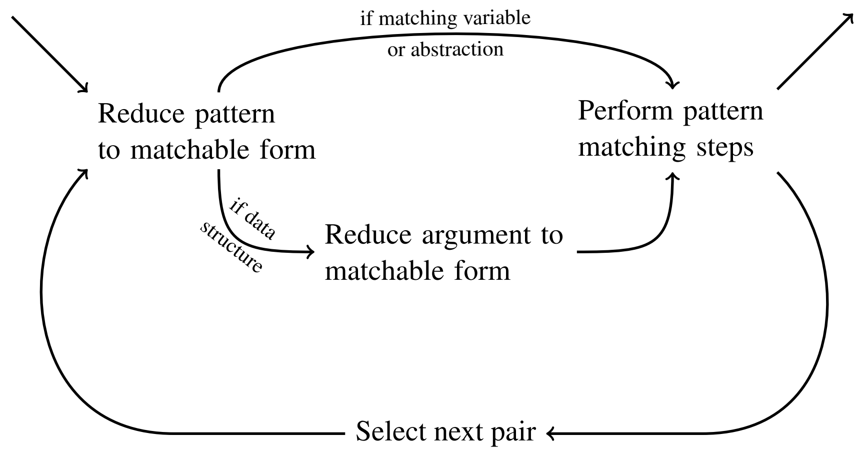

if matching variable or abstraction
Reduce pattern to matchable form
Perform pattern matching steps
if data structure
Reduce argument to matchable form
Select next pair

Figure 5: Matching-driven reduction strategy

$$\frac{t_1 \longrightarrow t_1'}{t_1 t_2 \longrightarrow t_1' t_2}$$

$$\frac{p \longrightarrow p'}{b\,\langle\theta|\mu|(a,p)\Delta\rangle \longrightarrow b\,\langle\theta|\mu|(a,p')\Delta\rangle}$$

$$\frac{a \longrightarrow a'}{b\,\langle\theta|\mu|(a,\hat{x})\Delta\rangle \longrightarrow b\,\langle\theta|\mu|(a',\hat{x})\Delta\rangle}\ x \notin \theta$$

$$\frac{a \longrightarrow a'}{b\,\langle\theta|\mu|(a,p_1 \bullet p_2)\Delta\rangle \longrightarrow b\,\langle\theta|\mu|(a',p_1 \bullet p_2)\Delta\rangle}$$

Figure 6: Context rules for matching-driven reduction

$$\begin{array}{rcll}
b\,\langle\theta|\tau|(a,\hat{x})\Delta\rangle & \longrightarrow_r & b^{\{x\mapsto a\}}\,\langle\theta|\tau\cup\{x\}|\Delta\rangle & \text{if } x\in\theta,\ x\notin\tau \text{ and } fn(a)\cap\theta=\emptyset \\
b\,\langle\theta|\theta|\emptyset\rangle & \longrightarrow_r & b^{\sigma} & \\
b\,\langle\theta|\tau|\emptyset\rangle & \longrightarrow_r & \bot & \text{if } \tau\neq\theta \\
b\,\langle\theta|\bot|\Delta\rangle & \longrightarrow_r & \bot &
\end{array}$$

Figure 7: Partial substitution rules

3.2 An Extension: Partial Substitution

Relaxing the matching procedure generates new possibilities of evaluation, which may bring more partial evaluation, more sharing or more parallelism. We explore here an extension of PPC_{EM} where the partial result of a matching can be applied to the function body before the matching process is completed.

Example 5. *Consider the following reduction:*

$$\begin{array}{ll} \multicolumn{2}{l}{([x]\hat{x}z \to (([\emptyset]x \to b)\hat{c}))\,(\hat{c}t)} \\ \longrightarrow_B & ([x]\hat{x}z \to (b\,\langle\emptyset|\emptyset|(\hat{c},x)\rangle))\,(\hat{c}t) \end{array}$$

The matching $\langle\emptyset|\emptyset|(\hat{c},x)\rangle$ is blocked because of the presence of the variable x in the pattern. Still, the external application can be evaluated:

$$\begin{array}{ll} \longrightarrow_B & (b\,\langle\emptyset|\emptyset|(\hat{c},x)\rangle)\,\langle x|\emptyset|(\hat{c}t,\hat{x}z)\rangle \\ \longrightarrow_\bullet^2 & (b\,\langle\emptyset|\emptyset|(\hat{c},x)\rangle)\,\langle x|\emptyset|(\hat{c}\bullet t,\hat{x}\bullet z)\rangle \\ \longrightarrow_m & (b\,\langle\emptyset|\emptyset|(\hat{c},x)\rangle)\,\langle x|\emptyset|(\hat{c},\hat{x})(t,z)\rangle \\ \longrightarrow_m & (b\,\langle\emptyset|\emptyset|(\hat{c},x)\rangle)\,\langle x|\{x\mapsto\hat{c}\}|(t,z)\rangle \end{array}$$

Now, the external matching $\langle x|\{x\mapsto\hat{c}\}|(t,z)\rangle$ is also blocked because of the variable z. However, its partial result is a substitution for x which, if applied, may unlock the internal matching. Indeed, allowing this partial substitution could lead to a reduction like:

$$\begin{array}{ll} \longrightarrow & (b\,\langle\emptyset|\emptyset|(\hat{c},\hat{c})\rangle)\,\langle x|\{x\mapsto\hat{c}\}|(t,z)\rangle \\ \longrightarrow_m & (b\,\langle\emptyset|\emptyset|\emptyset\rangle)\,\langle x|\{x\mapsto\hat{c}\}|(t,z)\rangle \\ \longrightarrow_r & b\,\langle x|\{x\mapsto\hat{c}\}|(t,z)\rangle \end{array}$$

where the internal matching is finally solved!

This kind of power may be of interest in two situations:

- By allowing more reduction in open terms, we gain more partial evaluation capabilities. This may be interesting for greater sharing and efficient evaluation [HG91].
- Suppose now that z is replaced in the example by a possibly big term. In a parallel implementation we could complete the external matching and evaluate the internal one in parallel. As pointed out in [FMS06], this might represent another gain in efficiency.

A light variation on PPC_{EM} gives this new power to our formalism. The principle of this variant is to systematically apply partial results (substitutions) as soon as they are obtained. Hence they do not need to be remembered in the object representing ongoing matching operations. Only a list of used variables is remembered for linearity verification.

The object representing a matching is now $\langle\theta|\tau|\Delta\rangle$ where τ is either $\perp$ or the list of the names of the matching variables that have already been used. Now the test of disjoint union of substitutions is replaced by a simple test against τ, while the final check compares θ and τ.

Initialisation, structural application, and most matching rules are the same in this variant. The only differences are for the first matching rule and the resolution rules, which are now as in Figure 7.

Any PPC_{EM} term can be translated into a term of this new calculus by applying the following transformation: turn any $b\,\langle\theta|\sigma|\Delta\rangle$ into $b^\sigma\,\langle\theta|dom(\sigma)|\Delta\rangle$ (there is nothing to change in a failed matching).

The simulation between PPC_{EM} and this extension is only one way: any reduction of PPC_{EM} is mapped by the previous morphism to a reduction sequence, but the converse is not true. Indeed the calculus with partial substitution allows new reductions, as pictured in Example 5. Confluence for this variant seems to be provable using the same technique as for plain PPC_{EM}.

Conclusion

The *Pure Pattern Calculus* is a compact framework modelling pattern matching with dynamic patterns. However, the conciseness of *PPC* is due to the use of several meta-level notions which deepens the gap between the calculus and implementation-related problems. This contribution defines the *Pure Pattern Calculus with Explicit Matching*, a refinement which is confluent and simulates *PPC*, and allows reasoning on the pattern matching mechanisms.

This enables a very simple definition of new reduction strategies in the spirit of call-by-name, which is new in this kind of framework since the reduction of the argument of a function depends on the pattern of the function, pattern which is itself a dynamic object. In the same direction, it would be interesting to express standardisation in pattern calculi (as presented for example in [KLR10]) using explicit matching.

References

[CK04] S. Cerrito and D. Kesner: Pattern Matching as Cut Elimination. *TCS*, 323:71–127, 2004. doi:10.1016/j.tcs.2004.03.032.

[CFK04] H. Cirstea, G. Faure and C. Kirchner: A Rho-Calculus of Explicit Constraint Application. *5th Workshop on Rewriting Logic and Applications. ENTCS*, vol. 117, 51–67, 2005. doi:10.1016/j.entcs.2004.06.029.

[FMS06] M. Fernández, I. Mackie, F.-R. Sinot: Interaction Nets vs the Rho-Calculus: Introducing Bigraphical Nets. *ENTCS*, 154(3):19–32, 2006. doi:10.1016/j.entcs.2006.05.004.

[For02] J. Forest: A Weak Calculus with Explicit Operators for Pattern Matching and Substitution. *13th International Conference on Rewriting Techniques and Applications. LNCS*, 2378:174–191, 2002. doi:10.1007/3-540-45610-4_13.

[HG91] C. K. Holst and D. K. Gomard: Partial Evaluation is Fuller Laziness. *PEPM'91*, 223–233, 1991. doi:10.1145/115866.115890.

[Jay09] B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.

[JK09] B. Jay and D. Kesner: First-Class Patterns. *J. Funct. Programming*, 19(2):191–225, 2009. doi:10.1017/S0956796808007144.

[KLR10] D. Kesner, C. Lombardi and A. Ríos: Standardisation for Constructor Based Pattern Calculi. *5th International Workshop on Higher-Order Rewriting: HOR 2010.*

[Klo80] J. W. Klop: *Combinatory Reduction Systems. Ph.D. Thesis, Mathematisch Centrum, Amstermdam, 1980*

[KvOdV08] J. W. Klop, V. van Oostrom, and R. de Vrijer: Lambda Calculus with Patterns. *TCS*, 398:16–31, 2008. doi:10.1016/j.tcs.2008.01.019.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[vOo90] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR228, Vrije Universiteit, Amsterdam, 1990.

Higher-order Rewriting for Executable Compiler Specifications

Kristoffer H. Rose
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA

krisrose@us.ibm.com

In this paper we outline how a simple compiler can be completely specified using higher order rewriting in all stages: parsing, analysis/optimization, and code emission, specifically using the *crsx.sf.net* system for a small declarative language called “X” inspired by XQuery (for which we are building a production quality compiler in the same way).

1 Introduction

A compiler typically consists of a parser generating an abstract syntax tree (AST) for some source language (SL), a “normalization” to a canonical form in an intermediate language (IL), some rewrites inserting analysis results into and performing simplifications of the IL, and finally code emission to the target language (TL).

$$\begin{tikzcd}
\mathrm{SL} \arrow[r, "\mathit{Parse}"] & \mathrm{AST} \arrow[r, "\mathit{Normalize}"] & \mathrm{IL} \arrow[loop below, "\mathit{Rewrite}"] \arrow[r, "\mathit{Emit}"] & \mathrm{TL}
\end{tikzcd}$$

Each arrow in the diagram can be understood as a rewriting:

1. parsing to an AST is a rewriting from the string of characters in the input file to a term representing the source program, usually formalized and implemented using some variation of context free grammars [12];
2. normalization of the AST into the IL involves rewrite rules to eliminate “syntactic sugar” and other redundant aspects of the source language;
3. rewriting of the IL involves adding annotations, simplifications, and sometimes using parts of the program itself like rewrite rules (for example for inlining defined functions); often some rewrites depend on the result of other rewrites (like an optimization depending on an analysis); finally,
4. code emission is usually a direct expansion of the “finished” IL program into sequences (or templates) of instructions that are directly executable by a computer.

We’ll show how each of these steps is specified using the CRSX system [18, 19], an implementation of a variation of Combinatory Reduction Systems [11]. The actual samples we’ll present below are mere toys, of course, but they do illustrate the ideas in a manner that is consistent with a production compiler that we are building for XQuery [2].

We first summarize the CRSX system notation, including the extensions, in Section 2, before we introduce the parser specification in Section 3 followed by the normalizer rules in Section 4. Section 5 then explains a few simple sample rewrites, and Section 6 presents code emission rules. Finally, we conclude and discuss some related work in Section 7.

2 CRSX Summary

Our setting is *Combinatory Reduction Systems* [11] as realized by the "CRSX" system [19]. Here we briefly summarize the used notation and where it differs from reference CRS.

Terms are constructed from the basic grammar

$$t ::= v \mid \{e\}C[s,\ldots,s] \mid \{e\}M[t,\ldots,t] \qquad \text{(Terms)}$$
$$s ::= \vec{v}.t \mid t \qquad \text{(Scope)}$$
$$e ::= M \mid e; v : t \mid e; C : t \qquad \text{(Environment)}$$

where variables, v, are written with a lower case letter (including composite units like `v"$x"`), meta-variables, M, must include a hash mark (#) in the name, and all other units (including literal constants) are constructors, C.

Term formation is as shown, where constructions $\{e\}C[s,\ldots,s]$ are non-standard in two ways:

- Each subterm of a construction is a *scope*, which may include a vector of distinct variable "binders" ($\vec{v}$ denotes $v_1v_2\ldots v_n$ for $n > 0$), which can then occur as variables inside the scope (with the usual caveat that the innermost possible scope is used for each particular variable name; this is the only location where the formalism accepts abstraction).
- Each construction has an associated *environment* component, which is a collection of mappings from constructors and variables to terms (in addition to permitting meta-variables for pattern matching against environments).

Meta-applications $\{e\}M[t,\ldots,t]$ are used in rewrite rules of the form

$$\mathit{name}[\mathit{options}] : \mathit{pattern} \rightarrow \mathit{contraction}$$

with the following extended version of the CRS conventions:

- The *name* becomes the name of the rule; it can be replaced with "-" to use a default name.
- The *options* is a comma-separated list of instructions to relax the requirement that all used meta-variables occur exactly once on each side of the rule, that all variables are explicitly scoped, and that all pattern meta-applications permit all in-scope variables (to avoid accidental η-style rules).
- The *pattern* is a term that must be a construction wherein contained meta-applications are applied exclusively to distinct bound variables. The pattern defines what the rule will *match*: specifically the rule will match any subterm where the top constructor matches including have the same number of parameters and binders on the parameter scopes, matching all required environment members, and matching the shape of each parameter term recursively with the addition that pattern meta-applications match any corresponding parameter term provided only the included bound variables occur in the matched term (as usual for CRS; we give examples later). The mapping from the meta-variables with the parameter bound variables to the real term and its bound variables is called a *valuation*; CRSX extends valuations to also map *whole environment meta-variables* and *free variables* to parts of the matched term.
- The $\rightarrow$ is the Unicode U2192 character.
- The *contraction* explains what the matched subterm should be replaced with by the rewrite step. Constructions stand for themselves. Meta-applications stand for copies of what the meta-variable

matched where in turn the matched bound variables are *substituted* by the corresponding arguments provided in the contraction meta-application, in usual CRS fashion. Variables bound in the contraction just stand for themselves but free variables either stand for occurrences of the variable they matched or, as a special feature can be declared “fresh,” which means a new globally unique fresh variable is created [17]. Environments in the contraction can reference matched environment meta-variables extended with additional bindings.

Finally, the CRSX parser permits the following abbreviations borrowed from λ calculus and programming languages:

- Parenthesis are allowed around every term, so `(`t`)` is the same as t;
- $c\,\vec{v}.t$ abbreviates $c[v_1.c[v_2.\cdots c[v_n.t]\cdots]]$ (think $\lambda xyz.t$);
- t_1t_2 abbreviates `@[`t_1`,`t_2`]` and is left recursive so $t_1t_2t_3$ is the same as $(t_1t_2)t_3$;
- t_1`;`t_2 abbreviates `$Cons[`t_1`,`t_2`]` and is right recursive with the addition that omitted segments correspond to `$Nil`, so `(`t_1`;`t_2`;)` corresponds to the term `$Cons[`t_1`,$Cons[`t_2`,$Nil]]`; and
- empty brackets `[]` can be omitted.

3 Parser

The first component of our X compiler is the parsing from X syntax to the AST, which are terms in a higher-order abstract syntax representation [16] of X. Thus the parser has to be instructed for every production in the language how the AST subterm for that production must look, including what binders should be introduced and how they can occur. Figure 1 shows the actual file used to achieve this with the CRSX system’s PG parser generator. (Note that like all files used by the CRSX system, the parser generator file is a Unicode text file which permits us to use special characters.)

The grammar itself is specified as follows:

- `//` introduces comments.
- The first line declares the external “class” name we’ll use for the parser as well as the default and other externally visible non-terminals that the parser can be explicitly requested to parse.
- The rest of the file consists of units that start with a name or some special keyword and end with a period.
- The unit starting with `meta` gives the special notation used for meta-variables when writing rules involving parsed expressions; we’ll return to this in the following section and here just remark that we use a notation for meta-variables inside parsed text which is a subset of the CRSX meta-variable notation, and the unit starting with `skip` declares the white space convention.
- In general, non-terminals are written in angle brackets, like `<P>`, terminals (or defined tokens) are written as simple identifiers, like `v`, and literal tokens are written as strings like `","`.
- Units starting with a non-terminal name are the proper productions. In productions, non-terminals and terminals stand for themselves, we use parenthesis `()` for grouping, and vertical bar `|` for choice—all else is annotations, explained below. So the first two productions could have been written as

```
<P> ::= <E> .
<E> ::= <S> ("," <E> |) .
```

```
// Grammar for X (simple XQuery-like language).
class net.sf.crsx.samples.x.X : <P>, <E>, <S>, <Q>

meta[<E>] ::= "#<PRODUCTION_NAME>" i?, "[", "]" . // Meta-applications over AST.

skip ::= " " | "\r" | "\n" | "\t" .               // White space.

<P> ::= {program} <E> .                           // Program.

<E> ::= <S>:#S (","_$ ⟦#S⟧ <E> | ⟦#S⟧) .          // Expression.


<S> ::= "(" (<E> | {empty}) ")"                   // Simple expression.
     |  "element"_$ <N> "{" <E> "}"
     |  {query} <Q>
     |  "if"_$ <S> "then" <S> "else" <S>
     |  {call} <N> "(" (<E> | {empty}) ")"
     |  v_?
     |  {literal} <L>
     .

<Q> ::= "for"_$ v_x "in" <S> <Q>[x]              // Query.
     |  "let"_$ v_x ":=" <S> <Q>[x]
     |  "where"_$ <S> <Q>
     |  "return"_$ <S>
     .

token v ::= "$" n .                              // Variable tokens.

<N> ::= n_$ .                                    // Names.
token n ::= [A-Za-z_] [A-Za-z0-9_-]* .

<L> ::= l_$ .                                    // Literals.
token l ::= i | s .
token i ::= [0-9]+ .
token s ::= "'" (¬[\'] | "''")* "'" .
```

Figure 1: *x.pg*—parsing X to AST.

```
// for $x in child(doc()) for $y in child(doc()) where eq($x,$y) return plus($x,$y)
"program"[
 "query"[
  "for"[
   "call"["child", "call"["doc", "empty"]],
   v"$x" .
    "for"[
     "call"["child", "call"["doc", "empty"]],
     v"$y" .
      "where"[
       "call"["eq", ","[v"$x", v"$y"]],
       "return"["call"["plus", ","[v"$x", v"$y"]]]]]]]]
```

Figure 2: Example parse from X program to AST.

if we were not interested in generating an AST term.

- Units starting with `token` give the regular expression for the defined token. We use conventional regular expression syntax with character classes written in `[]`s (negated by a preceding ¬ and including ranges), choice with `|`, optionality and repetition with `?+*`, and literal characters as strings.

The purpose of the parsing, however, is to build an AST for the parsed X program. This is achieved by the annotations in the productions.

- The default behaviour is that tokens are (parsed but) ignored and non-terminals are parsed and submitted as subterms to the current context.
- When a production includes a name in braces, like `{program}` in the `<P>` production, this specifies that the production generates an AST term with the tag `program` with all following subtrees as children (specifically up to the end of the current choice).
- When a token is followed by `_$` then this specifies that an AST term using the token as the tag with all following subtrees as children (up to the end of the surrounding choice). So in the `<N>` production, the `n` token is directly used as the tag (with no children since there are no following parsed non-terminals).
- When a generated subterm is followed by a colon (`:`) and a meta-variable name starting with a hash `#`, then this means that the subtree generated from the non-terminal is not echoed to the context but stored with that name for later use in an inserted term in double brackets ⟦...⟧. So, for example, we can read the `<E>` production as follows:
  1. Parse the `<S>` subterm and remember it as `#S` instead of including it in the context.
  2. If the next token is a comma then the result is a term rooted by a comma-tag and with two subtems: the one stored as `#S` generated by ⟦`#S`⟧ and the one generated by the following `<E>`.
  3. Otherwise, the result is just what was stored as `#S` generated by (the second) ⟦`#S`⟧ without any additional tag.

  (The fact that a tag can be omitted is a powerful feature that permits us to confuse the `<E>` and `<S>` non-terminals in the normalization rules, as we shall see.)

- An annotation of `_x`, where `x` can be any lower case variable name identifier, promotes the token value to a *scoped identifier definition* and makes `[x]` after a single non-terminal in the same production indicate that the scope of `x` is that non-terminal. So, for example, in the first choice of the `<Q>` production, the `v` token is used as a variable name which is scoped in the `<Q>` subterm.
- Finally, `_?` after a token indicates that the token must be an *occurrence* of a bound variable.

In summary, the parser specification looks like many other abstract syntax tree generation notations, such as MetaPRL [9] or ANTLR Tree Grammars [15], except for the additional direct support for higher order abstract syntax by explicitly specifying the scoping and a pleasantly compact way to generate terms where tokens are used directly as constructors, which reduces the size of large parsers considerably.

Figure 2 shows a sample AST printed by the CRSX engine for the term shown in the comment. The generated tags are quoted because they would otherwise be mistaken for other CRSX syntax; similarly, actual CRSX variables that do not start with a lower case letter are written as `v"$x"`, *etc.*, which allows us to retain the original X names in the AST. Notice how the AST term binds two variables, one for each `"for"` construct, following the CRSX constraint that binders are only permitted on construction subterms.

4 Normalization

Our sample intermediate language is a variant of nested-relational algebra [21, 5] modified to make the binders of dependent operators explicit so we can exploit the higher-order rewriting capabilities, *e.g.*, we write the map operator as

$$\mathtt{Map}[\mathtt{Dep}[\mathtt{id}.p_2], p_1]$$

with an explicit `Dep` dependency abstraction to scope the "context tuple" (usually denoted by a context sensitive symbol like `ID` in relational algebra).

The actual normalization rules are shown in Figure 3, and exercise most of the features of CRSX:

- We first check that we have the grammar from Section 3 loaded. The grammar enables two notations:
  1. In CRSX syntax, `%P`⟦...⟧ denotes *inline parsing* of the ... text using the `<P>` production of some grammar (that must have been loaded in advance).
  2. Inside parsed text, `#P` denotes *any* subterm where a `<P>` subterm is allowed; for disambiguation, such subterms further permit a numeric marker like `#P2`. (This is what the `meta` declaration in Figure 1 is for.)

  The first rule then expresses that a `<P>`-program containing an `<E>` subterm (they all do) rewrites to the shown `Algebraic`-term, where the `N`-subterm is the one representing the compilation scheme that will lead to the entire AST being normalized recursively.
- Notice that the right hand side of the first rule introduces a binder: `id` is bound in the invocation of `N`. In all the rules for `N` we shall explicitly refer to this variable, however, in those cases it will (locally) be a *free* variable where we do not know the binder.

  Thus all the following rules include the *option* `Free[id]` to indicate that the pattern can use `id` to match a free variable. (This is otherwise not permitted as it is likely to be the result of mistyping.)

  Matching of free variables in this way is inherently problematic for confluence, because it breaks the confluence of developments: if the variable is substituted by something then the rule no longer applies! Thus we need an assurance that *variables that are matched against and substitued are*

```
// N: Normalization scheme: compile from X AST to nested relational algebra IL.
N[(
$CheckGrammar['net.sf.crsx.samples.x.X'] ; // we need to parse X fragments

// Program.
N[%P⟦ #E ⟧] → Algebraic[Dep[id.N[#E, id]]] ;

// Expressions: N[expression, input-tuple] rewrites to operator.

-[Free[id]] : N[%E⟦ ( #S , #E )     ⟧, id] → Concat[N[#S, id], N[#E, id]] ;
-[Free[id]] : N[%S⟦ ()              ⟧, id] → Empty ;
-[Free[id]] : N[%S⟦ #L              ⟧, id] → Literal[#L] ;
-[Free[id]] : N[%S⟦ element #N {#E} ⟧, id] → Element[Literal[#N], N[#E, id]] ;
-[Free[id]] : N[%S⟦ #N(#E)          ⟧, id] → Call[#N, N[#E, id]] ;

-[Free[id]] : N[%S⟦ if #S then #S1 else #S2 ⟧, id]
 → Conditional[N[#S, id], N[#S1, id], N[#S2, id]] ;

-[Free[f,id]] : N[f, id] → Extract[id, f] ;

// Queries.
-[Free[id]] : N[%S⟦ #Q ⟧, id] → NQ[#Q, id, t.t] ;

// NQ[query source, input-variable, t.prefix-operator[t]]

-[Free[id],Fresh[f]] :
NQ[%Q⟦ for $v in #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id3.MapConcat[Dep[id2.
                        Map[Dep[id1.Tuple[ACons[f id1, ANil]]], N[#S, id2]]], #op[id3]]] ;

-[Free[id],Fresh[f]] :
NQ[%Q⟦ let $v := #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id2.MapConcat[Dep[id1.Tuple[(f N[#S, id1];)]], #op[id2]]] ;

-[Free[id]] :
NQ[%Q⟦ where #S #Q ⟧, id, t.#op[t]]
 → NQ[#Q, id, id2.Select[Dep[id1.N[#S, id1]], #op[id2]]] ;

-[Free[id]] :
NQ[%Q⟦ return #S ⟧, id, t.#op[t]]
 → Map[Dep[id1.N[#S, id1]], #op[id]] ;

)]
```

Figure 3: *N.crs*—normalizing X terms to nested-relational algebra.

```
Algebraic[
 (Dep id .
  Map[
   (Dep id1 . Call["plus", Concat[Extract[id1, v"$x"], Extract[id1, v"$y"]]]),
   Select[
    (Dep id1_1 . Call["eq", Concat[Extract[id1_1, v"$x"], Extract[id1_1, v"$y"]]]),
    MapConcat[
     (Dep id2 .
      Map[(Dep id1_2 . Tuple[ACons[(v"$y" id1_2), ANil]]), Call["child", Call["doc", Empty]]]),
     MapConcat[
      (Dep id2_1 .
       Map[(Dep id1_3 . Tuple[ACons[(v"$x" id1_3), ANil]]), Call["child", Call["doc", Empty]]]),
      id]]]]))]
```

Figure 4: Normalized version of sample query.

disjoint. For the present system this is ensured by the AST data structures being pure input data in the sense that no rule produces an AST construction, and no AST data is allowed to escape from the `N` wrapper. (The other way to ensure non-substitution is to create globally fresh free variables since only bound variables can be substituted.)

- The next block of rules defines all the easy cases of normalization of sequences, literals, element creation, function calls, conditional, and finally field extraction, which does not involve any X syntax because all fields are converted to free field tag variables, as we shall see.
- Finally, queries are translated backwards [8] using an "operator accumulator" third argument with the `NQ` helper compilation scheme. The first two rules of the `NQ` scheme involve replacing a bound variable with a globally fresh one, which is achieved by the use of higher-order matching and rewriting:
  1. the pattern of the rules includes the fragment `#Q[$v]`, which establishes that the `<Q>` subterm should be matched with "tracking" of all occurrences of the variable bound by the `for` or `let` construct, respectively (the notation used here is determined by the `meta` declaration in the parser description file);
  2. the rules include the option `Fresh[f]`, which makes the used `f` variable in the rules denote a fresh variable instance for each rewrite;
  3. the replacement (or *contraction*) of the rules includes the fragment `#Q[f]`, which substitutes the variable matched in that position with the new fresh variable `f`.

If we try to normalize the same term as before, CRSX outputs what is shown in Figure 4. Notice how the bound variables from the X program are now converted to field tags, which are free variables in the CRSX representation of the nested-relational algebra.

5 Rewriting

The purpose of using a relational algebra intermediate language is usually to rewrite queries to a more optimized form. Figure 5 contains a few such standard optimizations:

- The rules are not related to a compilation scheme and can thus "fire" at any time. This means that implementations should do some kind of completion procedure [13] to ensure that the rules are

```
// R scheme: basic traditional relational optimizations.
R[(
RemoveDepMap[Weak[#dop]] : Dep[id1.MapConcat[#dop[], id1]] → #dop ;
Productize[Weak[#op1]] : MapConcat[Dep[id.#op1[]], #op2] → Product[#op1, #op2] ;
)]
```

Figure 5: *R.crs*—simple relational optimizations.

```
Algebraic[
 (Dep id .
  Map[
   (Dep id1 . Call["plus", Concat[Extract[id1, v"$x"], Extract[id1, v"$y"]]]),
   Select[
    (Dep id1_1 . Call["eq", Concat[Extract[id1_1, v"$x"], Extract[id1_1, v"$y"]]]),
    Product[
     Map[(Dep id1_2 . Tuple[ACons[(v"$y" id1_2), ANil]]), Call["child", Call["doc", Empty]]],
     Product[
      Map[(Dep id1_3 . Tuple[ACons[(v"$x" id1_3), ANil]]), Call["child", Call["doc", Empty]]],
      id]]]]))]
```

Figure 6: Rewritten version of sample query.

applied properly, for example inserting a check for the application of these rules when a `Dep` term in one of the involved constructors is created.

- The `RemoveDepMap` rule includes the special `Weak[#dop]` option. This option states that the pattern for the `#dop` meta-variable *may* have an incomplete list of binders to indicate that the missing binders do not occur (free) in matching subterms. We exploit this in the pattern by not listing the one bound variable, `id1`, as an argument to the meta-application of `#dop` to ensure that the subterm matching the meta-application does not contain `id1`, which permits us to use it in the replacement without providing a substitution for `id1`. Thus the rule states that nesting of a dependent operator can be ignored if the dependent operator does not in fact depend on the nested tuple.
- Similarly, the `Productize` rule states that if the dependent operator of nesting is independent of the dependency then the two can be rewritten to a simple product. The final rewrite here merely permits delaying tests, which allows combining the tests.

We shall not show any specific rules that perform annotation but just mention that they typically take the form of an “annotation scheme” like

```
{id:#cType}Type[id] → #cType
```

where an environment in `{}`s is used to pass the types of variables to the individual subterms and construct their type (for the specifics of the CRSX environment notation see the appendix). For more complex analyses, inference rules like

$$\frac{\rho \vdash p_2 : t_2 \quad \rho + (i : t_2) \vdash p_1 : t}{\rho \vdash \mathtt{Map}[\mathtt{Dep}\ i.p_1, p_2] : t}$$

are encoded with generated rule schemes that rewrite terms like $\{\rho\}\vdash ?[p]$ to $\vdash ![t]$ when the rules can prove $\rho \vdash p : t$, which is encoded for the above rule as follows (shown without options):

```
{#rho}"⊢?"[Map[Dep i.#p1[i], #p2]]
  → {#rho}"⊢??"[∀ i."⊢?1"[i, #p1[i], #p2, {#rho}"⊢?"[#p2]]] ;
{#rho}"⊢?1"[i, #p1, #p2, "⊢!"[#t2]]
  → {#rho}"⊢?2"[i, #p1, #p2, #t2, {#rho;i:#t2}"⊢?"[#p1]] ;
{#rho}"⊢?2"[i, #p1, #p2, #t2, "⊢!"[#t]] → "⊢!!"[i, #p1, #p2, #t2, #t] ;
{#rho}"⊢??"[∀ i."⊢!!"[i, #p1[i], #p2, #t2, #t]] → "⊢!"[#t] ;
```

that introduce helper translation schemes to build the proof of the inference rules in a strictly deterministic left to right fashion. (This is automated by a CRSX meta-rewrite system in the real compiler.)

6 Code Emission

The generated code will use data flow macros, as is established practice for such compilers, but using higher-order terms. The rules for code emission are shown in Figure 7, and correspond closely to the usual operational semantics of the nested-relational operators:

- The top level emission translation scheme is `E`, which creates a "main" target program with explicit binders for the input and output channels.
- The body of the main program is a "pipe," which connects the input to the program and the program to the output. It is implemented by `TPipe`, the workhorse that creates a pair of a handler and a cursor, where the cursor is iterated over once for each value received by the handler: this iteration is what enables the identification of "tuple" with the usual "frame" because a tuple of values sent to a handler is the same as the frame of registers received by the iteration code through the cursor.
- The subsceheme `E2` translates each algebraic construct to an explicit data flow. Concatenation, for example, is achieved by doing the code in sequence with output to the same handler.
- Function call is interesting as the data flow architecture dictate that the way to instantiate a new frame for executing the function is to create a handler to send the function's arguments to and then invoke the function including the handler to which the result should be sent.
- Records (in relational algebra called "tuples") are represented as terms by recursive lists with a member per field.
- We use CRSX variables as "data flow register" represented by field tags, cursors representing the current value of an iteration, and handlers that can receive values for iteration; one can say that we use free CRSX variables similar to the way traditional code generation uses an "infinite register model."
- Control instructions combine existing pieces of code; the `TSwitch` code generator is the only branch construct that receives a single value on a handler and delegates to the branch marked with that value (or, for elements, the tag of the value).
- The data manipulation macros correspond to usual register lookup, frame copy, and frame merge operations.
- The last rules show how relational algebraic operators are translated into pipes and merges.

Running our example through code emission gives the result shown in Figure 8.[1]

[1] The mechanisms used are rather crude. Notice for example how the `Product` operators result in the code building element containers to cache the columns.

```
// E scheme: emit executable "pipeline code" from nested-relational algebra.
E[(

// Main program is a pipe from input cursor to output handler.
E[Algebraic[Dep id.#op[id]]] → TMain[in out.TPipe[h.TCopy[in, h], c.E2[#op[c], out]]] ;

// E2[ operator, handler ] generates code for operator to send the result value to the handler.

-[Free[h]]     : E2[Concat[#1, #2]   , h] → TSeq[E2[#1, h], E2[#2, h]] ;

-[Free[h]]     : E2[Empty            , h] → TNoop ;

-[Free[h]]     : E2[Literal[#N]      , h] → TLiteral[#N, h] ;

-[Free[h]]     : E2[Element[#1, #2]  , h]
 → TMakeElement[labelh.E2[#1, labelh], contenth.E2[#2, contenth], h] ;

-[Free[h]]     : E2[Call[#fun, #args], h] → TCall[#fun, argsh . E2[#args, argsh], h] ;

-[Free[c,f,h]] : E2[Extract[c,f]     , h] → TPick[c, f, h] ;
-[Free[h]]     : E2[Tuple[#fs]       , h] → MkT[#fs, TDNil, vh.TNoop, h] ;

// Helper to generate tuples.
-[Free[f,h]] : MkT[ACons[f #, #fs], #td, vh.#e[vh], h]
 → MkT[#fs, TDCons[f, #td], vh.TSeq[E2[#,vh], #e[vh]], h] ;
-[Free[h]]   : MkT[ANil           , #td, vh.#e[vh], h] → TMakeTuple[#td, vh.#e[vh], h] ;

-[Free[h]] : E2[Conditional[#,#1,#2], h]
 → TSwitch[caseh . E2[#, caseh], TCase[True, E2[#1, h], TOtherwise[E2[#2, h]]]] ;

// Basic queries.

-[Free[h]] : E2[Map[Dep id.#dop[id], #], h] → TPipe[h1.E2[#, h1], c1.E2[#dop[c1], h]] ;

-[Free[h]] : E2[Select[Dep id.#dop[id], #], h]
 → TPipe[h1.E2[#, h1],
  c1.TSwitch[caseh . E2[#dop[c1], caseh], TCase[True, TCopy[c1, h], TOtherwise[TEmpty]]]] ;

-[Free[h]] : E2[MapConcat[Dep id.#dop[id], #], h]
 → TPipe[h1.E2[#, h1], c1.TPipe[h2.E2[#dop[c1], h2], c2.TMerge[c1, c2, h]]] ;

-[Free[h]] : E2[Product[#1, #2], h]
 → TPipe[h2.TMakeElement[lh.TLiteral['Columns', lh], ch.E2[#2,ch],
  c2.TPipe[h1.E2[#1, h1], c1.TPipe[h2.TCall["child", nh.TCopy[c2,nh]], c3.TMerge[c1, c3, h]]]]] ;

)]
```

Figure 7: *E.crs*—emit code.

```
TMain[
 in out .
  TPipe[
   h. TCopy[in, h],
   id .
    TPipe[
     h1 .
      TPipe[
       h1_1 .
        TPipe[
         h2 .
          TMakeElement[
           lh. TLiteral[Columns, lh],
           ch .
            TPipe[
             h2_1. TMakeElement[lh_1. TLiteral[Columns, lh_1], ch_1. TCopy[id, ch_1]],
             c2 .
              TPipe[
               h1_2 .
                TPipe[
                 h1_3. TCall["child", argsh. TCall["doc", argsh_1. TNoop, argsh], h1_3],
                 id1. TMakeTuple[TDCons[v"$x", TDNil], vh. TSeq[TCopy[id1, vh], TNoop], h1_2]],
               c1. TPipe[h2_2. TCall["child", nh. TCopy[c2, nh]], c3. TMerge[c1, c3, ch]]]]],
         c2_1 .
          TPipe[
           h1_4 .
            TPipe[
             h1_5. TCall["child", argsh_2. TCall["doc", argsh_3. TNoop, argsh_2], h1_5],
             id1_1. TMakeTuple[TDCons[v"$y", TDNil], vh_1. TSeq[TCopy[id1_1, vh_1], TNoop], h1_4]],
           c1_1. TPipe[h2_3. TCall["child", nh_1. TCopy[c2_1, nh_1]], c3_1. TMerge[c1_1, c3_1, h1_1]]]],
       id1_2 .
        TSwitch[
         caseh .
          TCall["eq", argsh_4. TSeq[TPick[id1_2, v"$x", argsh_4], TPick[id1_2, v"$y", argsh_4]], caseh],
         TCase[True, TCopy[id1_2, h1], TOtherwise[TEmpty]]]],
     id1_3 .
      TCall["plus", argsh_5. TSeq[TPick[id1_3, v"$x", argsh_5], TPick[id1_3, v"$y", argsh_5]], out]]]]
```

Figure 8: Sample emitted code.

One important issue that we have to resolve in practice is to get all the optimizations to be applied *before* code generation. This requires a study of the critical pairs of the system. The system as presented here, for example, has an overlap between the `RemoveDepMap` optimization rule and the `E2 MapConcat` rule. The solution in this case is *not* traditional completion as that will effectively mean that all optimizations have to be equivalently implemented in the IL and TL but rather we simple block the cases for code generation that can be handled by an optimization rule. So the actual `E2 MapConcat` rule looks like this:

```
-[Free[h]] : E2[MapConcat[Dep id.$[NotMatch,#dop[],#dop[id]], #], h]
→ TPipe[h1.E2[#, h1], c1.TPipe[h2.E2[#dop[c1], h2], c2.TMerge[c1, c2, h]]] ;
```

(In practice, such choices are delegated to an analysis phase which drops cookies of some kind into the term to serve as enablers of the overlapping rewrite steps.)

7 Discussion

At the end what remains is to put all the pieces together. The driver is the top-level X symbol introduced by parsing. We add a small "driver file" that essentially rewrites $\mathtt{E}[\![\mathtt{N}[\![q]\!]]\!]$ for queries q.

I have found that this kind of architecture is quite consistent with what compiler development teams expect even if the notations used are of a more formal nature than most developers usually work with. The support for traditional "compiler block diagrams" like the one in the introduction, where the fact that each analysis and translation is specified independently makes using a structured approach realistic. The chaotic nature of the resulting execution of the specification comes out as an advantage and our implementation using a standard functional innermost-needed strategy often ends up interleaving the stages of the compilation in interesting ways, for example eliminating dead code before type checking, usually making mistakes in dependencies blatantly obvious. (Indeed, rewriting permits tweaking the reduction order or using tricks such as completion to discover bad dependencies early.) However, debugging of rule systems is very different from usual debugging in that mistakes show up as "unsimplified blobs or term," which is different from actual crashes (and requires strict discipline in naming the various modular components in a globally identifyable way, something we have side-stepped in this brief presentation).

Although we have not covered it here, we have observed that the rewrite systems obtained can even themselves be translated mechanically to low-level code, making it feasible to implement the actual production compiler direclty from the rewrite rules. Important factors in this has been the disciplined use of systems that can be transformed into orthogonal constructor systems, for which a table-driven normalizing strategy can be used in almost all cases (there is a performance penalty for some substitution cases).

The CRSX system implements higher order rewriting fully in the form of CRS, thus can handle full substitution and thus express transformations such as inlining. However, it turns out that many specific systems share with the small ones presented here the property that they use only "explicit substitution" style rewrites, which only permits observing variables [1]. Indeed it seems that the fact that the approach is *not* functional or a full logical framework is an advantage: the expressive power of explicit substitution is strictly smaller (in a complexity sense) than general functions.

Finally, a crucial component in using rewriting for specifying large rule sets as is the case in the real compiler is the strict shape requirements on rules: basically every aspect of a rule that is not strictly linear and only substitutes bound variables for bound variables without any constraints is an error unless it is explicitly requested: this purely syntactic approach catches numerous errors early.

Related Work. The area of verifying a compiler specification is well established using both hand-written and mechanical proofs [6]. Work has also been done on linking correct compiler specification and implementations using generic proof theoretic tools [14]. Tools supporting mechanical generation of compilers from specifications, such as SDF+ASF [3] and Stratego [4], have focused on compilers restricted to first-order representations of intermediate languages used by the compiler and on using explicit rewriting strategies to guide compilation. Our goal is the opposite: to only specify dependencies between components of the compiler and leave the actual rewriting strategy to the system (in practice using analysis-guided rule transformations coupled with a generic normalizing strategy).

We are only aware of one published work that uses higher order features with compiler construction, namely the work by Hickey and Nogin on specifying compilers using logical frameworks [9]. The resulting specification looks very similar to ours, and indeed one can see the code synthesis that could be done for their logic system as similar to the code generation we are employing. Also, both systems employ embedded source language syntax and higher-order abstract syntax. However, there are differences as well. First, CRSX is explicitly designed to implement just the kind of rewrite systems that we have described, and is tuned to generate code that drives transformation through lookup tables. Second, variables are first class in CRSX and not linked to meta-level abstraction, thus closer to the approach used by explicit substitution for CRS [1] and “nominal” rewriting [7]. This permits us, for example, to use an assembly language with mutable registers. Third, we find that the focus on local rewriting rules is easier to explain to compiler writers, and the inclusion of environments and inference rules in the basic notation further helps. Finally, the CRSX engine has no assumed strategy so we find the notion of local correctness easier to grasp.

What’s Next? With CRSX we continue to experiment with pushing the envelope for supporting more higher-order features without sacrificing efficiency.

An important direction is to connect with nominal rewriting and understand the relationship between what the two formalisms can express.

Another interesting direction for both performance and analysis is to introduce explicit *weakening* operators that “unbind” a given bound variable in a part of its scope. While used in this way with explicit substitution [20, 10], the interaction with higher-order rewriting is not yet clear.

In companion papers we explain the details of the translation from the supported three forms of rules, “recursive compilation scheme,” “chaotic annotation rules,” and “deterministic inference rules,” into effective native executables, and we explain annotations that make it feasible to avoid rewriting-specific static mistakes.

Acknowledgements. The author is grateful for insightful comments by the anonymous referees including being made aware of the work in logical frameworks.

References

[1] Roel Bloo & Kristoffer H. Rose (1996): *Combinatory Reduction Systems with Explicit Substitution that Preserve Strong Normalisation*. In Harald Ganzinger, editor: *RTA ’96—Rewriting Techniques and Applications. Lecture Notes in Computer Science* 1103, Rutgers University, Springer-Verlag, New Brunswick, New Jersey, pp. 169–183, doi:10.1007/3-540-61464-8_51.

[2] Scott Boag, Don Chamberlain, Mary F. Fernández, Daniela Florescu, Jonathan Robie & Jérôme Siméon (2007): *XQuery 1.0: An XML Query Language*. W3C Recommendation, World Wide Web Consortium. Available at http://www.w3.org/TR/2007/REC-xquery-20070123/.

[3] M.G.J. van den Brand, J. Heering, P. Klint & P. A. Olivier (2002): *Compiling Language Definitions: The ASF+SDF Compiler*. *ACM Transactions on Programming Languages and Systems* 24(4), pp. 334–368, doi:10.1145/567097.567099.

[4] Martin Bravenboer, Arthur van Dam, Karina Olmos & Eelco Visser (2006): *Program Transformation with Scoped Dynamic Rewrite Rules*. *Fundamenta Informaticae* 69(1–2), pp. 123–178.

[5] Sophie Cluet & Guido Moerkotte (1993): *Nested Queries in Object Bases*. In: *In Proc. Int. Workshop on Database Programming Languages*. pp. 226–242.

[6] Maulik A. Dave (2003): *Compiler verification: a bibliography*. *SIGSOFT Softw. Eng. Notes* 28(6), pp. 2–2, doi:10.1145/966221.966235.

[7] Maribel Fernández & Murdoch J. Gabbay (2007): *Nominal rewriting*. *Inf. Comput.* 205(6), pp. 917–965, doi:10.1016/j.ic.2006.12.002.

[8] Giorgio Ghelli, Nicola Onose, Kristoffer H. Rose & Jérôme Siméon (2007): *A better semantics for XQuery with side-effects*. In: *DBPL’07: Proceedings of the 11th international conference on Database programming languages*. Springer-Verlag, Berlin, Heidelberg, pp. 81–96, doi:10.1007/978-3-540-75987-4_6.

[9] Jason Hickey & Aleksey Nogin (2006): *Formal Compiler Construction in a Logical Framework*. *Higher-Order and Symb. Comp.* 19(2-3), pp. 197–230, doi:10.1007/s10990-006-8746-6.

[10] Delia Kesner & Fabien Renaud (2009): *The Prismoid of Resources*. In: *34th International Symposium on Mathematical Foundations of Computer Science (MFCS)*. *LNCS* 5734, Springer-Verlag, Novy Smokovec, High Tatras, Slovakia, pp. 464–476, doi:`10.1007/978-3-642-03816-7_40`.

[11] Jan Willem Klop, Vincent van Oostrom & Femke van Raamsdonk (1993): *Combinatory Reduction Systems: Introduction and Survey*. *Theoretical Computer Science* 121, pp. 279–308, doi:`10.1016/0304-3975(93)90091-7`.

[12] Donald E. Knuth (1968): *Semantics of Context-Free Languages*. *Mathematical Systems Theory* 2(2), pp. 127–145.

[13] Donald E. Knuth & P. Bendix (1970): *Simple Word Problems in Universal Algebras*. In J. Leech, editor: *Computational Problems in Abstract Algebra*. Pergamon Press, Elmsford, N.Y., pp. 263–297.

[14] Koji Okuma & Yasuhiko Minamide (2003): *Executing Verified Compiler Specification*. In Atsushi Ohori, editor: *APLAS 2003—First Asian Symposium on Programming Languages and Systems*. *Lecture Notes in Computer Science* 2895, Springer, Beijing, China, pp. 178–194, doi:`10.1007/978-3-540-40018-9_13`.

[15] Terence Parr (2008): *ANTLR v3 Tree Grammars*. Available at `http://www.antlr.org/wiki/display/ANTLR3/Tree+construction`.

[16] Frank Pfenning, & Conal Elliot (1988): *Higher-Order Abstract Syntax*. *SIGPLAN Notices* 23(7), pp. 199–208, doi:`10.1145/960116.54010`.

[17] Kristoffer H. Rose (1996): *Operational Reduction Models for Functional Programming Languages*. Ph.D. thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø. `http://krisrose.net/thesis.pdf`.

[18] Kristoffer H. Rose (2007): *CRSX – An Open Source Platform for Experimenting with Higher Order Rewriting*. Presented in absentia at HOR 2007—`http://kristoffer.rose.name/papers`.

[19] Kristoffer H. Rose (2010): *Combinatory Reduction Systems with Extensions*. `http://crsx.sourceforge.net`.

[20] Kristoffer H. Rose, Roel Bloo & Frédéric Lang (2009): *On Explicit Substitution with Names*. IBM Research Report RC24909, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA. Available at `http://domino.research.ibm.com/library/cyberdig.nsf/reportnumber/rc24909`. To appear in Journal of Automated Reasoning.

[21] Mark A. Roth, Herry F. Korth & Abraham Silberschatz (1988): *Extended algebra and calculus for nested relational databases*. *ACM Trans. Database Syst.* 13(4), pp. 389–417, doi:`10.1145/49346.49347`.

Uncurrying for Innermost Termination and Derivational Complexity[*]

Harald Zankl,[1] Nao Hirokawa,[2] and Aart Middeldorp[1]

[1] Institute of Computer Science, University of Innsbruck, Austria
{harald.zankl,aart.middeldorp}@uibk.ac.at

[2] School of Information Science, Japan Advanced Institute of Science and Technology, Japan
hirokawa@jaist.ac.jp

First-order applicative term rewriting systems provide a natural framework for modeling higher-order aspects. In earlier work we introduced an uncurrying transformation which is termination preserving and reflecting. In this paper we investigate how this transformation behaves for innermost termination and (innermost) derivational complexity. We prove that it reflects innermost termination and innermost derivational complexity and that it preserves and reflects polynomial derivational complexity. For the preservation of innermost termination and innermost derivational complexity we give counterexamples. Hence uncurrying may be used as a preprocessing transformation for innermost termination proofs and establishing polynomial upper and lower bounds on the derivational complexity. Additionally it may be used to establish upper bounds on the innermost derivational complexity while it neither is sound for proving innermost non-termination nor for obtaining lower bounds on the innermost derivational complexity.

1 Introduction

Proving termination of first-order applicative term rewrite systems is challenging since the rules lack sufficient structure. But these systems are important since they provide a natural framework for modeling higher-order aspects found in functional programming languages. Since proving termination is easier for innermost than for full rewriting we lift some of the recent results from [8] from full to innermost termination. For the properties that do not transfer to the innermost setting we provide counterexamples. Furthermore we show that the uncurrying transformation is suitable for proving upper bounds on the (innermost) derivational complexity.

We remark that our approach on proving innermost termination also is beneficial for functional programming languages that adopt a lazy evaluation strategy since applicative term rewrite systems modeling functional programs are left-linear and non-overlapping. It is well known that for this class of systems termination and innermost termination coincide (see [5] for a more general result).

The remainder of this paper is organized as follows. After recalling preliminaries in Section 2, we show that uncurrying preserves innermost non-termination (but not innermost termination) in Section 3. In Section 4 we show that it preserves and reflects derivational complexity of rewrite systems while it only reflects innermost derivational complexity. Section 5 reports on experimental results and we conclude in Section 6.

[*]This research is supported by FWF (Austrian Science Fund) project P18763 and the Grant-in-Aid for Young Scientists Nos. 20800022 and 22700009 of the Japan Society for the Promotion of Science.

E. Bonelli (Ed.): 5th International Workshop
on Higher-Order Rewriting (HOR'10)
EPTCS 49, 2011, pp. 46–57, doi:10.4204/EPTCS.49.4

2 Preliminaries

In this section we fix preliminaries on rewriting, complexity and uncurrying.

2.1 Term Rewriting

We assume familiarity with term rewriting [1, 17]. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a set of variables disjoint from $\mathcal{F}$. By $\mathcal{T}(\mathcal{F},\mathcal{V})$ we denote the set of terms over $\mathcal{F}$ and $\mathcal{V}$. The size of a term t is denoted $|t|$. A rewrite rule is a pair of terms (ℓ,r), written $\ell \to r$, such that ℓ is not a variable and all variables in r occur in ℓ. A term rewrite system (TRS for short) is a set of rewrite rules. A TRS $\mathcal{R}$ is said to be duplicating if there exist a rewrite rule $\ell \to r \in \mathcal{R}$ and a variable x that occurs more often in r than in ℓ.

Contexts are terms over the signature $\mathcal{F} \cup \{\Box\}$ with exactly one occurrence of the fresh constant $\Box$ (called hole). The expression $C[t]$ denotes the result of replacing the hole in C by the term t. A substitution σ is a mapping from variables to terms and $t\sigma$ denotes the result of replacing the variables in t according to σ. Substitutions may change only finitely many variables (and are thus written as $\{x_1 \mapsto t_1,\dots,x_n \mapsto t_n\}$). The set of positions of a term t is defined as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\}$ if t is a variable and as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\} \cup \{iq \mid q \in \mathcal{P}\mathsf{os}(t_i)\}$ if $t = f(t_1,\dots,t_n)$. Positions are used to address occurrences of subterms. The subterm of t at position $p \in \mathcal{P}\mathsf{os}(t)$ is defined as $t|_p = t$ if $p = \epsilon$ and as $t|_p = t_i|_q$ if $p = iq$. We say a position p is to the right of a position q if $p = p_1ip_2$ and $q = q_1jq_2$ with $p_1 = q_1$ and $i > j$. For a term t and positions $p,q \in \mathcal{P}\mathsf{os}(t)$ we say $t|_p$ is to the right of $t|_q$ if p is to the right of q.

A rewrite relation is a binary relation on terms that is closed under contexts and substitutions. For a TRS $\mathcal{R}$ we define $\to_{\mathcal{R}}$ to be the smallest rewrite relation that contains $\mathcal{R}$. We call $s \to_{\mathcal{R}} t$ a rewrite step if there exist a context C, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. In this case we call $\ell\sigma$ a redex and say that $\ell\sigma$ has been contracted. A root rewrite step, denoted by $s \to_{\mathcal{R}}^{\epsilon} t$, has the shape $s = \ell\sigma \to_{\mathcal{R}} r\sigma = t$ for some $\ell \to r \in \mathcal{R}$. A rewrite sequence is a sequence of rewrite steps. The set of normal forms of a TRS $\mathcal{R}$ is defined as $\mathit{NF}(\mathcal{R}) = \{t \in \mathcal{T}(\mathcal{F},\mathcal{V}) \mid t \text{ contains no redexes}\}$. A redex $\ell\sigma$ in a term t is called innermost if proper subterms of $\ell\sigma$ are normal forms, and rightmost innermost if in addition $\ell\sigma$ is to the right of any other redex in t. A rewrite step is called innermost (rightmost innermost) if an innermost (rightmost innermost) redex is contracted, written $\xrightarrow{\mathsf{i}}$ and $\xrightarrow{\mathsf{ri}}$, respectively.

If the TRS $\mathcal{R}$ is not essential or clear from the context the subscript $_{\mathcal{R}}$ is omitted in $\to_{\mathcal{R}}$ and its derivatives. As usual, $\to^+$ ($\to^*$) denotes the transitive (reflexive and transitive) closure of $\to$ and $\to^m$ its m-th iterate. A TRS is terminating (innermost terminating) if $\to^+$ ($\xrightarrow{\mathsf{i}}{}^+$) is well-founded.

Let $\mathcal{P}$ be a property of TRSs and let Φ be a transformation on TRSs with $\Phi(\mathcal{R}) = \mathcal{R}'$. We say Φ *preserves* $\mathcal{P}$ if $\mathcal{P}(\mathcal{R})$ implies $\mathcal{P}(\mathcal{R}')$ and Φ *reflects* $\mathcal{P}$ if $\mathcal{P}(\mathcal{R}')$ implies $\mathcal{P}(\mathcal{R})$. Sometimes we call Φ $\mathcal{P}$ preserving if Φ preserves $\mathcal{P}$ and $\mathcal{P}$ reflecting if Φ reflects $\mathcal{P}$, respectively.

2.2 Derivational Complexity

For complexity analysis we assume TRSs to be finite and (innermost) terminating.

Hofbauer and Lautemann [10] introduced the concept of derivational complexity for terminating TRSs. The idea is to measure the maximal length of rewrite sequences (derivations) depending on the size of the starting term. Formally, the *derivation height* of a term t (with respect to a finitely branching and well-founded order $\to$) is defined on natural numbers as $\mathsf{dh}(t,\to) = \max\{m \in \mathbb{N} \mid t \to^m u \text{ for some } u\}$. The *derivational complexity* $\mathsf{dc}_{\mathcal{R}}(n)$ of a TRS $\mathcal{R}$ is then defined as $\mathsf{dc}_{\mathcal{R}}(n) = \max\{\mathsf{dh}(t,\to_{\mathcal{R}}) \mid |t| \leqslant n\}$.

Similarly we define the *innermost* derivational complexity as $\mathrm{idc}_{\mathcal{R}}(n) = \max\{\mathrm{dh}(t, \xrightarrow{\mathrm{i}}_{\mathcal{R}}) \mid |t| \leqslant n\}$. Since we regard finite TRSs only, these functions are well-defined if $\mathcal{R}$ is (innermost) terminating. If $\mathrm{dc}_{\mathcal{R}}(n)$ is bounded by a linear, quadratic, cubic, ... function or polynomial, $\mathcal{R}$ is said to have linear, quadratic, cubic, ... or polynomial derivational complexity. A similar convention applies to $\mathrm{idc}_{\mathcal{R}}(n)$.

For functions $f, g\colon \mathbb{N} \to \mathbb{N}$ we write $f(n) \in \mathcal{O}(g(n))$ if there are constants $M, N \in \mathbb{N}$ such that $f(n) \leqslant M \cdot g(n) + N$ for all $n \in \mathbb{N}$.

One popular method to prove polynomial upper bounds on the derivational complexity is via triangular matrix interpretations [13], which are a special instance of monotone algebras. An $\mathcal{F}$-algebra $\mathcal{A}$ consists of a non-empty carrier A and a set of interpretations $f_{\mathcal{A}}$ for every $f \in \mathcal{F}$. By $[\alpha]_{\mathcal{A}}(\cdot)$ we denote the usual evaluation function of $\mathcal{A}$ according to an assignment α which maps variables to values in A. An $\mathcal{F}$-algebra $\mathcal{A}$ together with a well-founded order $\succ$ on A is called a *monotone algebra* if every $f_{\mathcal{A}}$ is monotone with respect to $\succ$. Any monotone algebra $(\mathcal{A}, \succ)$ induces a well-founded order on terms: $s \succ_{\mathcal{A}} t$ if for any assignment α the condition $[\alpha]_{\mathcal{A}}(s) \succ [\alpha]_{\mathcal{A}}(t)$ holds. A TRS $\mathcal{R}$ is compatible with a monotone algebra $(\mathcal{A}, \succ_{\mathcal{A}})$ if $l \succ_{\mathcal{A}} r$ for all $l \to r \in \mathcal{R}$.

Matrix interpretations $(\mathcal{M}, \succ)$ (often just denoted $\mathcal{M}$) are a special form of monotone algebras. Here the carrier is $\mathbb{N}^d$ for some fixed dimension $d \in \mathbb{N} \setminus \{0\}$. The order $\succ$ is defined on $\mathbb{N}^d$ as $(u_1, \ldots, u_d) \succ (v_1, \ldots, v_d)$ if $u_1 >_{\mathbb{N}} v_1$ and $u_i \geqslant_{\mathbb{N}} v_i$ for all $2 \leqslant i \leqslant d$. If every $f \in \mathcal{F}$ of arity n is interpreted as $f_{\mathcal{M}}(\vec{x_1}, \ldots, \vec{x_n}) = F_1\vec{x_1} + \cdots + F_n\vec{x_n} + \vec{f}$ where $F_i \in \mathbb{N}^{d \times d}$ for all $1 \leqslant i \leqslant n$ and $\vec{f} \in \mathbb{N}^d$ then monotonicity of $\succ$ is achieved by demanding $F_{i(1,1)} \geqslant 1$ for any $1 \leqslant i \leqslant n$. Such interpretations have been introduced in [2].

A matrix interpretation where for every $f \in \mathcal{F}$ all F_i ($1 \leqslant i \leqslant n$ where n is the arity of f) are upper triangular is called *triangular* (abbreviated by TMI). A square matrix A of dimension d is of *upper triangular* shape if $A_{(i,i)} \leqslant 1$ and $A_{(i,j)} = 0$ if $i > j$ for all $1 \leqslant i, j \leqslant d$. The next theorem is from [13].

Theorem 1. *If a TRS $\mathcal{R}$ is compatible with a TMI $\mathcal{M}$ of dimension d then* $\mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n^d)$.

Recent generalizations of this theorem are reported in [14, 18].

2.3 Uncurrying

This section recalls definitions and results from [8].

An *applicative* term rewrite system (ATRS for short) is a TRS over a signature that consists of constants and a single binary function symbol called application which is denoted by the infix and left-associative symbol $\circ$. In examples we often use juxtaposition instead of $\circ$. Every ordinary TRS can be transformed into an ATRS by currying. Let $\mathcal{F}$ be a signature. The currying system $\mathcal{C}(\mathcal{F})$ consists of the rewrite rules

$$f_{i+1}(x_1, \ldots, x_i, y) \to f_i(x_1, \ldots, x_i) \circ y$$

for every n-ary function symbol $f \in \mathcal{F}$ and every $0 \leqslant i < n$. Here $f_n = f$ and, for every $0 \leqslant i < n$, f_i is a fresh function symbol of arity i. The currying system $\mathcal{C}(\mathcal{F})$ is confluent and terminating. Hence every term t has a unique normal form $t{\downarrow}_{\mathcal{C}(\mathcal{F})}$. For instance, $\mathsf{f}(\mathsf{a}, \mathsf{b})$ is transformed into $\mathsf{f\ a\ b}$. Note that we write f for f_0.

Next we recall the uncurrying transformation from [8]. Let $\mathcal{R}$ be an ATRS over a signature $\mathcal{F}$. The *applicative arity* $\mathrm{aa}(f)$ of a constant $f \in \mathcal{F}$ is defined as the maximum n such that $f \circ t_1 \circ \cdots \circ t_n$ is a subterm in the left- or right-hand side of a rule in $\mathcal{R}$. This notion is extended to terms as follows:

$\mathcal{R}$	$\mathcal{U}(\mathcal{R})$	$\mathcal{R}{\downarrow}_{\mathcal{U}(\mathcal{R})}$	$\mathcal{R}_\eta$	$\mathcal{R}_\eta{\downarrow}_{\mathcal{U}(\mathcal{R})}$
$\mathsf{id}\ x \to x$	$\mathsf{id} \circ x \to \mathsf{id}_1(x)$	$\mathsf{id}_1(x) \to x$	$\mathsf{id}\ x \to x$	$\mathsf{id}_1(x) \to x$
$\mathsf{f}\ x \to \mathsf{id}\ \mathsf{f}\ x$	$\mathsf{id}_1(x) \circ y \to \mathsf{id}_2(x,y)$	$\mathsf{f}_1(x) \to \mathsf{id}_2(\mathsf{f},x)$	$\mathsf{f}\ x \to \mathsf{id}\ \mathsf{f}\ x$	$\mathsf{f}_1(x) \to \mathsf{id}_2(\mathsf{f},x)$
	$\mathsf{f} \circ x \to \mathsf{f}_1(x)$		$\mathsf{id}\ x\ y \to x\ y$	$\mathsf{id}_2(x,y) \to x \circ y$

Table 1: Some (transformed) TRSs

aa$(t) =$ aa(f) if t is a constant f and aa$(t_1) - 1$ if $t = t_1 \circ t_2$. Note that aa(t) is undefined if the head symbol of t is a variable. The uncurrying system $\mathcal{U}(\mathcal{R})$ consists of the rewrite rules

$$f_i(x_1,\ldots,x_i) \circ y \to f_{i+1}(x_1,\ldots,x_i,y)$$

for every constant $f \in \mathcal{F}$ and every $0 \leqslant i <$ aa(f). Here $f_0 = f$ and, for every $i > 0$, f_i is a fresh function symbol of arity i. We say that $\mathcal{R}$ is *left head variable free* if aa(t) is defined for every non-variable subterm t of a left-hand side of a rule in $\mathcal{R}$. This means that no subterm of a left-hand side in $\mathcal{R}$ is of the form $t_1 \circ t_2$ where t_1 is a variable. The uncurrying system $\mathcal{U}(\mathcal{R})$, or simply $\mathcal{U}$, is confluent and terminating. Hence every term t has a unique normal form $t{\downarrow}_{\mathcal{U}}$. The *uncurried* system $\mathcal{R}{\downarrow}_{\mathcal{U}}$ is the TRS consisting of the rules $\ell{\downarrow}_{\mathcal{U}} \to r{\downarrow}_{\mathcal{U}}$ for every $\ell \to r \in \mathcal{R}$. However the rules of $\mathcal{R}{\downarrow}_{\mathcal{U}}$ are not enough to simulate an arbitrary rewrite sequence in $\mathcal{R}$. The natural idea is now to add $\mathcal{U}(\mathcal{R})$, but still $\mathcal{R}{\downarrow}_{\mathcal{U}(\mathcal{R})} \cup \mathcal{U}(\mathcal{R})$ is not enough as shown in the next example from [8].

Example 2. Consider the TRS $\mathcal{R}$ in Table 1. Based on aa$(\mathsf{id}) = 2$ and aa$(\mathsf{f}) = 1$ we get three rules in $\mathcal{U}(\mathcal{R})$ and can compute $\mathcal{R}{\downarrow}_{\mathcal{U}(\mathcal{R})}$. The TRS $\mathcal{R}$ is non-terminating but $\mathcal{R}{\downarrow}_{\mathcal{U}(\mathcal{R})} \cup \mathcal{U}(\mathcal{R})$ is terminating.

Let $\mathcal{R}$ be a left head variable free ATRS. The η*-saturated* ATRS $\mathcal{R}_\eta$ is the smallest extension of $\mathcal{R}$ such that $\ell \circ x \to r \circ x \in \mathcal{R}_\eta$ whenever $\ell \to r \in \mathcal{R}_\eta$ and aa$(\ell) > 0$. Here x is a variable that does not appear in $\ell \to r$. In the following we write $\mathcal{U}^+_\eta(\mathcal{R})$ for $\mathcal{R}_\eta{\downarrow}_{\mathcal{U}(\mathcal{R})} \cup \mathcal{U}(\mathcal{R})$. Note that applicative arities are computed before η-saturation.

Example 3. Consider again Table 1. Since aa$(\mathsf{id}) = 2$ but aa$(\mathsf{id}\ x) = 1$ for the rule $\mathsf{id}\ x \to x$ in $\mathcal{R}$ this explains the rule $\mathsf{id}\ x\ y \to x\ y$ in $\mathcal{R}_\eta$. Note that $\mathcal{U}^+_\eta(\mathcal{R})$ is non-terminating.

For a term t over the signature of the TRS $\mathcal{U}^+_\eta(\mathcal{R})$, we denote by $t{\downarrow}_{\mathcal{C}'}$ the result of identifying different function symbols in $t{\downarrow}_{\mathcal{C}}$ that originate from the same function symbol in $\mathcal{F}$. For a substitution σ, we write $\sigma{\downarrow}_{\mathcal{U}}$ for the substitution $\{x \mapsto \sigma(x){\downarrow}_{\mathcal{U}} \mid x \in \mathcal{V}\}$.

From now on we assume that every ATRS is left-head variable free.

We conclude this preliminary section by recalling some results from [8].

Lemma 4 ([8, Lemma 20])**.** *Let σ be a substitution. If t is head variable free then $t{\downarrow}_{\mathcal{U}}\sigma{\downarrow}_{\mathcal{U}} = (t\sigma){\downarrow}_{\mathcal{U}}$.* □

Lemma 5 ([8, Lemma 15])**.** *If $\mathcal{R}$ is an ATRS then $\to_{\mathcal{R}} = \to_{\mathcal{R}_\eta}$.* □

Lemma 6 ([8, Lemmata 26 and 27])**.** *Let $\mathcal{R}$ be an ATRS. If s and t are terms over the signature of $\mathcal{U}^+_\eta(\mathcal{R})$ then (1) $s \to_{\mathcal{R}{\downarrow}_{\mathcal{U}}} t$ if and only if $s{\downarrow}_{\mathcal{C}'} \to_{\mathcal{R}} t{\downarrow}_{\mathcal{C}'}$ and (2) $s \to_{\mathcal{U}} t$ implies $s{\downarrow}_{\mathcal{C}'} = t{\downarrow}_{\mathcal{C}'}$.* □

Lemma 7 ([8, Proof of Theorem 16])**.** *Let $\mathcal{R}$ be an ATRS. If $s \to_{\mathcal{R}} t$ then $s{\downarrow}_{\mathcal{U}} \to^+_{\mathcal{U}^+_\eta(\mathcal{R})} t{\downarrow}_{\mathcal{U}}$.* □

Consequently our transformation is shown to be termination preserving and reflecting.

Theorem 8 ([8, Theorems 16 and 28])**.** *Let $\mathcal{R}$ be an ATRS. The ATRS $\mathcal{R}$ is terminating if and only if the TRS $\mathcal{U}^+_\eta(\mathcal{R})$ is terminating.* □

3 Innermost Uncurrying

Before showing that our transformation reflects innermost termination we show that it does not preserve innermost termination. Hence uncurrying may not be used as a preprocessing transformation for innermost non-termination proofs.

Example 9. Consider the ATRS $\mathcal{R}$ consisting of the rules

$$\mathsf{f}\ x \to \mathsf{f}\ x \qquad\qquad \mathsf{f} \to \mathsf{g}$$

In an innermost sequence the first rule is never applied and hence $\mathcal{R}$ is innermost terminating. The TRS $\mathcal{U}_\eta^+(\mathcal{R})$ consists of the rules

$$\mathsf{f}_1(x) \to \mathsf{f}_1(x) \qquad \mathsf{f} \to \mathsf{g} \qquad \mathsf{f}_1(x) \to \mathsf{g} \circ x \qquad \mathsf{f} \circ x \to \mathsf{f}_1(x)$$

and is not innermost terminating due to the rule $\mathsf{f}_1(x) \to \mathsf{f}_1(x)$.

The next example shows that $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ does not imply $s{\downarrow}_{\mathcal{U}} \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{U}_\eta^+(\mathcal{R})} t{\downarrow}_{\mathcal{U}}$. This is not a counterexample to soundness of uncurrying for innermost termination, but it shows that the proof for the "if-direction" of Theorem 8 (which is based on Lemma 7) cannot be adopted for the innermost case without further ado.

Example 10. Consider the ATRS $\mathcal{R}$ consisting of the rules

$$\mathsf{f} \to \mathsf{g} \qquad\qquad \mathsf{a} \to \mathsf{b} \qquad\qquad \mathsf{g}\ x \to \mathsf{h}$$

and the innermost step $s = \mathsf{f}\ \mathsf{a} \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{g}\ \mathsf{a} = t$. We have $s{\downarrow}_{\mathcal{U}} = \mathsf{f} \circ \mathsf{a}$ and $t{\downarrow}_{\mathcal{U}} = \mathsf{g}_1(\mathsf{a})$. The TRS $\mathcal{U}_\eta^+(\mathcal{R})$ consists of the rules

$$\mathsf{f} \to \mathsf{g} \qquad \mathsf{a} \to \mathsf{b} \qquad \mathsf{g}_1(x) \to \mathsf{h} \qquad \mathsf{g} \circ x \to \mathsf{g}_1(x)$$

We have $s{\downarrow}_{\mathcal{U}} \xrightarrow{\mathsf{i}}_{\mathcal{U}_\eta^+(\mathcal{R})} \mathsf{g} \circ \mathsf{a}$ but the step from $\mathsf{g} \circ \mathsf{a}$ to $t{\downarrow}_{\mathcal{U}}$ is not innermost.

The above problems can be solved if we consider terms that are not completely uncurried. The next lemmata prepare for the proof. Below we write $s \rhd t$ if t is a proper subterm of s.

Lemma 11. *Let $\mathcal{R}$ be an ATRS. If s is a term over the signature of $\mathcal{R}$, $s \in \mathit{NF}(\mathcal{R})$, and $s \to_{\mathcal{U}}^* t$ then $t \in \mathit{NF}(\mathcal{R}_\eta{\downarrow}_{\mathcal{U}})$.*

Proof. From Lemma 6(2) we obtain $s{\downarrow}_{\mathcal{C}'} = t{\downarrow}_{\mathcal{C}'}$. Note that $s{\downarrow}_{\mathcal{C}'} = s$ because s is a term over the signature of $\mathcal{R}$. If $t \notin \mathit{NF}(\mathcal{R}_\eta{\downarrow}_{\mathcal{U}})$ then $t \to_{\mathcal{R}_\eta{\downarrow}_{\mathcal{U}}} u$ for some term u. Lemma 6(1) yields $t{\downarrow}_{\mathcal{C}'} \to_{\mathcal{R}_\eta} u{\downarrow}_{\mathcal{C}'}$ and Lemma 5 yields $s \to_{\mathcal{R}} u{\downarrow}_{\mathcal{C}'}$. Hence $s \notin \mathit{NF}(\mathcal{R})$, contradicting the assumption. The proof is summarized in the following diagram:

$$\begin{tikzcd}
s \arrow[rr, "*", "\mathcal{U}"'] \arrow[d, phantom, "="] & & t \arrow[rr, "\mathcal{R}_\eta{\downarrow}_{\mathcal{U}}"'] \arrow[d, "*", "\mathcal{C}'"'] & & u \arrow[d, "*", "\mathcal{C}'"'] \\
s{\downarrow}_{\mathcal{C}'} \arrow[rr, phantom, "\overset{\text{Lemma 6(2)}}{=}"] & & t{\downarrow}_{\mathcal{C}'} \arrow[rr, "\text{Lemma 6(1)}", "\mathcal{R}_\eta"'] \arrow[rr, bend right, "\mathcal{R}", "\text{Lemma 5}"'] & & u{\downarrow}_{\mathcal{C}'}
\end{tikzcd}$$

□

Lemma 12. $\to_{\mathcal{U}}^* \cdot \rhd \subseteq \rhd \cdot \to_{\mathcal{U}}^*$

Proof. Assume $s \to_{\mathcal{U}}^* t \rhd u$. We show that $s \rhd \cdot \to_{\mathcal{U}}^* u$ by induction on s. If s is a variable or a constant then there is nothing to show. So let $s = s_1 \circ s_2$. We consider two cases.

- If the outermost $\circ$ has not been uncurried then $t = t_1 \circ t_2$ with $s_1 \to_{\mathcal{U}}^* t_1$ and $s_2 \to_{\mathcal{U}}^* t_2$. Without loss of generality assume that $t_1 \unrhd u$. If $t_1 = u$ then $s \rhd s_1 \to_{\mathcal{U}}^* t_1$. If $t_1 \rhd u$ then the induction hypothesis yields $s_1 \rhd \cdot \to_{\mathcal{U}}^* u$ and hence also $s \rhd \cdot \to_{\mathcal{U}}^* u$.
- If the outermost $\circ$ has been uncurried in the sequence from s to t then the head symbol of s_1 cannot be a variable and $\mathrm{aa}(s_1) > 0$. Hence we may write $s_1 = f \circ t_1 \circ \cdots \circ t_i$ and $t = f_{i+1}(t'_1, \ldots, t'_i, s'_2)$ with $t_j \to_{\mathcal{U}}^* t'_j$ for all $1 \leqslant j \leqslant i$ and $s_2 \to_{\mathcal{U}}^* s'_2$. Clearly, $t'_j \unrhd u$ for some $1 \leqslant j \leqslant i$ or $s'_2 \unrhd t$. In all cases the result follows with the same reasoning as in the first case. □

The next lemma states (a slightly more general result than) that an innermost root rewrite step in an ATRS $\mathcal{R}$ can be simulated by an innermost rewrite sequence in $\mathcal{U}_\eta^+(\mathcal{R})$.

Lemma 13. *For every ATRS $\mathcal{R}$ the inclusion ${}_{\mathcal{U}}^*\!\leftarrow \cdot \xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon} \subseteq \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} \cdot {}_{\mathcal{U}}^*\!\leftarrow$ holds.*

Proof. We prove that $s \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} r{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}} \; {}_{\mathcal{U}}^*\!\leftarrow r\sigma$ whenever $s \; {}_{\mathcal{U}}^*\!\leftarrow \ell\sigma \xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon} r\sigma$ for some rewrite rule $\ell \to r$ in $\mathcal{R}$. By Lemma 4 and the confluence of $\mathcal{U}$,

$$s \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{*} (\ell\sigma){\downarrow_{\mathcal{U}}} = \ell{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}} \to_{\mathcal{U}_\eta^+(\mathcal{R})} r{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}} \; {}_{\mathcal{U}}^*\!\leftarrow r\sigma$$

It remains to show that the sequence $s \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{*} (\ell\sigma){\downarrow_{\mathcal{U}}}$ and the step $\ell{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}} \to_{\mathcal{U}_\eta^+(\mathcal{R})} r{\downarrow_{\mathcal{U}}}\sigma{\downarrow_{\mathcal{U}}}$ are innermost with respect to $\mathcal{U}_\eta^+(\mathcal{R})$. For the former, let $s \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{*} C[u] \xrightarrow{\mathsf{i}}{}_{\mathcal{U}} C[u'] \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{*} (\ell\sigma){\downarrow_{\mathcal{U}}}$ with $u \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{\epsilon} u'$ and let t be a proper subterm of u. Obviously $\ell\sigma \to_{\mathcal{U}}^* C[u] \rhd t$. According to Lemma 12, $\ell\sigma \rhd v \to_{\mathcal{U}}^* t$ for some term v. Since $\ell\sigma \xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon} r\sigma$, the term v is a normal form of $\mathcal{R}$. Hence $t \in \mathit{NF}(\mathcal{R}_\eta{\downarrow_{\mathcal{U}}})$ by Lemma 11. Since $u \xrightarrow{\mathsf{i}}{}_{\mathcal{U}}^{\epsilon} u'$, t is also a normal form of $\mathcal{U}$. Hence $t \in \mathit{NF}(\mathcal{U}_\eta^+(\mathcal{R}))$ as desired. For the latter, let t be a proper subterm of $(\ell\sigma){\downarrow_{\mathcal{U}}}$. According to Lemma 12, $\ell\sigma \rhd u \to_{\mathcal{U}}^* t$. The term u is a normal form of $\mathcal{R}$. Hence $t \in \mathit{NF}(\mathcal{R}_\eta{\downarrow_{\mathcal{U}}})$ by Lemma 11. Obviously, $t \in \mathit{NF}(\mathcal{U})$ and thus also $t \in \mathit{NF}(\mathcal{U}_\eta^+(\mathcal{R}))$. □

The next example shows that it is not sound to replace $\xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon}$ by $\xrightarrow{\mathsf{i}}{}_{\mathcal{R}}$ in Lemma 13.

Example 14. Consider the ATRS $\mathcal{R}$ consisting of the rules

$$\mathsf{f} \to \mathsf{g} \qquad\qquad \mathsf{f}\ x \to \mathsf{g}\ x \qquad\qquad \mathsf{a} \to \mathsf{b}$$

Consequently the TRS $\mathcal{U}_\eta^+(\mathcal{R})$ consists of the rules

$$\mathsf{f} \to \mathsf{g} \qquad \mathsf{f}_1(x) \to \mathsf{g}_1(x) \qquad \mathsf{a} \to \mathsf{b} \qquad \mathsf{f} \circ x \to \mathsf{f}_1(x) \qquad \mathsf{g} \circ x \to \mathsf{g}_1(x)$$

We have $\mathsf{f}_1(\mathsf{a}) \; {}_{\mathcal{U}}^*\!\leftarrow \mathsf{f} \circ \mathsf{a} \xrightarrow{\mathsf{i}}{}_{\mathcal{R}} \mathsf{g} \circ \mathsf{a}$ but $\mathsf{f}_1(\mathsf{a}) \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} \cdot {}_{\mathcal{U}}^*\!\leftarrow \mathsf{g} \circ \mathsf{a}$ does not hold. To see that the latter does not hold, consider the two reducts of $\mathsf{g} \circ \mathsf{a}$ with respect to $\to_{\mathcal{U}}^*$: $\mathsf{g}_1(\mathsf{a})$ and $\mathsf{g} \circ \mathsf{a}$. We have neither $\mathsf{f}_1(\mathsf{a}) \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} \mathsf{g}_1(\mathsf{a})$ nor $\mathsf{f}_1(\mathsf{a}) \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} \mathsf{g} \circ \mathsf{a}$.

In order to extend Lemma 13 to non-root positions, we have to use rightmost innermost evaluation. This avoids the situation in the above example where parallel redexes become nested by uncurrying.

Lemma 15. *For every ATRS $\mathcal{R}$ the inclusion ${}_{\mathcal{U}}^*\!\leftarrow \cdot \xrightarrow{\mathsf{ri}}{}_{\mathcal{R}} \subseteq \xrightarrow{\mathsf{i}}{}_{\mathcal{U}_\eta^+(\mathcal{R})}^{+} \cdot {}_{\mathcal{U}}^*\!\leftarrow$ holds.*

Proof. Let $s \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} t = C[\ell\sigma] \xrightarrow{\mathsf{ri}}_{\mathcal{R}} C[r\sigma] = u$ with $\ell\sigma \xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon} r\sigma$. We use induction on C. If $C = \Box$ then $s \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} t \xrightarrow{\mathsf{i}}{}_{\mathcal{R}}^{\epsilon} u$. Lemma 13 yields $s \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{U}^+_\eta(\mathcal{R})} \cdot \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} u$. For the induction step we consider two cases.

- Suppose $C = \Box \circ s_1 \circ \cdots \circ s_n$ and $n > 0$. Since $\mathcal{R}$ is left head variable free, $\mathrm{aa}(\ell)$ is defined. If $\mathrm{aa}(\ell) = 0$ then $s = t' \circ s'_1 \circ \cdots \circ s'_n \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} \ell\sigma \circ s_1 \circ \cdots \circ s_n \xrightarrow{\mathsf{i}}_{\mathcal{R}} r\sigma \circ s_1 \circ \cdots \circ s_n$ with $t' \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} \ell\sigma$ and $s'_j \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} s_j$ for $1 \leqslant j \leqslant n$. The claim follows using Lemma 13 and the fact that innermost rewriting is closed under contexts. If $\mathrm{aa}(\ell) > 0$ we have to consider two cases. In the case where the leftmost $\circ$ symbol in C has not been uncurried we proceed as when $\mathrm{aa}(\ell) = 0$. If the leftmost $\circ$ symbol of C has been uncurried, we reason as follows. We may write $\ell\sigma = f \circ u_1 \circ \cdots \circ u_k$ where $k < \mathrm{aa}(f)$. We have $t = f \circ u_1 \circ \cdots \circ u_k \circ s_1 \circ \cdots \circ s_n$ and $u = r\sigma \circ s_1 \circ \cdots \circ s_n$. There exists an i with $1 \leqslant i \leqslant \min\{\mathrm{aa}(f), k+n\}$ such that $s = f_i(u'_1, \ldots, u'_k, s'_1, \ldots, s'_{i-k}) \circ s'_{i-k+1} \circ \cdots \circ s'_n$ with $u'_j \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} u_j$ for $1 \leqslant j \leqslant k$ and $s'_j \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} s_j$ for $1 \leqslant j \leqslant n$. Because of rightmost innermost rewriting, the terms $u_1, \ldots, u_k, s_1, \ldots, s_n$ are normal forms of $\mathcal{R}$. According to Lemma 11 the terms $u'_1, \ldots, u'_k, s'_1, \ldots, s'_n$ are normal forms of $\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}$. Since $i - k \leqslant \mathrm{aa}(\ell)$, $\mathcal{R}_\eta$ contains the rule $\ell \circ x_1 \circ \cdots \circ x_{i-k} \to r \circ x_1 \circ \cdots \circ x_{i-k}$ where $x_1, \ldots, x_{i-k}$ are pairwise distinct variables not occurring in ℓ. Therefore $\tau = \sigma \cup \{x_1 \mapsto s_1, \ldots, x_{i-k} \mapsto s_{i-k}\}$ is a well-defined substitution. We obtain
$$\begin{array}{lll} s & \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{U}^+_\eta(\mathcal{R})} & f_i(u_1{\downarrow_{\mathcal{U}}}, \ldots, u_k{\downarrow_{\mathcal{U}}}, s_1{\downarrow_{\mathcal{U}}}, \ldots, s_{i-k}{\downarrow_{\mathcal{U}}}) \circ s'_{i-k+1} \circ \cdots \circ s'_n \\ & \xrightarrow{\mathsf{i}}_{\mathcal{U}^+_\eta(\mathcal{R})} & (r \circ x_1 \circ \cdots \circ x_{i-k}){\downarrow_{\mathcal{U}}}\tau{\downarrow_{\mathcal{U}}} \circ s'_{i-k+1} \circ \cdots \circ s'_n \\ & {}_{\mathcal{U}}^{*}\!\!\leftarrow & (r \circ x_1 \circ \cdots \circ x_{i-k})\tau \circ s_{i-k+1} \circ \cdots \circ s_n \;=\; r\sigma \circ s_1 \circ \cdots \circ s_n \;=\; t \end{array}$$
where we use the confluence of $\mathcal{U}$ in the first sequence.

- In the second case we have $C = s_1 \circ C'$. Clearly $C'[\ell\sigma] \xrightarrow{\mathsf{ri}}_{\mathcal{R}} C'[r\sigma]$. If $\mathrm{aa}(s_1) \leqslant 0$ or if $\mathrm{aa}(s_1)$ is undefined or if $\mathrm{aa}(s_1) > 0$ and the outermost $\circ$ has not been uncurried in the sequence from t to s then $s = s'_1 \circ s' \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} s_1 \circ C'[\ell\sigma] \xrightarrow{\mathsf{ri}}_{\mathcal{R}} s_1 \circ C'[r\sigma] = u$ with $s'_1 \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} s_1$ and $s' \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} C'[\ell\sigma]$. If $\mathrm{aa}(s_1) > 0$ and the outermost $\circ$ has been uncurried in the sequence from t to s then we may write $s_1 = f \circ u_1 \circ \cdots \circ u_k$ where $k < \mathrm{aa}(f)$. We have $s = f_{k+1}(u'_1, \ldots, u'_k, s')$ for some term s' with $s' \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} C'[\ell\sigma]$ and $u'_i \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} u_i$ for $1 \leqslant i \leqslant k$. In both cases we obtain $s' \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{U}^+_\eta(\mathcal{R})} \cdot \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} C'[r\sigma]$ from the induction hypothesis. Since innermost rewriting is closed under contexts, the desired $s \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{U}^+_\eta(\mathcal{R})} \cdot \mathrel{{}_{\mathcal{U}}^{*}\!\!\leftarrow} u$ follows. $\Box$

By Lemma 15 and the equivalence of rightmost innermost and innermost termination [16] we obtain the main result of this section.

Theorem 16. *An ATRS $\mathcal{R}$ is innermost terminating if $\mathcal{U}^+_\eta(\mathcal{R})$ is innermost terminating.* $\Box$

4 Derivational Complexity

In this section we investigate how the uncurrying transformation affects derivational complexity for full and innermost rewriting.

4.1 Full Rewriting

It is sound to use uncurrying as a preprocessor for proofs of upper bounds on the derivational complexity:

Theorem 17. *If $\mathcal{R}$ is a terminating ATRS then* $\mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(\mathrm{dc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n))$.

Proof. Consider an arbitrary maximal rewrite sequence $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_m$ which we can transform into the sequence

$$t_0{\downarrow_{\mathcal{U}}} \to^+_{\mathcal{U}^+_\eta(\mathcal{R})} t_1{\downarrow_{\mathcal{U}}} \to^+_{\mathcal{U}^+_\eta(\mathcal{R})} t_2{\downarrow_{\mathcal{U}}} \to^+_{\mathcal{U}^+_\eta(\mathcal{R})} \cdots \to^+_{\mathcal{U}^+_\eta(\mathcal{R})} t_m{\downarrow_{\mathcal{U}}}$$

using Lemma 7. Moreover, $t_0 \to^*_{\mathcal{U}^+_\eta(\mathcal{R})} t_0{\downarrow_{\mathcal{U}}}$ holds. Therefore, $\mathrm{dh}(t_0, \to_{\mathcal{R}}) \leqslant \mathrm{dh}(t_0, \to_{\mathcal{U}^+_\eta(\mathcal{R})})$. Hence $\mathrm{dc}_{\mathcal{R}}(n) \leqslant \mathrm{dc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n)$ holds for all $n \in \mathbb{N}$. □

Next we show that uncurrying preserves polynomial complexity. Hence we disregard duplicating (exponential complexity, cf. [9]) and empty (constant complexity) ATRSs. A TRS $\mathcal{R}$ is called *length-reducing* if $\mathcal{R}$ is non-duplicating and $|\ell| > |r|$ for all rules $\ell \to r \in \mathcal{R}$. The following lemma is an easy consequence of [9, Theorem 23]. Here for a relative TRS $\mathcal{R}/\mathcal{S}$ the derivational complexity $\mathrm{dc}_{\mathcal{R}/\mathcal{S}}(n)$ is based on the rewrite relation $\to_{\mathcal{R}/\mathcal{S}}$ which is defined as $\to^*_{\mathcal{S}} \cdot \to_{\mathcal{R}} \cdot \to^*_{\mathcal{S}}$.

Lemma 18. *Let $\mathcal{R}$ be a non-empty non-duplicating TRS over a signature containing at least one symbol of arity at least two and let $\mathcal{S}$ be a length-reducing TRS. If $\mathcal{R} \cup \mathcal{S}$ is terminating then* $\mathrm{dc}_{\mathcal{R}\cup\mathcal{S}}(n) \in \mathcal{O}(\mathrm{dc}_{\mathcal{R}/\mathcal{S}}(n))$. □

Note that the above lemma does not hold if the TRS $\mathcal{R}$ is empty.

Theorem 19. *Let $\mathcal{R}$ be a non-empty ATRS. If* $\mathrm{dc}_{\mathcal{R}}(n)$ *is in* $\mathcal{O}(n^k)$ *then* $\mathrm{dc}_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}(n)$ *and* $\mathrm{dc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n)$ *are in* $\mathcal{O}(n^k)$.

Proof. Let $\mathrm{dc}_{\mathcal{R}}(n)$ be in $\mathcal{O}(n^k)$ and consider a maximal rewrite sequence of $\to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}$ starting from an arbitrary term t_0:

$$t_0 \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}} t_1 \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}} \cdots \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}} t_m$$

By Lemma 6 we obtain the sequence $t_0{\downarrow_{\mathcal{C}'}} \to_{\mathcal{R}} t_1{\downarrow_{\mathcal{C}'}} \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_m{\downarrow_{\mathcal{C}'}}$. Thus, $\mathrm{dh}(t_0, \to_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}) \leqslant \mathrm{dh}(t_0{\downarrow_{\mathcal{C}'}}, \to_{\mathcal{R}})$. Because $|t_0{\downarrow_{\mathcal{C}'}}| \leqslant 2|t_0|$, we obtain $\mathrm{dc}_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}(n) \leqslant \mathrm{dc}_{\mathcal{R}}(2n)$. From the assumption the right-hand side is in $\mathcal{O}(n^k)$, hence $\mathrm{dc}_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}(n)$ is in $\mathcal{O}(n^k)$. Since $\mathrm{dc}_{\mathcal{R}}(n)$ is in $\mathcal{O}(n^k)$, $\mathcal{R}$ must be non-duplicating and terminating. Because $\mathcal{U}$ is length-reducing, Lemma 18 yields that $\mathrm{dc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n)$ also is in $\mathcal{O}(n^k)$. □

In practice it is recommendable to investigate $\mathrm{dc}_{\mathcal{R}_\eta{\downarrow_{\mathcal{U}}}/\mathcal{U}}(n)$ instead of $\mathrm{dc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n)$, see [19]. The next example shows that uncurrying might be useful to enable criteria for polynomial complexity.

Example 20. Consider the ATRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{add}\ x\ \mathsf{0} \to x \qquad\qquad \mathsf{add}\ x\ (\mathsf{s}\ y) \to \mathsf{s}\ (\mathsf{add}\ x\ y)$$

The system $\mathcal{U}^+_\eta(\mathcal{R})$ consists of the rules

$$\mathsf{add}_2(x,\mathsf{0}) \to x \qquad\qquad \mathsf{add}_2(x,\mathsf{s}_1(y)) \to \mathsf{s}_1(\mathsf{add}_2(x,y))$$

$$\mathsf{add}_1(x) \circ y \to \mathsf{add}_2(x,y) \qquad \mathsf{add} \circ x \to \mathsf{add}_1(x) \qquad \mathsf{s} \circ x \to \mathsf{s}_1(x)$$

The 2-dimensional TMI $\mathcal{M}$

$$\mathsf{add2}_{\mathcal{M}}(\vec{x},\vec{y}) = \circ_{\mathcal{M}}(\vec{x},\vec{y}) = \begin{pmatrix}1\,1\\0\,1\end{pmatrix}\vec{x} + \begin{pmatrix}1\,1\\0\,1\end{pmatrix}\vec{y} \qquad \mathsf{add1}_{\mathcal{M}}(\vec{x}) = \mathsf{s1}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix}1\,0\\0\,1\end{pmatrix}\vec{x} + \begin{pmatrix}0\\1\end{pmatrix}$$
$$\mathsf{add}_{\mathcal{M}} = \mathsf{s}_{\mathcal{M}} = \mathsf{0}_{\mathcal{M}} = \begin{pmatrix}0\\1\end{pmatrix}$$

orients all rules in $\mathcal{U}^+_\eta(\mathcal{R})$ strictly, inducing a quadratic upper bound on the derivational complexity of $\mathcal{U}^+_\eta(\mathcal{R})$ according to Theorem 1 and by Theorem 17 also of $\mathcal{R}$. In contrast, the TRS $\mathcal{R}$ itself does not admit such an interpretation of dimension 2. To see this, we encoded the required condition as a satisfaction problem in non-linear arithmetic over the integers. MiniSmt [20][1] can prove this problem unsatisfiable by simplifying it into a trivially unsatisfiable constraint. Details can be inferred from the website mentioned in Footnote 4.

4.2 Innermost Rewriting

Next we consider innermost derivational complexity. Let $\mathcal{R}$ be an innermost terminating TRS. From a result by Krishna Rao [16, Section 5.1] which has been generalized by van Oostrom [15, Theorems 2 and 3] we infer that $\mathrm{dh}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = \mathrm{dh}(t, \xrightarrow{\mathsf{ri}}_{\mathcal{R}})$ holds for all terms t.

Theorem 21. *If $\mathcal{R}$ is an innermost terminating ATRS then* $\mathrm{idc}_{\mathcal{R}}(n) \in \mathcal{O}(\mathrm{idc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n))$.

Proof. Consider a maximal rightmost innermost rewrite sequence $t_0 \xrightarrow{\mathsf{ri}}_{\mathcal{R}} t_1 \xrightarrow{\mathsf{ri}}_{\mathcal{R}} t_2 \xrightarrow{\mathsf{ri}}_{\mathcal{R}} \cdots \xrightarrow{\mathsf{ri}}_{\mathcal{R}} t_m$. Using Lemma 15 we obtain a sequence

$$t_0 \xrightarrow{\mathsf{i}}{}^+_{\mathcal{U}^+_\eta(\mathcal{R})} t'_1 \xrightarrow{\mathsf{i}}{}^+_{\mathcal{U}^+_\eta(\mathcal{R})} t'_2 \xrightarrow{\mathsf{i}}{}^+_{\mathcal{U}^+_\eta(\mathcal{R})} \cdots \xrightarrow{\mathsf{i}}{}^+_{\mathcal{U}^+_\eta(\mathcal{R})} t'_m$$

for terms $t'_1, t'_2, \ldots, t'_m$ such that $t_i \to^*_{\mathcal{U}} t'_i$ for all $1 \leqslant i \leqslant m$. It follows that $\mathrm{dh}(t_0, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = \mathrm{dh}(t_0, \xrightarrow{\mathsf{ri}}_{\mathcal{R}}) \leqslant \mathrm{dh}(t_0, \xrightarrow{\mathsf{i}}_{\mathcal{U}^+_\eta(\mathcal{R})})$ and we conclude $\mathrm{idc}_{\mathcal{R}}(n) \in \mathcal{O}(\mathrm{idc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n))$. □

As Example 9 showed, uncurrying does not preserve innermost termination. Similarly, it does not preserve innermost polynomial complexity even if the original ATRS has linear innermost derivational complexity.

Example 22. Consider the non-duplicating ATRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{f} \to \mathsf{s} \qquad\qquad \mathsf{f}\ (\mathsf{s}\ x) \to \mathsf{s}\ (\mathsf{s}\ (\mathsf{f}\ x))$$

Since the second rule is never used in innermost rewriting, $\mathrm{idc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$ is easily shown by induction on n. We show that the innermost derivational complexity of $\mathcal{U}^+_\eta(\mathcal{R})$ is at least exponential. The TRS $\mathcal{U}^+_\eta(\mathcal{R})$ consists of the rules

$$\mathsf{f} \to \mathsf{s} \qquad \mathsf{f}_1(x) \to \mathsf{s}_1(x) \qquad \mathsf{f}_1(\mathsf{s}_1(x)) \to \mathsf{s}_1(\mathsf{s}_1(\mathsf{f}_1(x))) \qquad \mathsf{f} \circ x \to \mathsf{f}_1(x) \qquad \mathsf{s} \circ x \to \mathsf{s}_1(x)$$

and one can verify that $\mathrm{dh}(\mathsf{f}_1^n(\mathsf{s}_1(x)), \xrightarrow{\mathsf{i}}_{\mathcal{U}^+_\eta(\mathcal{R})}) \geqslant 2^n$ for all $n \geqslant 1$. Hence, $\mathrm{idc}_{\mathcal{U}^+_\eta(\mathcal{R})}(n+3) \geqslant 2^n$ for all $n \geqslant 0$.

[1]`http://cl-informatik.uibk.ac.at/software/minismt/`

Table 2: Innermost termination for 213 ATRSs.

subterm	matrix (1)	matrix (2)	matrix (3)	matrix (4)
$-$ / $+$	$-$ / $+$	$-$ / $+$	$-$ / $+$	$-$ / $+$
42 / 55	67 / 102	111 / 142	113 / 144	114 / 145

Table 3: (Innermost) derivational complexity for 195 (213) ATRSs.

	TMI (1)	TMI (2)	TMI (3)	TMI (4)
	$-$ / $+$	$-$ / $+$	$-$ / $+$	$-$ / $+$
dc	3 / 4	10 / 14	12 / 26	12 / 28
idc	3 / 4	10 / 14	12 / 26	12 / 28

5 Experimental Results

The results from this paper are implemented in the termination prover T$_\mathsf{T}$T$_2$ [12].[2] Version 7.0.2 of the termination problem data base (TPDB)[3] contains 195 ATRSs for full rewriting and 18 ATRSs for innermost rewriting. All tests have been performed on a single core of a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and 64 GB of main memory.

Experiments[4] give evidence that uncurrying allows to handle significantly more systems. For proving innermost termination we considered two popular termination methods, namely the subterm criterion [7] and matrix interpretations [2] of dimensions one to four. The implementation of the latter is based on SAT solving (cf. [2]). For a matrix interpretation of dimension d we used $5-d$ bits to represent natural numbers in matrix coefficients. An additional bit was used for intermediate results. Both methods are integrated within the dependency pair framework using dependency graph reasoning and usable rules as proposed in [3, 4, 6].

Table 2 shows the number of systems that could be proved innermost terminating. In the table $+$ ($-$) indicates that uncurrying has (not) been used as preprocessing step, e.g., for the subterm criterion the number of successful proofs increases from 42 to 55 if uncurrying is used as a preprocessing transformation. For the setting based on matrix interpretations the gains are even larger. In the table, the numbers in parentheses denote the dimensions of the matrices.

Table 3 shows how uncurrying improves the performance of T$_\mathsf{T}$T$_2$ for derivational complexity. In this table we used TMIs as presented in Theorem 1. Coefficients of TMIs are represented with $\max\{2, 5-d\}$ bits; again an additional bit is allowed for intermediate results. If uncurrying is used as preprocessing transformation, TMIs can, e.g., show 14 systems to have at most quadratic derivational complexity while without uncurrying the method only applies to 10 systems. Since T$_\mathsf{T}$T$_2$ has no special methods for proving *innermost* derivational complexity, the numbers in rows dc and idc coincide.

[2]`http://cl-informatik.uibk.ac.at/software/ttt2/`

[3]`http://termination-portal.org/wiki/TPDB`

[4]`http://cl-informatik.uibk.ac.at/software/ttt2/10hor/`

6 Conclusion

In this paper we studied properties of the uncurrying transformation from [8] for innermost rewriting and (innermost) derivational complexity. The significance of these results has been confirmed empirically.

For proving (innermost) termination of applicative systems we mention transformation $\mathcal{A}$ [3] as related work. The main benefit of the approach in [3] is that in contrast to our setting no auxiliary uncurrying rules are necessary. However, transformation $\mathcal{A}$ only works for *proper* ATRSs without head variables in the (left- and) right-hand sides of rewrite rules. Here proper means that any constant always appears with the same applicative arity.

We are not aware of other investigations dedicated to (derivational) complexity analysis of ATRSs. However, we remark that transformation $\mathcal{A}$ preserves derivational complexity.This is straightforward from [11, Lemma 2.1(3)].

As future work we plan to incorporate the results for innermost termination into the dependency pair processors presented in [8].

References

[1] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press.

[2] J. Endrullis, J. Waldmann & H. Zantema (2008): *Matrix Interpretations for Proving Termination of Term Rewriting*. Journal of Automated Reasoning 40(2-3), pp. 195–220. Available at `http://dx.doi.org/10.1007/s10817-007-9087-9`.

[3] J. Giesl, R. Thiemann & P. Schneider-Kamp (2005): *Proving and Disproving Termination of Higher-Order Functions*. In: Proc. 5th International Workshop on Frontiers of Combining Systems. LNCS 3717, pp. 216–231. Available at `http://dx.doi.org/10.1007/11559306_12`.

[4] J. Giesl, R. Thiemann, P. Schneider-Kamp & S. Falke (2006): *Mechanizing and Improving Dependency Pairs*. Journal of Automated Reasoning 37(3), pp. 155–203. Available at `http://dx.doi.org/10.1007/s10817-006-9057-7`.

[5] B. Gramlich (1995): *Abstract Relations between Restricted Termination and Confluence Properties of Rewrite Systems*. Fundamenta Informaticae 24(1-2), pp. 3–23.

[6] N. Hirokawa & A. Middeldorp (2005): *Automating the Dependency Pair Method*. Information and Computation 199(1-2), pp. 172–199. Available at `http://dx.doi.org/10.1016/j.ic.2004.10.004`.

[7] N. Hirokawa & A. Middeldorp (2007): *Tyrolean Termination Tool: Techniques and Features*. Information and Computation 205(4), pp. 474–511. Available at `http://dx.doi.org/10.1016/j.ic.2006.08.010`.

[8] N. Hirokawa, A. Middeldorp & H. Zankl (2008): *Uncurrying for Termination*. In: Proc. 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS (LNAI) 5330, pp. 667–681. Available at `http://dx.doi.org/10.1007/978-3-540-89439-1_46`.

[9] N. Hirokawa & G. Moser (2008): *Automated Complexity Analysis Based on the Dependency Pair Method*. In: Proc. 4th International Joint Conference on Automated Reasoning. LNCS (LNAI) 5195, pp. 364–380. Available at `http://dx.doi.org/10.1007/978-3-540-71070-7_32`.

[10] D. Hofbauer & C. Lautemann (1989): *Termination Proofs and the Length of Derivations*. In: Proc. 3rd International Conference on Rewriting Techniques and Applications. LNCS 355, pp. 167–177. Available at `http://dx.doi.org/10.1007/3-540-51081-8_107`.

[11] R. Kennaway, J.W. Klop, M.R. Sleep & F.-J. de Vries (1996): *Comparing Curried and Uncurried Rewriting*. Journal of Symbolic Computation 21(1), pp. 15–39.

[12] M. Korp, C. Sternagel, H. Zankl & A. Middeldorp (2009): *Tyrolean Termination Tool 2*. In: *Proc. 20th International Conference on Rewriting Techniques and Applications*. *LNCS* 5595, pp. 295–304. Available at `http://dx.doi.org/10.1007/978-3-642-02348-4_21`.

[13] G. Moser, A. Schnabl & J. Waldmann (2008): *Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations*. In: *Proc. 28th International Conference on Foundations of Software Technology and Theoretical Computer Science*. *LIPIcs* 2, pp. 304–315. Available at `http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2008.1762`.

[14] F. Neurauter, H. Zankl & A. Middeldorp (2010): *Revisiting Matrix Interpretations for Polynomial Derivational Complexity of Term Rewriting*. In: *Proc. 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. *LNCS (ARCoSS)* 6397, pp. 550–564. Available at `http://dx.doi.org/10.1007/978-3-642-16242-8_39`.

[15] V. van Oostrom (2007): *Random Descent*. In: *Proc. 18th International Conference on Rewriting Techniques and Applications*. *LNCS* 4533, pp. 314–328. Available at `http://dx.doi.org/10.1007/978-3-540-73449-9_24`.

[16] M.R.K. Krishna Rao (2000): *Some Characteristics of Strong Innermost Normalization*. *Theoretical Computer Science* 239, pp. 141–164. Available at `http://dx.doi.org/10.1016/S0304-3975(99)00215-7`.

[17] TeReSe (2003): *Term Rewriting Systems*. *Cambridge Tracts in Theoretical Computer Science* 55, Cambridge University Press.

[18] J. Waldmann (2010): *Polynomially Bounded Matrix Interpretations*. In: *Proc. 21st International Conference on Rewriting Techniques and Applications*. *LIPIcs* 6, pp. 357–372. Available at `http://dx.doi.org/10.4230/LIPIcs.RTA.2010.357`.

[19] H. Zankl & M. Korp (2010): *Modular Complexity Analysis via Relative Complexity*. In: *Proc. 21st International Conference on Rewriting Techniques and Applications*. *LIPIcs* 6, pp. 385–400. Available at `http://dx.doi.org/10.4230/LIPIcs.RTA.2010.385`.

[20] H. Zankl & A. Middeldorp (2010): *Satisfiability of Non-Linear (Ir)rational Arithmetic*. In: *Proc. 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. *LNCS (LNAI)* 6355, pp. 481–500.

A standardisation proof for algebraic pattern calculi

Delia Kesner
PPS, CNRS and Université Paris Diderot
France
Delia.Kesner@pps.jussieu.fr

Carlos Lombardi
Depto. de Ciencia y Tecnología
Univ. Nacional de Quilmes
Argentina
clombardi@unq.edu.ar

Alejandro Ríos
Depto. de Computación
Facultad de Cs. Exactas y Naturales
Univ. de Buenos Aires – Argentina
rios@dc.uba.ar

This work gives some insights and results on standardisation for call-by-name pattern calculi. More precisely, we define standard reductions for a pattern calculus with constructor-based data terms and patterns. This notion is based on reduction steps that are needed to match an argument with respect to a given pattern. We prove the Standardisation Theorem by using the technique developed by Takahashi [14] and Crary [2] for λ-calculus. The proof is based on the fact that any development can be specified as a sequence of head steps followed by internal reductions, i.e. reductions in which no head steps are involved.

1 Introduction

Pattern Calculi: Several calculi, called *pattern calculi*, have been proposed in order to give a formal description of pattern matching; i.e. the ability to analyse the different possible forms of the argument of a function in order to decide among different alternative definition clauses.

The *pattern matching* operation is the kernel of the evaluation mechanism of all these formalisms, basically because reduction can only be fired when the argument passed to a given function matches its pattern specification. An analysis of various pattern calculi based on different notions of pattern matching operations and different sets of allowed patterns can be found in [8].

Standardisation: A fundamental result in the λ-calculus is the *Standardisation Theorem*, which states that if a term M β-reduces to a term N, then there is a *standard* β-reduction sequence from M to N which can be seen as a canonical way to reduce terms. This result has several applications, e.g. it is used to prove the non-existence of reduction between given terms. One of its main corollaries is the quasi-leftmost-reduction theorem, which in turn is used to prove the non-existence of a normal form for a given term.

A first study on standardisation for call-by-name λ-calculus appears in [3]. Subsequently, several standardisation methods have been devised, for example [1] Section 11.4, [14], [9] and [13].

While leftmost-outermost reduction gives a standard strategy for call-by-name λ-calculus, more refined notions of reductions are necessary to define standard strategies for call-by-value λ-calculus [13], first-order term rewriting systems [6, 15], Proof-Nets [4], etc.

All standard reduction strategies require the definition of some *selected* redex by means of a partial function from terms to redexes; they all give priority to the selected step, if possible. This selected redex is sometimes called *external* [11], but we will refer here to it as the *head redex* of a term.

It is also worth mentioning a generic standardisation proof [12] that can uniformly treat cal-by-name and call-by-value λ-calculus. It is parameterized over the set of values that allow to fire the beta-reduction rule. However, the set of values are defined there in a global sense, while in pattern calculi being a value strongly depends on the form of the given pattern.

Standardisation in Pattern Calculi: For call-by-name λ-calculus, any term of the form $(\lambda x.M)N$ is a redex, and the head redex for such a term is the whole term. In pattern calculi any term of the form

$(\lambda p.M)N$ is a redex candidate, but not necessarily a redex. The parameter p in such terms can be more complex than a single variable, and the whole term is not a redex if the argument N does not match p, i.e., if N does not verify the structural conditions imposed by p. In this case we will choose as head a reduction step lying inside N (or even inside p) which makes p and N be closer to a possible match. While this situation bears some resemblance with *call-by-value* λ-calculus [13], there is an important difference: both the fact of $(\lambda p.M)N$ being a redex, and whether a redex inside N could be useful to get p and N closer to a possible match, depend on *both N and p*.

The aim of this contribution is to analyse the existence of a standardisation procedure for pattern calculi in a direct way, i.e. without using any complicated encoding of such calculi into some general computational framework [10]. This direct approach aims to put to evidence the fine interaction between reduction and pattern matching, and gives a standardisation *algorithm* which is specified in terms of the combination of computations of independent terms with partial computations of terms depending on some pattern. We hope to be able to extend this algorithmic approach to more sophisticated pattern calculi handling open and dynamic patterns [7].

The paper is organized as follows. Section 2 introduces the calculus, Section 3 gives the main concepts needed for the standardisation proof and the main results, Section 4 presents some lemmas used in the main proofs, Sections 5 and 6 show the main results used in the Standardisation Theorem proof and then the theorem itself; finally, Section 7 concludes and gives future research directions.

2 The calculus

We will study a very simple form of pattern calculus, consisting of the extension of standard λ-calculus with a set of constructors and allowing constructed patterns. This calculus appears for example in Section 4.1 in [8].

Definition 2.1 (Syntax) *The calculus is built upon two different enumerable sets of symbols, the variables x,y,z,w and the constants c,a,b; its syntactical categories are:*

$$\begin{array}{llcl@{\qquad}llcl}
\textbf{Terms} & M,N,Q,R & ::= & x \mid c \mid \lambda p.M \mid MM & \textbf{DataTerms} & D & ::= & c \mid DM \\
\textbf{Patterns} & p,q & ::= & x \mid d & \textbf{DataPatterns} & d & ::= & c \mid dp
\end{array}$$

Free and bound variables of terms are defined as expected as well as α-conversion.

Definition 2.2 (Substitution) *A susbsitution θ is a function from variables to terms with finite domain, where $\mathtt{dom}(\theta) = \{x \ : \ \theta(x) \neq x\}$. The extension of θ to terms is defined as expected. We denote $\theta \ ::= \ \{x_1/M_1, \ldots, x_n/M_n\}$ wherever $\mathtt{dom}(\theta) \subseteq \{x_1, \ldots, x_n\}$. Moreover, for θ, ν substitutions, X a set of variables, we define*

$$\begin{array}{rcl}
\mathtt{var}(\theta) & ::= & \mathtt{dom}(\theta) \bigcup \left(\cup_{x \in \mathtt{dom}(\theta)} \mathtt{fv}(\theta x) \right) \\
\nu\theta & ::= & \left(\cup_{x \in \mathtt{dom}(\theta)} \{x/\nu(\theta x)\} \right) \bigcup \left(\cup_{x \in (\mathtt{dom}(\nu) - \mathtt{dom}(\theta))} \{x/\nu x\} \right) \\
\theta\,|_X & ::= & \cup_{x \in X \cap \mathtt{dom}(\theta)} \{x/\theta x\}
\end{array}$$

Definition 2.3 (Matching) *Let p be a pattern and M a term which do not share common variables. Matching on p and M is a partial function yielding a substitution and defined by the following rules ($\uplus$ on substitutions denotes disjoint union with respect to their domains, being undefined if the domains have a non-empty intersection):*

$$\frac{}{x \ll^{\{x/N\}} N} \qquad \frac{}{c \ll^{\emptyset} c} \qquad \frac{d \ll^{\theta_1} D \quad p \ll^{\theta_2} N \quad \theta_1 \uplus \theta_2 \text{ defined}}{dp \ll^{\theta_1 \uplus \theta_2} DN}$$

We write $p \ll M$ iff $\exists \theta \; p \ll^{\theta} M$. Remark that $p \ll M$ implies that p is linear.

Definition 2.4 (Reduction step) *We consider the following reduction steps modulo α-conversion:*

$$\frac{M \to M'}{MN \to M'N}\,\mathsf{SAppL} \qquad \frac{N \to N'}{MN \to MN'}\,\mathsf{SAppR} \qquad \frac{p \ll^{\theta} N}{(\lambda p.M)N \to \theta M}\,\mathsf{SBeta} \qquad \frac{M \to M'}{\lambda p.M \to \lambda p.M'}\,\mathsf{SAbs}$$

By working modulo α-conversion we can always assume in rule (SBeta) that p and N do not share common variables in order to compute matching.

Lemma 2.5 (Basic facts about the calculus)

a. *(data pattern/term structure) Let $d \in$ **DataPatterns** (resp. $D \in$ **DataTerms**), then $d = cp_1 \dots p_n$ (resp. $D = cM_1 \dots M_n$) for some $n \geq 0$.*

b. *(data patterns only match data terms) Let $d \in$ **DataPatterns**, M a term, such that $d \ll M$. Then $M \in$ **DataTerms**.*

c. *(minimal matches) If $p \ll^{\theta} M$ then $\mathtt{dom}(\theta) = \mathtt{fv}(p)$.*

d. *(uniqueness of match) If $p \ll^{\theta_1} M$ and $p \ll^{\theta_2} M$, then $\theta_1 = \theta_2$.*

Crucial to the standardisation proof is the concept of development, we formalize it through the relation $\triangleright$, meaning $M \triangleright N$ iff there is a development (not necessarily complete) with source M and target N.

Definition 2.6 (Term and substitution development) *We define the relation $\triangleright$ on terms and a corresponding relation $\blacktriangleright$ on substitutions. The relation $\triangleright$ is defined by the following rules:*

$$\frac{}{M \triangleright M}\,\mathsf{DRefl} \qquad \frac{M \triangleright M'}{\lambda p.M \triangleright \lambda p.M'}\,\mathsf{DAbs}$$

$$\frac{M \triangleright M' \quad N \triangleright N'}{MN \triangleright M'N'}\,\mathsf{DApp} \qquad \frac{M \triangleright M' \quad \theta \blacktriangleright \theta' \quad p \ll^{\theta} N}{(\lambda p.M)N \triangleright \theta' M'}\,\mathsf{DBeta}$$

and $\blacktriangleright$ is defined as follows: $\theta \blacktriangleright \theta'$ iff $\mathtt{dom}(\theta) = \mathtt{dom}(\theta')$ and $\forall x \in \mathtt{dom}(\theta) \; . \; \theta x \triangleright \theta' x$

2.1 Head step

The definition of head step will take into account the terms $(\lambda p.M)N$ even if $p \not\ll N$. In such cases, the head redex will be inside N as the patterns in this calculus are always normal forms (this will not be the case for more complex pattern calculi).

The selection of the head redex inside N depends on both N and p. This differs from standard call-by-value λ-calculus, where the selection depends only on N.

We show this phenomenon with a simple example. Let a, b, c be constants and $N = (aR_1)R_2$, where R_1 and R_2 are redexes. The redexes in N needed to achieve a match with a certain pattern p, and thus the selection of the head redex, depend on the pattern p.

Take for example different patterns $p_1 = (ax)(by), p_2 = (abx)y, p_3 = (abx)(cy), p_4 = (ax)y$, and consider the term $Q = (\lambda p.M)N$. If $p = p_1$, then it is not necessary to reduce R_1 (because it already matches

x) but it is necessary to reduce R_2, because no redex can match the pattern by; hence R_2 will be the head redex in this case. Analogously, for p_2 it is necessary to reduce R_1 but not R_2, for p_3 both are needed (in this case we will choose the leftmost one) and p_4 does match N, hence the whole Q is the head redex. This observation motivates the following definition.

Definition 2.7 (Head step) *The relations $\underset{h}{\rightarrow}$ (head step) and $\underset{p}{\rightsquigarrow}$ (preferred needed step to match pattern p) are defined as follows:*

$$\frac{M \underset{h}{\rightarrow} M'}{MN \underset{h}{\rightarrow} M'N}\ \mathsf{HApp1} \qquad \frac{p \ll^{\theta} N}{(\lambda p.M)N \underset{h}{\rightarrow} \theta M}\ \mathsf{HBeta} \qquad \frac{N \underset{p}{\rightsquigarrow} N'}{(\lambda p.M)N \underset{h}{\rightarrow} (\lambda p.M)N'}\ \mathsf{HPat}$$

$$\frac{M \underset{h}{\rightarrow} M'}{M \underset{d}{\rightsquigarrow} M'}\ \mathsf{PatHead} \qquad \frac{D \underset{d}{\rightsquigarrow} D'}{DM \underset{dp}{\rightsquigarrow} D'M}\ \mathsf{Pat1} \qquad \frac{M \underset{p}{\rightsquigarrow} M' \quad d \ll D}{DM \underset{dp}{\rightsquigarrow} DM'}\ \mathsf{Pat2}$$

The rule $\mathsf{PatHead}$ is intended for data patterns only, not being valid for variable patterns; we point this by writing a d (data pattern) instead of a p (any pattern) in the arrow subscript inside the conclusion.

We observe that the rule analogous to HPat in the presentation of standard reduction sequences for call-by-value λ-calculus in both [13] and [2] reads

$$\frac{N \underset{h}{\rightarrow} N'}{(\lambda p.M)N \underset{h}{\rightarrow} (\lambda p.M)N'}$$

reflecting the N-only-dependency feature aforementioned.

We see also that a head step in a term like $(\lambda p.M)N$ determined by rule HPat will lie inside N, but the same step will not necessarily be considered head if we analyse N alone.

It is easy to check that if $M \underset{p}{\rightsquigarrow} M'$ then $p \not\ll M$, avoiding any overlap between HBeta and HPat and also between $\mathsf{Pat1}$ and $\mathsf{Pat2}$. This in turn implies that all terms have at most one head redex. We remark also that the head step depends not only on the pattern structure but also on the match or lack of match between pattern and argument.

Lemma 2.8 (Basic facts about head steps)

a. *(head reduction only if abstraction in head) Let M be a term such that $M \underset{h}{\rightarrow} M'$ for some M'. Then $M = (\lambda p.M_{01})M_1 \ldots M_n$ with $n \geq 1$.*

b. *(head reduction only if no match) Let M be a term such that $M \underset{h}{\rightarrow} M'$ for some M', $d \in$ **DataPatterns**. Then $d \not\ll M$.*

c. *($\underset{p}{\rightsquigarrow}$ only if $\underset{h}{\rightarrow}$ or data term) Let p be a pattern and let M be a term such that $M \underset{p}{\rightsquigarrow} M'$ for some M'. Then either $M \in$ **DataTerms** or $M \underset{h}{\rightarrow} M'$.*

Proof Item (a) is trivial. Item (b) uses Item (a) and L. 2.5:(b). Item (c) is trival by definition of $\underset{p}{\rightsquigarrow}$. □

3 Main concepts and ideas needed for the standardisation proof

In order to build a standardisation proof for constructor based pattern calculi we chose to adapt the one in [14] for the call-by-name λ-calculus, later adapted to call-by-value λ-calculus in [2], over the classical presentation of [13].

The proof method relies on a **h-development** property stating that any development can be split into a leading sequence of head steps followed by a development in which no head steps are performed; this is our Corollary 5.4 which corresponds to the so-called “main lemma” in the presentations by Takahashi and Crary.

Even for a simple form of pattern calculus such as the one presented in this contribution, both the definitions (as we already mentioned when defining head steps) and the proofs are non-trivial extensions of the corresponding ones for standard λ-calculus, even in the framework of call-by-value. As mentioned before, the reason is the need to take into account, for terms involving the application of a function to an argument, the pattern of the function parameter when deciding whether a redex inside the argument should be considered as a head redex.

In order to formalize the notion of “development without occurrences of head steps”, an *internal development* relation will be defined. The dependency on both N and p when analysing the reduction steps from a term like $(\lambda p.M)N$ is shown in the rule IApp2.

Definition 3.1 (Internal development) *The relations* $\overset{int}{\triangleright}$ *(internal development) and* $\overset{int}{\triangleright}_p$ *(internal development with respect to the pattern p) are defined as follows:*

$$\frac{}{M \overset{int}{\triangleright} M}\,\mathsf{IRefl} \qquad \frac{M \triangleright M'}{\lambda p.M \overset{int}{\triangleright} \lambda p.M'}\,\mathsf{IAbs} \qquad \frac{M \neq \lambda p.M_1 \quad M \overset{int}{\triangleright} M' \quad N \triangleright N'}{MN \overset{int}{\triangleright} M'N'}\,\mathsf{IApp1}$$

$$\frac{M \triangleright M' \quad N \overset{int}{\triangleright}_p N'}{(\lambda p.M)N \overset{int}{\triangleright} (\lambda p.M')N'}\,\mathsf{IApp2} \qquad \frac{N \triangleright N' \quad p \ll N}{N \overset{int}{\triangleright}_p N'}\,\mathsf{PMatch} \qquad \frac{N \overset{int}{\triangleright} N'}{N \overset{int}{\triangleright}_c N'}\,\mathsf{PConst}$$

$$\frac{N \notin \textbf{DataTerms} \quad N \overset{int}{\triangleright} N'}{N \overset{int}{\triangleright}_{dp} N'}\,\mathsf{PNoCData} \qquad \frac{D \overset{int}{\triangleright}_d D' \quad M \triangleright M' \quad d \not\ll D}{DM \overset{int}{\triangleright}_{dp} D'M'}\,\mathsf{PCDataNo1}$$

$$\frac{D \triangleright D' \quad M \overset{int}{\triangleright}_p M' \quad d \ll D \quad p \not\ll M}{DM \overset{int}{\triangleright}_{dp} D'M'}\,\mathsf{PCDataNo2}$$

$$\frac{D \triangleright D' \quad M \triangleright M' \quad d \ll D \quad p \ll M \quad dp \not\ll DM}{DM \overset{int}{\triangleright}_{dp} D'M'}\,\mathsf{PCDataNo3}$$

Remark that rule PCDataNo3 is useful to deal with non-linear patterns.

Thus for example, $ab((\lambda y.y)c) \overset{int}{\triangleright}_{axx} abc$ since $ab \triangleright ab$, $(\lambda y.y)c \triangleright c$, $ax \ll ab$, $x \ll (\lambda y.y)c$ but $axx \not\ll ab((\lambda y.y)c)$.

We observe also that if $N \overset{int}{\triangleright} N'$ or $N \overset{int}{\triangleright}_p N'$ then $N \triangleright N'$.

The following lemma analyses data / non-data preservation

Lemma 3.2 (Development and data)

a. *(internal development cannot create data terms) Let $M \notin$ **DataTerms**, N such that $M \overset{int}{\triangleright} N$. Then $N \notin$ **DataTerms***

b. *(development from data produces always data) Let $M \in$ **DataTerms**, N such that $M \triangleright N$. Then $N \in$ **DataTerms***

The formal description of the h-development condition takes a form of an additional binary relation. This relation corresponds to the one called *strong parallel reduction* in [2].

Definition 3.3 (H-development) *We define the relations $\underset{h}{\triangleright}$ and $\underset{h}{\blacktriangleright}$. Let M, N be terms; ν, θ substitutions.*

a. $M \underset{h}{\triangleright} N$ *iff* *(i)* $M \triangleright N$, *(ii)* $\exists Q$ s.t. $M \underset{h}{\rightarrow}^{*} Q \overset{int}{\triangleright} N$, *(iii)* $\forall p \,.\, \exists Q_p$ s.t. $M \underset{p}{\overset{*}{\rightsquigarrow}} Q_p \overset{int}{\triangleright}_p N$.

b. $\nu \underset{h}{\blacktriangleright} \theta$ *iff* *(i)* $Dom(\nu) = Dom(\theta)$, *(ii)* $\forall x \in Dom(\nu) \,.\, \nu x \underset{h}{\triangleright} \theta x$.

The clause *(iii)* in the definition of $\underset{h}{\triangleright}$ shows the dependency on the patterns that was already noted in the definitions of head step and internal development.

This clause is needed when proving that all developments are h-developments; let's grasp the reason through a brief argument. Suppose we want to prove that a development inside N in a term like $(\lambda p.M)N$ is an h-development. The rules to be used in this case are HPat (Def. 2.7) and IApp2 (Def. 3.1). Therefore we need to perform an analysis *relative to the pattern* p; and this is exactly expressed by clause *(iii)*. Consequently the proof of clause *(ii)* for a term needs to consider clause *(iii)* (instantiated to a certain pattern) for a subterm; this is achieved by including clause *(iii)* in the definition and by performing an inductive reasoning on terms.

4 Auxiliary results

We collect in this section some results needed to complete the main proofs in this article.

Lemma 4.1 (pattern-head reduction only if there is no match)
Let M, N be terms, p a pattern, such that $M \underset{p}{\rightsquigarrow} N$. Then $p \not\ll M$.

Proof Using L. 2.8:(b). □

Lemma 4.2 (development cannot lose matches)
Let M, N be terms, p a pattern, such that $M \triangleright N$ and $p \ll^{\nu} M$. Then $p \ll^{\theta} N$ for some θ such that $\nu \blacktriangleright \theta$.

Proof Induction on $p \ll^{\nu} M$. The axioms can be checked trivially. For the rule, let $M = M_1 M_2$, $N = N_1 N_2$, $p = p_1 p_2$ and $\nu = \nu_1 \uplus \nu_2$; p is linear since it matches a term . The only rules applicable for $M \triangleright N$ are DRefl or DApp; DBeta is not applicable because $M_1 \in$ **DataTerms**. If DRefl was used, the lemma holds trivially taking $\theta = \nu$. If DApp was used, we apply the IH on both hypotheses obtaining $p_i \ll^{\theta_i} N_i$ with $\nu_i \blacktriangleright \theta_i$; by L. 2.5:(c) and the linearity of p we know $\theta = \theta_1 \uplus \theta_2$ is well-defined; it is easy to check that θ satisfies the lemma conditions. □

Lemma 4.3 ($\overset{int}{\triangleright}_p$ cannot create match)
Let M, N be terms, p a pattern, such that $M \overset{int}{\triangleright}_p N$. Then $p \not\ll M$ implies $p \not\ll N$.

Proof Induction on $M \overset{int}{\triangleright}_p N$ by rule analysis

PMatch not applicable as $p \not\ll M$.

PConst in this case the condition $p \not\ll M$ implies $p \not\ll N$ equates to $M \neq p$ implies $N \neq p$, as p is a constant.
The rule premise reads $M \overset{int}{\triangleright} N$: if rule IRefl was used then $N \neq p$ by hypothesis, else the $\overset{int}{\triangleright}$ rule conclusions exclude the possibility of N being a constant.

PNoCData $M \notin$ **DataTerms** and $M \overset{int}{\triangleright} N$ by rule hyp., then $N \notin$ **DataTerms** by L. 3.2:(a), finally $p \not\ll N$ by L. 2.5:(b).

PCDataNo1 By the IH, as rule hyp. includes both $D \overset{int}{\triangleright}_d D'$ and $d \not\ll D$ being $M = DT$ and $p = dp'$.

PCDataNo2 Similar to the former considering $p = dp'$ and using $T \overset{int}{\triangleright}_{p'} T'$ and $p' \not\ll T$.

PCDataNo3 In this case $M = DM'$, $p = dp'$, $d \ll^{\theta} D$, $p' \ll^{\theta'} M'$ and $dp' \not\ll DM'$. We necessarily have that $\theta \uplus \theta'$ is not defined hence p is not linear so that $p \not\ll N$ also holds.

□

Lemma 4.4 (left-pattern-head implies whole-pattern-head)
Let p_1, p_2 be patterns and M_1, N_1, M_2 be terms such that $M_1 \underset{p_1}{\rightsquigarrow} N_1$. Then $M_1M_2 \underset{p_1p_2}{\rightsquigarrow} N_1M_2$.

Proof It is clear that $p_1 \notin Var$, because there is no N_1 such that $M_1 \underset{x}{\rightsquigarrow} N_1$ if $x \in Var$.

If PatHead applied in $M_1 \underset{p_1}{\rightsquigarrow} N_1$, then $M_1 \underset{h}{\rightarrow} N_1$, by HApp1 $M_1M_2 \underset{h}{\rightarrow} N_1M_2$, and finally by PatHead $M_1M_2 \underset{p_1p_2}{\rightsquigarrow} N_1M_2$.

If either Pat1 or Pat2 applied in $M_1 \underset{p_1}{\rightsquigarrow} N_1$, then M_1 is clearly a data term, Then $M_1M_2 \underset{p_1p_2}{\rightsquigarrow} N_1M_2$ by Pat1. □

Lemma 4.5 (matching is compatible with substitution)
Let M be a term, p a pattern and θ a substitution such that $p \ll^{\theta} M$. Then for any substitution ν, the following holds: $p \ll^{\gamma} \nu M$ where $\gamma = \nu\theta \mid_{\mathtt{fv}(p)}$.

Proof By induction on the match. The axioms can be checked trivially given L. 2.5:(c).

We analyze the rule applied in this context

$$\frac{d \ll^{\theta_1} M_1 \qquad p' \ll^{\theta_2} M_2}{dp' = p \ll^{\theta = \theta_1 \uplus \theta_2} M = M_1M_2}$$

Applying the IH on both hypotheses and then using the rule gives $dp' \ll^{(\nu\theta_1)|_{\mathtt{fv}(d)} \uplus (\nu\theta_2)|_{\mathtt{fv}(p)}} M_1M_2$; an easy check of $(\nu\theta_1) \mid_{\mathtt{fv}(d)} \uplus (\nu\theta_2) \mid_{\mathtt{fv}(p')} = (\nu(\theta_1 \uplus \theta_2)) \mid_{\mathtt{fv}(dp')}$ concludes the proof. □

Lemma 4.6 (development is compatible with substitution)
Let M, N be terms and ν, θ substitutions, such that $M \triangleright N$ and $\nu \blacktriangleright \theta$. Then $\nu M \triangleright \theta N$

Proof By induction on $M \triangleright N$ by rule analysis.

For DRefl the thesis amounts to $\nu M \triangleright \theta M$, which can be checked by a simple induction on M. DAbs and DApp can be simply verified by the IH.

For DBeta first we mention a technical result which will be used. Let θ, τ be substitutions such that $\mathtt{dom}(\tau) \cap \mathtt{var}(\theta) = \emptyset$, then

$$\big((\theta\tau)\mid_{\mathtt{dom}(\tau)}\big)\,\theta = \theta\tau \tag{1}$$

this can be easily checked comparing the effect of applying both substitutions to an arbitrary variable.

Let's analyze the rule premises and conclusion applied in this context

$$\frac{M_1 \triangleright M_1' \qquad \tau \blacktriangleright \tau' \qquad p \ll^{\tau} M_2}{M = (\lambda p.M_1)M_2 \triangleright \tau' M_1' = N}$$

As we can freely choose the variables appearing in p, we assume $\mathtt{fv}(p) \cap (\mathtt{var}(\nu) \cup \mathtt{var}(\theta)) = \emptyset$. By L. 2.5:(c) we know $\mathtt{dom}(\tau) = \mathtt{dom}(\tau') = \mathtt{fv}(p)$.

We apply the IH on $M_1 \triangleright M_1'$ and also on $\tau x \triangleright \tau' x$ for each $x \in \mathtt{dom}(\tau)$ to conclude $\nu M_1 \triangleright \theta M_1'$ and $(\nu\tau)\mid_{\mathtt{dom}(\tau)} \blacktriangleright (\theta\tau')\mid_{\mathtt{dom}(\tau)}$ respectively. Furthermore, from $p \ll^{\tau} M_2$ and L. 4.5 we conclude $p \ll^{(\nu\tau\mid_{\mathtt{dom}(\tau)})} \nu M_2$.

We use DBeta from the three conclusions above to obtain

$$\nu M = (\lambda p.\nu M_1)(\nu M_2) \triangleright \big((\theta\tau')\mid_{\mathtt{dom}(\tau)}\big)(\theta M_1')$$

To check $\theta N = \theta(\tau' M_1') = \big((\theta\tau')\mid_{\mathtt{dom}(\tau)}\big)(\theta M_1')$ it is enough to verify $\theta\tau' = \big((\theta\tau')\mid_{\mathtt{dom}(\tau)}\big)\theta$, the latter can be easily checked by (1).

□

Lemma 4.7 (head reduction is compatible with substitution)

(i) Let M, N be terms and ν a substitution such that $M \underset{h}{\rightarrow} N$. Then $\nu M \underset{h}{\rightarrow} \nu N$.

(ii) Let M, N be terms, p a pattern and ν a substitution such that $M \underset{p}{\rightsquigarrow} N$. Then $\nu M \underset{p}{\rightsquigarrow} \nu N$.

Proof (sketch)
Both items are proved by simultaneous induction on $M \underset{h}{\rightarrow} N$ and $M \underset{p}{\rightsquigarrow} N$.

We use L. 4.5 for case HBeta, the IH and L. 4.5 for case Pat2, and just the IH for the remaining cases. □

5 H-developments

The aim of this section is to prove that all developments are h-developments.

We found easier to prove separately that the h-development condition is compatible with the language constructs, diverging from the structure of the proofs in [2].

Lemma 5.1 ($\underset{h}{\triangleright}$ is compatible with abstraction)
Let M, N be terms such that $M \underset{h}{\triangleright} N$. Then $\lambda q.M \underset{h}{\triangleright} \lambda q.N$ for any pattern q.

Proof Part *(i)* trivially holds by hyp. *(i)* and DAbs.

Part *(ii)*: by hyp. *(i)* and IAbs we get $\lambda q.M \overset{int}{\triangleright} \lambda q.N$. Then $Q = \lambda q.M$.

Part *(iii)*: if $p \in Var$ then PMatch applies, if p is a constant or a compound data pattern then PConst or PNoCData apply respectively as $(\lambda q.M) \overset{int}{\triangleright} (\lambda q.N)$. In all cases we obtain $(\lambda q.M) \overset{int}{\triangleright}_p (\lambda q.N)$. Then $Q = \lambda q.M$. □

Lemma 5.2 ($\underset{h}{\triangleright}$ is compatible with application)
Let M_1, M_2, N_1, N_2 be terms such that $M_1 \underset{h}{\triangleright} N_1$ and $M_2 \underset{h}{\triangleright} N_2$. Then $M_1M_2 \underset{h}{\triangleright} N_1N_2$.

Proof Part *(i)* is immediate by the hypotheses *(i)* and DApp.

Let's prove part *(ii)*.

We first use hypothesis *(ii)* on $M_1 \underset{h}{\triangleright} N_1$ to obtain $M_1 \underset{h}{\rightarrow}^* Q_1 \overset{int}{\triangleright} N_1$ and subsequently apply HApp1 to $M_1 \underset{h}{\rightarrow}^* Q_1$ to get

$$M_1M_2 \quad \underset{h}{\rightarrow}^* \quad Q_1M_2 \tag{2}$$

Either Q_1 is an abstraction or not.

Assume Q_1 is not an abstraction. Since $Q_1 \overset{int}{\triangleright} N_1$ and $M_2 \triangleright N_2$, we apply IApp1 so that $Q_1M_2 \overset{int}{\triangleright} N_1N_2$; this together with (2) gives the desired result.

Now assume $Q_1 = \lambda p.Q_{12}$. We use the hyp. *(iii)* on $M_2 \underset{h}{\triangleright} N_2$, obtaining $M_2 \underset{p}{\rightsquigarrow}^* Q_2 \overset{int}{\triangleright}_p N_2$ and then we apply HPat to get

$$Q_1M_2 \quad \underset{h}{\rightarrow}^* \quad Q_1Q_2 \tag{3}$$

Moreover, as $Q_1 = \lambda p.Q_{12} \overset{int}{\triangleright} N_1$, the only applicable rules are IRefl or IAbs, and in both cases $N_1 = \lambda p.N_{12}$ and $Q_{12} \triangleright N_{12}$.
We now use IApp2 with premises $Q_{12} \triangleright N_{12}$ and $Q_2 \overset{int}{\triangleright}_p N_2$ to get

$$Q_1Q_2 = (\lambda p.Q_{12})Q_2 \quad \overset{int}{\triangleright} \quad (\lambda p.N_{12})N_2 = N_1N_2 \tag{4}$$

The desired result is obtained by (2), (3) and (4).

Let's prove part *(iii)*.

If $p \in Var$ we are done by *(i)* and PMatch; we thus get $M_1M_2 \overset{int}{\triangleright}_p N_1N_2$ so that $Q = M_1M_2$.

If $p = c$ then using *(ii)* we obtain $M_1M_2 \underset{h}{\rightarrow}^* Q \overset{int}{\triangleright} N_1N_2$ for some Q ; we apply PatHead and PConst to get $M_1M_2 \overset{*}{\underset{c}{\rightsquigarrow}} Q$ and $Q \overset{int}{\triangleright}_c N_1N_2$ respectively, concluding the proof for this case.

Consider $p = p_1p_2$ with p_1 a data pattern and p_2 a pattern.

We use the hyp. *(iii)* on $M_1 \underset{h}{\triangleright} N_1$, getting $M_1 \overset{*}{\underset{p_1}{\rightsquigarrow}} Q_1 \overset{int}{\triangleright}_{p_1} N_1$. Let us define R_1 as follows: if there is a data term in the sequence $M_1 \overset{*}{\underset{p_1}{\rightsquigarrow}} Q_1$ then R_1 is the first of such terms; otherwise R_1 is Q_1. In both cases $M_1 \overset{*}{\underset{p_1}{\rightsquigarrow}} R_1 \overset{*}{\underset{p_1}{\rightsquigarrow}} Q_1$. We necessarily have $M_1 \underset{h}{\rightarrow}^* R_1$ by PatHead, then $M_1M_2 \underset{h}{\rightarrow}^* R_1M_2$ by HApp1 and subsequently $M_1M_2 \underset{p}{\rightsquigarrow} R_1M_2$ by PatHead.

We conclude $M_1M_2 \overset{*}{\underset{p}{\rightsquigarrow}} Q_1M_2$, trivially if $Q_1 = R_1$, and applying Pat1 to $R_1 \overset{*}{\underset{p_1}{\rightsquigarrow}} Q_1$ to obtain $R_1M_2 \overset{*}{\underset{p}{\rightsquigarrow}} Q_1M_2$ otherwise.

If $Q_1 = (\lambda q.Q_1')$ then we use the hyp. *(iii)* on $M_2 \underset{h}{\triangleright} N_2$ getting $M_2 \overset{*}{\underset{q}{\rightsquigarrow}} Q_2 \overset{int}{\triangleright}_q N_2$.

We apply HPat to $M_2 \underset{q}{\rightsquigarrow}^* Q_2$ getting $Q_1M_2 \underset{h}{\rightarrow}^* Q_1Q_2$; therefore we obtain $Q_1M_2 \overset{*}{\underset{p}{\rightsquigarrow}} Q_1Q_2$ by $\mathsf{PatHead}$.

In the other side $Q_1 = (\lambda q.Q_1') \triangleright N_1$, therefore $N_1 = (\lambda q.N_1')$ and $Q_1' \triangleright N_1'$.

We apply $\mathsf{IApp2}$ to $Q_1' \triangleright N_1'$ and $Q_2 \overset{int}{\triangleright}_q N_2$ to obtain $Q_1Q_2 \overset{int}{\triangleright} N_1N_2$, therefore $Q_1Q_2 \overset{int}{\triangleright}_p N_1N_2$ by $\mathsf{PNoCData}$. We thus get the desired result taking $Q_p = Q_1Q_2$.

If Q_1 is not an abstraction and $Q_1 \notin$ **DataTerms**, then only PConst or $\mathsf{PNoCData}$ can justify $Q_1 \overset{int}{\triangleright}_{p_1} N_1$, thus implying $Q_1 \overset{int}{\triangleright} N_1$; this together with the hypothesis *(i)* $M_2 \triangleright N_2$ gives $Q_1M_2 \overset{int}{\triangleright} N_1N_2$ by $\mathsf{IApp1}$, hence $Q_1M_2 \overset{int}{\triangleright}_p N_1N_2$ by $\mathsf{PNoCData}$. We get the desired result by taking $Q_p = Q_1M_2$.

If $Q_1 \in$ **DataTerms** we anaylise the different alternatives for the matching between p_1p_2 and Q_1M_2.

Assume $p_1 \not\ll Q_1$. In this case we apply $\mathsf{PCDataNo1}$ to $Q_1 \overset{int}{\triangleright}_{p_1} N_1$ and $M_2 \triangleright N_2$ to obtain $Q_1M_2 \overset{int}{\triangleright}_p N_1N_2$ and thus the desired result holds by taking $Q_p = Q_1M_2$.

Assume $p_1 \ll Q_1$ and $p_2 \not\ll M_2$. In this case we use the hyp. *(iii)* on $M_2 \underset{h}{\triangleright} N_2$ to get $M_2 \overset{*}{\underset{p_2}{\rightsquigarrow}} Q_2 \overset{int}{\triangleright}_{p_2} N_2$, then apply $\mathsf{Pat2}$ to get $Q_1M_2 \overset{*}{\underset{p}{\rightsquigarrow}} Q_1Q_2$. Finally from $Q_1 \overset{int}{\triangleright}_{p_1} N_1$ and $Q_2 \overset{int}{\triangleright}_{p_2} N_2$ we obtain $Q_1Q_2 \overset{int}{\triangleright}_p N_1N_2$ by either $\mathsf{PCDataNo2}$, $\mathsf{PCDataNo3}$ or PMatch. We get the desired result by taking $Q_p = Q_1Q_2$.

Finally assume $p_1 \ll Q_1$ and $p_2 \ll Q_2$. In this case the hypotheses imply in particular $Q_1 \triangleright N_1$ and $M_2 \triangleright N_2$. We thus conclude $Q_1M_2 \overset{int}{\triangleright}_p N_1N_2$ using either PMatch or $\mathsf{PCDataNo3}$ (depending on whether $p \ll Q_1M_2$ or not), getting the desired result by taking $Q_p = Q_1M_2$.

□

Now we proceed with the proof of the h-development property. The generalization of the statement involving $\underset{h}{\blacktriangleright}$ is needed to conclude the proof[1], as can be seen in the DBeta case below.

Lemma 5.3 (Generalized h-developments property)
Let M, N be terms and ν, θ substitutions, such that $M \triangleright N$ and $\nu \underset{h}{\blacktriangleright} \theta$.
Then $\nu M \underset{h}{\triangleright} \theta N$

Proof By induction on $M \triangleright N$ analyzing the rule used in the last step of the derivation.

DRefl in this case $N = M$, we proceed by induction on M

- $M = x \in Dom(\nu)$, in this case $\nu M = \nu x \underset{h}{\triangleright} \theta x = \theta N$ by hypothesis.
- $M = x \notin Dom(\nu)$, in this case $\nu M = x \underset{h}{\triangleright} x = \theta N$.
- $M = M_1M_2$, in this case$\nu M_1 \underset{h}{\triangleright} \theta M_1$ and $\nu M_2 \underset{h}{\triangleright} \theta M_2$ hold by the IH. The desired result is obtained by L. 5.2.
- $M = \lambda p.M_1$, in this case $\nu M_1 \underset{h}{\triangleright} \theta M_1$ holds by the IH. The desired result is obtained by L. 5.1.

[1]In [2] the compatibility of h-development with substitutions is stated as a separate lemma; for pattern calculi we could not find a proof of compatibility with substitution independent of the main h-development result.

DAbs in this case $M = \lambda p.M_1, N = \lambda p.N_1, M_1 \triangleright N_1$.

Using the IH on $M_1 \triangleright N_1$ we obtain $\nu M_1 \underset{h}{\triangleright} \theta N_1$, the desired result is obtained by L. 5.1.

DApp in this case $M = M_1 M_2, N = N_1 N_2, M_i \triangleright N_i$.

Using the IH on both rule premises we obtain $\nu M_i \underset{h}{\triangleright} \theta N_i$, the desired result is obtained by L. 5.2.

DBeta Let's write down the rule instantiation

$$\frac{M_{12} \triangleright N_{12} \quad \tau \blacktriangleright \tau' \quad q \ll^{\tau} M_2}{M = (\lambda q.M_{12})M_2 \triangleright \tau' N_{12} = N}$$

(i) can be obtained by hypotheses $M \triangleright N$ and $\nu \underset{h}{\blacktriangleright} \theta$, and then L. 4.6.

For [*(iii)* if $p \in Var$] we are done by *(i)* and PMatch.

For [*(iii)* if $p = d$] and also for *(ii)* : we know both $M \underset{h}{\to} \tau M_{12}$ and $M \underset{p}{\rightsquigarrow} \tau M_{12}$, then by L. 4.7

$$\nu M \underset{h}{\to} \nu(\tau M_{12}) \quad \text{and} \quad \nu M \underset{p}{\rightsquigarrow} \nu(\tau M_{12}) \tag{5}$$

We apply the IH on each $\tau x \triangleright \tau' x$, obtaining $(\nu\tau)x = \nu(\tau x) \underset{h}{\triangleright} \theta(\tau' x) = (\theta\tau')x$ for all $x \in Dom(\tau)$. Moreover, if $x \in Dom(\nu) - Dom(\tau)$ then $(\nu\tau)x = \nu x \underset{h}{\triangleright} \theta x = (\theta\tau')x$ by hypothesis.

Consequently, $\nu\tau \underset{h}{\blacktriangleright} \theta\tau'$. Now we use the IH on $M_{12} \triangleright N_{12}$ taking $\nu\tau \underset{h}{\blacktriangleright} \theta\tau'$ as second hypothesis to obtain

$$\nu(\tau M_{12}) = (\nu\tau)M_{12} \underset{h}{\triangleright} (\theta\tau')N_{12} = \theta(\tau' N_{12}) = \theta N$$

This result along with (5) concludes the proof for both parts.

□

Corollary 5.4 (H-development property)
Let M, N be terms such that $M \triangleright N$. Then $M \underset{h}{\triangleright} N$.

6 Standardisation

The part of the standardisation proof following the proof of the h-development property coincides in structure with the proof given in [2].

First we will prove that we can get, for any reduction involving head steps that follows an internal development, another reduction in which the head steps are at the beginning. The name given to the Lemma 6.1 was taken from [2].

This proof needs again to consider explicitly the relations relative to patterns, for similar reasons to those described when introducing h-development in section 3.

Lemma 6.1 (Postponement)

(i) *if $M \overset{int}{\triangleright} N \underset{h}{\to} R$ then there exists some term N' such that $M \underset{h}{\to} N' \triangleright R$*

(ii) *for any pattern p, if $M \overset{int}{\triangleright}_p N \underset{p}{\rightsquigarrow} R$ then there exists some term N'_p such that $M \underset{p}{\rightsquigarrow} N'_p \triangleright R$*

Proof For *(i)*, if the rule used in $M \overset{int}{\rhd} N$ is IRefl, then the result is immediate taking $N' = R$. Therefore, in the following we will ignore this case.

We prove *(i)* and *(ii)* by simultaneous induction on M taking into account the previous observation.

variable in this case it must be $N = M$ for both *(i)* and *(ii)* and neither $M \underset{h}{\to} R$ nor $M \underset{p}{\leadsto} R$ for any p, R.

abstraction in this case N must also be an abstraction for both *(i)* and *(ii)* and neither $N \underset{h}{\to} R$ nor $N \underset{p}{\leadsto} R$ for any p, R.

application in this case $M = M_1 M_2$

We prove *(i)* first, analysing the possible forms of M_1

- Assume M_1 is not an abstraction

  In this case $\mathsf{IApp1}$ applies, so we know $N = N_1 N_2$, $M_1 \overset{int}{\rhd} N_1$, and $M_2 \rhd N_2$.

  Since $M_1 \overset{int}{\rhd} N_1$, N_1 is not an abstraction, then the only applicable rule for $N \underset{h}{\to} R$ is $\mathsf{HApp1}$, hence $R = R_1 N_2$ and $N_1 \underset{h}{\to} R_1$.

  Now we use the IH on $M_1 \overset{int}{\rhd} N_1 \underset{h}{\to} R_1$ to get $M_1 \underset{h}{\to} N_1' \rhd R_1$, then we obtain $M = M_1 M_2 \underset{h}{\to} N_1' M_2$ by $\mathsf{HApp1}$.

  Finally we apply DApp to $N_1' \rhd R_1$ and $M_2 \rhd N_2$ to get $N_1' M_2 \rhd R_1 N_2 = R$, which concludes the proof for this case.
- Now assume $M_1 = \lambda p.M_{12}$ and $p \not\ll M_2$

  Since $M = (\lambda p.M_{12})M_2 \overset{int}{\rhd} N$, the only rule that applies is $\mathsf{IApp2}$, then $N = (\lambda p.N_{12})N_2$, $M_{12} \rhd N_{12}$, and $M_2 \overset{int}{\rhd}_p N_2$. By L. 4.3 we obtain $p \not\ll N_2$, so the only applicable rule in $N = (\lambda p.N_{12})N_2 \underset{h}{\to} R$ is HPat, then $R = (\lambda p.N_{12})R_2$ and $N_2 \underset{p}{\leadsto} R_2$.

  Now we use the IH *(ii)* on $M_2 \overset{int}{\rhd}_p N_2 \underset{p}{\leadsto} R_2$, to get $M_2 \underset{p}{\leadsto} N_2' \rhd R_2$.

  We obtain $M = (\lambda p.M_{12})M_2 \underset{h}{\to} (\lambda p.M_{12})N_2'$ by HPat, then we get $(\lambda p.M_{12}) \rhd (\lambda p.N_{12})$ by DAbs on $M_{12} \rhd N_{12}$, finally we apply DApp to the previous result and $N_2' \rhd R_2$ to obtain $(\lambda p.M_{12})N_2' \rhd (\lambda p.N_{12})R_2 = R$ which concludes the proof for this case.
- Finally, assume $M_1 = \lambda p.M_{12}$ and $p \ll^{\nu} M_2$

  Again, the only rule that applies in $M = (\lambda p.M_{12})M_2 \overset{int}{\rhd} N$ is $\mathsf{IApp2}$, then $N = (\lambda p.N_{12})N_2$, $M_{12} \rhd N_{12}$, and $M_2 \overset{int}{\rhd}_p N_2$. Now, by L. 4.2 we obtain $p \ll^{\theta} N_2$ for some substitution θ such that $\nu \blacktriangleright \theta$, then the applied rule in $N \underset{h}{\to} R$ is HBeta (the case HPat being excluded by L. 4.1), hence $R = \theta N_{12}$

  It is clear that $M \underset{h}{\to} \nu M_{12}$. By L. 4.6 we obtain $\nu M_{12} \rhd \theta N_{12} = R$, which concludes the proof for this case.

For *(ii)* we proceed by a case analysis of p

If $p \in Var$ then there is no R such that $N \underset{p}{\leadsto} R$ for any term N.

If $p \ll M$ then by L. 4.2 $p \ll N$, and therefore by L. 4.1 there can be no R such that $N \underset{p}{\leadsto} R$.

If $p = c$ then $p \not\ll M$, hence $M \overset{int}{\triangleright}_p N \underset{p}{\rightsquigarrow} R$ implies $M \overset{int}{\triangleright} N \underset{h}{\to} R$ as PConst and PatHead are the only possibilities for this case respectively. We use part *(i)* to obtain $M \underset{h}{\to} N' \triangleright R$, and $M \underset{p}{\rightsquigarrow} N'$ by PatHead which concludes the proof for this case.

If $p = d\, p_2$ and $M \notin$ **DataTerms**, then the only possibilities for $M \overset{int}{\triangleright}_p N \underset{p}{\rightsquigarrow} R$ are PNoCData and PatHead respectively, then $M \overset{int}{\triangleright} N \underset{h}{\to} R$. We use part *(i)* to obtain $M \underset{h}{\to} N' \triangleright R$, and $M \underset{p}{\rightsquigarrow} N'$ by PatHead which concludes the proof for this case.

Now assume $p = d\, p_2$, $M \in$ **DataTerms**, and $p \not\ll M$. We must analyse three possibilities

- $d \not\ll M_1$.

  In this case only PCDataNo1 applies for $M \overset{int}{\triangleright}_p N$, therefore $N = N_1 N_2$ with $M_1 \overset{int}{\triangleright}_d N_1$ and $M_2 \triangleright N_2$. By L. 4.3 we know $d \not\ll N_1$ and moreover N_1 is a data term (as can be seen by L. 3.2) thus not having head redexes, so the only possible rule for $N \underset{p}{\rightsquigarrow} R$ is Pat1, then $R = R_1 N_2$ with $N_1 \underset{d}{\rightsquigarrow} R_1$.

  Now we use the IH on the derivation $M_1 \overset{int}{\triangleright}_d N_1 \underset{d}{\rightsquigarrow} R_1$ to get $M_1 \underset{d}{\rightsquigarrow} N_1' \triangleright R_1$, therefore $M = M_1 M_2 \underset{p}{\rightsquigarrow} N_1' M_2$ by Pat1.

  Moreover as $N_1' \triangleright R_1$ and $M_2 \triangleright N_2$ hence $N_1' M_2 \triangleright R_1 N_2 = R$, which concludes the proof for this case.
- $d \ll M_1$ and $p_2 \not\ll M_2$.

  In this case only PCDataNo2 applies for $M \overset{int}{\triangleright}_p N$, therefore $N = N_1 N_2$ with $M_1 \triangleright N_1$ and $M_2 \overset{int}{\triangleright}_{p_2} N_2$. By L. 4.2 and L. 4.3 respectively, we obtain both $d \ll N_1$ and $p_2 \not\ll N_2$. Moreover N is a data term (as can be seen by L. 3.2) thus not having head redexes. Hence the only possibility for $N \underset{p}{\rightsquigarrow} R$ is Pat2, then $R = N_1 R_2$ with $N_2 \underset{p_2}{\rightsquigarrow} R_2$

  We now use the IH on $M_2 \overset{int}{\triangleright}_{p_2} N_2 \underset{p_2}{\rightsquigarrow} R_2$ to get $M_2 \underset{p_2}{\rightsquigarrow} N_2' \triangleright R_2$, and by Pat2 $M = M_1 M_2 \underset{p}{\rightsquigarrow} M_1 N_2'$

  We also use DApp on $M_1 \triangleright N_1$ and $N_2' \triangleright R_2$ to get $M_1 N_2' \triangleright N_1 R_2 = R$, which concludes the proof for this case.
- $d \ll M_1$, $p_2 \ll M_2$ and $d\, p_2 \not\ll M_1 M_2$.

  $d \ll M_1$ implies (L 2.5:(b)) $M_1 \in$ **DataTerms** so that from $M = M_1 M_2 \overset{int}{\triangleright}_p N$ we can only have $N = N_1 N_2$ with $M_1 \triangleright N_1$ and $M_2 \triangleright N_2$. L. 4.2 gives $d \ll N_1$ and $p_2 \ll N_2$. L. 3.2:(b) gives $N \in$ **DataTerms**. To show $N \underset{p}{\rightsquigarrow} R$ we have three possibilities: PatHead is not possible since $N \in$ **DataTerms** (c.f. L 2.8:(a)), Pat1 is not possible since $d \ll M_1$ (c.f. L 4.1), Pat2 is not possible since $p_2 \ll N_2$ (c.f. L 4.1).

□

Corollary 6.2
Let M, N, R be terms such that $M \overset{int}{\triangleright} N \underset{h}{\to} R$. Then $\exists N'$ s.t. $M \underset{h}{\to}^* N' \overset{int}{\triangleright} R$.

Proof Immediate by L. 6.1 and Corollary 5.4. □

Now we generalize the h-development concept to a sequence of developments. The name given to Lemma 6.3 was taken from [2].

Lemma 6.3 (Bifurcation)
Let M,N be terms such that $M \triangleright^* N$. Then $M \underset{h}{\to}^* R \overset{int}{\triangleright^*} N$ for some term R.

Proof Induction on the length of $M \triangleright^* N$. If $M = N$ the result holds trivially.

Assume $M \triangleright Q \triangleright^* N$. By C. 5.4 and IH respectively, we obtain $M \underset{h}{\to}^* S \overset{int}{\triangleright} Q$ and $Q \underset{h}{\to}^* T \overset{int}{\triangleright^*} N$ for some terms S and T. Now we use Corollary 6.2 (many times) on $S \overset{int}{\triangleright} Q \underset{h}{\to}^* T$ to get $S \underset{h}{\to}^* R \overset{int}{\triangleright} T$.

Therefore $M \underset{h}{\to}^* S \underset{h}{\to}^* R \overset{int}{\triangleright} T \overset{int}{\triangleright^*} N$ as we desired. □

Using the previous results, the standardisation theorem admits a very simple proof.

Definition 6.4 (Standard reduction sequence) *The standard reduction sequences are the sequences of terms $M_1;\dots;M_n$ which can be generated using the following rules.*

$$\frac{M_2;\dots;M_k \quad M_1 \underset{h}{\to} M_2}{M_1;\dots;M_k}\ \mathsf{StdHead} \qquad \frac{M_1;\dots;M_k}{(\lambda p.M_1);\dots;(\lambda p.M_k)}\ \mathsf{StdAbs}$$

$$\frac{M_1;\dots;M_j \quad N_1;\dots;N_k}{(M_1 N_1);\dots(M_j N_1);(M_j N_2);\dots;(M_j N_k)}\ \mathsf{StdApp} \qquad \frac{}{x}\ \mathsf{StdVar}$$

Theorem 6.5 (Standardisation)
Let M,N be terms such that $M \triangleright^* N$. Then there exists a standard reduction sequence $M;\dots;N$.

Proof By L. 6.3 we have $M \underset{h}{\to}^* R \overset{int}{\triangleright^*} N$; we observe that it is enough to obtain a standard reduction sequence $R;\dots;N$, because we subsequently apply StdHead many times.

Now we proceed by induction on N

- $N \in Var$; in this case $R = N$ and we are done.
- $N = \lambda p.N_1$; in this case $R = \lambda p.R_1$ and $R_1 \triangleright^* N_1$. By IH we obtain a standard reduction sequence $R_1;\dots;N_1$, then by StdAbs so is $R = \lambda p.R_1;\dots;\lambda p.N_1 = N$.
- $N = N_1 N_2$, so $R = R_1 R_2$ and $N_i \triangleright^* R_i$. We use the IH on both reductions to get two standard reduction sequences $N_i;\dots;R_i$, then we join them using StdApp.

□

7 Conclusion and further work

We have presented an elegant proof of the Standardisation Theorem for constructor-based pattern calculi.

We aim to generalize both the concept of standard reduction and the structure of the Standardisation Theorem proof presented here to a large class of pattern calculi, including both open and closed variants as the Pure Pattern Calculus [7]. It would be interesting to have sufficient conditions for a pattern calculus to enjoy the standardisation property. This will be close in spirit with [8] where an abstract confluence proof for pattern calculi is developed.

The kind of calculi we want to deal with imposes challenges that are currently not handled in the present contribution, such as open patterns, reducible (dynamic) patterns, and the possibility of having `fail` as a decided result of matching. Furthermore, the possibility of decided `fail` combined with compound patterns leads to the convenience of studying forms of *inherently parallel* standard reduction strategies.

The abstract axiomatic Standardisation Theorem developed in [5] could be useful for our purpose. However, while the axioms of the abstract formulation of standardisation are assumed to hold in the proof of the standardisation result, they need to be defined and verified for each language to be standardised. This could be nontrivial, as in the case of TRS [6, 15], where a meta-level matching operation is involved in the definition of the rewriting framework. We leave this topic as further work.

References

[1] H.P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, Amsterdam.

[2] K. Crary (2009): *A Simple Proof of Call-by-Value Standardization*. Technical Report CMU-CS-09-137, Carnegie-Mellon University.

[3] H.B. Curry & R. Feys (1958): *Combinatory Logic*. North-Holland Publishing Company, Amsterdam.

[4] J.-Y. Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–101.

[5] G. Gonthier, J.-J. Lévy & P.-A. Melliès (1992): *An abstract standardisation theorem*. In: *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science, 22-25 June 1992, Santa Cruz, California, USA*, IEEE Computer Society, pp. 72–81.

[6] G. Huet & J.-J. Lévy (1991): *Computations in orthogonal rewriting systems*. In: Jean-Louis Lassez & Gordon Plotkin, editors: *Computational Logic, Essays in Honor of Alan Robinson*, MIT Press, pp. 394–443.

[7] C.B. Jay & D. Kesner (2006): *Pure Pattern Calculus*. In: Peter Sestoft, editor: *European Symposium on Programming*, number 3924 in LNCS, Springer-Verlag, pp. 100–114.

[8] C.B. Jay & D. Kesner (2009): *First-class patterns*. *Journal of Functional Programming* 19(2), pp. 191–225.

[9] Ryo Kashima (2000): *A Proof of the Standardization Theorem in λ-Calculus*. Research Reports on Mathematical and Computing Sciences C-145, Tokyo Institute of Technology.

[10] J.W. Klop, V. van Oostrom & R.C. de Vrijer (2008): *Lambda calculus with patterns*. *Theoretical Computer Science* 398(1-3), pp. 16–31.

[11] Paul-André Melliès (1996): *Description Abstraite des Systèmes de Réécriture*. Ph.D. thesis, Université Paris VII.

[12] Luca Paolini & Simona Ronchi Della Rocca (2004): *Parametric parameter passing Lambda-calculus*. *Information and Computation* 189(1), pp. 87–106.

[13] G. Plotkin (1975): *Call-by-name, call-by-value and the Lambda-calculus*. *Theoretical Computer Science* 1(2), pp. 125–159.

[14] M. Takahashi (1995): *Parallel reductions in lambda-calculus*. *Information and Computation* 118(1), pp. 120–127.

[15] Terese (2003): *Term Rewriting Systems*, *Cambridge Tracts in Theoretical Computer Science* 55. Cambridge University Press.