HOR 2012

6th International Workshop on Higher-Order Rewriting

Makoto Hamana (ed.)

RTA'12 Affiliated Workshop
Nagoya, Japan
June 2, 2012

Preface

This is the proceedings volume of the 6th International Workshop on Higher- Order Rewriting (HOR 2012), which was held on June 2, 2012, in Nagoya, Japan.

The aim of HOR is to provide an informal forum to discuss all aspects of higher-order rewriting. The topics of the workshop include applications, foundations, frameworks, implementations, and semantics.

HOR is a biannual meeting. HOR 2002 was a part of FLoC 2002 in Copenhagen, Denmark. HOR 2004 was part of the RDP 2004 in Aachen, Germany. HOR 2006 was part of FLoC 2006 in Seattle, USA. HOR 2007 was part of RDP 2007 in Paris, France. HOR 2010 was part of FloC 2010 in Edinburgh, UK. HOR 2012 is an RTA 2012 workshop in Nagoya, Japan.

The present volume provides final versions of six accepted contributed extended abstracts of talks selected for the workshop. HOR 2012 had also a tool session. The following termination checkers for higher-order rewriting systems was presented:

- Carsten Fuhs: Haskell termination tool
- Rene Thiemann: Isabelle termination tool
- Takahito Aoto and Toshiyuki Yamada: Simply-typed TRS termination tool
- Cynthia Kop: WANDA, termination tool for Algebraic Functional Systems

We are grateful to these authors for accepting to present thier tools at HOR. We are also very grateful to Zhenjiang Hu (National Institute of Informatics, Japan) for kindly accepting to give an invited talk at HOR 2012.

Many thanks to the program committee that I had the pleasure to chair, consisting of Andreas Abel, Frederic Blanqui, Stefan Kahrs and Fer-Jan de Vries. The tasks were the decisions on the invited speakers, a timely refereeing of the submitted abstracts and the final decisions on the program. A heartfelt thank you to the HOR workshop series steering committee, Delia Kesner and Femke van Raamsdonk, who offered me to take this responsibility for the higher-order rewriting community and gave valuable advice throughout. Finally, we would like to thank the organizing committee of RTA 2012, and in particular Naoki Nishida, for all help in the preparation of the workshop.

June 2012
Makoto Hamana (Gunma University, Japan)

Contents

Can Graph Transformation be Bidirectionalized? Bidirectional Semantics of Structural Recursion on Graphs

Zhenjiang Hu
National Institute of Informatics, Japan
hu@nii.ac.jp

Bidirectional transformations provide a novel mechanism for synchronizing and maintaining the consistency of information between input and output. Despite many promising results on bidirectional transformations, these have been limited to lists and trees. We challenge the problem of bidirectional transformations on graphs, by proposing a formal definition of a well-behaved bidirectional semantics for UnCAL, a graph algebra. Our key idea is to treat graphs as compactable infinite trees and to manipulate trees by structural recursion. Specifically, we carefully refine the existing forward evaluation of structural recursion so that it can produce sufficient trace information for later backward evaluation, and we use the trace information for backward evaluation to reflect updates on the view to the source. We prove our bidirectional evaluation is well-behaved.

Transformation by Templates for Simply-Typed Term Rewriting

Yuki Chiba
School of Information Science, Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa, 923-1292, Japan
chiba@jaist.ac.jp

Takahito Aoto
RIEC, Tohoku University
2-1-1 Katahira, Aoba-ku, Sendai, Miyagi, 980-8577, Japan
aoto@nue.riec.tohoku.ac.jp

We extend a framework of program transformation by templates based on first order term rewriting (Chiba et al., 2005) to simply typed term rewriting (Yamada, 2001), which is a framework of higher order term rewriting. A pattern matching algorithm to apply templates for transforming a simply typed term rewriting system is given and the correctness of the algorithm is shown.

1 Program transformation by templates and simply typed term pattern

Huet and Lang [4] introduced a framework of program transformation by templates. In this framework, the second-order matching algorithm plays an important role—how to apply the transformation template to a given program is specified by the solution of the matching algorithm. Curien et al. [3] gave an efficient matching algorithm. Yokoyama et al. [7] presented a sufficient condition of patterns to have at most one solution. De Moor and Sittampalam [5] gave a matching algorithm containing third-order matching. In all of these frameworks, programs and program schemas are represented by lambda terms and higher-order substitutions are performed by β-reduction.

In Chiba et al. [1, 2], we introduced a framework of program transformation by templates *based on term rewriting*. Contrast to the framework mentioned above, programs and program schemas are given by term rewriting systems (TRSs for short) and TRS patterns, where TRS patterns is a TRS in which pattern variables are used in the place of function symbols. For example, a program transformation template $\langle \mathscr{P}, \mathscr{P}', \mathscr{H} \rangle$ is given like this:

$$\mathscr{P} = \left\{ \begin{array}{ll} \mathsf{f}(\mathsf{a}) & \to \mathsf{b} \\ \mathsf{f}(\mathsf{c}(u_1, v_1)) & \to \mathsf{g}(u_1, \mathsf{f}(v_1)) \\ \mathsf{g}(\mathsf{b}, u_2) & \to u_2 \\ \mathsf{g}(\mathsf{d}(u_3, v_3), w_3) & \to \mathsf{d}(u_3, \mathsf{g}(v_3, w_3)) \end{array} \right\}, \mathscr{P}' = \left\{ \begin{array}{ll} \mathsf{f}(u_4) & \to \mathsf{f}_1(u_4, \mathsf{b}) \\ \mathsf{f}_1(\mathsf{a}, u_5) & \to u_5 \\ \mathsf{f}_1(\mathsf{c}(u_6, v_6), w_6) & \to \mathsf{f}_1(v_6, \mathsf{g}(w_6, u_6)) \\ \mathsf{g}(\mathsf{b}, u_7) & \to u_7 \\ \mathsf{g}(\mathsf{d}(u_8, v_8), w_8) & \to \mathsf{d}(u_8, \mathsf{g}(v_8, w_8)) \end{array} \right\},$$

$$\mathscr{H} = \{ \ \mathsf{g}(\mathsf{b}, u_1) \approx \mathsf{g}(u_1, \mathsf{b}), \ \ \mathsf{g}(\mathsf{g}(u_2, v_2), w_2) \approx \mathsf{g}(u_2, \mathsf{g}(v_2, w_2)) \ \}.$$

The TRS pattern $\mathscr{P}$ is a schema of input programs to be transformed and $\mathscr{P}'$ is a schema whose instantiations become the output programs. This template is used for a program transformation from recursive programs to iterative programs. For example, to transform the following TRS

$$\mathscr{R}_{\mathsf{sum}} = \left\{ \begin{array}{lllllll} \mathsf{sum}([]) & \to & 0, & \quad \mathsf{sum}(x_1{:}y_1) & \to & +(x_1, \mathsf{sum}(y_1)) \\ +(0, x_2) & \to & x_2, & \quad +(\mathsf{s}(x_3), y_3) & \to & \mathsf{s}(+(x_3, y_3)) \end{array} \right\}$$

Submitted to:
HOR 2012

we perform a pattern matching with $\mathscr{P}$ against $\mathscr{R}_{\mathsf{sum}}$. Then a *term homomorphism* φ satisfying $\mathscr{R}_{\mathsf{sum}} = \varphi(\mathscr{P})$ is obtained using a matching algorithm [1]. Now the iterative form of $\mathscr{R}_{\mathsf{sum}}$ is obtained as

$$\mathscr{R}'_{\mathsf{sum}} = \varphi(\mathscr{P}') = \left\{ \begin{array}{lllllll} \mathsf{sum}(x_4) & \to & \mathsf{sum1}(x_4,0) & & & & \\ \mathsf{sum1}([\,],x_5) & \to & x_5, & & \mathsf{sum1}(x_6{:}y_6,z_6) & \to & \mathsf{sum1}(y_6,+(z_6,x_6)), \\ +(0,x_7) & \to & x_7, & & +(\mathsf{s}(y_8),z_8) & \to & \mathsf{s}(+(y_8,z_8)) \end{array} \right\}.$$

The term homomorphism φ is also used to generate the following set $\mathscr{E}_{\mathsf{sum}} = \varphi(\mathscr{H})$ of equations.

$$\mathscr{E}_{\mathsf{sum}} = \{\ +(0,x_1) \approx +(x_1,0),\quad +(+(x_2,v_2),w_2) \approx +(x_2,+(v_2,w_2))\ \}$$

These equations are used at the verification of the correctness of transformation not only in the framework based on term rewriting but also the one based on lambda calculus. We showed that the correctness of transformation may be (partly) verified by proving properties of term rewriting systems for suitable templates [1, 2]. All recursive programs must be described as program schemes in the framework of lambda calculus in order to use fixed point combinator. In contrast, they can be defined by using case splitting in our framework. Because of universally quantified nature of variables in rewrite rules, second-order matching algorithms on lambda term are not directly applicable in our setting [1].

Because this framework is based on the first order term rewriting, it is difficult to deal with higher order programs. In this paper, we extend the framework of program transformation by templates on first order term rewriting to *simply typed term rewriting* [6], which is one of the simplest frameworks of higher order term rewriting. Our key idea is introducing two kinds of function application, one for function application on STTRSs and one for applying second order matching solution.

The first question to develop such a framework is which term language presenting program schemas to chose. We first introduce a set of *basic types* B and a set *type variables* U. The idea is that in program templates types are not fixed and type variables contained in templates are instantiated at the time of concrete transformations. We call simple types over $B \cup U$ *type pattern* and refer *types* those over B. Next we assume that each *constant* and *local variable* are associated with its type and type pattern, respectively and each *pattern variable* is associated with its argument type patterns and result type pattern. The set of pattern variables whose argument type patterns are $\tau_1,\ldots,\tau_n$ and result type patterns are τ is denoted as $\mathscr{X}^{\tau_1\times\cdots\times\tau_n\Rightarrow\tau}$.

Definition 1 (Term pattern) *The set* $\mathrm{T}^\tau(\Sigma,\mathscr{X},\mathscr{V})$ *of* term pattern of type pattern τ *over constants* $\Sigma = \bigcup_{\tau\in\mathrm{ST}(B)}\Sigma^\tau$*, pattern variable* $\mathscr{X} = \bigcup_{\tau,\tau_1,\ldots,\tau_n\in\mathrm{ST}(B,U)}\mathscr{X}^{\tau_1\times\cdots\tau_n\Rightarrow\tau}$ *and local variables* $\mathscr{V} = \bigcup_{\tau\in\mathrm{ST}(B,U)}\mathscr{V}^\tau$ *is defined as: (1)* $\Sigma^\tau\cup\mathscr{V}^\tau \subseteq \mathrm{T}^\tau(\Sigma,\mathscr{X},\mathscr{V})$*, (2)* $s\in\mathrm{T}^{\tau_1\times\cdots\times\tau_n\to\tau}(\Sigma,\mathscr{X},\mathscr{V})$ $(n\geq 1)$ *and* $t_i\in\mathrm{T}^{\tau_i}(\Sigma,\mathscr{X},\mathscr{V})$ *for all* $i\in\{1,\ldots,n\}$ *imply* $(s\ t_1\ \cdots\ t_n)\in\mathrm{T}^\tau(\Sigma,\mathscr{X},\mathscr{V})$*, and (3)* $p\in\mathscr{X}^{\tau_1\times\cdots\times\tau_n\Rightarrow\tau}$ $(n\geq 0)$ *and* $t_i\in\mathrm{T}^{\tau_i}(\Sigma,\mathscr{X},\mathscr{V})$ *for all* $i\in\{1,\ldots,n\}$ *imply* $p\langle t_1,\ldots,t_n\rangle\in\mathrm{T}^\tau(\Sigma,\mathscr{X},\mathscr{V})$*.*

Here note that two kinds of function application are introduced—one is $(s\ t_1\ \cdots\ t_n)$ for the function application on object language (simply typed terms) and the other is $p\langle t_1,\ldots,t_n\rangle$ for the function application to be used instantiating program templates to concrete programs. This is contrast to the second-order matching frameworks on lambda terms where these two kinds of function application are identified.

Let $B = \{\mathsf{Nat},\mathsf{List}\}$, $U = \{\alpha,\beta,\gamma,\delta,\varepsilon\}$, $\Sigma = \{\ [\,]^{\mathsf{List}}, {:}^{\mathsf{Nat}\times\mathsf{List}\to\mathsf{List}}, @^{\mathsf{List}\times\mathsf{List}\to\mathsf{List}}, \mathsf{map}^{(\mathsf{Nat}\to\mathsf{Nat})\times\mathsf{List}\to\mathsf{List}}, \mathsf{mapapp}^{(\mathsf{Nat}\to\mathsf{Nat})\times\mathsf{List}\times\mathsf{List}\ \to\mathsf{List}}\ \}$, and $\mathscr{X} = \{\mathsf{a}^{\Rightarrow\alpha}, \mathsf{b}^{\gamma\Rightarrow\delta}, \mathsf{c}^{\varepsilon\times\alpha\Rightarrow\alpha}, \mathsf{d}^{\varepsilon\Rightarrow\varepsilon}, \mathsf{e}^{\varepsilon\times\gamma\times\delta\Rightarrow\delta}, \mathsf{f}^{\alpha\times\beta\times\gamma\Rightarrow\delta},$

$$\textbf{Constant}\quad [A \trianglelefteq A, \sigma] \vdash \sigma \qquad \textbf{Var1}\quad [\alpha \trianglelefteq \tau', \sigma] \vdash \sigma \quad \text{if } \sigma(\alpha) = \tau'$$

$$\textbf{Var2}\quad [\alpha \trianglelefteq \tau', \sigma] \vdash \{\alpha := \tau'\} \circ \sigma \quad \text{if } \alpha \notin \mathrm{dom}(\sigma)$$

$$\textbf{Function}\quad \frac{[\tau_1 \trianglelefteq \tau_1', \sigma] \vdash \sigma' \quad [\tau_2 \times \cdots \times \tau_n \to \tau \trianglelefteq \tau_2' \times \cdots \times \tau_n' \to \tau', \sigma'] \vdash \sigma''}{[\tau_1 \times \tau_2 \times \cdots \times \tau_n \to \tau \trianglelefteq \tau_1' \times \tau_2' \times \cdots \times \tau_n' \to \tau', \sigma] \vdash \sigma''}$$

Figure 1: Type matching rules

$\mathsf{g}^{\alpha\times\gamma\Rightarrow\delta}, \mathsf{h}^{\alpha\times\beta\Rightarrow\alpha}, \mathsf{r}^{\beta\Rightarrow\alpha}\}$. An example of program transformation template is like this:

$$\mathscr{P} = \left\{\begin{array}{lll} \mathsf{f}\langle u,v,w\rangle & \to & \mathsf{g}\langle \mathsf{h}\langle u,v\rangle, w\rangle \\ \mathsf{g}\langle \mathsf{a}\langle\rangle, u\rangle & \to & \mathsf{b}\langle u\rangle \\ \mathsf{g}\langle \mathsf{c}\langle u,v\rangle, w\rangle & \to & \mathsf{e}\langle u,w,\mathsf{g}\langle v,w\rangle\rangle \\ \mathsf{h}\langle \mathsf{a}\langle\rangle, u\rangle & \to & \mathsf{r}\langle u\rangle \\ \mathsf{h}\langle \mathsf{c}\langle u,v\rangle, w\rangle & \to & \mathsf{c}\langle \mathsf{d}\langle u\rangle, \mathsf{h}\langle v,w\rangle\rangle \end{array}\right\}, \mathscr{P}' = \left\{\begin{array}{lll} \mathsf{f}\langle \mathsf{a}\langle\rangle, v, w\rangle & \to & \mathsf{g}\langle \mathsf{r}\langle v\rangle, w\rangle \\ \mathsf{f}\langle \mathsf{c}\langle u,v\rangle, w, z\rangle & \to & \mathsf{e}\langle \mathsf{d}\langle u\rangle, z, \mathsf{f}\langle v,w,z\rangle\rangle \\ \mathsf{g}\langle \mathsf{a}\langle\rangle, u\rangle & \to & \mathsf{b}\langle u\rangle \\ \mathsf{g}\langle \mathsf{c}\langle u,v\rangle, w\rangle & \to & \mathsf{e}\langle u,w,\mathsf{g}\langle v,w\rangle\rangle \\ \mathsf{h}\langle \mathsf{a}\langle\rangle, u\rangle & \to & \mathsf{r}\langle u\rangle \\ \mathsf{h}\langle \mathsf{c}\langle u,v\rangle, w\rangle & \to & \mathsf{c}\langle \mathsf{d}\langle u\rangle, \mathsf{h}\langle v,w\rangle\rangle \end{array}\right\}, \mathscr{H} = \emptyset$$

This transformation template can be used to transform simply typed term rewriting systems (STTRSs):

$$\left\{\begin{array}{lll} \mathsf{mapapp}\ f\ xs\ ys & \to & \mathsf{map}\ f\ (@\ xs\ ys) \\ \mathsf{map}\ f\ [\,] & \to & [\,] \\ \mathsf{map}\ f\ (:\ x\ xs) & \to & :\ (f\ x)\ (\mathsf{map}\ f\ xs) \\ @\ [\,]\ ys & \to & ys \\ @\ (:\ x\ xs)\ ys & \to & :\ x\ (@\ xs\ ys) \end{array}\right\} \Rightarrow \left\{\begin{array}{lll} \mathsf{mapapp}\ f\ [\,]\ ys & \to & \mathsf{map}\ f\ ys \\ \mathsf{mapapp}\ f\ (:\ x\ xs)\ ys & \to & :\ (f\ x)\ (\mathsf{mapapp}\ f\ xs\ ys) \\ \mathsf{map}\ f\ [\,] & \to & [\,] \\ \mathsf{map}\ f\ (:\ x\ xs) & \to & :\ (f\ x)\ (\mathsf{map}\ f\ xs) \\ @\ [\,]\ ys & \to & ys \\ @\ (:\ x\ xs)\ ys & \to & :\ x\ (@\ xs\ ys) \end{array}\right\}.$$

One may wonder why there exist common rules in input and output which seem unnecessary for specifying the transformation. They, however, are prepared to guarantee the correctness of the transformation. We note that transformations can be applied even if additional rules (other than target rules) are involved.

In the rest of the paper, we show how such a transformation by templates on STTRSs can be done.

2 Pattern matching algorithm

In this section, we present our pattern matching algorithm. Since term patterns possibly contains type variables, type matching is performed during the pattern matching algorithm on the fly to obtain type consistent term homomorphisms. Type matching can be done as in the first order matching except that it has to be checked incrementally. A *type substitution* is a mapping from type variables to type patterns.

Definition 2 (Type matching) *Let τ be a type pattern, τ' a type and σ and σ' type substitutions. We write $[\tau \trianglelefteq \tau', \sigma] \vdash \sigma'$ if there is an inference by applying the rules listed in Figure 1.*

A *context* is a term containing *holes* $\Box_i$. $\mathrm{C}_n(\Sigma)$ is the set of contexts containing only holes $\Box_1, \ldots, \Box_n$. $C\langle s_1, \ldots, s_n\rangle$ is the result of a context $C \in \mathrm{C}_n(\Sigma)$ replacing $\Box_i$ by s_i. We write $C \in \mathrm{C}^{\tau_1\times\cdots\times\tau_n\Rightarrow\tau}(\Sigma)$ if $\Box_i^{\tau_i}$ for all $\Box_i \in C$ and C^τ. A *term homomorphism* is a mapping φ from $\mathscr{V} \cup \mathscr{X}$ to $\mathscr{V} \cup \mathrm{C}(\Sigma)$ satisfying: (1) the restriction of φ on $\mathscr{V}$ is an injective mapping from $\mathrm{dom}_{\mathscr{V}}(\varphi) = \{x \in \mathscr{V} \mid \varphi(x) \neq x\}$ to $\mathscr{V}$ and (2) $\varphi(p) \in \mathrm{C}_n(\Sigma)$ for any $p \in \mathscr{X}$, where $n = \mathrm{arity}(p)$. We put $\mathrm{dom}_{\mathscr{X}}(\varphi) = \{p \in \mathscr{X} \mid \varphi(p) \neq p\langle \Box_1, \ldots, \Box_{\mathrm{arity}(p)}\rangle\}$.

$$\textbf{Bound}\qquad \frac{\langle S \cup \{x^{\tau} \trianglelefteq y^{\tau'}\}, \sigma, \varphi\rangle}{\langle S, \sigma', \varphi \cup \{x \mapsto y\}\rangle}\ \text{ if }\ \begin{array}{l}\varphi(x) = y \wedge \sigma' = \sigma, \text{ or}\\ x \notin \mathrm{dom}(\varphi) \wedge y \notin \mathrm{range}(\varphi) \wedge [\tau \trianglelefteq \tau', \sigma] \vdash \sigma'\end{array}$$

$$\textbf{Remove}\qquad \frac{\langle S \cup \{f^{\tau} \trianglelefteq f^{\tau}\}, \sigma, \varphi\rangle}{\langle S, \sigma, \varphi\rangle}$$

$$\textbf{Split}\qquad \frac{\langle S \cup \{(s_1\ \cdots\ s_n)^{\tau} \trianglelefteq (t_1\ \cdots\ t_n)^{\tau'}\}, \sigma, \varphi\rangle}{\langle S \cup \{s_i \trianglelefteq t_i \mid 1 \le i \le n\}, \sigma', \varphi\rangle}\ \text{ if } [\tau \trianglelefteq \tau', \sigma] \vdash \sigma'$$

$$\textbf{Extend}\qquad \frac{\langle S \cup \{p\langle s_1, \ldots, s_n\rangle^{\tau} \trianglelefteq C\langle t_1, \ldots, t_n\rangle^{\tau'}\}, \sigma, \varphi\rangle}{\langle \{p \mapsto C\}(S \cup \{s_i \trianglelefteq t_i \mid \Box_i \in C\}), \sigma', \varphi \cup \{p \mapsto C\}\rangle}\ \text{ if } [\tau \trianglelefteq \tau', \sigma] \vdash \sigma'$$

Figure 2: Rules of pattern matching algorithm

$$\begin{array}{l}
\quad\langle\{\mathsf{g}\langle \mathsf{c}\langle u,v\rangle, w\rangle \trianglelefteq \mathsf{map}\ f\ (:\ x\ xs),\ \mathsf{e}\langle u,w,\mathsf{g}\langle v,w\rangle\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \emptyset, \emptyset\rangle\\
\Longrightarrow \langle\{w \trianglelefteq f, \mathsf{c}\langle u,v\rangle \trianglelefteq :\ x\ xs,\ \mathsf{e}\langle u,w,(\mathsf{map}\ w\ v)\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \{\delta := \mathsf{List}\}, \{\mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\}\rangle\\
\Longrightarrow \langle\{\mathsf{c}\langle u,v\rangle \trianglelefteq :\ x\ xs,\ \mathsf{e}\langle u,w,(\mathsf{map}\ w\ v)\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \left\{\begin{array}{l}\delta := \mathsf{List},\\ \gamma := \mathsf{Nat} \to \mathsf{Nat}\end{array}\right\}, \left\{\begin{array}{l}w \mapsto f,\\ \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\end{array}\right\}\rangle\\
\Longrightarrow \langle\{u \trianglelefteq x, v \trianglelefteq xs, \mathsf{e}\langle u,w,(\mathsf{map}\ w\ v)\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \left\{\begin{array}{l}\delta := \mathsf{List},\\ \gamma := \mathsf{Nat} \to \mathsf{Nat}\\ \alpha := \mathsf{List}\end{array}\right\}, \left\{\begin{array}{l}w \mapsto f,\\ \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\\ \mathsf{c} \mapsto :\ \Box_1\ \Box_2\end{array}\right\}\rangle\\
\Longrightarrow \langle\{v \trianglelefteq xs, \mathsf{e}\langle u,w,(\mathsf{map}\ w\ v)\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \left\{\begin{array}{ll}\delta := \mathsf{List}, & \gamma := \mathsf{Nat} \to \mathsf{Nat}\\ \alpha := \mathsf{List}, & \varepsilon := \mathsf{Nat}\end{array}\right\}, \left\{\begin{array}{ll}w \mapsto f, & \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\\ \mathsf{c} \mapsto :\ \Box_1\ \Box_2, & u \mapsto x\end{array}\right\}\rangle\\
\Longrightarrow \langle\{\mathsf{e}\langle u,w,(\mathsf{map}\ w\ v)\rangle \trianglelefteq :\ (f\ x)\ (\mathsf{map}\ f\ xs)\}, \left\{\begin{array}{ll}\delta := \mathsf{List}, & \gamma := \mathsf{Nat} \to \mathsf{Nat}\\ \alpha := \mathsf{List}, & \varepsilon := \mathsf{Nat}\end{array}\right\}, \left\{\begin{array}{ll}w \mapsto f, & \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\\ \mathsf{c} \mapsto :\ \Box_1\ \Box_2, & u \mapsto x\\ v \mapsto xs & \end{array}\right\}\rangle\\
\Longrightarrow \langle\{u \trianglelefteq x, w \trianglelefteq f, \mathsf{map}\ w\ v \trianglelefteq \mathsf{map}\ f\ xs\}, \left\{\begin{array}{ll}\delta := \mathsf{List}, & \gamma := \mathsf{Nat} \to \mathsf{Nat}\\ \alpha := \mathsf{List}, & \varepsilon := \mathsf{Nat}\end{array}\right\}, \left\{\begin{array}{ll}w \mapsto f, & \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1\\ \mathsf{c} \mapsto :\ \Box_1\ \Box_2, & u \mapsto x\\ v \mapsto xs, & \mathsf{e} \mapsto :\ (\Box_2\ \Box_1)\ \Box_3\end{array}\right\}\rangle\\
\overset{*}{\Longrightarrow} \langle\emptyset, \left\{\begin{array}{ll}\delta := \mathsf{List}, & \gamma := \mathsf{Nat} \to \mathsf{Nat}\\ \alpha := \mathsf{List}, & \varepsilon := \mathsf{Nat}\end{array}\right\}, \left\{\begin{array}{lll}w \mapsto f, & \mathsf{g} \mapsto \mathsf{map}\ \Box_2\ \Box_1, & \mathsf{c} \mapsto :\ \Box_1\ \Box_2\\ u \mapsto x, & v \mapsto xs, & \mathsf{e} \mapsto :\ (\Box_2\ \Box_1)\ \Box_3\end{array}\right\}\rangle
\end{array}$$

Figure 3: Derivation of **ST-Match**

φ is *consistent with* a type substitution σ if (1) $\varphi(x) \in \mathscr{V}^{\sigma(\tau)}$ for any $x \in \mathscr{V}^{\tau} \cap \mathrm{dom}_{\mathscr{V}}(\varphi)$ and (2) $\varphi(p) \in \mathrm{C}^{\sigma(\tau_1) \times \cdots \times \sigma(\tau_n) \Rightarrow \sigma(\tau)}(\Sigma)$ for any $p \in \mathscr{X}^{\tau_1 \times \cdots \times \tau_n \Rightarrow \tau} \cap \mathrm{dom}_{\mathscr{X}}(\varphi)$.

A *matching pair* $s \trianglelefteq t$ is a pair of a term pattern s and a term t, and a *matching problem* is a finite set of matching pairs. Given a matching problem S, our pattern matching algorithm solves whether there exist term homomorphism φ and type substitution σ such that φ is consistent with σ, and $\varphi(s) = t$ holds for any $s \trianglelefteq t \in S$. Our algorithm is given by inference rules acting on a *configuration* $\langle S, \sigma, \varphi\rangle$.

Definition 3 (ST-Match) *Let $\Longrightarrow$ be a relation on configurations defined by: $\langle S, \sigma, \varphi\rangle \Longrightarrow \langle S', \sigma', \varphi'\rangle$ if $\langle S, \sigma, \varphi\rangle$ is rewritten to $\langle S', \sigma', \varphi'\rangle$ by applying the rules listed in Figure 2. The reflexive transitive closure of $\Longrightarrow$ is denoted by $\overset{*}{\Longrightarrow}$. The procedure **ST-Match** is given as follows*

Input: *a matching problem S*

Output: *a pair $\langle \varphi, \sigma\rangle$ of a term homomorphism φ and a type substitution σ if $\langle S, \emptyset, \emptyset\rangle \overset{*}{\Longrightarrow} \langle \emptyset, \sigma, \varphi\rangle$.*

Note that **Extend** rule select an appropriate context *non-deterministically*—our algorithm intends conciseness but not to describe how it can be implemented efficiently. In Figure 3, we present a derivation of **ST-Match** for a pattern matching problem involved in the program transformation in Section 1.

Theorem 4 (Termination of ST-Match) ***ST-Match** terminates for any input.*

$$\langle\varphi,\sigma\rangle \vdash x^{\tau} \unlhd y^{\tau'} \quad \text{if } \varphi(x)=y \wedge \sigma(\tau)=\tau' \qquad \frac{\langle\varphi,\sigma\rangle \vdash s_1 \unlhd t_1 \quad \cdots \quad \langle\varphi,\sigma\rangle \vdash s_n \unlhd t_n}{\langle\varphi,\sigma\rangle \vdash (s_1 \ \cdots \ s_n)^{\tau} \unlhd (t_1 \ \cdots \ t_n)^{\tau'}} \ \text{if } \sigma(\tau)=\tau'$$

$$\langle\varphi,\sigma\rangle \vdash c^{\tau} \unlhd c^{\tau} \qquad \frac{\langle\varphi,\sigma\rangle \vdash s_{i_1} \unlhd t_{i_1} \quad \cdots \quad \langle\varphi,\sigma\rangle \vdash s_{i_m} \unlhd t_{i_m}}{\langle\varphi,\sigma\rangle \vdash p\langle s_1,\ldots,s_n\rangle^{\tau} \unlhd C\langle t_1,\ldots,t_n\rangle^{\tau'}} \ \text{if} \begin{cases} \{i_1,\ldots,i_m\}=\{i \mid \Box_i \in C\}, \\ \sigma(\tau)=\tau', \text{ and } \varphi(p)=C \end{cases}$$

Figure 4: Inference rules for checking solutions of configurations

3 Correctness of pattern matching algorithm

In this section, we show soundness and completeness of **ST-Match** in order to ensure the correctness of the algorithm. We say a pair $\langle\varphi,\sigma\rangle$ of a term homomorphism φ and a type substitution σ is a *solution* of a matching problem S if (1) φ is consistent with σ and (2) $\varphi(s)=t$ for all $s \unlhd t \in S$.

Definition 5 (Solution of configuration) *(a) We write $\langle\varphi,\sigma\rangle \vdash s \unlhd t$ if there is an inference tree by applying the rules in Figure 4.*

(b) A pair $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ of a term homomorphism $\tilde{\varphi}$ and a type substitution $\tilde{\sigma}$ is a solution *of a configuration $\langle S,\sigma,\varphi\rangle$ if (1) $\langle\varphi,\sigma\rangle \vdash s \unlhd t$ for all $s \unlhd t \in S$, (2) $\sigma \subseteq \tilde{\sigma}$, and (3) $\varphi \subseteq \tilde{\varphi}$.*

For ensuring type consistency of outputs of **ST-Match**, we consider the followings.

Lemma 6 (1) Suppose $\langle S,\sigma,\varphi\rangle \Longrightarrow \langle S',\sigma',\varphi'\rangle$. If $x \in \mathscr{V}^{\tau}$ implies $\varphi(x) \in \mathscr{V}^{\sigma(\tau)}$ for any $x \in \mathrm{dom}_{\mathscr{V}}(\varphi)$, then $x \in \mathscr{V}^{\tau}$ implies $\varphi'(x) \in \mathscr{V}^{\sigma(\tau)}$ for any $x \in \mathrm{dom}_{\mathscr{V}}(\varphi')$. (2) Suppose $\langle\tilde{\varphi},\tilde{\sigma}\rangle \vdash s \unlhd t$. $p \in \mathscr{X}^{\tau_1\times\cdots\times\tau_n\Rightarrow\tau}$ implies $\tilde{\varphi}(p) \in \mathrm{C}^{\tilde{\sigma}(\tau_1)\times\cdots\times\tilde{\sigma}(\tau_n)\Rightarrow\tilde{\sigma}(\tau)}(\Sigma)$ for any $p \in \mathscr{X}(s) \cap \mathrm{dom}_{\mathscr{X}}(\tilde{\varphi})$.

Next lemma can be shown by using straightforward case splitting of applied rules.

Lemma 7 *Suppose $\langle S,\sigma,\varphi\rangle \Longrightarrow \langle S',\sigma',\varphi'\rangle$ and $x \in \mathscr{V}^{\tau}$ implies $\varphi(x) \in \mathscr{V}^{\sigma(\tau)}$ for any $x \in \mathrm{dom}_{\mathscr{V}}(\varphi)$. If $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ is a solution of $\langle S',\sigma',\varphi'\rangle$, then $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ is also a solution of $\langle S,\sigma,\varphi\rangle$.*

From lemmas above, we obtain the following result of soundness.

Theorem 8 (Soundness of ST-Match) *If **ST-Match** produces $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ for an input S, then $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ is a solution of S.*

In order to show completeness of **ST-Match**, we show the following lemmas.

Lemma 9 *Let $\langle S,\sigma,\varphi\rangle$ be a configuration with $S \neq \emptyset$. If $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ is a solution of $\langle S,\sigma,\varphi\rangle$, then there exists a solution $\langle S',\sigma',\varphi'\rangle$ such that (1) $\langle S,\sigma,\varphi\rangle \Longrightarrow \langle S',\sigma',\varphi'\rangle$ and (2) $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ is a solution of $\langle S',\sigma',\varphi'\rangle$.*

Lemma 10 *Let $\langle\tilde{\varphi},\tilde{\sigma}\rangle$ be a solution of a configuration $\langle S,\sigma,\varphi\rangle$. If there exist a term homomorphism φ' and a type substitution σ' such that $\langle S,\sigma,\varphi\rangle \Longrightarrow \langle\emptyset,\sigma',\varphi'\rangle$, then (1) $\sigma' \subseteq \tilde{\sigma}$, and (2) $\varphi' \subseteq \tilde{\varphi}$.*

We now obtain the following theorem from lemmas above.

Theorem 11 (Completeness of ST-Match) *Let Φ be the set of outputs of **ST-Match** for input matching problem S. If $\langle\varphi,\sigma\rangle$ is a solution of S, then there exists $\langle\tilde{\varphi},\tilde{\sigma}\rangle \in \Phi$ such that $\tilde{\varphi} \subseteq \varphi$ and $\tilde{\sigma} \subseteq \sigma$.*

References

[1] Y. Chiba, T. Aoto & Y. Toyama (2005): *Program transformation by templates based on term rewriting*. In: *Proc. of PPDP 2005*, ACM Press, pp. 59–69.

[2] Y. Chiba, T. Aoto & Y. Toyama (2010): *Program transformation templates for tupling based on term rewriting*. *IEICE Transactions* 93-D(5), pp. 963–973.

[3] R. Curien, Z. Qian & H. Shi (1996): *Efficient second-order matching*. In: *Proc. of RTA 1996*, *LNCS* 1103, Springer-Verlag, pp. 317–331.

[4] G. Huet & B. Lang (1978): *Proving and applying program transformations expressed with second order patterns*. *Acta Informatica* 11, pp. 31–55.

[5] O. de Moor & G. Sittampalam (2001): *Higher-order matching for program transformation*. *TCS* 269(1–2), pp. 135–162.

[6] T. Yamada (2001): *Confluence and termination of simply typed term rewriting systems*. In: *Proc. of RTA 2001*, *LNCS* 2051, Springer-Verlag, pp. 338–352.

[7] T. Yokoyama, Z. Hu & M. Takeichi (2004): *Deterministic second-order patterns*. *IPL* 89(6), pp. 309–314.

Confluence via Critical Valleys

Vincent van Oostrom
Department of Philosophy, Utrecht University, The Netherlands
Vincent.vanOostrom@phil.uu.nl

A recent result due to Hirokawa and Middeldorp expresses that a left-linear first-order term rewriting systems is confluent, if its critical pairs are joinable and its *critical pair* system, comprising the steps of the critical peaks as rules, is relatively terminating with respect to the original term rewriting system. That result captures both confluence of orthogonal first-order term rewriting systems and of terminating left-linear first-order term rewritings having joinable critical pairs. Here we extend it in three ways:

- we generalise the result from first- to higher-order rewriting;
- we show that instead of the critical pair system, it suffices to consider only a *critical valley* system, comprising as rules reductions from the source of a critical peak to the targets of the first multisteps (if these exist) of the valley completing the peak; and
- we show that *development closed* critical pairs, where the target of the inner step of a critical peak reduces in a multistep to the target of the outer step of the peak, need not be considered when constructing the critical valley system.

1 Confluence via Critical Valleys

Let a *critical valley* system for a locally confluent term rewriting system $\mathscr{R}$ be a system $\mathscr{S}$ over the same signature comprising for *each* critical peak $\underline{s}_0 \leftarrow_{\mathscr{R},\mathrm{root}} \underline{t} \rightarrow_{\mathscr{R}} \underline{r}_0$ such that not $\underline{s}_0 \mathrel{\leftarrow\!\!\!\circ\!-}_{\mathscr{R}} \underline{r}_0$, and *some* valley $\underline{s}_0 \mathrel{-\!\circ\!\!\!\rightarrow}^n_{\mathscr{R}} \underline{s}_n = \underline{r}_m \mathrel{\leftarrow\!\!\!\circ\!-}^m_{\mathscr{R}} \underline{r}_0$ completing it, rules $\underline{t} \rightarrow \underline{s}_1$ if $n \geq 1$ and $\underline{t} \rightarrow \underline{r}_1$ if $m \geq 1$. Referring the reader to [2] for no(ta)tions and results used, we generalize [1, Thm. 16 and p. 497]:

Theorem (Critical Valley)**.** *A left-linear locally confluent first- or higher-order term rewriting system $\mathscr{R}$ is confluent if $\mathscr{S}/\mathscr{R}$ is terminating for some critical valley system $\mathscr{S}$ for $\mathscr{R}$.*

Proof. Since $\rightarrow_{\mathscr{R}} \subseteq \mathrel{-\!\circ\!\!\!\rightarrow}_{\mathscr{R}} \subseteq \twoheadrightarrow_{\mathscr{R}}$ holds for all term rewriting systems, it suffices [4, Proposition 1.1.11 and Lemma 11.6.24] to show confluence of $\mathrel{-\!\circ\!\!\!\rightarrow}_{\mathscr{R}}$, for which in turn it suffices [3, Theorem 3] to show that its labelling defined by $t \blacktriangleright_{\hat{t}} s$ if $\hat{t} \twoheadrightarrow_{\mathscr{R}} t \mathrel{-\!\circ\!\!\!\rightarrow}_{\mathscr{R}} s$, is decreasing with respect to the order $(\mathscr{S}/\mathscr{R})^+$. In particular, we show that for given $\hat{t}_i$, a peak $t_0 \blacktriangleleft_{\hat{t}_0} t \blacktriangleright_{\hat{t}_1} t_1$ contracting the multi-redexes U_0,U_1, can be completed into a decreasing diagram by a conversion of shape $\blacktriangleright_{\hat{t}_1} \cdot \blacklozenge^* \cdot \blacktriangleleft_{\hat{t}_0}$, where all steps in the conversion $\blacklozenge^*$ have labels $\mathscr{S}/\mathscr{R}$-smaller than a $\hat{t}_i$, by induction on the amount of overlap between the patterns of redexes in U_0,U_1:
(0) Then $U_0 \cup U_1$ is a set of non-overlapping redexes and contracting them in t yields a common $\mathrel{-\!\circ\!\!\!\rightarrow}$-reduct t' of the t_i by the Triangle Theorem 10 of [2], so $t_0 \blacktriangleright_{\hat{t}_1} t' \blacktriangleleft_{\hat{t}_0} t_1$, since $\hat{t}_i \twoheadrightarrow_{\mathscr{R}} t \twoheadrightarrow_{\mathscr{R}} t_{1-i}$.
(>0) Let $u_i \in U_i$ with $s_0 \leftarrow_{u_0} t \rightarrow_{u_1} r_0$ be induced by a critical peak $\underline{s}_0 \leftarrow_{\mathscr{R}} \underline{t} \rightarrow_{\mathscr{R}} \underline{r}_0$ with, w.l.o.g., u_1 innermost, and distinguish cases on whether $\underline{s}_0 \mathrel{\leftarrow\!\!\!\circ\!-}_{\mathscr{R}} \underline{r}_0$ or not:
($\top$) By Claim 23 of [2] there exists a peak $t_0 \blacktriangleleft_{\hat{t}_0} r_0 \blacktriangleright_{\hat{t}_1} t_1$ contracting multi-redexes U'_0,U'_1 having a smaller amount of overlap than U_0,U_1 had, and we conclude by the induction hypothesis.
($\bot$) There is a valley $\underline{s}_0 \mathrel{-\!\circ\!\!\!\rightarrow}^n_{\mathscr{R}} \underline{s}_n = \underline{r}_m \mathrel{\leftarrow\!\!\!\circ\!-}^m_{\mathscr{R}} \underline{r}_0$ such that $\underline{t} \rightarrow \underline{s}_1$ if $n \geq 1$ and $\underline{t} \rightarrow \underline{r}_1$ if $m \geq 1$.

Submitted to:
HOR 2012

If $n \geq 1$, the induction hypothesis can be applied to $t_0 \blacktriangleleft_{\hat{t_0}} s_0 \blacktriangleright_{\hat{t_1}} s_1$ as $\underline{t}$ and $U_0 - \{u_0\}$ do not overlap in t by innermostness of u_1 and the tree-structure of terms so neither do their descendants in s_1 after u_0, yielding a decreasing diagram $t_0 \blacktriangleright_{\hat{t_1}} \cdot \blacktriangleleft\!\blacktriangleright^{*} \cdot \blacktriangleleft_{\hat{t_0}} s_1$ hence, relabeling its last step, also $t_0 \blacktriangleright_{\hat{t_1}} \cdot \blacktriangleleft\!\blacktriangleright^{*} \cdot \blacktriangleleft_{s_1} s_1$, where all steps except the first have labels $\mathscr{S}/\mathscr{R}$-smaller than a $\hat{t_i}$.

If $m \geq 1$ the induction hypothesis can be applied to $r_1 \blacktriangleleft_{\hat{t_0}} r_0 \blacktriangleright_{\hat{t_1}} t_1$ as $\underline{t}$ and V_0 overlap more in t than $\underline{r}_0$ and the residuals of V_0 after v_0 do in r_0 by innermostness of t and the tree-structure of terms, yielding a decreasing diagram $r_1 \blacktriangleright_{\hat{t_1}} \cdot \blacktriangleleft\!\blacktriangleright^{*} \cdot \blacktriangleleft_{\hat{t_0}} t_1$ hence, relabeling its first step, also $r_1 \blacktriangleright_{r_1} \cdot \blacktriangleleft\!\blacktriangleright^{*} \cdot \blacktriangleleft_{\hat{t_0}} t_1$ where all steps except the last have labels $\mathscr{S}/\mathscr{R}$-smaller than a $\hat{t_i}$.

- If $n, m \geq 1$ then we may join the above conversions by the following labelling induced by a suffix of the local confluence valley $s_1 \blacktriangleright_{s_1}^{n-1} s_n = r_m \blacktriangleleft_{r_1}^{m-1} r_1$;
- If $n = 0$ and $m \geq 1$, then we may join $t_0 \blacktriangleleft\!\!\blacktriangleleft_{r_1} r_1$ and the second conversion above;
- If $n \geq 1$ and $m = 0$, then we may join the first conversion above and $s_1 \blacktriangleright\!\!\blacktriangleright_{s_1} t_0$; and
- The case that $n = 0 = m$ cannot occur as then $\underline{s}_0 \not\leftarrow\!\circ\!\!-_{\mathscr{R}} \underline{r}_0$. □

References

[1] Nao Hirokawa & Aart Middeldorp (2010): *Decreasing diagrams and relative termination*. In: *Proceedings of IJCAR 2010*, *Lecture Notes in Computer Science* 6173, Springer, pp. 487–501. doi:10.1007/978-3-642-14203-1_41.

[2] Vincent van Oostrom (1997): *Developing developments*. *Theoretical Computer Science* 175(1), pp. 159–181. doi:10.1016/S0304-3975(96)00173-9.

[3] Vincent van Oostrom (2008): *Confluence by Decreasing Diagrams, Converted*. In Andrei Voronkov, editor: *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 2007. Proceedings*, *Lecture Notes in Computer Science* 5117, Springer, pp. 306–320. (`doi:10.1007/978-3-540-70590-1_21`).

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

Termination of higher-order rewriting in dependent type calculi[*]

Jean-Pierre Jouannaud
INRIA-LIAMA
Beijing, China
Software Chair, Tsinghua University
Beijing, China
jeanpierre.jouannaud@gmail.com

Jianqi Li
Tsinghua University
Beijing, China
jianqili@gmail.com

Automatable methods exist for showing termination of either first-order or higher-order rewrite rules, in particular by defining syntactic orderings on the set of terms. These methods assume the absence of fully polymorphic as well as of dependent types. In this paper, we consider the case of dependent type calculi such as Edinburgh's Logical Framework. We define a transformation from LF-terms to terms of a lambda-calculus typable in a functional type discipline augmented with type constructors, which preserves both termination and non-termination. Termination of rewrite rules in the source language can then be proved by comparing their transforms in the target language with the computability path ordering of Blanqui, Jouannaud and Rubio.

1 Introduction

Methods exist for showing termination of higher-order rewriting based on plain pattern matching, in particular by defining syntactic orderings on terms. These methods assume absence of fully polymorphic types and of dependent types as well. In this paper, we consider the (much simpler indeed) case of dependent type calculi such as Edinburgh's LF [5], a Logical Framework rich enough for encoding other logical systems. We define a well-founded order on the set of LF-terms via a transformation to terms of a lambda-calculus typable in a functional type discipline augmented with (algebraic) type constructors. Since the transformation preserves reductions, termination in the target language implies termination in the source language.

LF is a type theory of Barendregt's cube with monomorphic dependent types and constant type constructors. We consider here a minor extension enjoying function symbols and type constructors with arities, which is convenient for expressing dependent rewriting. The target language is a simply-typed lambda-calculus enriched with function symbols and type constructors with arities. This choices has 3 advantages: the target vocabulary is as close as possible from the source vocabulary; the transformed rules can be compared in the computability path ordering (CPO) of Blanqui, Jouannaud and Rubio [3]; the technique can be adapted to show termination of higher-order rewriting based on higher-order pattern matching [7].

[*]Funded by Tsinghua National Laboratory for Information Science and Technology (TNList) Cross-discipline Foundation 2011-9, National Basic Research Program of China(973 Program) 2010CB328003.

Preliminary Report. Final version to appear in:
HOR 2012

2 Edinburgh Logical Framework

2.1 Terms

Let $\mathcal{S}$ be the set of *data types* (*constants*), $\mathcal{V}$ be the set of *variables*. Type constants have arity 0 while their kind may be functional. Object constant have an arity upper-indexing their name, a facility later used by CPO. There are three sorts of terms in LF [5], objects, types (or type families), and kinds. The syntax of LF-expressions is given by following grammar:

Kinds	K	$:=$	$\text{TYPE} \mid \Pi x : A.K$
Type families	A, B	$:=$	$a \mid \Pi x : A.B \mid \lambda x : A.B \mid A\ M$
Objects	M, N	$:=$	$x \mid \lambda x : A.M \mid M\ N \mid f^n(M_1, ..., M_n)$

The application M N may also be written as $@(M, N)$. We assume the notion of free and bound variables in a term U, denoted respectively by $\text{Fv}(U)$ and $\text{Bv}(U)$. We consider terms as trees which nodes are labelled by the term constructors, λx and Πx being considered as unary constructors. A position in a term is a concatenation of natural numbers describing the path to a given node, concatenation being denoted by "$\cdot$". We write $u|_p$ for the subterm of u at position p, and $u[v]_p$ for the term obtained by replacement of $u|_p$ by v at p.

2.2 Typing judgements

We assume the notion of (non-capturing) substitution, which application is written in postfix form. In LF, all expressions are typed, objects by types and types by kinds, kinds themselves being checked for validity (which is equivalent to typing them all by a special constant which cannot itself be typed). Typing environment are pairs $\Sigma; \Gamma$ made of a signature and a context. Contexts $\Gamma := \texttt{nil} \mid \Gamma, x : A$ define the variables while signatures $\Sigma := \texttt{nil} \mid \Sigma, a : K \mid \Sigma, f^n : A$ define the constants. What distinguishes dependent from non-dependent calculi is that the order of constants or variables in the environment is determined by their types. This has important consequences on typing, especially in the expression of the so-called substitution lemma.

Five kinds of judgement are recursively defined by the LF type system given in part at Figure 1: $\vdash_{\mathcal{S}} \Sigma$ for "Σ is a valid signature"; $\Sigma \vdash_{\mathcal{C}} \Gamma$ for "Γ is a valid context assuming $\vdash_{\mathcal{S}} \Sigma$"; $\Sigma; \Gamma \vdash_{\mathcal{K}} K$ for "K is a valid kind assuming $\Sigma \vdash_{\mathcal{C}} \Gamma$"; $\Sigma; \Gamma \vdash_{\mathcal{F}} A : K$ for "A has kind K assuming $\Sigma; \Gamma \vdash_{\mathcal{K}} K$"; $\Sigma; \Gamma \vdash_{\mathcal{O}} M : A$ for "M has type A assuming $\Sigma; \Gamma \vdash_{\mathcal{F}} A : K$"; and $\Sigma; \Gamma \vdash C : D$ as shorthand for any of the previous two judgements. Where the notation $M\{N/x\}$ denoting the (capture avoiding) substitution of the free variable x by N in M, $A \circ \{M_1/x_1, ..., M_n/x_n\}$ the sequential substitutions, $\equiv$ the equivalence relation derived from the β-reduction rules of LF.

Given a valid signature Σ and a context Γ valid in Σ, the set $\mathcal{T}_\Sigma$ of valid terms is the (disjoint) union of the sets $\mathcal{K}$ of valid *kinds*, $\mathcal{F}$ of valid *type families* and $\mathcal{O}$ of valid *objects*.

Our dependent calculus will be refered to as LF_Σ to stress the signature Σ (LF for short). In dependent type calculi, applying the substitution lemma several times introduces naturally an order on the application of elementary substitutions. We use the notation $M \circ \{M_i/x_i\}_{\{1,...,n\}}$ to denote the sequential application to M of the *elementary* substitutions $\{M_1/x_1\}$, ..., $\{M_n/x_n\}$ in this order $\circ$ denotes the postfix application of a substitution executed in a sequential way.

In LF, the usual rule schemata of the lambda calculus apply at both the object and type levels, making four different rules. In particular: $(\lambda x : A.M)\ N \rightarrow_{\beta_F} M\{N/x\}$ and $(\lambda x : A.B)\ M \rightarrow_{\beta_O} B\{M/x\}$. We write $A \equiv B$ and say that A and B are *convertible* if they are in the least monotone equivalence generated by the LF rules.

$$\textbf{Sig.}: \quad [\textsc{Const}]\ \frac{\vdash_{\mathcal{S}} \Sigma \qquad \Sigma;\mathtt{nil} \vdash_{\mathcal{F}} A_i, A : \text{TYPE}}{\vdash_{\mathcal{S}} \Sigma, f^n : \Pi\{x_i : A_i\}_{\{1,\dots,n\}}.A}\ f^n \notin dom(\Sigma)$$

$$\textbf{Kinds}: \quad [\textsc{Univ}]\ \frac{\Sigma \vdash_{\mathcal{C}} \Gamma}{\Sigma;\Gamma \vdash_{\mathcal{K}} \text{TYPE}} \qquad [\textsc{Prod}]\ \frac{\Sigma;\Gamma,\ x : A \vdash_{\mathcal{K}} K}{\Sigma;\Gamma \vdash_{\mathcal{K}} \Pi x : A.K}$$

$$\textbf{Types}: \quad [\textsc{Prod}]\ \frac{\Sigma;\Gamma,\ x : A \vdash_{\mathcal{F}} B : \text{TYPE}}{\Sigma;\Gamma \vdash_{\mathcal{F}} \Pi x : A.B : \text{TYPE}} \quad [\textsc{Abs}]\ \frac{\Sigma;\Gamma,\ x : A \vdash_{\mathcal{F}} B : K}{\Sigma;\Gamma \vdash_{\mathcal{F}} \lambda x : A.B : \Pi x : A.K}$$

$$[\textsc{App}]\ \frac{\Sigma;\Gamma \vdash_{\mathcal{F}} A : \Pi x : B.K \qquad \Sigma;\Gamma \vdash_{\mathcal{O}} M : B}{\Sigma;\Gamma \vdash_{\mathcal{F}} A M : K\{M/x\}}$$

$$\textbf{Obj.}: \quad [\textsc{Conv}]\ \frac{\Sigma;\Gamma \vdash_{\mathcal{O}} M : A \qquad \Sigma;\Gamma \vdash_{\mathcal{F}} A' : \text{TYPE}}{\Sigma;\Gamma \vdash_{\mathcal{O}} M : A'}\ A \equiv A'$$

$$[\textsc{Abs}]\ \frac{\Sigma;\Gamma,\ x : A \vdash_{\mathcal{O}} M : B}{\Sigma;\Gamma \vdash_{\mathcal{O}} \lambda x : A.M : \Pi x : A.B} \quad [\textsc{App}]\ \frac{\Sigma;\Gamma \vdash_{\mathcal{O}} M : \Pi x : A.B \quad \Sigma;\Gamma \vdash_{\mathcal{O}} N : A}{\Sigma;\Gamma \vdash_{\mathcal{O}} M N : B\{N/x\}}$$

$$[\textsc{Fun}]\ \frac{\Sigma;\Gamma \vdash_{\mathcal{O}} M_i : A_i\{M_j/x_j\}_{\{1,\dots,i-1\}}\ i \in \{1,\dots,n\}}{\Sigma;\Gamma \vdash_{\mathcal{O}} f^n(M_1,\dots,M_n) : A \circ \{M_i/x_i\}_{\{1,\dots,n\}}}\ f^n : \Pi\{x_i : A_i\}_{\{1,\dots,n\}}.A \in \Sigma$$

Figure 1: **Fragment of LF Typing rules**

2.3 Dependent rewriting

Definition 1 (Plain dependent rewriting system) *Given a signature* Σ, *a* plain dependent rewriting system *is a set* $\{\Delta_i \vdash l_i \to r_i : A_i\}_i$ *of quadruples made for every index i of a context* Δ_i*, a left-hand side object term* l_i*, a right-hand side object term* r_i*, and a type* A_i *such that* $\Sigma;\Delta_i \vdash_{\mathcal{O}} l_i : A_i$ *and* $\Sigma;\Delta_i \vdash_{\mathcal{O}} r_i : A_i$.

Given a rewriting system R, one step rewriting $\Sigma;\Gamma \vdash_R s \longrightarrow^p_R t$ *is a relation defined as:*

1. *s and t are types or objects which are typable under* $\Sigma;\ \Gamma$.
2. $\Delta \vdash l \to r : A \in R$*, where* Δ *and* Γ *have no variable in common.*
3. $s = s[l \circ \gamma]_p$ *and* $t = s[r \circ \gamma]_p$.

where γ *is a dependent substitution ordered according to* Δ.

3 Flattening

Encoding *LF* into the simply-typed λ-calculus is a well-known technique already used by Harper, Honsell and Plotkin in their seminal *LF* paper. It allows to reduce the strong normalization property of dependently typed calculi problem to that of a simpler calculus without dependent types. The encoding is based on erasing type dependencies, and recording them via wrappers. The technique has been used successfully for many variants of the Calculus of Constructions [4].

We describe now piece by piece the specific transformation used here from LF terms to $\lambda_{\rho(\Sigma)}$, our simply typed target language which signature built from Σ is $\rho(\Sigma)$. We start with types.

$\mathcal{T}_{\mathcal{S}} = * \mid a \mid \mathcal{T}_{\mathcal{S}} \to \mathcal{T}_{\mathcal{S}}$, where $a : K \in \Sigma$ and $*$ is a new constant type.

We then define an *erasure* transformation for families, kinds, and environments:

(1)$\|\text{TYPE}\| = *$	(2)$\|\Pi x : A.K\| = \|A\| \to \|K\|$	(3)$\|a\| = a$
(4)$\|\Pi x : A.B\| = \|A\| \to \|B\|$	(5)$\|\lambda x : A.B\| = \|B\|$	(6)$\|A\ M\| = \|A\|$

$$\begin{array}{rcl} |\Sigma| & = & \{a : |K| \quad | \ a : K \in \Sigma\} \\ & \cup & \{f^n : |A_1| \times \ldots \times |A_n| \to |A| \quad | \ f^n : \Pi\{x_i : A_i\}_{\{1,\ldots,n\}}.A \in \Sigma\} \\ |\Gamma| & = & \{x : |A| \quad | \ x : A \in \Gamma\} \end{array}$$

We extend the signature Σ by algebraic symbols that record in their name any important erased type information, using *lf* and *lo* to encode abstractions at the family and object levels:

$\rho(\Sigma) = |\Sigma| \cup \{pi_\sigma : * \times (\sigma \to *) \to * | \ \sigma \in \mathcal{T_S}\} \cup \{lf_\sigma,\ lo_\sigma : * \times \sigma \to \sigma | \ \sigma \in \mathcal{T_S}\}$

The set of *raw terms* $\mathcal{A}_{\rho(\Sigma)}$ of $\lambda_{\rho(\Sigma)}$ is then defined by the following grammar:
$s,\ t,\ t_i = x \mid \lambda x.t \mid @(s\ t) \mid g(t_1, ..., t_n)$, where $g \in \rho(\Sigma)$.

Here comes the transformation from source to target language:

(1)	$[a : K]$	$= a$
(2)	$[\Pi x : A.B : \text{TYPE}]$	$= pi_{\|A\|}([A],\ \lambda x : \|A\|.[B])$
(3)	$[\lambda x : A.B : \Pi x : A.K]$	$= \lambda x : \|A\|.lf_{\|K\|}([A],\ [B])$
(4)	$[A\ M : K]$	$= @([A]\ [M])$
(5)	$[x : A]$	$= x$
(6)	$[\lambda x : A.M : \Pi x : A.B]$	$= \lambda x : \|A\|.lo_{\|B\|}([A],\ [M])$
(7)	$[M\ N : A]$	$= @([M]\ [N])$
(8)	$[f^n(M_1, ..., M_n) : A]$	$= f^n([M_1], ..., [M_n])$

and the type system in the target language:

$$[1]\ \frac{}{\rho(\Sigma); \Delta \vdash x : \sigma}\ x : \sigma \in \Delta \qquad [3]\ \frac{\rho(\Sigma); \Delta \vdash s : \sigma \to \tau \qquad \rho(\Sigma); \Delta \vdash t : \sigma}{\rho(\Sigma); \Delta \vdash @(s\ t) : \tau}$$

$$[2]\ \frac{\rho(\Sigma); \Delta,\ x : \sigma \vdash t : \tau}{\rho(\Sigma); \Delta \vdash \lambda x : \sigma.t : \sigma \to \tau} \qquad [4]\ \frac{\rho(\Sigma); \Delta \vdash s_i : \sigma_i}{\rho(\Sigma); \Delta \vdash g(s_1,\ ...,\ s_n) : \sigma}\ \begin{array}{l} g^n : \sigma_1 \times \ldots \times \sigma_n \\ \quad \to \sigma \in \rho(\Sigma) \end{array}$$

Figure 2: **Typing rules for** $\lambda_{\rho(\Sigma)}$ where a context Δ is made of pairs $(x \in \mathcal{V}\ :\ \sigma_i \in \mathcal{T_S})$

We end up our journey with the translations $[\beta\eta]$ of $\beta\eta$-reductions in $\lambda_{\rho(\Sigma)}$:
$@((\lambda x : |A|.lf_{|K|}([A],\ [B]))\ [N]) \longrightarrow_{\beta_\mathcal{F}} [B]\{[N]/x\}$ if $(\lambda x : A.B : \Pi x : A.K)\ N$ valid in LF_Σ
$@((\lambda x : |A|.lo_{|B|}([A],\ [M]))\ [N]) \longrightarrow_{\beta_\mathcal{O}} [M]\{[N]/x\}$ if $(\lambda x : A.M : \Pi x : A.B)\ N$ valid in LF_Σ
$(\lambda x : |A|.lf_{|K|}([A],\ @([B]\ x))) \longrightarrow_{\eta_\mathcal{F}} [B]$ if $\lambda x : A.@(B\ x) : K$ valid in LF_Σ, x not free in B
$(\lambda x : |A|.lo_{|B|}([A],\ @([M]\ x))) \longrightarrow_{\eta_\mathcal{O}} [M]$ if $\lambda x : A.@(M\ x) : B$ valid in LF_Σ, x is not free in M

Theorem 2 (Correctness) *Let Σ be a signature and R be a dependent term rewriting system. Then R is terminating in LF_Σ iff $[R\beta\eta]$ is terminating in $\lambda_{\rho(\Sigma)}$.*

4 Example illustrating the transformation and CPO-comparison

We recall the fragment of the definition of CPO needed for the coming example.

Definition 3 (Primitive CPO) *Assume $\Delta \vdash_{\rho(\Sigma)} s : \sigma$ and $\Delta \vdash_{\rho(\Sigma)} t : \tau$, then $s : \sigma \succ^X t : \tau$ iff (a) $t \in X$ or either of the following cases holds:*

1. $s = f(\bar{s})$ with $f \in \rho(\Sigma)$ and either of *(a)* $t \in X$

(b) $t = g(\bar{t})$ *with* $f =_{\rho(\Sigma)} g \in \rho(\Sigma)$ *and* $\bar{s}(\succ_{\mathcal{T}_S})_{mul}\, \bar{t}$

(c) $t = g(\bar{t})$ *with* $f >_{\rho(\Sigma)} g \in \rho(\Sigma), up\{@\}$ *and* $s \succ^{X} \bar{t}$

(d) $t = \lambda y : \beta.w$ *and* $s \succ^{X \cup \{z\}} w\{z/y\}$ *for* $z : \beta$ *fresh*

(e) $u \succeq_{\mathcal{T}_S} t$ *for some* $u \in \bar{s}$

Here is our example, we give first the signatures and then the rewriting rules:

nat : TYPE $\quad 0 : nat \quad +1 : \Pi x : nat.nat$

$List : \Pi m : nat.$TYPE $\quad nil : List\ 0$

$cons : \Pi\{x : nat,\ m : nat,\ l : List\ m\}.List\ m+1$

$map : \Pi\{m : nat,\ l : List\ m\}.(\Pi f : \Pi\{x : nat\}.nat.List\ m)$

$foldr : \Pi\{m : nat,\ l : list\ m,\ y : nat\}.(\Pi g : \Pi\{x_1 : nat,\ x_2 : nat\}.nat.nat)$

1. $map(0,\ l) \to \lambda f : \Pi\{x : nat\}.nat.nil$
2. $map(m+1,\ cons(m,\ x,\ l)) \to \lambda f : \Pi\{x : nat\}.nat.cons((f\ x),\ map(m,\ l))$
3. $foldr(0,\ l,\ y) \to \lambda g : \Pi\{x_1 : nat,\ x_2 : nat\}.nat.y$
4. $foldr(m+1,\ cons(m,\ z,\ l),\ y) \to \lambda g : \Pi\{x_1 : nat,\ x_2 : nat\}.nat.(g\ z\ (foldr(m,\ l,\ y)\ g))$

Here are the typing constraints and precedence on constants needed to carry out the example:
$List >_{\mathcal{T}_S} nat \quad foldr > map > cons > nil > lf_{\sigma} > lo_{\tau} > List > nat$, where $\sigma,\ \tau \in \mathcal{T}_S$
and $lf_{\sigma} > lf_{\tau}$, $lo_{\sigma} > lo_{\tau}$ for any $\sigma >_{\mathcal{T}_S} \tau$

Note that type constant from the dependent world become both type constants and term constants in the flattened world. We restrict our attention to rule 4, which translation is:

$[l_4] = foldr(m+1,\ cons(m,\ z,\ l),\ y)$

$[r_4] = \lambda g : nat \to nat \to nat.lo_{nat}(lf_{nat}(nat,\ \lambda x_1 : nat.lf_{nat}(nat,\ \lambda x_2 : nat.nat)),$
$\quad @(@(g\ z)\ @(foldr(m,\ l,\ y)\ g)))$

The initial CPO judgement $[l_4] \succ_{\mathcal{T}_S} [r_4]$ is reduced to goal 1 by CPO 1(d).

1. $[l_4] \succ^{\{g\}} lo_{nat}(lf_{nat}(nat,\ \lambda x_1 : nat.lf_{nat}(nat,\ \lambda x_2 : nat.nat)),$
   $@(@(g\ z)\ @(foldr(m,\ l,\ y)\ g)))$. Since $foldr > lo_{nat}$, CPO 1(c) yields goals 2 and 3;
2. $[l_4] \succ^{\{g\}} lf_{nat}(nat,\ \lambda x_1 : nat.lf_{nat}(nat,\ \lambda x_2 : nat.nat))$, which yields by CPO 1(c);
3. $[l_4] \succ^{\{g\}} @(@(g\ z)\ @(foldr(m,\ l,\ y)\ g))$, which yields goals 6 and 7 by CPO 1(c);
4. $[l_4] \succ^{\{g\}} nat$, which holds by $foldr > nat$ and CPO 1(c);
5. $[l_4] \succ^{\{g\}} \lambda x_1 : nat.lf_{nat}(nat,\ \lambda x_2 : nat.nat)$. By CPO 1(d), which reduces to goal 8;
6. $[l_4] \succ^{\{g\}} @(g\ z)$, which yields goals 9 and 10 by CPO 1(c);
7. $[l_4] \succ^{\{g\}} @(foldr(m,\ l,\ y)\ g)$, which yields goals 11 and 12 by CPO 1(c);
8. $[l_4] \succ^{\{g,\ x_1\}} lf_{nat}(nat,\ \lambda x_2 : nat.nat)$. By CPO 1(c), which reduces to goal 13 and 14;
9. $[l_4] \succ^{\{g\}} g$, which holds by CPO 1(a);
10. $[l_4] \succ^{\{g\}} z$, with $List >_{\mathcal{T}_S} nat$, which holds by successively use CPO 1(e) twice;
11. $[l_4] \succ^{\{g\}} foldr(m,\ l,\ y)$, which yields goals 15 and 16 by CPO 1(b);
12. $[l_4] \succ^{\{g\}} g$, which holds by CPO 1(a);
13. $[l_4] \succ^{\{g,\ x_1\}} nat$, which holds by $foldr > nat$ and CPO 1(c);
14. $[l_4] \succ^{\{g,\ x_1\}} \lambda x_2 : nat.nat$, which reduce to goal 17 by CPO 1(d);
15. $m+1 \succ_{\mathcal{T}_S} m$, which holds by $nat =_{\mathcal{T}_S} nat$ and CPO 1(e);
16. $cons(m,\ z,\ l) \succ_{\mathcal{T}_S} l$, which holds by $List =_{\mathcal{T}_S} List$ and CPO 1(e);
17. $[l_4] \succ^{\{g,\ x_1,\ x_2\}} nat$, which holds by $foldr > nat$ and CPO 1(c). DONE.

5 Conclusion

The reader may wonder why not using the computational closure introduced in [2] for the simply typed λ-calculus, and then developped along the years by Blanqui [1]. Indeed, the computational closure captures the initial version of HORPO [6], but is not known to include CPO, although it is likely to be the case, at least for type-preserving CPO-reductions. A more fundamental reason is that, unlike CPO, the computational closure is not a syntax-directed definition, hence has limited practical usage. CPO can therefore be seen as the best possible, computionally efficient approximation of the computational closure.

The next (easy) step in this work will be to define CPO directly on *LF* terms, and prove it is SN by using our transformation from *LF* to $\lambda_{\rho(\Sigma)}$.

References

[1] Frédéric Blanqui (2007): *Computability Closure: Ten Years Later.* In Hubert Comon-Lundh, Claude Kirchner & Hélène Kirchner, editors: *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, *LNCS* 4600, Springer-Verlag, pp. 68–88.

[2] Frédéric Blanqui, Jean-Pierre Jouannaud & Mitsuhiro Okada (1999): *The Calculus of algebraic Constructions.* In P. Narendran & M. Rusinowitch, editors: *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, *LNCS* 1631, Springer-Verlag, Berlin, pp. 301–316.

[3] Frédéric Blanqui, Jean-Pierre Jouannaud & Albert Rubio (2008): *The computability path ordering: The end of a quest.* In Michael Kaminski & Simone Martini, editors: *Proceedings of the 17th Annual Conference of the European Association for Computer Science Logic (CSL'08)*, *LNCS* 5213, Springer, Berlin, pp. 1–14.

[4] Herman Geuvers & Mark-Jan Nederhof (1991): *Modular Proof of Strong Normalization for the Calculus of Constructions.* *Journal of Functional Programming* 1(2), pp. 155–189.

[5] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics.* *Journal of the ACM* 40(1), pp. 143–184.

[6] Jean-Pierre Jouannaud & Albert Rubio (1999): *The higher-order recursive path ordering.* In Giuseppe Longo, editor: *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, IEEE Computer Society, pp. 402–411.

[7] Jean-Pierre Jouannaud & Albert Rubio (2006): *Higher-order orderings for normal rewriting.* In Frank Pfenning, editor: *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA'06)*, *LNCS* 4098, Springer-Verlag, Berlin, pp. 387–399.

Higher Order Rewriting for Real Programmers

Kristoffer H. Rose
IBM Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, USA; ⟨*krisrose@us.ibm.com*⟩

We report on the mechanisms and patterns of the CRSX system specifically for supporting "real" programming in the form of compiler systems by programmers with cursory training in formal systems, in the hope of a discussion of how higher order rewriting can be used more widely.

1 Introduction

We are presently leading the implementation of a "real" compiler in IBM using the CRSX "combinatory reduction systems with extensions" higher order rewriting tool for building compilers [10]. A part of this task is to transfer knowledge of how to *program and maintain systems written using higher order rewriting to production programmers*. For most programmers the rewriting foundations remain theoretic issues that have at best been briefly mentioned in their education, *e.g.*, rewriting may have been presented as an explanation for how pattern matching works or how invariants of programs are proven by case analysis, with the obvious qualification that real programs always make choices sequentially if they are more complex than a single table lookup. Or λ calculus might be mentioned as the theory behind inner classes or closures, except of course for the fact that in real programming languages, closures can only capture very simple constants and possibly a few restricted pointer values from the context where they are created.

This leads to the fact that in the few areas where programmers do need to think about terms and bindings, a separate collection of techniques is developed to deal well with these issues. For the area that CRSX is focused on, *compiler writing*, the issues for example show up when compiler writers go to great lengths to express creation and simple manipulations of *syntax trees*, *c.f.* [1, §5.2] or [2, §4], and indeed most popular parser generators include special notation for creating and manipulating syntax tree nodes [7, 11, 9]. Another example is that each intermediate stage of the compiler uses a distinct notion of *binding* such a the *symbol table* used during parsing and analysis; *locations* used for intermediate code and optimizations; *labels*, *registers*, and even *flow edge*, used for code generation.

Indeed the tuning of each of these areas to a craft (or even fine art) has led to significant resistance to writing "real" compilers based on formal methods, as witnessed by the relative lack of success to introduce this to the mainstream compiler writing culture in spite of a significant effort invested in dedicated projects with advanced tools targeted at the task [8, 3, 4, 5]. We find it fair to say that the cited systems have failed to achieve widespread adoption for three reasons (to a varying degree in each case):

Ambition: Only require a semantic specification of the *meaning* of the language to implement and automatically deduce the compiler program.

Abstraction: Require deep understanding of the underlying *formalism* before being useful to the programmer.

Generality: Focus on general logic or other foundational principles and less on support for specific data structures needed for compiler writing (such as symbol tables and hash encodings).

In this discussion paper we will explain how CRSX is attempting to mitigate these issues, and hope to spawn a discussion of whether this can lead to a *revival of the use of higher order rewriting techniques* *in compiler rewriting* as well as other areas of programming where the tasks to solve involve notions related to manipulation of trees with some kind of binding or structured references.

2 Example CRSX Issues

In this section we illustrate how several aspects of CRSX programming have been explained to engineers in our compiler project.

2.1 Notation. All rewriting examples are given in CRSX [10] syntax (we make a point of not introducing it formally here) except that font style is significant, using META_n and **Primitive**, where actual CRSX syntax has meta-variables `#meta_n` and primitives `$Primitive`, respectively.

To start, first order term rewriting is quite easy to explain.

2.2 Example. Consider the following CRSX *sorts* for expressions (E) and instructions (I)

E ::= Plus[E, E] | Zero ;
I ::= PLUS | PUSHZERO | I[**List**[I]] ;

The traditional compositional translation from expressions to instructions is expressed by

CodeOf[E] :: I ;
CodeOfZero: CodeOf[Zero] $\rightarrow$ PUSHZERO ;
CodeOfPlus: CodeOf[Plus[LEFT, RIGHT]] $\rightarrow$ I[(CodeOf[LEFT]; CodeOf[RIGHT]; PLUS;)];

We explain that words like CodeOf and Plus are *defined constructors* with arguments in [...]s, and words like LEFT are *meta-variables* that stand for whatever is really found as the appropriate subterm; meta-variables in general occur precisely once on each side of the arrow. The *forms* for each sort must be declared (we shall extend them later), and each function (or "scheme") must be declared with a separate *sort declaration*: here we declare (with ::) that CodeOf must take an E argument and produce as its result an I instruction.[1] Specifically, each *rule* has a name (CodeOfZero, CodeOfPlus), a *pattern* left of the arrow that describes subterms the rule can fire for, and the *contraction* right of the arrow with the replacement subterm. Furthermore, in CRSX we write *lists* of things like in C with each member of the list terminated by a semicolon (;) with sort **List**[MEMBERSORT]. Rewriting proceeds by finding a place where a rule can fire and then update the term at that place with what is obtained by the rewriting rule: for CodeOfPlus a CodeOf directly applied to a Plus tree node fires the rule, replacing the CodeOf node with the instruction to the right.

2.3 Discussion (restrictions)**.** From a formal rewriting perspecive, the explanation in Example 2.2 only works because the example obeys some severe restrictions:

1. the rule is completely compositional and linear (both left and right);
2. there is a clear separation of "function" (CodeOf) and "data" (Plus) symbols;
3. the rule does not involve any binders;
4. the rule does not depend on something in the context;
5. the rule is closed, *i.e.*, pattern and contraction have no (common) free variables;
6. the rule contraction does not introduce any fresh variables;

[1]Indeed the declaration is how CRSX distinguishes function and data symbols, thus encouraging (but not enforcing) constructor systems.

7. there is no substitution; and
8. there are no nested applications.

The rest of this section shows patterns for breaking each restriction and how it is explained for CRSX programmers; for space reasons the explanation is compacted but I hope they lead to a discussion of whether there are better ways to implement and/or explain these (and more) real life examples, however, it is important to understand that *the majority of rules needed for a typical compiler follow the restrictions.*

2.4 Example. Restriction 1 is frequently broken when *analysis* rules are in play, like

−[**Copy**[PROG]]: Compile[PROG] → Optimize[Analyze[PROG], PROG] ;

The Compile rule translates a to an Optimize construct where the first argument is obtained by an Analyze stage and the second is a separate copy of the original PROG, which is why we explicitly declare that it is copied.

2.5 Example. A typical example where we break restriction 2 is if we add rules like

PlusZero[**Mixed**[Plus]]: Plus[Zero, RIGHT] → RIGHT ;

The **Mixed**[Plus] option ensures that the system does not protest that a data symbol (Plus) is used at the root of a pattern but instead invokes a *completion* procedure to make sure that the rule fires when it can before the regular functional patterns (like CodeOf) are investigated.[2]

2.6 Example. If the input syntax tree is first order (*e.g.*, when using a non-CRSX parser generator) then we may want to maintain a *symbol table* to introduce proper binders with declarations like the following, breaking restrictions 3 and 4 explicitly. Consider the sorts

RAWE ::= RawLet[RAWE, **String**, RAWE] | RawVar[**String**] ;
E ::= Let[E, v:E . E] | v ;

The first declaration defines first order "raw" syntax trees where variables are still **String** tokens, the second how the desired higher order syntax trees have Let expressions that use a proper CRSX binder (v) of E sort *and* explicitly have to declare that an E can be a *syntactic variable* with the "$|v$" choice. We then add the scheme N for translating raw to "cooked" expressions:

{**String** : E} N[RAWE] :: E ;
NLet: {ST} N[RawLet[VAL, NM, BODY]] → Let[N[VAL], v . {ST; NM : v} N[BODY]] ;
NVar[**Discard**[ST, NM]]: {ST; NM : VAR} N[RawVar[NM]] → VAR ;
NNoVar: {ST; ¬NM} N[RawVar[NM]] → **$[Error**, $[:, "Unknown␣variable:␣", NM]] ;

The sort declaration for the N function explicitly states that it requires a *symbol table* parameter (written as a {...} prefix) that maps **String** tokens to E terms (really syntactic variables). The NLet rule uses the symbol table {ST} and introduces a new symbol: the contraction creates the binder v scoped over the translation of BODY using N with a symbol table extended to map NM to v. NVar also accesses the symbol table but further *looks up* a raw NM and, if a variable VAR corresponds to it, then that is the result rule. This is a specific kind of non-left-linearity allowed: a meta-variable of **String** sort matched in the normal part of a pattern can be used in the immediately preceding environment part to look up the value corresponding to the constant. Since we are not reusing any of ST or NM in the contraction we must explicitly break restriction 1 with a **Discard** option. Finally, rule NNoVar handles the (orthogonal) case where NM has *not* been declared (and reports an error in that case using the built in **$**[**Error**, ...]).

[2]Such optimization rules typically break confluence but, with proper notification of the problematic overlaps, completion in fact usually works and gives the result that the compiler writer expects.

2.7 Example. Restrictions 5 and 6 are broken, for example, when traversing terms with syntactic variables. We extend the code generation sorts with register instructions and a new register sort, R:

> I ::= ... | POPSTORE[R] | PUSHLOAD[R] ;
> R ::= *r* ;

We can now add code generation rules for the register operations.

> CodeOfLet[**Fresh**[x,r]]: {Γ}CodeOf[Let[VAL, v.BODY[v]]]
> $\rightarrow$ I[({Γ}CodeOf[VAL]; POPSTORE[r]; {Γ; x:r}CodeOf[BODY[x]];)] ;
> CodeOfVar[**Free**[x,r]]: {ST; x:r}CodeOf[x] $\rightarrow$ PUSHLOAD[r] ;

The CodeOfLet rule has the first *binder pattern* we have seen, v.BODY[v], in the style of combinatory reduction systems. This requires a careful explanation that when a meta-variable in a pattern is inside a scope, like BODY is in the scope of v, then we must provide (in [...]s) the precise list of variables in scope that can occur inside BODY as a *place holder* for the contraction: in the contraction we can then replace all occurrences of the variable (here v) with another variable (here x), which must be freshly created and thus is declared with a **Fresh**[x] option as it breaks restriction 6. Notice that even though we replace all occurrences of a variable with another variable, this does not count as a substitution and so does not break restriction 7. We also create a free register variable for use by the store/load instruction pair. Additionally, when fresh variables are used in this way then the only way they can be matched later is as in rule CodeOfVar, which uses **Free**[x] to permit breaking restriction 5; like in Example 2.6 the *lookup* of a variable in the list of properties does not count as non-linearity. Finally, we remark that the proper sort declaration for CodeOf is now

> {E:R}CodeOf[E] :: I ;

The explained replacement in fact follows usual formal rewriting tradition yet this point turns out to be the most difficult to explain to programmers, and thus the aspect of CRSX where it is most important that the system supports the programmer with severe constraints.

In contrast, the use of a "fresh" variable that is globally allocated such as for registers, or permitting special "free" variables in patterns, is easier to explain but has a more profound impact in the formal model, needing less structured semantics such as provided by "nominal" rewriting [6], but the right formal model is not yet clear.

2.8 Example. For some applications, notably inlining, we wish to break restriction 7. The simplest such rule would be an inlining Let rule.

> CodeOfLetInline[**Copy**[VAL]]: CodeOf[Let[VAL, v.BODY[v]]] $\rightarrow$ BODY[VAL] ;

The rule is explained by noting that the variable substitution from before generalizes to "full" substitution shown here with the caveat that since we do not know how many occurrences of v has in BODY, we must allow for VAL to be copied. If we have a linear version of Let then a substitution rule can be written with special ^{1}-marked "linear" variables,

> CodeOfLinearLetInline: CodeOf[LinearLet[VAL, v^1.BODY[v^1]]] $\rightarrow$ BODY[VAL] ;

since at most one copy is to be created.

2.9 Example. In some cases it is necessary to write nested applications and break restriction 8. The most common case is when the result of an analysis is passed through the environment to a later transaction stage. We typically specify the analysis with a function with *polymorphic* signature

$\forall a.\mathrm{Ana}[\textsc{e},\ ok^1{:}\,\textsc{ok}.a] :: a$;

which is then used in code like

Start[**Copy**[E]]: $\{\Gamma\}$G[E] $\rightarrow$ Ana[E, ok^1.G2[ok, E]] ;
Cont: $\{\Gamma\}$G2[**OK**, E] $\rightarrow \ldots$;

This is explained by recalling that if G2 was not *guarded* as shown with the special linear ok^1 parameter[3] when passed to Ana in the Start rule, then the Cont rule could fire prematurely before $\{\Gamma\}$ is populated by the rules for Ana. The sort OK is a special built-in sort containing just the **OK** token.

This concludes the explanation of the restrictions. We further give a brief example of the final language support feature of custom syntax.

2.10 Example (embedded syntax)**.** Example 2.2 can be rewritten to introduce and use embedded syntax by writing

E ::= $[\![\langle\textsc{e}\rangle + \langle\textsc{e}\rangle]\!]$ | $[\![0]\!]$;
C ::= $[\![\langle\textsc{c}\rangle\ \langle\textsc{c}\rangle$ PLUS$]\!]$ | $[\![$PUSHZERO$]\!]$;
CodeOf[E] :: C ;
CodeOf[E$[\![0]\!]$] $\rightarrow$ C$[\![$PUSHZERO$]\!]$;
CodeOf[E$[\![\langle\textsc{left}\rangle + \langle\textsc{right}\rangle]\!]$] $\rightarrow$ C$[\![\langle$CodeOf[LEFT]$\rangle$ $\langle$CodeOf[RIGHT]$\rangle$ PLUS$]\!]$;

where the double brackets encapsulate embedded syntax fragments wherein the angle brackets “escape” to normal term notation to permit meta-variables or sub-translation as appropriate (here assuming the E scheme translates from the *exp* syntactic category to the *code* one).

The rules here are basically explained as follows: (1) E is a syntactic sort where one of the allowed forms is the shown sum where two expressions are separated by +, an the other is the plain symbol 0;[4] (2) C is a syntactic form that permits the shown postfix PLUS instruction and a PUSHZERO constant; (3) CodeOf is a compilation scheme that takes an argument of sort E and generates a result of sort C; (4) when CodeOf is used to translate a sum expression then it generates code for the operands followed by the PLUS instruction, and for a zero it generates the single PUSHZERO instruction. Notice that we have taken some liberties compared to the usual use of semantic specifications by unifying the notation: “semantic” brackets $[\![\ldots]\!]$ here just mean “subterm in native embedded syntax,” and simple angle brackets $\langle\ldots\rangle$ simple mean “subterm in meta-notation (CRSX).” Also note that when using syntax in this way, CRSX decides what the real data constructor names to use will be, so *all* constructions of the involved sorts must use the SORT$[\![\ldots]\!]$ notation.

2.11 Remark. Finally, a word on *modularity*: our current medium sized compiler for the XQuery language, with basic type analysis, static reduction, and code generation, comprises around three thousand rules, and can be compiled as a single program, so CRSX has no serious separate compilation features.

3 Conclusion

We have argued that higher order rewriting systems in general, and CRSX in particular, is ready to be introduced into the mainstream programming culture on par with tools such as parser generators and GUI builders.

[3]Linear variables are only permitted in an outermost position of their scope encapsulated only in forms that are irreducible with a variable, and cannot be syntactic, which ensures they are preserved—CRSX does not implement general linearity.

[4]The spacing and other lexical aspects of the parser have to be specified separately.

In addition to the pure language issues discussed here, there are other necessary components. The first is a *development environment*; again this is not in itself sufficient to drive adoption as witnessed by the highly developed envrironment of the CENTAUR system [4], especially it is important that the developed artifacts can be integrated easily with outside components. Another necessary factor is efficiency: the final production compiler needs to be as fast as hand written compilers, however, this turns out to be less of a problem in practice as manually written compilers tend to grow heavy under the weight of the involved (more or less) ad hoc internal data structure manipulation and consistency checks.

In CRSX we have started experimenting with an integration of less compositional features of compilers. The integration of contextual (or reduction context) "big step" rules has been done using conditional rewriting techniques and we are working on further techniques such as "subterm hashing" (used for common subexpression elimination) and constraint solving.

Finally, we hope that the presentation of this material can lead to a constructive discussion of what the higher order rewriting community can do to have the impact on the computing community at large that the power and elegance of the theory and technology deserves!

Acknowledgements. Thanks to Cynthia Kop for developing the foundations of the used sort system and completion procedure while visiting Watson during the summer of 2011.

References

[1] A. V. Aho, R. Sethi & J. D. Ullman (1986): *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[2] Andrew Appel (1998): *Modern Compiler Implementation in Java*. Cambridge University Press.

[3] D. Bjørner & C.B. Jones (1982): *Formal Specification and Software Development*. Prentice Hall International.

[4] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual (1988): *Centaur: the system*. In: *SIGSOFT'88—Third Annual Symposium on Software Development Environments*, Boston, USA.

[5] 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.

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

[7] Étienne Gagnon (1998): *SableCC, an Object-Oriented Compiler Framework*. Ph.D. thesis, School of Computer Science, McGill University. Available at `http://sablecc.sourceforge.net/thesis/thesis.html`.

[8] Peter Mosses (1979): *SIS – Semantics Implementation System, Reference Manual and User's Guide*. Technical Report MD-30, DAIMI (Computer Science Department), Aarhus University.

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

[10] Kristoffer H. Rose (2011): *CRSX – Combinatory Reduction Systems with Extensions*. In Manfred Schmidt-Schauß, editor: *RTA '11—22nd International Conference on Rewriting Techniques and Applications*, *Leibniz International Proceedings in Informatics (LIPIcs)* 10, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Novi Sad, Serbia, pp. 81–90, doi:10.4230/LIPIcs.RTA.2011.81. CRSX system available from `http://crsx.sf.net`.

[11] Sun: *JavaCC[tm]: JJTree Reference Documentation*. Available at `http://javacc.java.net/doc/JJTree.html`.

The permutative λ-calculus (extended abstract)

Beniamino Accattoli
INRIA and LIX (École Polytechnique)

Delia Kesner
PPS (CNRS and Université Paris-Diderot)

We introduce the permutative λ-calculus, an extension of λ-calculus with three equations and one reduction rule for permuting constructors, generalising many calculi in the literature, in particular Regnier's sigma-equivalence and Moggi's assoc-equivalence. We prove confluence modulo the equations and preservation of beta-strong normalisation (PSN) by means of an auxiliary substitution calculus. The proof of confluence relies on M-developments, a new notion of development for λ-terms.

1 Introduction

Background. The standard operational semantics of λ-calculus is given by β-reduction. However, this unique notion of reduction is often extended with some other rewriting rules allowing to permute constructors. This arises in different contexts and comes with many different motivations. A typical example is the postponement of *erasing* steps, which is obtained by introducing one particular such permutation rule [7]. Four other notable motivations for introducing permutations are: making redexes more visible [9], analysing the relation between λ-terms and Proof-Nets [14], proving the completeness of CPS-translation for the call-by-value λ-calculus [15], translating Moggi's monadic metalanguage into λ-calculus [6]. The rewriting theory of these permutation rules is often tricky, in particular when proving strong normalisation or preservation of strong normalisation (PSN) [10, 4, 12, 16, 5]. This is indeed the major and usually difficult question arising in all these extensions: to prove that if t is a β-strongly-normalising λ-term then t is also strongly-normalising with respect to the extended reduction relation.

The permutative λ-calculus. The permutative λ-calculus $\Lambda_{\hat{\mathtt{p}}}$ introduced in this paper extends λ-calculus with three *equations* and one rewriting rule for permuting constructors. It sensibly generalises all previous extended λ-calculi by taking — when possible — the permutations as *equivalences*, and not as *reductions*. This is a key point of our approach. We show that the permutative λ-calculus preserves β-strong normalisation and is Church-Rosser modulo the equivalences, the strongest possible form of confluence for a reduction relation modulo an equivalence. Whenever an orientation of the equations (or a subset of them) yields a terminating reduction $\rightsquigarrow$ then the system where the equations are replaced by $\rightsquigarrow$ enjoys PSN. Thus, our result subsumes all PSN results of the kind in the literature.

The proof technique. We study the permutative λ-calculus through an auxiliary and new calculus with **explicit substitutions (ES)** called $\lambda\mathtt{sub}$. In this calculus β-reduction is split into two subsystems: $\rightarrow_{\mathtt{dB}}$ which creates a substituted term $t[x/u]$, *i.e.* a term t affected by a *delayed/explicit* substitution $[x/u]$, and $\rightarrow_{\mathtt{sub}}$ which executes the ES $[x/u]$ — getting $t\{x/u\}$ — and hence completes β-reduction. This simple calculus is then enriched with various equivalences — thus getting the equational $\lambda\mathtt{sub}$-calculus — obtained by what might be called an *extension by continuity*: if t and u are equivalent λ-terms in $\Lambda_{\hat{\mathtt{p}}}$ and they $\rightarrow_{\mathtt{dB}}$-reduce to t' and u', respectively, then t' and u' are equivalent in the equational $\lambda\mathtt{sub}$. This requires to consider equivalences on terms with ES and not only on λ-terms.

Submitted to:

PSN. We prove PSN for the permutative λ-calculus by reducing this problem to PSN for the *equational* $\lambda\mathtt{sub}$-calculus, which in turn reduces to an existing result for the structural λ-calculus [3].

Confluence. Confluence of the permutative λ-calculus turns out to be delicate, and our proof is one of the main contributions of the paper. Indeed, confluence of $\Lambda_{\hat{\mathtt{p}}}$ does not follow from confluence of λ-calculus. The usual Tait–Martin Löf technique does not work, since the equations may create/hide redexes. While confluence of many reduction systems can usually be proved by means of developments [8], this notion does not suffice in the case of $\Lambda_{\hat{\mathtt{p}}}$, again because the equations create redexes. Its stronger variant, known as superdevelopments [11] or L-developments [1] — which also reduces some created redexes — does not work either. We then introduce a new form of development called M-development and then derive confluence for $\Lambda_{\hat{\mathtt{p}}}$, showing that M-developments satisfy an appropriate extension of Van Oostrom Z-property [13] to rewriting modulo. A key point is that M-developments are defined and studied through the equational $\lambda\mathtt{sub}$-calculus, where the splitting of β-reduction in terms of dB and sub becomes crucial to allow a fine analysis of redex creation. A nice fact is that our proof technique is modular, in the sense that one can choose to arbitrarily orient all or only some of the equations as rewriting rules while keeping the proof essentially unchanged. Moreover, our proof does not rely on confluence of λ-calculus.

Proof-Nets. Our work is the final product of a long-term study of the relation between ES and Linear Logic Proof-Nets. Here we present the implications of our study on λ-calculus, a language without ES, which is of a more general interest. No knowledge of Proof-Nets is assumed in the paper.

This work has been published in the proceedings of LPAR 2012 [2], to which we refer for the details.

References

[1] Beniamino Accattoli & Delia Kesner (2010): The Structural lambda-Calculus. In Anuj Dawar & Helmut Veith, editors: *Proc. of 24th Computer Science Logic (CSL)*, *Lecture Notes in Computer Science* 6247, Springer-Verlag, pp. 381–395.

[2] Beniamino Accattoli & Delia Kesner (2012): The Permutative -Calculus. In: *LPAR*, *Lecture Notes in Computer Science* 7180, Springer, pp. 23–36. Available at `http://dx.doi.org/10.1007/978-3-642-28717-6_5`.

[3] Beniamino Accattoli & Delia Kesner (2012): Preservation of strong normalisation modulo permutations for the structural calculus. *LMCS* 8.

[4] René David (2011): A short proof that adding some permutation rules to preserves SN. *Theoretical Computer Science* 412(11), pp. 1022–1026.

[5] José Espírito Santo (2011): A note on preservation of strong normalisation in the λ-calculus. *Theoretical Computer Science* 412(12-14), pp. 169–183.

[6] José Espírito Santo, Ralph Matthes & Luís Pinto (2009): Types for Proofs and Programs. chapter Monadic Translation of Intuitionistic Sequent Calculus, Springer-Verlag, Berlin, Heidelberg, pp. 100–116. Available at `http://dx.doi.org/10.1007/978-3-642-02444-3_7`.

[7] Philippe de Groote (1993): The Conservation Theorem revisited. In: *TLCA*, *Lecture Notes in Computer Science* 664, Springer, pp. 163–178. Available at `http://dx.doi.org/10.1007/BFb0037105`.

[8] J. Roger Hindley (1978): Reductions of Residuals are Finite. *Transactions of the American Mathematical Society* 240, pp. 345–361.

[9] Fairouz Kamareddine (2000): Postponement, conservation and preservation of strong normalization for generalized reduction. *Journal of Logic and Computation* 10(5), pp. 721–738.

[10] A. J. Kfoury & J. B. Wells (1995): New Notions of Reduction and Non-Semantic Proofs of beta-Strong Normalization in Typed lambda-Calculi. In Dexter Kozen, editor: *10th Annual IEEE Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, pp. 311–321.

[11] Jan-Willem Klop, Vincent van Oostrom & Femke van Raamsdonk (1993): Combinatory reduction systems: introduction and survey. *Theoretical Computer Science* 121(1/2), pp. 279–308.

[12] Stéphane Lengrand (2008): Termination of lambda-calculus with the extra Call-By-Value rule known as assoc. *CoRR* abs/0806.4859.

[13] Vincent van Oostrom: Z. Slides available on `http://www.phil.uu.nl/~oostrom/publication/rewriting.html`.

[14] Laurent Regnier (1994): Une équivalence sur les lambda-termes. *Theoretical Computer Science* 2(126), pp. 281–292.

[15] Amr Sabry & Matthias Felleisen (1992): Reasoning about programs in continuation-passing style. In: *Proc. of LISP and functional programming*, LFP '92, ACM, New York, NY, USA, pp. 288–298, doi:http://doi.acm.org/10.1145/141471.141563.

[16] José Espírito Santo (2007): Delayed Substitutions. In: *RTA*, pp. 169–183. Available at `http://dx.doi.org/10.1007/978-3-540-73449-9_14`.

A Unified Approach to Fully Lazy Sharing

Thibaut Balabonski
Univ Paris Diderot, Sorbonne Paris Cité,
PPS, UMR 7126, CNRS
F-75205 Paris, France
thibaut.balabonski@pps.univ-paris-diderot.fr

We give an axiomatic presentation of sharing-via-labelling for weak λ-calculi, that makes it possible to formally compare many different approaches to fully lazy sharing, and obtain two important results. We prove that the known implementations of full laziness are all equivalent in terms of the number of β-reductions performed, although they behave differently regarding the duplication of terms. We establish a link between the optimality theories of weak λ-calculi and first-order rewriting systems by expressing fully lazy λ-lifting in our framework, thus emphasizing the first-order essence of weak reduction.

This **category B** paper is an extended abstract of a work presented at *POPL'12* [3].

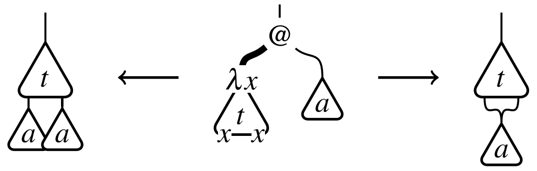
Since C.P. Wadsworth [19] introduced his graph-based algorithm for the efficient evaluation of the λ-calculus, many evaluation techniques have been proposed that try to minimize the number of β-reduction steps by using some *sharing* mechanism.

The point of sharing is the following. First remark that β-reduction, taken as such, potentially duplicates subterms at each reduction step, and that duplicating a subterm which is not in normal form also duplicates some unfinished work. Hence it would be interesting to have alternative formalisms in which the duplicated occurrences of a given original subterm would keep a unique shared representation. This would allows us to evaluate all the copies simultaneously, as if no duplication ever happened. This is what we call sharing.

This work is a unified study of several approaches to sharing in the λ-calculus.

1 Graphs and fully lazy sharing

The most intuitive way of expressing sharing might be by using graphs. In the pictures, the binary node @ represents application, and redexes[1] are marked with bold lines. For instance in the center of the following picture, an abstraction $\lambda x.t$ is applied to an argument a. The function body t contains two occurrences of the variable x: the argument a is thus logically duplicated (on the left).

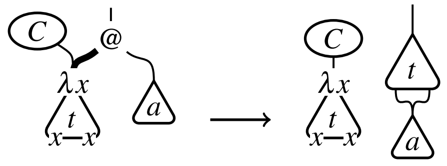


The simplest notion of sharing, which may be referred to as *lazy sharing*, or just *laziness*, prevents the previous duplication by keeping a physically unique, with two pointers to its location (right hand side part of the previous picture).

Here the term *laziness* is to be taken in literal sense: the duplications are postponed as long as it is possible, but some of them will eventually happen. For instance, a shared function has to be copied prior to any instantiation, as shown in the picture below.

[1] a **red***ucible* **ex***pression*, or **redex**, designs a place where an evaluation step can take place

Submitted to:
HOR 2012



A further level of sharing, called fully lazy sharing, is based upon the following remark: the *free expressions* [10] of a function body are not affected by the instantiation of the function, hence they need not be duplicated.

The first description of fully lazy sharing is in the *graph* evaluation technique presented by C.P. Wadsworth [19]. This graph reduction performs only a partial copy of a duplicated function body, by avoiding the copy of its maximal free expressions (see Example 1a).

Over the last 40 years, several works proposed various sharing mechanisms related to full laziness.

O. Shivers and M. Wand [16] enrich the graph structure of [19] to allow a simple and efficient implementation. For this they also use a different characterization of what has to be copied.

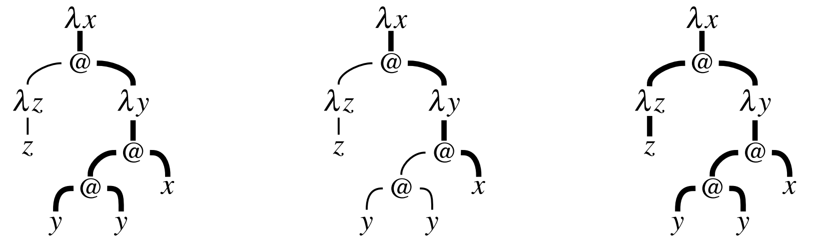
T. Blanc, J.-J. Lévy and L. Maranget [4] derive a graph implementation of fully lazy sharing using *labels* that characterize optimal sharing for a weak λ-calculus studied in [6]. This approach can copy *fewer* graph nodes of the duplicated abstractions (see Example 1b).

S. Peyton-Jones [10] reaches a simple graph formalism thanks to a fully-lazy version of λ-lifting. Following [8], fully-lazy λ-lifting turns (higher-order) λ-terms into (first-order) applicative expressions, after *extracting* the maximal free expressions of the functions.

Finally, Z. Ariola and M. Felleisen [1] and P. Sestoft [15] use the *extraction* of maximal free expressions to build fully lazy versions of (respectively) the call-by-need λ-calculus [2] and Launchbury's natural semantics for laziness [13]. Both solutions are based on *closures* represented by *let ... in ...* constructs. The former solution [1] uses a more restrictive definition of free expressions and hence may in some cases copy *more* nodes than the others (see Example 1c).

Example 1.

Bold lines identify the parts of the function that are duplicated by the different models.



(a) [19] (b) [4] (c) [1]

Summary. The following table sums up how each of the previous works gives its own view on fully lazy sharing, with different interpretations of the same main idea and using various combinations of technical tools that are sometimes hardly comparable. We use the symbol = to mean "as many copied nodes as [19]".

		Tools			
	Dupl.	Graphs	Extraction	Closures	Labels
[19]	=	✓			
[10]	=	✓	✓		
[1]	More		✓	✓	
[15]	=		✓	✓	
[16]	=	✓			
[4]	Fewer	✓			✓

This paper proposes a formal setting in which all the approaches mentioned above can be expressed. This allows us to formally compare them and leads us to the two following conclusions:

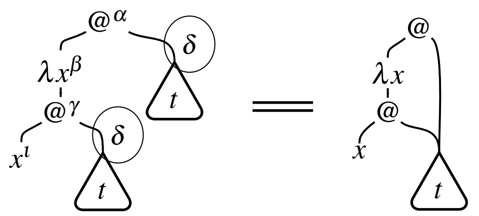
- The previous approaches correspond to at least three different graph implementations. This means that, strictly speaking, they do not all induce the same amount of sharing. Hence, despite the fact that all these approaches intend to implement the same idea their equivalence is not obvious.
- However, all these approaches have the same reduction space. This means that the different implementations of fully lazy sharing perform the same number of β-reductions. In other words, any further comparison of these approaches need not anymore take this parameter into account.

2 Contributions

2.1 An axiomatic framework for sharing-via-labelling.

We build an axiomatic framework which generalizes the work of T. Blanc, J.-J. Lévy and L. Maranget [4] and allows us to express all the previous approaches. We use labelled terms to describe the graphs realizing optimal sharing for a given notion of weak reduction. Various weak reduction notions are defined thanks to an axiomatic description of the parts of the program where reduction is forbidden. In any case the restrictions concern only evaluation in the body of a non-instantiated function, called partial evaluation. This implies that the call-by-value and call-by-name strategies are always valid. However, the different weak calculi may or may not be confluent (see [6]).

As in [14] the link between labelled terms and graphs is made by interpreting the label of a term as its location in memory, or graphically by its coordinate:



We call this principle *sharing-via-labelling*. The idea is also explored in [7]. In this setting the equality of labels corresponds to the physical equality of two terms, which should in turn imply their syntactic equality: two terms stored/drawn at the same place ought to be equal. The reduction of a graph-redex is simulated by the reduction of all the labelled term-redexes with a given label. One then needs to ensure that the *sharing property* (terms with equal labels are syntactically equal) is preserved by reduction.

This approach of sharing-via-labelling allows us to relate all the definitions of fully lazy sharing that do not rely on supercombinators and λ-lifting. In other words this axiomatic framework, which is designed in higher-order rewriting, covers the definitions of full laziness which directly operate in the higher-order world [19, 1, 15, 16, 4]. The remaining approaches using a translation to first-order rewriting by λ-lifting [8, 10] are studied separately. The translation to first-order by means of combinators of D. Turner [18] is out of the scope of the present paper, since these combinators simulate explicit substitutions and then introduce additional reduction steps.

Notably due to its axiomatic nature, our framework is not suitable for an immediate implementation. On the other hand, this approach teaches us something about full laziness in general and on its various concrete implementations. The novelty of our framework lies in the fact that it cannot be seen as a straightforward generalization of any of the aforementioned embodiments of full laziness taken in isolation: the axiomatization rather comes from an analysis of the similarities and the differences of all the concrete systems. This yields a new system whose specific properties may be understood as the *intersection* of the particular properties of the various concrete systems. In other words, our axiomatization tries to grasp the essence of full laziness.

2.2 A formal coding of higher order into first order by λ-lifting.

The λ-lifting program transformation turns a λ-term into a first-order term. The main feature of λ-lifting is the transformation of λ-abstractions into function symbols, also called supercombinators, over which first-order reduction rules are defined. As emphasized in [12], this transformation unveils a tight relation between weak λ-calculus and first-order rewriting.

Usual definitions of λ-liftings [9] proceed by first defining the transformation of λ-abstractions, and then iteratively applying the process to a λ-term until it contains no more λ-abstractions. Definitions differ in particular in the way in which a single λ-abstraction is transformed and on the order in which the iteration is applied. For instance, [10] describes a bottom-up transformation, while [15] iterates in an unspecified order. We ensure the consistence of these two views by giving a definition of fully-lazy λ-lifting in which the order of the iterative process is irrelevant.

Since λ-lifting is an iterative process that turns progressively a λ-term into a first-order term, none of the intermediate steps is in either of these worlds. Nevertheless, we would like to embody the source, the target, and all the intermediate steps of the transformation into a single formalism. To this aim we use Combinatory Reduction Systems (CRS), a higher-order rewriting framework introduced by J.W. Klop and reviewed in [11] that mixes abstractions and symbols. The β-reduction as well as the target first-order reduction have a straightforward encoding into CRS rules. Moreover, fully lazy λ-lifting itself can then be seen as a rewriting process: it is expressed as a confluent and strongly normalizing CRS reduction relation.

We provide a new proof of correctness of fully lazy λ-lifting by showing that the transformation preserves reduction sequences: each single reduction step in the source (resp. target) system is simulated by exactly one single step in the target (resp. source) system. The proof is small-step: the reduction sequences are proved to be preserved in every intermediate step of the transformation. Moreover, we prove that the notion of optimal sharing is also preserved, which has two consequences:

- The direct [19] and the λ-lifting based [10] approaches of full laziness are reduction-wise equivalent.
- Fully-lazy λ-lifting establishes a link between optimal sharing in the weak λ-calculus [4] and the better known optimality theory of first-order rewriting [14, 17]. This emphasizes in a new way the "first-order" nature of weak reduction, without any de Bruijn indices or explicit substitutions (contrary to [14]).

A final bonus remark is an incidental point which happens to have some theoretical significance:

while β-reduction and λ-lifting considered separately can be seen as orthogonal systems[2], their combination cannot. As far as the author is aware, the system derived in this paper is the first successful optimality-oriented labelling of a non-orthogonal system.

References

[1] Z.M. Ariola & M. Felleisen (1997): *The Call-By-Need lambda Calculus*. *J. Funct. Program.* 7(3), pp. 265–301.

[2] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky & P. Wadler (1995): *The Call-by-Need Lambda Calculus*. In: *POPL*, pp. 233–246.

[3] T. Balabonski (2012): *A Unified Approach to Fully Lazy Sharing*. In: *POPL*, pp. 469–480.

[4] T. Blanc, J.-J. Lévy & L. Maranget (2007): *Sharing in the Weak Lambda-Calculus Revisited*. In: *Reflections on Type Theory, Lambda Calculus and the Mind*.

[5] H.J.S. Bruggink (2003): *Residuals in Higher-Order Rewriting*. In: *RTA*, pp. 123–137.

[6] N. Çağman & J. R. Hindley (1998): *Combinatory Weak Reduction in Lambda Calculus*. *TCS* 198(1-2), pp. 239–247.

[7] D. Dougherty, P. Lescanne, L. Liquori & F. Lang (2005): *Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics: Extended Abstract*. *ENTCS* 127(5), pp. 57–82.

[8] R.J.M. Hughes (1982): *Super Combinators: A New Implementation Method for Applicative Languages*. In: *Symposium on LISP and Functional Programming*, pp. 1–10.

[9] T. Johnsson (1985): *Lambda Lifting: Transforming Programs to Recursive Equations*. pp. 190–203.

[10] S. Peyton Jones (1987): *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc.

[11] J.W. Klop, V. van Oostrom & F. van Raamsdonk (1993): *Combinatory Reduction Systems: Introduction and Survey*. *Theor. Comput. Sci.* 121(1&2), pp. 279–308.

[12] U. Dal Lago & S. Martini (2009): *On Constructor Rewrite Systems and the Lambda-Calculus*. In: *ICALP (2)*, pp. 163–174.

[13] J. Launchbury (1993): *A Natural Semantics for Lazy Evaluation*. In: *POPL*, pp. 144–154.

[14] L. Maranget (1991): *Optimal Derivations in Weak Lambda-calculi and in Orthogonal Terms Rewriting Systems*. In: *POPL*, pp. 255–269.

[15] P. Sestoft (1997): *Deriving a Lazy Abstract Machine*. *J. Funct. Program.* 7(3), pp. 231–264.

[16] O. Shivers & M. Wand (2004): *Bottom-up β-reduction: Uplinks and λ-DAGs*. Technical Report RS-04-38, BRICS.

[17] Terese (2003): *Term Rewriting Systems*. Cambridge Univ.Press.

[18] D.A. Turner (1979): *A new implementation technique for applicative languages*. *Software: Practice and Experience* 9(1), pp. 31–49.

[19] C. P. Wadsworth (1971): *Semantics and Pragmatics of the Lambda Calculus*. Ph.D. thesis.

[2] in brief, a system is orthogonal when no two rules are applicable to overlapping sets of positions of a term, see for instance [17, 5]