Preface

In various settings higher-order rewriting systems, that is, systems with bound variables, arise. The aim of this first workshop on higher-order rewriting (HOR) is to provide an informal setting to discuss recent work and work in progress concerning higher-order rewriting.

The topics of the workshop include, but are not limited to, applications, foundations, frameworks, implementation and semantics.

We are grateful to Vincent van Oostrom (Utrecht University, The Netherlands) and Joe Wells (Heriot-Watt University, Edinburgh, Scotland) for kindly accepting to give an invited talk at HOR'02. The first invited talk is shared with the Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS'02).

Further we would like to thank the organizers of FLoC'02, and in particular everyone who has been involved in the preparation of the affiliated workshops.

Delia Kesner (Université Paris-Sud, France)
Tobias Nipkow (Technische Universität München, Germany)
Femke van Raamsdonk (Vrije Universiteit, The Netherlands)

Contents

The Rho Cube : some results, some problems

Horatiu Cirstea, Claude Kirchner, Luigi Liquori, Benjamin Wack

The rewriting calculus [1], or Rho Calculus, integrates in a uniform way matching, rewriting and non determinism. Its abstraction mechanism is based on the rewrite rule formation: in a ρ-term of the form $l \to r$, we abstract on the ρ-term l, and it is worth noticing that when l is a variable x this corresponds exactly to the λ-term $\lambda x.r$. When an abstraction $l \to r$ is applied to the ρ-term u, which is denoted by $(l \to r)\bullet u$, the evaluation mechanism is based on the binding of the free variables present in l to the appropriate subterms of u. Indeed this binding is realized by matching l against u, and one of the characteristics of the calculus is to possibly use information in the matching process such as algebraic axioms like associativity or commutativity.

At that stage, non-determinism may come into play since the matching process can return zero, one, or several possibilities. For each of these variable bindings, the value of the variables are propagated in the term r yielding zero, one or several (finite or not) results. In a restricted way this is exactly what happen in the β-redex $(\lambda x.r)\, u$, which is simply denoted in the syntax of the Rho Calculus $(x \to r)\bullet u$, and where the match is trivially the substitution $\{x/u\}$. Therefore, the Rho Calculus strictly contains the λ-calculus and the possibility to express failure of evaluation or multiplicity of results is directly supported. The rewriting calculus is thus a very general and powerful formalism since it allows one to simply represent not only lambda-calculus and rewriting but also object calculi [2].

The static and dynamic semantics of the rewriting calculus have been extensively studied and more recently, the properties of the calculus in a typed context have been investigated. In particular, a new presentation *à la Church*, together with nine (8+1) type systems which can be placed in a *ρ-cube* that extends the λ-cube of Barendregt, has been proposed [3]. We study the properties of the different typed calculi and the relationship with corresponding logics.

We have proposed an original solution to the different problems related to the identification of the standard "λ" and "Π" abstractors [4, 5]; our approach is essentially based on the *complete* unification of the two operators into the only abstraction symbol present in the Rho Calculus, *i. e.* the "$\to$" operator. This unification is, so to speak, *built-in* in the definition of the Rho Calculus itself. The most powerful type system in our cube (namely the ninth one) is a variant of the plain Calculus of Constructions and it is essentially inspired from the *Extended Calculus of Constructions* (ECC) of Z. Luo [6]. In ECC, indeed, we have an infinite set of sorts, *i. e.* $s \in \{*, \Box_i\}$, with $i \in \mathbb{N}$, and the extra axiom $\vdash \Box_i : \Box_{i+1}$ with $i \in \mathbb{N}$.

We have shown some classical properties for different typed calculi: correctness, subject reduction, consistency. We should notice that, in our case, the additional type system ECC is essential since an infinite hierarchy of sorts is needed. Since the plain ρ-calculus is not confluent, we have to deal with confluent strategies. Therefore, the typing rules have to be slightly modified in order to avoid possible clashes with the conditions imposed to recover confluence. As already mentioned in previous papers, having only one abstractor gives the possibility to introduce smart typing rules, but this leads to the failure of the uniqueness of typing. Still, we have an insight at how many distinct types a term can have, and we prove the uniqueness of typing for $\rho 2$, the polymorphic ρ-calculus.

References

[1] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[2] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[3] Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 166–180, Genova, Italy, April 2001.

[4] F. Kamareddine, R. Bloo, and R. Nederpelt. On π-conversion on the λ-cube and the Combination with Abbreviations. *Annals of Pure and Applied Logics*, 97(1-3):27–45, 1999.

[5] F. Kamareddine and R. Nederpelt. Canonical Typing and π-conversion in the λ-cube. *Journal of Functional Programming*, 6(2):85–109, 1996.

[6] Z. Luo. ECC: An Extended Calculus of Constructions. In *Proceedings of LICS*, pages 385–395, 1990.

Evaluation Strategies for Calculi with Explicit Pattern Matching and Substitutions

Julien FOREST
Laboratoire de Recherche en Informatique (CNRS UMR 8623)
Bât 490, Université Paris-Sud
91405 Orsay CEDEX, France.

Higher and First-Order Pattern calculi such as for example [BTKP93, CK99] were proposed as a theoretical model for programming languages with function definitions by cases (CAML [Obj], HASKELL [HPJe92]).

An *evaluation strategy* gives a deterministic way to proceed with evaluation of terms of a given calculus. Thus for example, the so called *lazy* and *strict* evaluation strategies are the most used in implementation of functional programming languages. Intuitively, a lazy strategy evaluates a subterm of a given term *if and only if* this evaluation is necessary to continue with the evaluation of the whole term. A strict strategy first evaluates the sub-terms of a term, even if these evaluation are not necessary, and only then evaluates the term. Functional languages such as CAML use strict evaluation strategies whether HASKELL implements a lazy strategy.

In this talk we are interested in the encoding of the Higher-Order Calculus by using explicit operators for pattern matching and substitutions. In particular, we present two different evaluation strategies for the TPC_{ES} calculus of Cerrito and Kesner [CK99]. For each of these strategies we specify two *evaluators*, namely a *big-step* one and a *small-step* one. This work on calculi with explicit pattern matching and substitutions can be seen as a extension of the work of Hardin, Maranget and Pagano [HMP95].

The more interesting case is the small-step lazy evaluator which breaks the orthogonality between *pattern matching* and *substitution* in such a way that *propagation* of substitution is only performed when an explicit substitution operator appears as the head constructor of the term being evaluated.

To illustrate the difference between the orthogonal approach and our, let us take the following TPC_{ES}-term:

$$M_0 = Let \ < x_1, x_2 > \ be \ < t_1, t_2 > \ in \ t_1$$

where is a term and is a *pattern*.

Any term of the form $Let\ N\ be\ P\ in\ M$ in the TPC_{ES} calculus can be understood as a classical λ-term $(\lambda P.\nu(M))\nu(N)$, where P is a complex pattern and ν is a canoniocal translation from TPC_{ES} to the λ-calculus. In order

to evaluate the term M_0 the first point consists in decomposing the pattern matching part of M_0. Indeed, the pair pattern $< t_1, t_2 >$ will be matched with the pair term $< x_1, x_2 >$, thus giving:

$$M_1 = Let\ x_1\ be\ t_1\ in\ (Let\ x_2\ be\ t_2\ in\ t_1)$$

Now, the variable pattern t_1 will be matched with the variable term x_1, thus giving:

$$M_2 = (Let\ x_2\ be\ t_2\ in\ t_1)[t_1/x_1]$$

Now, substitution must be propagated into the sub-terms of M_2, thus giving:

$$M_3 = Let\ x_2[s]\ be\ t_2\ in\ t_1[s] \text{ where } s = t_1/x_1$$

Here is the point where the orthogonal approach differs from non-orthogonal one.

The orthogonal approach will first propagate the substitution s into the sub-terms (in two steps), thus giving:

$$M_5^c = Let\ x_2\ be\ t_2\ in\ x_1$$

Then, the pattern t_2 will be matched with the term x_2 (in one step), thus giving:

$$M_6^c = x_1[t_2/x_2]$$

Finally, the evaluation of the substitution gives the final result

$$M_7^c = x_1$$

The strategy that we propose in the small-step lazy evaluator applied to the term M_3 consists in first matching (in one step) the pattern t_2 with the term $x_2[s]$, thus giving:

$$M_4^o = t_1[s][t_2/x_2[s]]$$

Finally, classical propagation of substitutions can be performed in only two steps in order to obtain the final result x_1. Thus for this particular example, we have shown that the non-orthogonal approch leads to less calculi and is lazier than the orthogonal approch. This result seems to be valide for any TPC_{ES}-term.

Finally, our purpose in this talk is to theoretically define what both strict and lazy strategies for calculi with explicit operators for pattern matching and substitution are. Moreover we show that in the case of the lazy strategy, one maybe should improve the laziness of the strategy by not treat pattern matching and substitution as two orthogonal operations. I hope this result should be extented to higher-order pattern matching calculi.

References

[BTKP93] Val Breazu-Tannen, Delia Kesner, and Laurence Puel. A typed pattern calculus. In Moshe Vardi, editor, *Eight Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 262–274. IEEE Computer Society Press, June 1993.

[CK99] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[HMP95] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional backends and compilers within the lambda-sigma calculus, 1995. Draft.

[HPJe92] Paul Hudak, Simon Peyton-Jones, and Philip Wadler (editors). Report on the programming language haskell, a non-strict, purely functional language (version 1.2). Sigplan Notices, 1992.

[Obj] The Objective Caml language. `http://caml.inria.fr/`.

Term Rewriting with Variable Binding

Makoto Hamana

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

The theory of higher-order rewriting is recently very active research area in rewriting. We present a variation of higher-order rewrite system, called Binding Term Rewriting Systems (BRSs). This is based on the recent development on the mathematical semantics of abstract syntax with variable binding by Fiore,Plotkin and Turi [FPT99]. This work is a continuation of our previous work on a logic programming with variable binding [Ham01], which was also based on Fiore-Plotkin-Turi's semantics. In contrast to most of other higher-order rewrite systems, BRSs do not use the lambda calculus for the binding mechanism, instead, own finer notion of binding is defined in the term language by type-theoretic approach.

A motivating example of our BRSs is a set of rewrite rules for the untyped lambda calculus, which can be considered as a typical example of a system involving variable binding. BRSs are defined in typed language, where the only base type ι and the types $[\mathsf{N}]\sigma$ of abstraction from a type σ are assumed. Assume three function symbols $\mathsf{app} : \iota, \iota \to \iota$, $\mathsf{lam} : [\mathsf{N}]\iota \to \iota$ and $\mathsf{sub} : [\mathsf{N}]\iota, \iota \to \iota$ for the lambda calculus. Then we can represent the lambda terms $\lambda a.M$ as the BRS's terms $\mathsf{lam}([a]M)$, where $[\_]$ is own abstraction operator of BRSs. For example, $\lambda c.\lambda d.d$ is represented by $\mathsf{lam}([c]\mathsf{lam}([d](\mathsf{nam}(d))))$ where $\mathsf{nam}(d)$ is a name which represents a variable d of the lambda term. Then β-rule can be written by a BRS's rewrite rule:

$$t : \iota : a \vdash \mathsf{app}(\mathsf{lam}([a]t), s) \to \mathsf{sub}([a]t, s) : \iota : \varnothing.$$

Here the term $\mathsf{sub}([a]t, s)$ expresses a usual meta-level substitution $t[a := s]$ in object-level (i.e. BRS's) term. To make it actually a substitution, we can define the following rewrite rules:

$$\begin{aligned}
y{:}\iota{:}\varnothing \vdash \mathsf{sub}([a]\mathsf{nam}(a), y) &\to y : \iota : \varnothing \\
y{:}\iota{:}b \vdash \mathsf{sub}([a]\mathsf{nam}(b), y) &\to \mathsf{nam}(b) : \iota : b \\
e_1{:}\iota{:}a,\ e_2{:}\iota{:}a,\ y{:}\iota{:}\varnothing \vdash \mathsf{sub}([a]\mathsf{app}(e_1, e_2), y) &\to \mathsf{app}(\mathsf{sub}([a]e_1, y), \mathsf{sub}([a]e_2, y)) : \iota : \varnothing \\
x{:}\iota{:}a, b,\ y{:}\iota{:}\varnothing \vdash \mathsf{sub}([a]\mathsf{lam}([b]x), y) &\to \mathsf{lam}([b](\mathsf{sub}([a]x, y))) : \iota : \varnothing.
\end{aligned}$$

BRS's rewrite rules are also typed. Behind the colons in the right-hand sides, type informations are written; the first is a usual type and the second is called a *stage* which means (possible) free names in the rewrite rules. The important thing in BRSs is that different names written in a term denote actually different names. This means that in the second rule, the name a and b can not be the same, so the term $\mathsf{sub}([a]\mathsf{nam}(a), t)$ can only match the first rule.

The front of $\vdash$ in the rules are typing contexts. For example, $y{:}\iota{:}\varnothing$ means that y is the type ι and can be substituted by a term having no names, because it has the stage $\varnothing$. And $x{:}\iota{:}a, b$ means that x can contain the free names a and b. The matching can be performed as similar to the first-order case*

* The matching algorithm in this term language can be given by a special case of the unification algorithm [Ham02a].

So for example, the term $\mathsf{sub}([a]\mathsf{lam}([b]f(\mathsf{nam}(a),\mathsf{nam}(b))),\ \mathsf{nam}(c))$ can match the last rule with the substitution $x \mapsto f(\mathsf{nam}(a),\mathsf{nam}(b))$. This is very different from other higher-order systems because this substitution makes the names a and b *caputered* by the binders, which is usually avoided by suitable α-renaming. This kind of capture is admitted only when the variable declaration has a stage which includes bound names such as $x{:}\iota{:}a, b$. Namely, stage controls the form of substitution.

Names are a bit similar to constants in TRSs, which cannot be substituted by terms directly like variables. So if we need to replace a name by some term, we define a set of rules for substitution such as sub in a BRS. Of course, we have α-renaming of terms, say, the term $\mathsf{sub}([d]\mathsf{nam}(d),\mathsf{nam}(c))$ can match the second rule. We have also a different kind of α-renaming, which can be considered as α-renaming between rules (not terms). The difference between λ-binder and BRS's binder $[\_]$ appears here. For instance, in the rewrite rule

$$x : \iota : \underline{a} \vdash f([\underline{a}]x) \to g([\underline{a}]x) : \iota : \varnothing$$

the underlined a's are actually the same a because we have such "α-renaming between rules". So α-renaming of this rule is replacing all these a's with some other name. This is not in the case of Higher-order rewrite systems, e.g. for the rewrite rule

$$f(\lambda\underline{a}.x) \to g(\lambda\underline{a}.x)$$

the underlined a's are not actually the same a because of usual α-conversion.

We first define the type system of the term language, then develop equational logic and rewriting for BRSs, and show a correspondence between them. Our system BRSs is rather similar to the first-order term rewriting system (TRSs) than other higher-order rewrite systems. In particular, our main result is BRS rewriting can be simulated by TRS by giving a translation from a BRS to a corresponding TRS. The translation is just forgetting all type and stage informations and regarding the binding and name operators as usual first-order function symbols, and taking all possible variants with respect to free and bound names of the original BRS's rewrite rules. Since the translated TRS's rules do not have the stage restriction, it may produce more rewriting than the original BRS's. Hence, the translated TRS is terminating, then so is the original BRS. As a result, it is possible to prove termination of BRSs by using the proving techniques for termination of TRSs. Also we will discuss comparison between BRSs and other higher-order rewrite systems.

For further details the reader is referred to the draft [Ham02b].

References

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.

[Ham01] M. Hamana. A logic programming language based on binding algebras. In N. Kobayashi and B. C. Pierce, editors, *4th International Symposium on Theoretical Aspects of Computer Software (TACS 2001), Sendai*, volume 2215 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2001.

[Ham02a] M. Hamana. Simple β_0-unification for terms with context holes. In *Proceedings of The 16th International Workshop on Unification (UNIF 2002)*, 2002. Satelite workshop of FLoC'02, Copenhagen, Denmark.

[Ham02b] M. Hamana. Term rewriting with variable binding. Draft. available from `http://www.keim.cs.gunma-u.ac.jp/~hamana/Papers/brs.ps`, 2002.

UNTYPED TERM REWRITING

Abstract, HOR'02

Daniel Leivant*

This note is a crude initial statement of a simple idea, with a rough outline of some of its potential applications and uses. Since the author is far from being an expert in the field of rewrite systems, he would be particularly grateful for feedback.

1. THE U-CALCULUS.

Term rewriting is word rewriting regulated by first-order functionality. More generally, in higher-order rewriting term application is governed simply-typed functionality. We consider a formalism **U** of *untyped* term rewriting, in which the parsing of terms by their applicative structure is a regulator still in force, but with no typing restriction. In this note we illustrate the power of the calculus with some examples, and outline potential relations with computability in higher type and reasoning about programs.

The syntax of **U** posits two kinds of identifiers: the *object variables* and the *functional parameters.* We use $x, y, \ldots$ and $\mathbf{f}, \mathbf{g} \ldots$ as syntactic parameters for these kinds, respectively. *Terms* are generated inductively from identifiers (of both kinds) by application: if $\mathbf{t}$ and $\mathbf{s}$ are terms, then so is $\mathbf{t}(\mathbf{s})$. We call a term *computational* if it is of the form $\mathbf{f}(\mathbf{t}_1)\cdots(\mathbf{t}_m)$, i.e. with a parameter in the lead. Thus $\mathbf{f}(x)$, $\mathbf{f}(\mathbf{f})$, $\mathbf{f}(x)(\mathbf{f})$, $\mathbf{f}(x(\mathbf{f}))$, $x(x)$, $x(\mathbf{f}(x))(\mathbf{f})$ and $x(\mathbf{f}(x)(\mathbf{f}))$ are all terms, but only the first four are computational. A *rule* is a pair $\mathbf{t} \Rightarrow \mathbf{t}'$, where $\mathbf{t}$ is computational. The *inference-rules* for rules are *instantiation*, *replacement*, and *composition:*

$$\frac{\mathbf{t} \Rightarrow \mathbf{t}'}{\{\mathbf{q}/x\}(\mathbf{t} \Rightarrow \mathbf{t}')} \qquad \frac{\mathbf{q} \Rightarrow \mathbf{q}'}{\{\mathbf{q}/x\}\mathbf{t} \Rightarrow \{\mathbf{q}'/x\}\mathbf{t}} \qquad \frac{\mathbf{t} \Rightarrow \mathbf{s} \quad \mathbf{s} \Rightarrow \mathbf{q}}{\mathbf{t} \Rightarrow \mathbf{q}}$$

A *procedure* is a finite set of rules.

Examples.

1. *First order term-rewriting.*
2. A special case of first-order term-rewriting is standard word rewriting: a word $\mathbf{a}_1 \cdots \mathbf{a}_k$ is identified with the term $\mathbf{a}_1(\cdots(\mathbf{a}_k(\boldsymbol{\varepsilon}))\cdots)$
3. SK Combinatory Logic.
4. *Simultaneous recursion equations,* whence the $\boldsymbol{\lambda}\mathbf{Y}$-calculus.

*Computer Science Department, Indiana University, Bloomington, IN 47405. leivant@cs.indiana.edu. Research partially supported by NSF grant CCR-9309824.

2. INTERPRETATION OF THE LAMBDA CALCULUS

The *Lambda calculus* can be simulated within Combinatory Logic. That simulation can also be described directly. Let us use sans-serif characters for the λ-calculus variables, to distinguish them from the *italic* characters we use for variables of **U**. We stipulate, without loss of generality, that no λ-term has a variable occurring both bound and free. For each λ-term M and a listing $\vec{\mathsf{x}} = \mathsf{x}_1 \ldots \mathsf{x}_r$ of λ-variables that includes all variables free in M, and no variable bound in M, we introduce a functional parameter (i.e. identifier), which we write as $\{M[\vec{\mathsf{x}}]\}$, intended to represent $\lambda\vec{\mathsf{x}}.M$. Writing $\vec{y}$ for the tuple $y_1 \cdots y_r$ of rewrite variables (of the same length, r, as $\vec{\mathsf{x}}$), the rules are as follows.

$$\begin{array}{rcl} \{\mathsf{x}_i[\vec{\mathsf{x}}]\}(\vec{y}) & \Rightarrow & y_i \\ \{(\lambda\mathsf{z}.M)[\vec{\mathsf{x}}]\} & \Rightarrow & \{M[\vec{\mathsf{x}},\mathsf{z}]\} \\ \{(M\,N)[\vec{\mathsf{x}}]\}(\vec{y}) & \Rightarrow & \{M[\vec{\mathsf{x}}]\}(\vec{y})(\{N[\vec{\mathsf{x}}]\}(\vec{y})) \end{array}$$

The reduction rules of the λ-calculus can be introduced explicitly as rewrite rules. However, β-conversion is already derived from the evaluation rules above for λ-terms, as follows.

Let $M_{z:=N}$ be the result of substituting N for the free occurrences of z in M. We use this notation when N is *substitutable in* M, i.e. when no variable is free in N and bound in M.[1] We write $M \approx M'$ if M and M' rewrite into a common term, i.e. $M \Rightarrow M''$ and $M' \Rightarrow M''$ for some M''.

LEMMA **1** [Substitution] *Let $\vec{\mathsf{x}}$ and $\vec{y}$ have equal length. Under the conventions above,*

$$\{M_{z:=N}[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y},\vec{y}') \approx \{M[\vec{\mathsf{x}},\mathsf{z},\vec{\mathsf{x}}']\}(\vec{y})(n)(\vec{y}')$$

where $n =_{\mathrm{df}} \{N[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y})(\vec{y}')$.

Proof. Induction on M. For M a λ-variable x other than z, and the **U**-variable y corresponding to it, we have

$$\begin{array}{l} \{\mathsf{x}_{z:=N}[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y})(\vec{y}') \\ \quad \equiv \{\mathsf{x}[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y})(\vec{y}') \\ \quad \Rightarrow y \end{array}$$

and

$$\{\mathsf{x}[\vec{\mathsf{x}},\mathsf{z},\vec{\mathsf{x}}']\}(\vec{y})(n)(\vec{y}') \Rightarrow y.$$

For $M \equiv \mathsf{z}$,

$$\begin{array}{l} \{\mathsf{z}_{z:=N}[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y})(\vec{y}') \\ \quad \equiv \{N[\vec{\mathsf{x}},\vec{\mathsf{x}}']\}(\vec{y})(\vec{y}') \\ \quad \equiv n \end{array}$$

[1] This condition can be refined of course, but to no particular benefit here.

and

$$\{\mathsf{z}[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}]\}(\vec{y})(n)(\vec{y'}) \Rightarrow n$$

For the induction step for λ-abstraction we have

$$\begin{aligned}
&\{(\lambda\mathsf{u}.M)[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}]\}(\vec{y})(n)(\vec{y'}) \\
&\quad \Rightarrow \{M[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}, \mathsf{u}]\}(\vec{y})(n)(\vec{y'}) \\
&\quad \approx \{M_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}, \mathsf{u}]\}(\vec{y})(\vec{y'}) \quad \text{by IH}
\end{aligned}$$

On the other hand,

$$\begin{aligned}
&\{(\lambda\mathsf{u}.M)_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}) \\
&\quad \equiv \{(\lambda\mathsf{u}.M_{z:=N})[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}) \quad \text{given the conventions on variables} \\
&\quad \Rightarrow \{M_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}, \mathsf{u}]\}(\vec{y})(\vec{y'})
\end{aligned}$$

Finally, for the induction step for application we have,

$$\begin{aligned}
&\{(M\,M')_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}) \\
&\quad \Rightarrow \{M_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}) \;\; (\{M'_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}))
\end{aligned}$$

while on the other hand,

$$\begin{aligned}
&\{(M\,M')[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}]\}(\vec{y})(n)(\vec{y'}) \\
&\quad \Rightarrow \{M[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}]\}(\vec{y})(n)(\vec{y'}) \;\; (M'[\vec{\mathsf{x}}, \mathsf{z}, \vec{\mathsf{x}'}]\}(\vec{y})(n)(\vec{y'})) \\
&\quad \approx \{M_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'}) \;\; (\{M'_{z:=N}[\vec{\mathsf{x}}, \vec{\mathsf{x}'}]\}(\vec{y})(\vec{y'})) \qquad \text{by IH} \qquad \dashv
\end{aligned}$$

From the Lemma we immediately obtain

PROPOSITION **2** *The rule of β-conversion is derived for the interpretation above of the λ-calculus. That is, under the conventions above,*

$$\{((\lambda\mathsf{z}.M)N)[\vec{\mathsf{x}}]\}(\vec{y}) \approx \{M_{z:=N}[\vec{\mathsf{x}}]\}(\vec{y})$$

Proof.

$$\begin{aligned}
&\{((\lambda\mathsf{z}.M)N)[\vec{\mathsf{x}}]\}(\vec{y}) \\
&\quad \Rightarrow \{(\lambda\mathsf{z}.M)[\vec{\mathsf{x}}]\}(\vec{y}) \; (\{N[\vec{\mathsf{x}}]\}(\vec{y})) \\
&\quad \Rightarrow \{M[\vec{\mathsf{x}}, \mathsf{z}]\}(\vec{y}) \; (\{N[\vec{\mathsf{x}}]\}(\vec{y})) \\
&\quad \approx \{M_{z:=N}[\vec{\mathsf{x}}]\}(\vec{y}) \qquad \text{by the Lemma} \qquad \dashv
\end{aligned}$$

The simulation above of the λ-calculus also yields an interpretation in **U** of higher order rewriting (HOR), say in the style of Nipkow [6]. The general rules of HOR are rewrite rules that refer to terms of the simply typed lambda calculus. Such terms can be simulated as above, giving rise to rules of **U**.

Many standard examples can be formulated in **U** directly. For instance,

$$map(f)(cons(x)(y)) \Rightarrow cons(f(x))(map(f)(y))$$

is admissible as it stands, and indeed is generic with respect to types (which is not the case in HOR).

3. PATTERN MATCHING AND THE ρ-CALCULUS

Fix a functional parameter R. Consider a pattern-matching rule $\mathbf{t} \Rightarrow \mathbf{u}$, where $\vec{x}$ is a list of the variables free in $\mathbf{t}$. The functional behavior of the pattern-matching rule is encapsulated by the term $\mathsf{R}(\mathbf{t})(\mathbf{u})$ and the rewrite rule $\mathsf{R}(\mathbf{t})(\mathbf{u})(\sigma\mathbf{t}) \Rightarrow \sigma\mathbf{u}$, where $\sigma = \{\vec{y}/\vec{x}\}$, with $\vec{y}$ a list of fresh variables.

Consider for example the ρ-term

$$[[(x \to x{+}1) \to (1 \to x)](a \to a{+}1)](1)$$

considered in [2, §2.1] as an example of a rule with no corresponding λ-term. To this corresponds the term

$$\mathsf{R}(\mathsf{R}(x)(x{+}1))(\mathsf{R}(1)(x))(\mathsf{R}(a)(a{+}1))(1).$$

Under the term-rewrite schema above

$$\mathsf{R}(\mathsf{R}(x)(x{+}1))(\mathsf{R}(1)(x))(\mathsf{R}(y)(y{+}1)) \Rightarrow \mathsf{R}(1)(y).$$

By instantiation

$$\mathsf{R}(\mathsf{R}(x)(x{+}1))(\mathsf{R}(1)(x))(\mathsf{R}(a)(a{+}1)) \Rightarrow \mathsf{R}(1)(a).$$

And so

$$\mathsf{R}(\mathsf{R}(x)(x{+}1))(\mathsf{R}(1)(x))(\mathsf{R}(a)(a{+}1))(1) \Rightarrow \mathsf{R}(1)(a)(1).$$

From the rewrite schema for R we also have $\mathsf{R}(1)(a)(y) \Rightarrow a$, so the final r.h.s. is a.

Thus the central idea of the ρ-calculus, i.e. abstraction on patterns, is captured in the $\mathbf{U}$-calculus. We conjecture that much of the ρ-calculus is interpretable in the $\mathbf{U}$-calculus.

4. FUNCTIONALS OVER INDUCTIVE DATA-SYSTEMS

Consider an inductively generated (multi-sorted) data-system C, such as the free algebra $\mathbb{N}$, whose constructors are the 0-ary $\mathbf{0}$ and the unary $\mathbf{s}$, or the free algebra $\mathbb{W} = \{0,1\}^*$ whose constructors are the 0-ary $\boldsymbol{\varepsilon}$ and the unary $\mathbf{0}$ and $\mathbf{1}$. The calculus $\mathbf{U}(C)$ differs from $\mathbf{U}$ only in treating the constructors of C as additional identifiers, apart from the variables and parameters. We start by adopting the most liberal approach, and let the calculus be oblivious to the functionality of the constructors, including their arities. For example, in $\mathbf{U}(\mathbb{N})$ we can form terms such as $\mathbf{0}(x(\mathbf{s}))(\mathbf{0})$. Still, the constructors differ from the variables in that they cannot be instantiated, and they differ from the parameters in that terms starting with them are not declared computational. The base-terms of C are represented in $\mathbf{U}(C)$ in Curried form. For example, if among the constructors of C are a binary $\mathbf{p}$ and a 0-ary $\boldsymbol{\varepsilon}$ then the base-term $\mathbf{p}(\mathbf{p}(\boldsymbol{\varepsilon},\boldsymbol{\varepsilon}),\boldsymbol{\varepsilon})$ is represented by $\mathbf{p}(\mathbf{p}(\boldsymbol{\varepsilon})(\boldsymbol{\varepsilon}))(\boldsymbol{\varepsilon})$.

Let $Tp(C)$ be the system of simple types over C: the base types are (names for) the sorts of C, and compound types are generated using $\rightarrow$ ($\times$ can be included with minor modifications). To simplify, let C be $\mathbb{N}$, and let ι be the name of the unique sort. A *program* of $\mathbf{U}(\mathbb{N})$ is a triplet $(P, \mathbf{f}, \tau)$ where P is a procedure, $\mathbf{f}$ a functional parameter, and τ a type of $Tp(\mathbb{N})$. We wish to define the computation of partial functionals of type τ by programs $(P, \mathbf{f}, \tau)$, starting with unary first-order functions. A partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is *computed* by $(P, \mathbf{f}, \iota \rightarrow \iota)$ if for all $n, m \in \mathbb{N}$, $\mathbf{f}(n) \Rrightarrow m$ is derived from P iff $f(n) = m$. An equivalent definition which generalizes better to higher types is this. Rephrase the definition of a program as above to include an additional parameter $\mathbf{a}$. The program $(P, \mathbf{f}, \mathbf{a}, \iota \rightarrow \iota)$ is said to *compute* $f : \mathbb{N} \rightarrow \mathbb{N}$ if for all $n, m \in \mathbb{N}$, $f(n) = m$ iff $\mathbf{f}(\mathbf{a}) \Rrightarrow m$ is derived from $P + (\mathbf{a} \Rrightarrow m)$.

For second order functionals we say that a partial functional of type $\xi = (\iota \rightarrow \iota) \rightarrow \iota$ is *computed* by program $(P, \mathbf{f}, \mathbf{a}, \xi)$, if for all unary (partial) function g over $\mathbb{N}$ and all $m \in \mathbb{N}$, $fg = m$ iff $\mathbf{f}(\mathbf{a}) \Rrightarrow m$ is derived from $P + \{ (\mathbf{a}(n) \Rrightarrow gn \mid n \in \mathbb{N}$ and gn is defined $\}$.

The generalization to all second-order types is trivial. However, the generalization of these definitions to functionals of order > 2 must involve continuity conditions. Note that a procedure P may be used in programs for different types.

5. COMPUTABILITY AS MODEL THEORY.

Here we switch to *undirected* rewrite systems, that is, reading $\Rrightarrow$ as equality, and using derivations in equational logic. The *vocabulary* V_P of a procedure P of $\mathbf{U}(C)$ consist of the parameters in P and the constructors of C. Let $\mathcal{T}_P$ be the set of all (untyped) terms over V_P. The *canonical model* $\mathcal{M}_P$ of P has as universe the quotient $\mathcal{T}_P / \cong$, where $\mathbf{t} \cong \mathbf{t}'$ iff there is an equational derivation of $\mathbf{t} \approx \mathbf{t}'$ from P. Herbrand falsely conjectured that a function f over $\mathbb{N}$ is computable iff it is the unique solution over $\mathbb{N}$ of a set P of recursion equations. A revised version of his conjecture does hold, though: f is computable iff it is the unique restriction to $\mathbb{N}$ of solutions of P in $\mathcal{M}_P$. For the definition above of second-order computability we believe that a similar statement holds: *A functional f over C, of type τ of rank ≤ 2, is total and computable iff it is the unique restriction to C_τ of solutions of P of type τ over $\mathcal{M}_P$.*

This model theoretic property now generalizes automatically to all types τ. MAJOR QUESTION: what is the computational notion that corresponds to this definition of computability in higher type, and how does it relate to familiar definitions, e.g. Kleene's definition of *total* computable functionals of higher type [3].

6. PROOF THEORY: WHERE TYPES COME FROM.

Let C be an inductive data-system, and consider unary relational identifiers for the sorts of C. Say C is the single-sorted $\mathbb{N}$, with $\mathbb{N}$ as sort-identifier. For a type $\tau \in Tp(\mathbb{N})$ we define a formula $T_\tau[x]$, over the vocabulary of rewrite terms augmented with $\mathbf{N}$, which states that (the object denoted by) x has semantic

(i.e. Curry-style) type τ. This is defined by recurrence on τ: $T_{\mathbf{N}}$ is $\mathbf{N}$, and $T_{\sigma\to\tau}[x] \equiv_{\mathrm{df}} \forall y.\, T_\sigma[y] \to T_\tau[x(y)]$. We say that an equational program $(P, \mathbf{f})$ *has semantic-type* τ if $\models \bar{P} \to T_\tau[\mathbf{f}]$. For first order types τ this is equivalent to $\mathcal{M}_P \models \bar{P} \to T_\tau[\mathbf{f}]$, and also to the totality of the function of type τ computed by $(P, \mathbf{f})$. We conjecture that the same holds true for all types.

The definition of T_τ can be trivially extended to polymorphic types τ of F_2 or even F_ω. For a type variable X define T_X to be X, understood as a variable for sets (unary relations). Then, define $T_{\forall X.\tau}[x] \equiv_{\mathrm{df}} \forall X.T_\tau[x]$.

Let $\mathbf{L}(C)$ be a natural deduction formulation of constructive logic in all finite types, with data-introduction and data-elimination (induction) rules for C [5]. Let $\boldsymbol{\lambda}(C)$ be F_ω augmented with the constructors of C and recurrence operators over C. Let $(P, \mathbf{f})$ be an untyped equational program over the data-system C. We know that for a first-order type τ, a derivation in $\mathbf{L}(C)$ for $\bar{P} \to T_\tau[\mathbf{f}]$ maps under the homomorphism of [4, 5] (a contracted form of Curry-Howard) to a term of $\boldsymbol{\lambda}(C)$ which defines the type τ-function computed by $(P, \mathbf{f})$. A major task is to extend this to higher types: rank-2 (i.e. second order functionals), finite-types (functionals of arbitrary type), and F_2 and F_ω types.

References

[1] J. Barwise, editor. *Handbook of Mathematical Logic*. North-Holland, Amsterdam, 1977.

[2] Horatiu Cistera and Claude Kirchner. Introduction to the rewriteing calculus. Research Report 3818, INRIA, 1999.

[3] A. Kechris and Y. Moschovakis. Recursion in higher types, 1977. In [1], 681–738.

[4] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, London, 1990.

[5] Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 114:117–153, 2002.

[6] Tobias Nipkow. Higher-order critical pairs. In *Sixth IEEE Symposium on Logic in in Computer Science*, pages 342–349, San Diego, 1991. IEEE Computer Society.

On head-rewriting paths in the $\lambda\sigma$-calculus

Paul-André Melliès

CNRS, Université Paris 7

Abstract

In this note, we illustrate our definition of head-rewriting path in the $\lambda\sigma$-calculus [6] and explicate the proof of a theorem appearing in [5]. The theorem states that every head-rewriting path $M \longrightarrow V$ in the $\lambda\sigma$-calculus projects (by σ-normalization) to the head-rewriting path $\sigma(M) \longrightarrow \sigma(V)$ in the λ-calculus.

The point of this note is not to present new material on the $\lambda\sigma$-calculus, but to summarize the results obtained by the author since 1995, and the counter-example [4]. All results discussed here are published in [5,6].

The general pattern of this work goes as follows:

(1.) start from a syntactical rewriting system $\mathcal{R}$ like the $\lambda\sigma$-calculus,
(2.) define (syntactically) the class $\mathcal{V}$ of *values* (= head-normal forms) of $\mathcal{R}$,
(3.) applying a stability theorem [6] to characterize (diagrammatically) the head-rewriting paths of $\mathcal{R}$,
(4.) uncover from diagrammatic step (3.) the syntactical nature of the head-rewriting paths of $\mathcal{R}$.

The methodology goes beyond the particular case of λ-calculi with explicit substitutions, and may be applied on any (interesting enough!) rewriting system. The methodology is particularly useful to analyze rewriting systems admitting critical pairs, which lack the usual "confluence methodology" inherited from Church, Rosser, Knuth and Bendix historical contributions.

1 Head-normal forms in the $\lambda\sigma$-calculus

The set $\mathcal{V}_{\lambda\sigma}$ of head-normal forms in the $\lambda\sigma$-calculus is defined in a similar fashion as the set $\mathcal{V}_\lambda$ of head-normal forms in the λ-calculus.

Definition 1 (head-normal forms) *A closed $\lambda\sigma$-term V is called a $\lambda\sigma$-head-normal form when it may be written as*

$$V \quad = \quad \lambda...\lambda(\mathbf{i}M_1...M_n)$$

where $\mathbf{i} = \mathbf{1}[\uparrow \circ \cdots \circ \uparrow]$ is a de Bruijn number, and $M_1, ..., M_n$ are $\lambda\sigma$-terms.

Obviously, every $\lambda\sigma$-term V in head-normal form σ-normalizes to λ-term $\sigma(V)$ in head-normal form. But the converse is not true: think of the $\lambda\sigma$-term $V[s]$ where V is a $\lambda\sigma$-term in head-normal form.

Now, let M denote the λ-term $M = (\lambda x.(\lambda y.y)x)V$ where V is a λ-term in head-normal form. This λ-term M induces a head-rewriting path to the set $\mathcal{V}_\lambda$ of head-normal forms in the λ-calculus:

$$(\lambda x.(\lambda y.y)x)V \longrightarrow (\lambda y.y)V \longrightarrow V$$

The $\lambda\sigma$-term $U(M) = (\lambda(\lambda \mathbf{1})\mathbf{1})V$ is the de Bruijn notation for the λ-term M. At this point, we have not yet defined what we mean by *head-rewriting path* of the $\lambda\sigma$-calculus. However, we may already indicate, in the informal style, several candidates of head-rewriting path from M to the set $\mathcal{V}_{\lambda\sigma}$ of head-normal forms, see figure 1 in appendix.

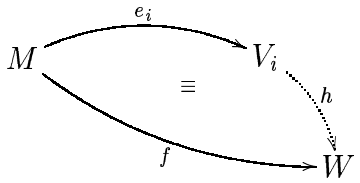
2 Head-rewriting paths in the $\lambda\sigma$-calculus (diagrammatically)

The central point of our work is a stability theorem proved diagrammatically [6] for a wide class of (possibly higher-order, possibly conflicting) rewriting systems. Here, for simplicity, we instantiate the diagrammatic theorem, and formulate it for the $\lambda\sigma$-calculus, and the set $\mathcal{V}_{\lambda\sigma}$ of head-normal forms.

Let $\equiv$ denote the Lévy permutation equivalence on paths. The theorem states that, for every $\lambda\sigma$-term M, there exists a cone

$$(M \xrightarrow{e_i} V_i)_{i\in I}$$

of paths from M to $\mathcal{V}$, *universal* in the sense that every path $f : M \longrightarrow W$ from M to a head-normal form W factorizes as $M \xrightarrow{e_i} V_i \xrightarrow{h} W$ (modulo $\equiv$) for *one and only one* index $i \in I$.



This leads to a diagrammatic definition of head-rewriting paths in the $\lambda\sigma$-calculus.

Definition 2 (head-rewriting path) *A $\lambda\sigma$-rewriting path $f : M \longrightarrow N$ is* head-rewriting *if it appears as an element $e_i : M \longrightarrow V_i$ of the universal cone $(e_i : M \longrightarrow V_i)_{i\in I}$.*

3 Head-rewriting paths in the $\lambda\sigma$-calculus (syntactically)

By its diagrammatic nature, definition 2 does not provide any information about the syntactic shape of head-rewriting paths in the $\lambda\sigma$-calculus. In good cases, typically in the λ-calculus, the shape of head-rewriting paths may be captured by a big-step semantics. Defining such a big-step semantics would be quite difficult in the $\lambda\sigma$-calculus. So, instead of introducing a heavy big-step semantics, we prefer to establish nice characterizing properties of the head-rewriting paths of the $\lambda\sigma$-calculus. Namely, we establish that

Theorem 3 (adequacy) *Every head-rewriting path*

$$M \xrightarrow{e_i} V_i$$

of the $\lambda\sigma$-calculus projects (by σ-normalization) to the unique head-rewriting path

$$\sigma(M) \xrightarrow{\sigma(e_i)} \sigma(V_i)$$

starting from $\sigma(M)$ in the λ-calculus.

The theorem appears in [5]. It somewhat counter-balances the counter-example [4] with a nice adequacy between the $\lambda\sigma$-calculus, and the λ-calculus.

Let us outline the proof here. Let $U(M)$ denote the de Bruijn notation of the λ-term M, seen as a $\lambda\sigma$-term. The main technical lemma follows:

Lemma 4 *Every standard rewriting path $f : M \longrightarrow U(N)$ in the $\lambda\sigma$-calculus is interpreted as a standard path $\sigma(f) : \sigma(M) \longrightarrow N$ in the λ-calculus.*

The result is more subtle than it looks at first sight. For instance, the translation from $\lambda\sigma$ to λ by σ-normalisation, does not preserve standardization in general. For instance, consider the standard $\lambda\sigma$-rewriting path

$$((\lambda\mathbf{11})\mathbf{1})[\underline{(\lambda\mathbf{1})\mathbf{1}} \cdot id] \longrightarrow (\underline{(\lambda\mathbf{11})\mathbf{1}})[\mathbf{1}[\mathbf{1} \cdot id] \cdot id] \longrightarrow (\mathbf{11})[\mathbf{1} \cdot id][\mathbf{1}[\mathbf{1} \cdot id] \cdot id]$$

The path translates as an inside-out computation in the λ-calculus:

$$(\lambda x.xx)(\underline{(\lambda y.y)\mathbf{1}}) \longrightarrow \underline{(\lambda x.xx)\mathbf{1}} \longrightarrow \mathbf{11}$$

The proof of lemma 4 starts with a definition, and a syntactic lemma relating occurrences and rewriting paths.

Definition 5 (enshrine,annihilate,preserve)

- *An occurrence x* enshrines *a redex $u : P \longrightarrow Q$ when x is a strict prefix of the occurrence of u.*
- *A redex u* annihilates *an occurrence x when the occurrence of u is prefix of x.*
- *A path $u_1; \cdots; u_n$* preserves *an occurrence x when none of the redexes u_i annihilates x.*

Lemma 6 *Suppose that an occurrence x enshrines a redex $r : P \longrightarrow Q$. Every standard path $f = r; g$ preserves the occurrence x when:*

- *x is a cons-node $(M \cdot s)$,*
- *x is a λ-node λM,*
- *x is an application node MN and the path f is a σ-path.*

From lemma 6, it follows that no $\lambda\sigma$-redex contracted in the standard path $f : M \longrightarrow U(N)$ ever occurs enshrined in a cons-node $M \cdot s$. Otherwise, the cons-node $M \cdot s$ would appear in the resulting $\lambda\sigma$-term $U(M)$, which would contradict the fact that $U(M)$ is the de Bruijn notation of a λ-term. From this, it follows in turn that no *Beta*-redex contracted in f ever occurs inside a substitution s. Of course, the remark does not extend to the σ-redexes contracted in f, as the following standard path $f : M \longrightarrow U(N) = P$ illustrates:

$$\mathbf{1}[\underline{(\mathbf{1} \cdot id) \circ (P \cdot id)}] \stackrel{Map}{\longrightarrow} \underline{\mathbf{1}[\mathbf{1}[P \cdot id] \cdot id \circ (P \cdot id)]} \stackrel{VarCons}{\longrightarrow} \underline{\mathbf{1}[P \cdot id]} \stackrel{VarCons}{\longrightarrow} P$$

The property implies that every *Beta*-redex contracted in $f : M \longrightarrow U(N)$ is translated as a *unique* β-redex in $\sigma(f)$. So, whenever $f : M \longrightarrow U(N)$ is standard, we may consider $\sigma(f)$ as a proper λ-rewriting path — and not just as an equivalence class of λ-rewriting paths modulo permutation of disjoint redexes, as usual projections of $\lambda\sigma$-rewriting paths in the λ-calculus, see [5].

The proof of lemma 4 ends by showing that any standardisation cell $\alpha : \sigma(f) \Rightarrow h$ in the λ-calculus mirrors as a standardisation cell $\beta : f \Rightarrow g$ in the $\lambda\sigma$-calculus, in the sense that $\sigma(g) = h$. This proves lemma 4.

Once this lemma 4 established by syntactic means, the rest of the proof of theorem 3 proceeds by abstract non-syntactic arguments. Let us illustrate this with theorem 8. We know from [6] that every head-rewriting path (in the λ-calculus, in the $\lambda\sigma$-calculus, in any axiomatic rewriting system) is *external* in the sense below:

Definition 7 (external path) *A rewriting path $e : M \longrightarrow N$ is* external *in the λ-calculus or the $\lambda\sigma$-calculus, when for every right-composable rewriting path $f : N \longrightarrow P$,*

$$N \stackrel{f}{\longrightarrow} P \text{ is standard} \quad \Rightarrow \quad M \stackrel{e}{\longrightarrow} N \stackrel{f}{\longrightarrow} P \text{ is standard.}$$

Next lemma follows easily from lemma 4, using abstract arguments enables by definition 7.

Theorem 8 *Every external rewriting path $e : M \longrightarrow N$ in the λ-calculus is translated as an external path $\sigma(e) : \sigma(M) \longrightarrow \sigma(N)$ in the λ-calculus.*

We should mention at this point a recent manuscript [1] where Eduardo Bonelli "axiomatizes" our syntactic proof of lemma 4, in order to generalize theorem 8 to a wide class of λ-calculi with explicit substitutions. The axiomatization is still awfully complicated, but the subject is very young, and there are good

chances to see a simplified version emerge at a later stage.

4 Needed strategies in the $\lambda\sigma$-calculus

Here, we generalize to rewriting systems with critical pairs, the usual notion of *needed path* for orthogonal rewriting systems. We start from the definition of Luc Maranget in [3] for orthogonal rewriting systems: A redex $r : M \longrightarrow N$ is needed when it has a residual (at least) after any rewriting path $f : M \longrightarrow P$, unless f contracts at some step a residual of r.

We want to extend the definition to rewriting systems with critical pairs. Consider a rewriting system like:

$$A \longrightarrow B \qquad A \longrightarrow C$$

The critical pair in A implies that the redex $r : A \longrightarrow B$ has no residual after the reduction $f : A \longrightarrow C$. From this, should one conclude that $r : A \longrightarrow B$ is not needed? Well, not really! It is possible that the word *needed* is inadapted to qualify $r : A \longrightarrow B$, and that it should be replaced by the word *unavoidable*. But if we choose to keep the word *needed*, then we should consider $r : A \longrightarrow B$ as needed, since after all, every path from A either contracts a residual of $r : A \longrightarrow B$, or a redex forming a critical pair with a residual of $r : A \longrightarrow B$. We adapt Maranget's definition accordingly:

Definition 9 (needed redex) *A redex $r : M \longrightarrow N$ is needed when every rewriting path $f : M \longrightarrow P$ such that $r \sqsubseteq f$, contracts a residual of r at least.*

Here, $r \sqsubseteq f$ means that there exists a path g such that f and $r;g$ are equal, modulo Lévy permutation equivalence. Check that the redex $r : A \longrightarrow B$ is needed in our previous example, and that the definition is equivalent to Maranget's one for orthogonal rewriting systems. Let us say that the definition of needed path is formalized further in [5].

Gérard Huet and Jean-Jacques Lévy prove that every needed strategy normalizes in an orthogonal rewriting system. This is not true any more in the presence of critical pairs. Consider the term A in the first order rewriting system:

$$A \longrightarrow A \qquad A \longrightarrow B$$

Observe that the redex $r : A \longrightarrow A$ is external (a fortiori needed) and that the term A has normal form B. However, the *needed strategy* computing $A \longrightarrow A$ does not normalise A. Next result shows that the reason for non-termination is that there exists an infinite number of paths from A to its normal form B, modulo Lévy permutation equivalence $\equiv$. In fact, one Lévy permutation class for each path:

$$A \longrightarrow A \longrightarrow \cdots \longrightarrow A \longrightarrow B$$

Consider a (first-order, higher-order, axiomatic) rewriting system in which, for every term M, there exists at most a finite number of normalizing paths, modulo Lévy permutation equivalence. Among examples of such rewriting systems:

- orthogonal rewriting systems, in which the number is at most one, by a famous result by Lévy,
- the $\lambda\sigma$-calculus, by theorem 3 (or theorem 8) and strong normalization of the substitution σ-calculus.

We prove in [5]

Theorem 10 (needed normalization) *Every needed strategy normalizes in such a rewriting system.*

Theorem 10 generalizes Huet and Lévy classical theorem for needed strategies in orthogonal rewriting systems, see [2]. It also proves normalization of needed strategies in the $\lambda\sigma$-calculus.

After proving this result, the author hoped to explain that way many recent normalization results on rewriting systems with critical pairs. A nice example is Femke van Raamsdonk's normalization theorem for needed strategies in weakly orthogonal rewriting systems [8]. Unfortunately, in a private communication, Vincent van Oostrom [7] gave two examples of weakly orthogonal rewriting systems, in which a term M may have an *infinite* number of normalizing paths, modulo Lévy permutation equivalence.

First example

$$a \longrightarrow Fa \qquad GF^n x \longrightarrow b \quad \text{for every natural number } n$$

Second example

$$a \longrightarrow Fa \qquad Fb \longrightarrow b \qquad Fx \longrightarrow b$$

5 Conclusion

The quest for a satisfactory description of rewriting systems with critical pairs, is only starting.

References

[1] E. Bonelli. Axiomatic Rewrite Systems: ES and Zones. *Personal communication,* 2002.

[2] G. Huet, J.-J. Lévy. "Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems". Rapport de recherche INRIA 359, 1979. Republished

as "Computations in orthogonal rewriting systems, I and II", in Jean-Louis Lassez and Gordon Plotkin, editors, *Computational logic, essays in honor of Alan Robinson*, pages 395—443. MIT Press, Cambridge, Massachussets, 1991.

[3] L. Maranget. *La stratégie paresseuse.* Thèse de l'Université Paris VII, 1992.

[4] P.-A. Melliès, Typed lambda calculi with explicit substitutions may not terminate. In *Proceedings Typed Lambda Calculus and Applications 95*, Edinburgh, Lecture Notes in Computer Science 902, Springer Verlag, 1995.

[5] ———————— Axiomatic Rewriting Theory II: The lambda-sigma-calculus enjoys finite normalisation cones. *Journal of Logic and Computation*, vol 10, number 3, pp. 461-487, 2000.

[6] ———————— Axiomatic Rewriting Theory IV: A stability theorem in Rewriting Theory. *Proceedings of the 14th Annual Symposium on Logic in Computer Science*, Indianapolis, 1998.

[7] V. van Oostrom. Two counter-examples to a conjecture on finite cones and weakly orthogonal systems. *Personal communication,* 2000.

[8] F. van Raamsdonk, *Confluence and normalisation for higher-order rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, March 1994.

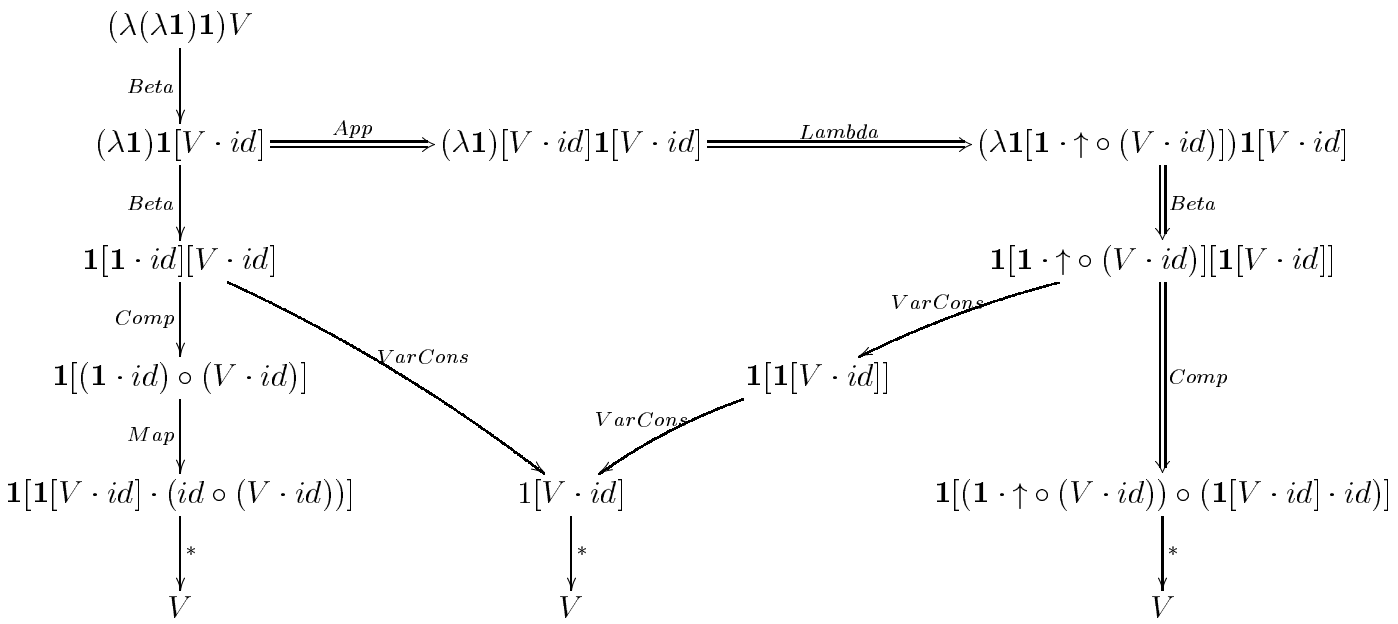


Fig. 1. Several head-rewriting paths in the $\lambda\sigma$-calculus

Higher-Order Pattern Disunification Revisited

Alberto Momigliano
Department of Mathematics and Computer Science
University of Leicester, Leicester, LE1 HR2, U.K.
am133@mcs.le.ac.uk

Higher-order pattern disunification is concerned with the (decidable) task of solving arbitrary quantified formulae on certain λ-terms, namely patterns, where the only predicate symbol is equality, call them *equational formulae*. The equality theory is the one induced by $\alpha\beta\eta$-conversion. Notwithstanding its interest, for example w.r.t. the issue of completeness of higher-order algebraic specifications, the problem has attracted little research, possibly due to its technical complexity.

The only account I am aware of is by Lugiez in [3], where he presents a set of rules which transform an equational formula on simply-typed patterns into a so-called *constrained solved form*. Alas, the treatment is very difficult to comprehend; indeed, the crux of the matter can be located in the issue of *dependency constraints*, already present, albeit not with the same impact, in pattern unification, namely in the Flex-Flex-Same case. We can provide a glimpse of the problem by asking, for example, what is the solution of a formula such as

$$\forall Y.\lambda x\,.\,Z\ x \neq_{\beta\eta} \lambda x\,.\,Y \tag{1}$$

where Y *does not* depend on x. Intuitively, Z equals a X which *must* depend on x. As the simply-typed λ-calculus is not strong enough to directly represent this, Lugiez modifies the language of terms to promote dependency constraints to first-class objects. Technically, for a variable $X : A_1 \to \ldots \to A_n \to a$, he introduces a notation $X = X[I]$ for $I \subseteq \{1 \ldots n\}$ meaning that X can be instantiated by a closed term that *must* depend on the i-th argument, $i \in I$. Those constraints are further generalized to allow I to be a sort of algebraic set expression and to occur in the quantificational prefix of a formula. A *constrained formula* is obtained by turning the original formula into a disjunction (resp. conjunction) consisting of all the dependency constraints induced by every free or existential (resp. universal) variable. For example, the above problem (1) yields the constrained form:

$$\begin{array}{l}(Z = Z[\emptyset] \wedge \forall Y.Y = Y[\emptyset] \wedge \lambda x\,.\,Z\ x \neq_{\beta\eta} \lambda x\,.\,Y) \vee \\ (Z = Z[\{1\}] \wedge \forall Y.Y = Y[\emptyset] \wedge \lambda x\,.\,Z\ x \neq_{\beta\eta} \lambda x\,.\,Y)\end{array}$$

which eventually transforms into the solved form $Z = Z[\{1\}] \wedge \top$. The technical

handling of those formulae is rather awkward as they require specialized rules which are, we feel, foreign to the main issue.

In [4], while tackling a related (though simpler) problem, namely the relative complement problem over higher-order patterns, we have employed a different, and, we maintain, neater method. To begin with, we distinguish the case when a pattern is *fully applied*, that is each free variable is applied to *every* bound variable mentioned in the lambda binder. With this restriction, which, by the way, plays an important role in functional logic programming and rewriting [2], the complement of a *linear* pattern, i. e. with no repeated occurrence of the same free variable is a straightforward extension of the first-order case. This applies to disunification as well: indeed, it is fairly immediate to provide a complete set of rules for this fragment, which generalize the first-order ones [1] and dependency constraints nor linearity are needed at all.

For the general case, the formulation of the problem suggests that we should enrich the λ-calculus with an internal notion of *strictness* at the level of types, so that we can directly express that a term must depend on a given variable. For reasons of symmetry we also add the dual concept of *invariance*, expressing that a given term does *not* depend on a given variable. Thus, we internalize dependency constraints into the type theory. Indeed, we generalize our language to include *strict functions* of type $A \xrightarrow{1} B$ (which are guaranteed to depend on their argument), *invariant functions* of type $A \xrightarrow{0} B$ (which are guaranteed *not* to depend on their argument) and the full function space $A \xrightarrow{u} B$. This yields the following language, where the different kind of abstraction and application are denoted by a post-fix label:

$$\begin{array}{llll} \textit{Labels} & k & ::= & 1 \mid 0 \mid u \\ \textit{Types} & A & ::= & a \mid A_1 \xrightarrow{k} A_2 \\ \textit{Terms} & M & ::= & c \mid x \mid \lambda x^k{:}A.\ M \mid (M_1\, M_2)^k \end{array}$$

The typing judgment uses a three-zoned calculus $\Gamma;\Omega;\Delta \vdash M : A$, which, logically speaking, yields a variant of relevance logic. It has nice meta-theoretical properties culminating in the existence of canonical forms (as proven in [4]).

Thus our formulation of problem (1) is to make the partially applied pattern $\lambda x\,.\,Y$ fully applied by inserting a vacuous application and then applying the appropriate rules, i.e.

$$\begin{array}{l} \forall(Y'{:}A \xrightarrow{0} a).\lambda x^u{:}A.\ Z\ x^u \neq_{\beta\eta} \lambda x^u{:}A.\ Y'\ x^0 \\ \quad \Longrightarrow^* \exists(X{:}A \xrightarrow{1} a).\lambda x^u{:}A.\ Z\ x^u =_{\beta\eta} \lambda x^u{:}A.\ X\ x^1 \end{array}$$

More in general, patterns in the simply-typed λ-calculus are embedded into the strict one by viewing constants (and bound variables) as strict functions, while higher order functions are kept undetermined. Moreover, partially applied patterns involved in dis-equations are linearized introducing the obvious constraints.

For instance, using the running example in Chapter 11 of [5], a (object-level) η-redex $(x).\ y\ x$ is encoded in the simply typed calculus as $abs(\lambda x :$

$term.\,(app\ Y)\ x)$, where the proviso $x \notin FV(y)$ is realized by using a a variable Y of type $Term$ and a partially applied pattern. It is then brought to full application similarly to the above and embedded as $abs(\lambda x^u\!:\!term.\,app\ (Y'\ x^0)^1\ x^1)^1$. The solution of the disunification problem

$$\forall(Y'\!:\!term \xrightarrow{0} term).\lambda x^u\!:\!term.\,Z\ x^u \neq_{\beta\eta} abs(\lambda x^u\!:\!term.\,app\ (Y'\ x^0)^1\ x^1)^1$$

via one branch of the Explosion rule includes

$$\exists(X_1\!:\!term \xrightarrow{1} term), (X_2\!:\!term \xrightarrow{u} term).$$
$$\lambda x^u\!:\!term.\,Z\ x^u =_{\beta\eta} abs(\lambda x^u\!:\!term.\,app\ (X_1\ x^1)^1\ (X_2\ x^u)^1)^1$$

Having internalized the dependencies constraints, the disunification procedure proceeds with relative ease through the usual procedure of elimination of universal quantifiers and transformation of existentially quantified formulae into solved form, namely *basic* formulae in the terminology of [1]. This is based on the adaptation of the rules of pattern unification over strict terms which have been shown to be finitary in [4]. Termination is shown along the lines of [3].

References

[1] H. Comon. Disunification: a survey. In J.-L. Lassez and G.Plotkin, editors, *Computational Logic*. MIT Press, Cambridge, MA, 1991.

[2] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.

[3] D. Lugiez. Positive and negative results for higher-order disunification. *Journal of Symbolic Computation*, 20(4):431–470, Oct. 1995.

[4] A. Momigliano and F. Pfenning. Higher-order pattern complement and the strict λ-calculus. *ACM Transactions on Computational Logic*, To appear.

[5] F. van Raamsdonk. *Term Rewriting Systems*, chapter Higher-Order Rewriting. Cambridge University Press, To appear.

On Proving Termination of Higher-Order Rewrite Systems by Dependency Pair Technique

Masahiko Sakai[1] and Keiichirou Kusakari[2]

[1] Department of Information Engineering, Nagoya University,
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
sakai@nuie.nagoya-u.ac.jp
[2] Research Institute of Electrical Communication, Tohoku University,
Katahira, Aoba-ku, Sendai, 980-8577, Japan
kusakari@nue.riec.tohoku.ac.jp

We will discuss the dependency pair (DP) technique for proving termination of Nipkow's higher-order rewrite systems (HRS). The DP technique is effective on term rewriting systems, because it gives us a mechanical support for proving non-simple termination. Sakai, Watanabe and Sakabe tried to extend the DP method to the higher-order case[1]. Since it requires a reduction quasi-ordering having the subterm property, the argument filtering method is not applicable any more that weakens their approach extremely. After that, we proposed another DP method[2] on HRS by introducing a notion of dependency forest and show that the termination property of a higher-order rewrite system R can be checked by the non-existence of an infinite R-chain, if R is non-duplicating or non-nested. Although no longer the subterm property is needed hence the argument filtering is applicable, the strong restriction, non-duplicating or non-nested, is required. We explain this method and discuss the possibility to weaken the restriction.

The key point of the DP technique is producing R' by adding rules, called DPs, to the given TRS R in order to transform infinite reduction sequence of R to an infinite reduction sequence of R' having infinite head-reductions, called DP-chain. Thus, proving termination is reduced to showing that no infinite DP-chain exists. In the transformation of the sequence, we take an appropriate subterm of each term in the original sequence. Generally, the head symbols in DP-chain are introduced by right-hand sides in R. Fortunately, it is easy to define DPs in first-order case, because each additional rules are obtained by replacing the right-hand side of a original rule with its subterms headed by defined symbol. In higher-order case, it is difficult to define DPs having such a good property, because the head symbols in DP-chain may be carried or even copied as instances of higher-order variables after their introduction. We introduce the notion of dependency forest in order to analyze and to trace dependencies between the introduction and the consumption of such symbols.

The following example shows why the restriction non-duplication or non-nested is required. Consider an HRS R;

$$R = \begin{cases} i(X) \to g(X, X), \\ g(h(\lambda x.F(x)), X) \to F(X) \end{cases}$$

Although we have only one dependency pair $\langle i^{\#}(X), g^{\#}(X, X)\rangle$, the following infinite reduction sequence exists; $i(h(\lambda x.i(x))) \to g(h(\lambda x.i(x)), h(\lambda x.i(x))) \to i(h(\lambda x.i(x))) \to \cdots$.

[1] Masahiko Sakai, Yoshitsugu Watanabe, Toshiki Sakabe, An Extension of Dependency Pair Method for Proving Termination of Higher-Order Rewrite Systems, IEICE Trans. on Information and Systems, Vol.E84-D, No.8, pp.1025–1032, 2001.
[2] Masahiko Sakai, Keiichirou Kusakari, On New Dependency Pair Method for Proving Termination of Higher-Order Rewrite Systems, The International Workshop on Rewriting in Proof and Computation (RPC'01), Sendai, Japan, October 25–27, pp.176–187, 2001.