Theoretical Computer Science 121 (1993) 279–308
Elsevier

Combinatory reduction systems: introduction and survey

Jan Willem Klop*

CWI, P.O. Box 94079, 1090 GB Amsterdam, Department of Mathematics and Computer Science, Free University, de Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

Vincent van Oostrom

Department of Mathematics and Computer Science, Free University, de Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

Femke van Raamsdonk**

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Dedicated to Corrado Böhm

Abstract

Klop, J.W., V. van Oostrom and F. van Raamsdonk, Combinatory reduction systems: introduction and survey, Theoretical Computer Science 121 (1993) 279–308.

Combinatory reduction systems, or CRSs for short, were designed to combine the usual first-order format of term rewriting with the presence of bound variables as in pure λ-calculus and various typed λ-calculi. Bound variables are also present in many other rewrite systems, such as systems with simplification rules for proof normalization. The original idea of CRSs is due to Aczel, who introduced a restricted class of CRSs and, under the assumption of orthogonality, proved confluence. Orthogonality means that the rules are nonambiguous (no overlap leading to a critical pair) and left-linear (no global comparison of terms necessary).

We introduce the class of orthogonal CRSs, illustrated with many examples, discuss its expressive power, and give an outline of a short proof of confluence. This proof is a direct generalization of Aczel's original proof, which is close to the well-known confluence proof for λ-calculus by Tait and Martin-Löf. There is a well-known connection between the parallel reduction featuring in the latter proof and the concept of "developments", and a classical lemma in the theory of λ-calculus is that of "finite developments", a strong normalization result. It turns out that the notion of "parallel reduction" used in Aczel's proof gives rise to a generalized form of developments which we call "superdevelopments" and on which we will briefly comment.

Correspondence to: J.W. Klop, CWI, P.O. Box 94079, 1090 GB Amsterdam, Department of Mathematics and Computer Science, Free University, de Boelelaan 1081, 1081 HV Amsterdam, The Netherlands. Email addresses of the authors: jwk@cwi.nl, oostrom@cs.vu.nl, femke@cwi.nl.

* Partially supported by ESPRIT BRA 6454 CONFER (CONcurrency and Functions: Evaluation and Reduction).

** Research done in the framework of NWO/SION project 612-316-606 "Extensions of orthogonal rewrite systems – syntactic properties".

We conclude with mentioning the results of a comparison of CRSs with the recently proposed and strongly related format of higher-order rewriting: Nipkow's HRSs (higher-order rewrite systems).

1. Introduction

We start in a somewhat informal way by discussing various issues of term rewriting with bound variables, or "higher-order rewriting" as it is often called nowadays. This is done in Sections 2–10. These sections are intended to give a gentle introduction to combinatory reduction systems (CRSs). In Sections 11–12 we give the formal (and quite lengthy) definition of CRSs. Section 13 contains an outline of a short confluence proof for orthogonal CRSs and a brief discussion of "superdevelopments". Section 14 mentions related work, and compares CRSs with the higher-order rewrite systems introduced by Nipkow [37]. Section 15 concludes with a discussion of current research issues for CRSs. The appendix presents several "large" examples of orthogonal CRSs, such as polymorphic second-order λ-calculus.

2. Definable extensions of λ-calculus

Although λ-calculus is able to define many data types, such as natural numbers with arithmetic operators, it is often more convenient to construct an extension of λ-calculus where such data types are explicitly added. Thus, one may consider, e.g., λ-calculus plus pairing given by the reduction or rewrite rules

$$(\lambda x.M)N \to M[x := N],$$

$$\mathtt{left}(\mathtt{pair}\, MN) \to M,$$

$$\mathtt{right}(\mathtt{pair}\, MN) \to N.$$

The reduction system can be simulated in "pure" untyped λ-calculus by taking the following terms: $\mathtt{left} := \lambda p.p(\lambda mn.m)$, $\mathtt{right} := \lambda p.p(\lambda mn.n)$, $\mathtt{pair} := \lambda mnz.zmn$. This translation has the property that $\mathtt{left}(\mathtt{pair}\, MN) \twoheadrightarrow M$ and $\mathtt{right}(\mathtt{pair}\, MN) \twoheadrightarrow N$ for all M and N, i.e. every step in the original system is simulated by a finite reduction sequence in λ-calculus. We will call an extension like this a (directly) definable extension of λ-calculus. It seems a natural minimal requirement for an extension to be definable that reduction can be simulated. Minimal but not sufficient. The encoding should not be too liberal. Consider for instance the reduction rule:

$$\mathtt{compare}\, MM \to \mathtt{equal}.$$

Reduction according to this rule can be simulated in λ-calculus by taking: $\mathtt{compare} := \lambda xy.I$ with $I = \lambda x.x$ and $\mathtt{equal} := I$. Then we have indeed $\mathtt{compare}\, MM \twoheadrightarrow \mathtt{equal}$. However, we also have $\mathtt{compare}\, MN \twoheadrightarrow \mathtt{equal}$, for all M, N.

This illustrates that a more sophisticated notion of definability has to be developed, which we will not attempt to do in the present paper. We claim that the translations presented in this paper are not too liberal.

3. Proper extensions of λ-calculus

One might wonder whether every reduction system consisting of λ-calculus extended with term rewriting rules is definable in λ-calculus. The `compare` rule of the previous section is a typical example of a reduction system for which this is not the case (for a reasonable notion of definability, cf. [30]).

If we add the rule

$$\mathtt{pair}(\mathtt{left}\, M)(\mathtt{right}\, M) \to M$$

to the `pair` rules, the system is no longer a directly definable one. Hence, this extension (called λ-calculus with surjective pairing) is a *proper* extension of λ-calculus. This has been proved by Barendregt [4].

In both cases the problem is the double occurrence of the meta-variable M in the left-hand side of a rule. Such a rule is called "not left-linear".

An example of another kind of a reduction system that cannot be defined in λ-calculus is obtained by adding the rules for parallel or:

$$\mathtt{or}\ M\ \mathtt{true} \to \mathtt{true},$$

$$\mathtt{or}\ \mathtt{true}\ M \to \mathtt{true}.$$

Again, there is no λ-term `or` implementing these rules in the direct sense given above. Now the problem is not non-left-linearity, but the inherent parallelism in the rules for `or`; and λ-calculus has a sequential evaluation [9].

4. λ-rewrite systems

Here we are not concerned with a study of definability in λ-calculus, which is an issue that has not yet been explored extensively. For recent progress on this subject, see [8]. However, the three examples of the previous section show that it is worthwhile to study extensions of λ-calculus with term rewriting rules. Let us indicate λ-calculus, with as only rule the one of β-reduction, by λ and abbreviate a term rewriting system without bound variables as TRS. A combination of λ-calculus and some TRS will be called a λ-TRS. They may be of two kinds: the ones where λ and the TRS have disjoint alphabets, in which case we denote by $\lambda \oplus R$ the extension of λ with the TRS R, and the ones where R contains the application operator just as λ, in which case we write $\lambda \cup R$. The three examples of extensions of λ-calculus are of the latter kind and illustrate the expressiveness of the class of λ-TRSs. We note that, in recent years, several studies have appeared of extensions of various typed λ-calculi with ordinary term rewriting rules, sometimes called "algebraic rewriting" [10, 11].

5. Metavariables with arity

In the next section we will investigate the expressiveness of λ-TRSs. We will especially be concerned with the study of rules with bound variables. In this section a notational device is introduced for writing rules with binding structures in an easy way.

In informal discussions on λ-calculus, one sometimes uses the sloppy but intuitively clear and convenient notation for the β-reduction rule: $(\lambda x.M(x))N \to M(N)$, instead of the usual notation given above employing the explicit substitution operator $[x := N]$. The sloppiness is in the use of $M(N)$: on its own this notation does not make sense, only in the context of having stated "let M be $M(x)$", as is done by writing $(\lambda x.M(x))N$, does it make sense to employ $M(N)$, then meaning $M[x := N]$. However, in the sequel we will give a perfectly rigorous semantics to this, up to now, sloppy notation.

This leads (after Aczel [2]) to the introduction of metavariables *with arity*. For example, $M(x)$ is a unary metavariable. Also, we will employ henceforth a special notation for metavariables: Z_k^n, where n denotes the arity, $n \geqslant 0$, and $k \geqslant 0$ is an enumerating index. For ease of reading however, we will just write $Z, Z', Z'', \ldots$, omitting the arity indication which is clear from the use of these metavariables. For the variables intended to be bound by some "quantifier" (or rather, "qualifier" as it qualifies the intention of how the binding is used) such as λ, μ, or indeed $\forall$, $\exists$, we write $x, y, z, \ldots$ For example, λ-calculus with surjective pairing now takes the following more pleasing form:

$$(\lambda x.Z(x))Z' \to Z(Z'),$$

$$\mathtt{left}(\mathtt{pair}\,ZZ') \to Z,$$

$$\mathtt{right}(\mathtt{pair}\,ZZ') \to Z',$$

$$\mathtt{pair}(\mathtt{left}\,Z)(\mathtt{right}\,Z) \to Z.$$

A feature of this notation is that it allows expression of a simple but frequently occurring type of side-condition. For example, the η-rule of λ-calculus is written as

$$\lambda x.Zx \to Z.$$

Usually, stating the η-rule, one adds the restriction "provided x does not occur in Z". However, our formal definition (Section 11) of the kind of rules we are introducing makes this superfluous: an instantiation of Z in $\lambda x.Zx$ will by definition not have free occurrences of x.

An example involving n-ary metavariables ("n-ary β-reduction") is

$$(\lambda x_1 \ldots x_n . Z(x_1, \ldots, x_n)) Z_1 \ldots Z_n \rightarrow Z(Z_1, \ldots, Z_n).$$

A pathological one, suggesting the ease of writing iterated substitutions, is

$$\sigma xy.\lambda z.zZ(x)Z'(y) \rightarrow Z(Z'(Z(Z'(\lambda z.z)))).$$

Note that, as in the case of the η-rule, an instance of $Z(x)$ is not allowed to contain free occurrences of y or of z, and instances of $Z'(y)$ are not allowed to contain free x's or z's.

6. Extensions of λ-calculus with rules with bound variables

Besides extensions of λ-calculus there are various other examples of rewrite systems with bound variables in which the feature of bound variables may be used in quite a different way. For example

$$\mu x.M \rightarrow M[x := \mu x.M]$$

as in the operational semantics for recursively defined concepts (e.g. in recursive procedures as in [15] and in processes defined by recursion [34]). In the notation just introduced, this rule is written as

$$\mu x.Z(x) \rightarrow Z(\mu x.Z(x)).$$

This rule is definable in pure λ-calculus by defining $\mu x.Z(x)$ as $Y_T(\lambda x.Z(x))$, with $Y_T = (\lambda xf.f(xxf))(\lambda xf.f(xxf))$, Turing's fixed point combinator. Indeed, we then have

$$\begin{aligned} \mu x.Z(x) &= Y_T(\lambda x.Z(x)) \\ &\twoheadrightarrow Z(Y_T(\lambda x.Z(x))) \\ &= Z(\mu x.Z(x)). \end{aligned}$$

Par abus de langage, let us say that we have defined μ by $Y_T\lambda$. In the precise CRS format below, μ is in fact defined by $BY_T\lambda$, where B is the composition combinator $(\lambda xyz.x(yz)$. Usually instead of B the infix notation employing $\circ$ is used, rendering μ as $Y_T \circ \lambda$.

Another example stems from proof theory. There one is concerned with proof normalization (cf. [41, 19]):

$$P(LZ)(\lambda x.Z'(x))(\lambda x.Z''(x)) \rightarrow Z'(Z),$$

$$P(RZ)(\lambda x.Z'(x))(\lambda x.Z''(x)) \rightarrow Z''(Z).$$

These rules are easily defined in λ (e.g. by taking $P = \lambda x.x$, $L = \lambda xyz.yx$ and $R = \lambda xyz.zx$). Also the pathological rule $\sigma xy.\lambda z.zZ(x)Z'(y) \rightarrow Z(Z'(Z(Z'(\lambda z.z))))$ can easily be defined in λ.

7. Definable extensions of λ-TRSs

Consider the following reduction system with rules with bound variables.

$$\gamma xy.F(x, y, Z(x, y)) \to C,$$

$$\gamma xy.F(Z(x, y), x, y) \to C,$$

$$\gamma xy.F(y, Z(x, y), x) \to C.$$

These γ-rules are immediately obtained, once we have at our disposal the TRS $\mathscr{F}$ with rewrite rules:

$$F(A, B, Z) \to C,$$

$$F(Z, A, B) \to C,$$

$$F(B, Z, A) \to C.$$

Then, putting $G = \lambda z.zAB$ we have in $\lambda \oplus \mathscr{F}$ the reduction $G(\lambda xy.F(x, y, Z(x, y))) \twoheadrightarrow C$, and similarly for the other two rules for γ; hence, we can define γ as $G\lambda$.

8. Proper extensions of λ-TRSs

With λ-TRSs as reduction format at our disposal, one can ask whether every system involving pattern matching and binding of variables can be written as a λ-TRS. This would mean that all reduction sequences could be neatly separated into a λ-part (β-reduction) and a pattern matching part (first-order term rewriting as in a TRS). It would be interesting if this were indeed the case. However, if binding structures for variables are used in other ways than for expressing a substitution mechanism, then we doubt that they can always be expressed by means of a λ-TRS. Two examples feeding this doubt are

$$\lambda x.Zx \to Z,$$

$$\rho x.xZ(x) \to Z(\Omega),$$

where $\Omega = (\lambda x.xx)(\lambda x.xx)$. With reference to the second rule (which is our preferred example since in combination with the β-rule of λ-calculus it is still orthogonal), the question is whether a λ-term R exists such that

$$R(\lambda x.xZ(x)) \twoheadrightarrow Z(\Omega).$$

We conjecture that such an R does not exist, also in the case when operators from a TRS (without bound variables) are used. The point is that $Z(x)$ cannot be extracted from the application $xZ(x)$, and trying to get rid of the prefixed x by some substitution also disturbs $Z(x)$ irreversibly (see the proof idea below). Note that it would be easy to find an R' such that $R'(\lambda x.xZ(x)) \twoheadrightarrow Z(I)$, where $I = \lambda x.x$. The same holds with $K = \lambda xy.x$ instead of I. Actually, if we admit an extension with a TRS containing application, we can extract $Z(x)$ from $xZ(x)$, namely by using an operator J with (in applicative notation) the rule $J(Z_1Z_2) \to Z_2$; but the extension would be inconsistent in the sense of making all terms interconvertible, as an easy exercise in λ-calculus shows.

Proof idea. Take $Z(x) = \Omega^n x = \Omega \ldots \Omega x$ (n times Ω). Now suppose there exists an R such that for all n we have $R(\lambda x.x(\Omega^n x)) \twoheadrightarrow Z(\Omega) = \Omega^{n+1}$. This reduction must have the form

$$R(\lambda x.x(\Omega^n x)) \twoheadrightarrow (\lambda x.x(\Omega^n x))S_1 \ldots S_k \to S_1(\Omega^n S_1)S_2 \ldots S_k$$
$$\twoheadrightarrow \Omega^n S_1 T_1 \ldots T_p S_2 \ldots S_k \twoheadrightarrow \Omega^{(n+1)}.$$

This is only possible if $p=0$, $k=1$ and $S_1 \twoheadrightarrow \Omega$, contradicting the fact that S_1 must have a head normal form. (It will require a lot of work to make this argument rigorous, also because of the allowed presence of TRS operators.)

Another example of a system with a curious use of bound variables is

$$\alpha x.\mathtt{or}(Z, x) \to Z,$$
$$\alpha x.\mathtt{or}(x, Z) \to Z.$$

As these examples illustrate, it seems very reasonable, if not necessary, to consider reduction systems more general than λ-TRSs. A format of this type, combining term rewriting and binding structures for variables, has been developed in [30], generalizing an idea of Aczel [2]. The resulting CRSs employ a notation of metavariables with arity. The following rules constitute an example of a CRS:

$$(\lambda x.Z(x))Z' \to Z(Z'),$$
$$\mathtt{left}(\mathtt{pair}\,ZZ') \to Z,$$
$$\mathtt{right}(\mathtt{pair}\,ZZ') \to Z',$$
$$\mu x.Z(x) \to Z(\mu x.Z(x)),$$
$$P(LZ)(\lambda x.Z'(x))(\lambda x.Z''(x)) \to Z'(Z),$$
$$P(RZ)(\lambda x.Z'(x))(\lambda x.Z''(x)) \to Z''(Z),$$
$$\sigma xy.\lambda z.zZ(x)Z'(y) \to Z(Z'(Z(Z'(\lambda z.z)))),$$
$$\gamma xy.F(x, y, Z(x, y)) \to C,$$
$$\gamma xy.F(Z(x, y), x, y) \to C,$$
$$\gamma xy.F(y, Z(x, y), x) \to C,$$
$$\rho x.xZ(x) \to Z(\Omega),$$
$$\alpha x.\mathtt{or}(Z, x) \to Z,$$
$$\alpha x.\mathtt{or}(x, Z) \to Z.$$

A formal definition of CRSs can be found in Section 11.

9. Orthogonality

We call a CRS *orthogonal* when its rewrite rules are independent of each other. More precisely: suppose that R and S are redexes in M, such that R contains the redex S. Suppose R is in fact an r-redex, where r is the name of a rewrite rule. Then we require, for orthogonality, that contraction of S does not affect the r-redex status of the subterm R' resulting from R. How can we guarantee this? By imposing the following two requirements:

(1) The CRS does not contain rules with a left-hand side in which some metavariable has multiple occurrences; in other words, the rules must be *left-linear*.

(2) Whenever a redex R contains a subredex S, then S must in fact be contained in one of the instantiated metavariables of the rule according to which R is a redex. In other words, the rules are *nonoverlapping*.

As to (1), note that multiple occurrences of bound variables in a left-hand side of a rule are allowed.

9.1. Examples

The CRS of the previous section is orthogonal. The one-rule system consisting of $\lambda x.Zx \to Z$ is orthogonal. However, $\lambda\beta\eta$-calculus, consisting of the two rules

$$(\lambda x.Z(x))Z' \to Z(Z'),$$

$$\lambda x.Zx \to Z$$

is overlapping, and hence not orthogonal. The following underlined terms suggest the overlaps:

$$\underline{(\lambda x.Z(x))}Z'.$$

The underlined part, not contained in a metavariable, may be instantiated to an η-redex.

$$\lambda x.\underline{Zx}.$$

The underlined part, not contained in a metavariable, may be instantiated to a β-redex.

The rules $\alpha x.\mathtt{or}(Z, x) \to Z$ and $\alpha x.\mathtt{or}(x, Z) \to Z$ exhibit a curious phenomenon. They are seemingly overlapping, namely, by instantiating Z to x in both left-hand sides. However, this is not allowed; legitimate instantiation of Z has no free occurrences of x, because these occurrences would be bound by αx. This will be clearer after introducing CRSs formally. Here we conclude that the rules for α are, surprisingly, nonoverlapping.

The rules $\lambda xy.F(Z(x,y))\to 0$, $\lambda xy.F(Z(y,x))\to 1$ are overlapping. Note that different instantiations may be used to show the overlap.

The rules $\lambda xy.F(x,Z(y))\to 0$, $\lambda xy.F(y,Z(x))\to 1$ are orthogonal. The rule $\lambda x\lambda y.Z(x,y)\to 0$ is self-overlapping.

10. Substructures

The λ-calculus is a "full" rewrite system since the inductive clauses describing the formation of terms are not subject to any restriction. There are useful "substructures" of λ-calculus where the term formation clauses do have some restrictions. A well-known example is the λI-calculus, where the abstraction clause reads: if M is a λI-term, then $\lambda x.M$ is a λI-term provided x occurs at least once freely in M. Another substructure of λ is given by the set of strongly normalizing terms (terms not admitting an infinite reduction); another by the set of weakly normalizing terms (terms having a normal form). A fourth example is the set of terms which are simply typable. All these substructures are closed under reduction; that is, when M is a term in the domain of the substructure, then all its reducts are also. We will take this property as the defining property for a substructure. In the theory of typed λ-calculi it is known as the *subject reduction property* (see also [7, Definition 12.9]).

Next to "full" CRSs, we now also admit all its substructures as CRSs. We will call CRSs which are not full (which have restricted term formation) *restricted* CRSs.

Since we are almost exclusively interested in the "reduction theory" of CRSs (rather than the equality theory, or convertibility theory), almost all propositions proved for full CRSs also hold for restricted CRSs. For instance, when a full CRS is confluent, all its sub-CRSs are also confluent. The only property we know which is sensitive for the difference between full and restricted is as follows. (See Fig. 1.)

Theorem 10.1. *Let R be an orthogonal full CRS. Let M be a term in R having a normal form N but also admitting an infinite reduction. Then N has an infinite expansion, i.e. an inverse reduction.*

For a proof, see [30]. Obviously, this "N-property" does not hold in general for restricted orthogonal CRSs, since the set of terms need not be closed under expansion (inverse reduction).

Admitting substructures as CRSs has an important consequence; i.e. the equivalence of the so-called applicative notation and the functional notation for TRSs and CRSs, as follows. In most of the examples mentioned above, we employed the applicative style of notation which is well known from λ-calculus and Combinatory Logic. (Instead of "applicative" one can also use the word "curried".) In an applicative system there is one binary operation @, application and all other operators are 0-ary, i.e. constants. The usual notation is to write (ts) instead of $@(t,s)$, and one adopts the well-known convention of "association to the left", to restore missing bracket pairs. In

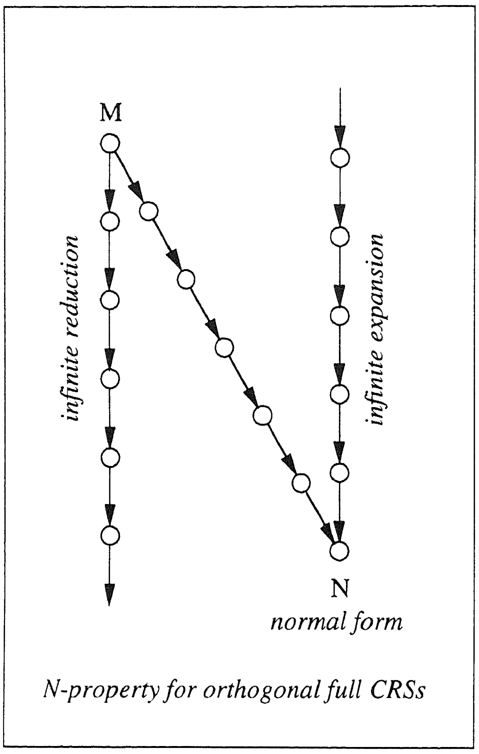

Fig. 1.

general systems there may be operators of any arity. We will also call general systems "functional" systems. So clearly the applicative systems form a subclass of the functional systems. Therefore, the question arises: is the functional notation more expressive than the applicative notation, or in other words, is the class of functional systems essentially larger than that of applicative systems? In some places in the literature this seems to be suggested. However, the answer is negative, once we have the notion of subsystem (sub-CRS) available, as introduced above (and more precisely below).

Example 10.2. Consider the functional TRS R:

$$A(x, 0) \to 0,$$

$$A(x, S(y)) \to S(A(x, y)),$$

defining addition A in terms of 0 and successor S. The applicative version R^{ap} of R is

$$Ax0 \to 0,$$

$$Ax(Sy) \to S(Axy),$$

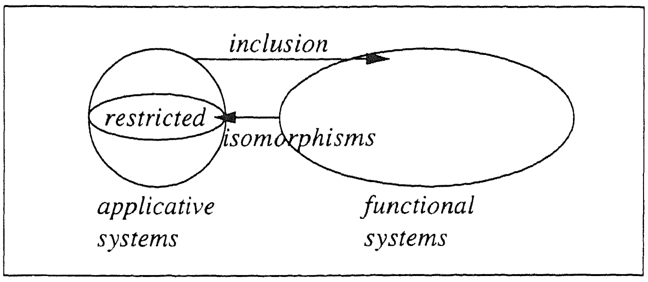

Fig. 2.

where the usual applicative notation (as in CL, Combinatory Logic) is used. That is, $Ax0$ is short for $@(@(A,x),0)$, where $@$ is application. Clearly, R^{ap} is not isomorphic to R, as there are "surplus" terms such as $A0$ or $A000$ or AAA that have no counterpart in R. But R is isomorphic to a substructure of R^{ap}, with terms that are inductively defined by
- $x, y, \ldots, 0$ is a term,
- if t, s are terms then Ats is a term,
- if t is a term then St is a term.

It is clear that, in general, a functional system is isomorphic in this way to a restricted applicative system (see Fig. 2). Thus, the styles of applicative and functional notations are equivalent and equally expressive.

11. Formal definition of a combinatory reduction system

11.1. Alphabet of a combinatory reduction system

A CRS is a pair consisting of an alphabet and a set of rewrite rules. In a CRS a distinction is made between metaterms and terms. The left- and right-hand side of a rule are metaterms, and rules act upon terms. This distinction is made in order to stress the point that a reduction rule acts as a scheme, so its left- and right-hand side are not ordinary terms. For instance, in a term rewriting system, $F(x)$ as a term is something different from $F(x)$ as the left-hand side of a reduction rule. In CRSs, metaterms occur only as the left- or right-hand side of a reduction rule. They may contain metavariables that indicate a position in a reduction rule where an arbitrary term can be substituted. Terms do not contain metavariables, but may contain variables. Taking this point of view, x in $F(x)$ as a term is a variable, and x in $F(x)$ as a left- or right-hand side of a reduction rule is a metavariable. In CRS notation, the former is written as $F(x)$ and the latter as $F(Z)$.

The alphabet of a CRS consists of

(1) a set $\mathrm{Var} = \{x_n \mid n \geqslant 0\}$ of *variables* (also written as $x, y, z, \ldots$),

(2) a set Mvar of *metavariables* $\{Z_n^k \mid k, n \geqslant 0\}$ (here k is the *arity* of Z_n^k),
(3) a set of *function symbols*, each with a fixed arity,
(4) a binary operator for abstraction, written as $[\_]\_$,
(5) improper symbols '(', ')' and ','.

The arities k of the metavariables Z_n^k can always be read off from the metaterm in which they occur – hence we will often suppress these superscripts. For example, in $(\lambda x.Z_0(x))Z_1$ the Z_0 is unary and Z_1 is 0-ary.

11.2. Term formation in a combinatory reduction system

Definition 11.1. The set MTerms of metaterms of a CRS with an alphabet as in Section 11.1 is defined inductively as follows:
(1) variables are metaterms;
(2) if t is a metaterm and x a variable, then $[x]t$ is a metaterm, obtained by *abstraction*;
(3) if F is an n-ary function symbol ($n \geqslant 0$) and $t_1, \ldots, t_n$ are metaterms, then $F(t_1, \ldots, t_n)$ is a metaterm;
(4) if $t_1, \ldots, t_k$ ($k \geqslant 0$) are metaterms, then $Z_n^k(t_1, \ldots, t_k)$ is a metaterm (in particular the Z_n^0 are metaterms).

Note that metavariables Z_n^{k+1} with arity >0 are not metaterms; they need arguments. Metaterms without metavariables are terms. The set of terms is denoted as Terms.

Notation.
(1) An iterated abstraction metaterm $[x_1]\ldots[x_{n-1}][x_n]t$ is written as $[x_1, \ldots, x_n]t$. For a unary function symbol F, we will often write $Fx_1 \ldots x_n.t$ instead of $F([x_1, \ldots, x_n]t)$. For instance, $\lambda x.t$ abbreviates $\lambda([x]t)$.
(2) We will adopt the following conventions:
- All occurrences of abstractors $[x_i]$ in a metaterm or term are different; e.g. $\lambda([x][x].t)$ is not legitimate, nor is $\lambda([x].@(t, \lambda([x].t')))$.
- Furthermore, terms differing only by a renaming of bound variables are considered syntactically equal. (The notion of "bound" is as in λ-calculus: an occurrence of a variable x is *bound* if it is in the scope of an abstractor $[x]$. It is *free* otherwise).

Definition 11.2. A (meta)term is *closed* if every variable occurrence is bound.

11.3. Rewrite rules of a combinatory reduction system

A *rewrite* (or *reduction*) rule in a CRS is a pair (s, t), written as $s \to t$, where s and t are metaterms such that:
(1) s and t are closed metaterms;
(2) s has the form $F(t_1, \ldots, t_n)$;

(3) the metavariables Z_n^k that occur in t also occur in s;

(4) the metavariables Z_n^k in s occur only in the form $Z_n^k(x_1, \ldots, x_k)$, where the x_i $(i=1, \ldots, k)$ are variables. Moreover, the x_i are pairwise distinct.

If, moreover, no metavariable Z_n^k occurs twice or more in s, the rewrite rule $s \to t$ is called *left-linear*.

Example 11.3. $@(\lambda([x]Z(x)), Z') \to Z(Z')$ is the left-linear rule of β-reduction in λ-calculus. Application is here expressed by the binary function symbol @.

12. Extracting the reduction relation

It requires some subtlety to extract from the rewrite rules the actual rewrite relation that they generate. First we define *substitutes* (we adopt this name from Kahrs [25]).

Definition 12.1. Let t be a term.

(1) Let $(x_1, \ldots, x_n)$ be an n-tuple of pairwise distinct variables. Then the expression $\underline{\lambda}(x_1, \ldots, x_n).t$ is an n-ary substitute. We use $\underline{\lambda}$ as a "metalambda" to distinguish it from the one of λ-calculus.

(2) The variables $x_1, \ldots, x_n$ occurring in t are bound in the substitute $\underline{\lambda}(x_1, \ldots, x_n).t$. They may be renamed in the usual way, provided no name clashes occur. Renamed versions of a substitute are considered identical. The free variables in $\underline{\lambda}(x_1, \ldots, x_n).t$ are the free variables of t except $x_1, \ldots, x_n$.

(3) An n-ary substitute $\underline{\lambda}(x_1, \ldots, x_n).t$ may be applied to an n-tuple $(t_1, \ldots, t_n)$ of terms from the CRS, resulting in the following simultaneous substitution:

$$(\underline{\lambda}(x_1, \ldots, x_n).t)(t_1, \ldots, t_n) = t[x_1 := t_1, \ldots, x_n := t_n].$$

Definition 12.2. A *valuation* is a map σ assigning to an n-ary metavariable Z an n-ary substitute:

$$\sigma(Z) = \underline{\lambda}(x_1, \ldots, x_n).t.$$

Valuations are extended to a homomorphism on metaterms as follows:

(1) $\sigma(x) = x$ for $x \in Var$;

(2) $\sigma([x]t) = [x]\sigma(t)$;

(3) $\sigma(F(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n))$;

(4) $\sigma(Z(t_1, \ldots, t_n)) = \sigma(Z)(\sigma(t_1), \ldots, \sigma(t_n))$.

So if $\sigma(Z) = \underline{\lambda}(x_1, \ldots, x_n).t$, then $\sigma(Z(t_1, \ldots, t_n)) = t[x_1 := \sigma(t_1), \ldots, x_n := \sigma(t_n)]$.

We will now formulate some "safety conditions" for instantiating rewrite rules to actual rewrite steps. Intuitively, we could summarize their description as follows: rename bound variables as much as possible in order to avoid name clashes, i.e. free variables x being captured unintentionally by abstractors $[x]$.

Definition 12.3. (1) Let $s \to t$ be a rewrite rule. A renaming of that rule (by renaming the bound variables in s, t) will be called a *variant* of the rule.

(2) Let σ be a valuation. Then a *variant* of σ originates by renaming the bound variables in the substitutes $\sigma(Z)$.

(3) Let $s \to t$ be a rewrite rule and σ a valuation. Then $s \to t$ is called *safe for* σ if, for no Z in s and t, the substitute $\sigma(Z)$ has a free variable x occurring in an abstraction $[x]$ of s or t.

(4) Furthermore, σ is called safe (with respect to itself) if there are no two substitutes $\sigma(Z)$ and $\sigma(Z')$ such that $\sigma(Z)$ contains a free variable x which appears also bound in $\sigma(Z')$.

Note that for every rewrite rule $s \to t$ and valuation σ there are variants σ' and $s' \to t'$ such that σ' is safe and $s' \to t'$ is safe for σ'. In the following we will suppose that all valuations are safe with respect to themselves and with respect to the reduction rules to which they are applied.

Example 12.4. The η-reduction rule variant $\lambda x.Zx \to Z$, or in full notation written as $\lambda([x]@(Z,x)) \to Z$, is not safe for σ with $\sigma(Z)=x$. The variant $\lambda y.Zy \to Z$ is safe for σ.

Definition 12.5. Let $\Box$ be a fresh symbol. A term with one or more occurrences of $\Box$ is called a *context*. A context with n occurrences of $\Box$ is written as $C[\ldots]$, and one with exactly one occurrence of $\Box$ as $C[\ ]$. The result of replacing the n occurrences of $\Box$ from left to right by terms $t_1, \ldots, t_n$ is written as $C[t_1, \ldots, t_n]$. We call s a *subterm* of t if there exists a context $C[\ ]$ such that $t = C[s]$.

Definition 12.6. (1) Let $s \to t$ be a rewrite rule version which is safe for the safe valuation σ. Then $\sigma(s) \to \sigma(t)$ is called a *rewrite* or *contraction*. The term $\sigma(s)$ is called a *redex*.

(2) Let $\sigma(s) \to \sigma(t)$ be a rewrite, and $C[\ ]$ a context. Then $C[\sigma(s)] \to C[\sigma(t)]$ is called a *rewrite step* (or reduction step).

(3) $\twoheadrightarrow$ is the reflexive–transitive closure of the one step rewrite relation $\to$ on terms. If $s \twoheadrightarrow t$ then we say that s *reduces* to t and t is called a *reduct* of s.

Remark. We need $s \to t$ to be safe for σ to prevent variable capture when evaluating the left-hand side of the rule. We need σ to be safe (with respect to itself) because otherwise undesired variable captures take place in evaluating the right-hand sides of rules. For example, consider $Z(Z')$ with σ such that $\sigma(Z) = \underline{\lambda}(y).(\lambda x.xy)$ and $\sigma(Z') = x$ (so σ is not safe). Then $\sigma(Z(Z')) = \sigma(Z)(\sigma(Z')) = (\underline{\lambda}(y).(\lambda x.xy))(x) = \lambda x.xx$, with variable capture. Note that free variables in the rewrite $\sigma(s) \to \sigma(t)$ may be captured by the context $C[\ ]$ in which it is embedded to form a rewrite step $C[\sigma(s)] \to C[\sigma(t)]$; but that is intended!

Example 12.7. In this example we write t^σ instead of $\sigma(t)$. We reconstruct a step according to the β-reduction rule of λ-calculus (written in the usual, applicative, notation):

$$(\lambda x.Z(x))Z' \to Z(Z').$$

Let the valuation $Z^\sigma = \underline{\lambda}(u).yuu$, $Z'^\sigma = ab$ be given. Then we have the reduction step:

$$\begin{aligned}((\lambda x.Z(x))Z')^\sigma &= (\lambda x.Z(x)^\sigma)Z'^\sigma \\ &= (\lambda x.Z^\sigma(x^\sigma))Z'^\sigma \\ &= (\lambda x.(\underline{\lambda}u.yuu)(x))(ab) \\ &= (\lambda x.yxx)(ab)\end{aligned}$$

$$\to$$

$$\begin{aligned}(Z(Z'))^\sigma &= Z^\sigma(Z'^\sigma) \\ &= (\underline{\lambda}u.yuu)(ab) \\ &= y(ab)(ab).\end{aligned}$$

Note that in the CRS format there is no need for explicitly requiring that some variables are not allowed to occur in instances of metavariables. For instance, in $F([x]Z)$, an instance of Z is not allowed to contain free occurrences of x. In λ-calculus such a requirement cannot be made in the system itself; it has to be stated in the metalanguage, as is done for the η-rule. In this sense the CRS formalism is more expressive than that of λ-calculus.

This requirement, discussed in Section 12.3 (3), is necessary: consider e.g. the rule $\tau x.xZ \to Z$. Suppose we did not require that Z cannot have free occurrences of x. Then $\tau x.xx \to x$; but that would mean that a closed term rewrites to an open term, i.e. free variables appear out of the blue, which of course is disallowed. One may ask why this is not the case for the rule $\tau x.xZ(x) \to Z(x)$; the answer is that this is not a legitimate rule because the right-hand side is not a closed metaterm.

We will now give a more precise definition of overlap and orthogonality.

Definition 12.8. Let R be a CRS containing rewrite rules $\{r_i = s_i \to t_i \mid i \in I\}$.

(1) R is *nonoverlapping* if the following holds:

- Let the left-hand side s_i of r_i be in fact $s_i(Z_1(\bar{x}_1), \ldots, Z_m(\bar{x}_m))$ where all metavariables in s_i are displayed and $\bar{x}_i$ is short for $(x_{i_1}, \ldots, x_{i_{ki}})$ with k_i the arity of Z_i. Now if the r_i-redex $\sigma(s_i(Z_1(\bar{x}_1), \ldots, Z_m(\bar{x}_m)))$ contains an r_j-redex $(i \neq j)$, then this r_j-redex must be already contained in one of the $\sigma(Z_p(\bar{x}_p))$.
- Likewise if the r_i-redex properly contains an r_i-redex.

(2) R is *left-linear* if all s_i are linear. A metaterm is linear if it does not contain multiple occurrences of the same metavariable. (Example: $\rho x.xZ(x)$ is linear; $\alpha xy.F(Z(x), Z(y))$ is not linear.)

(3) R is *orthogonal* if it is nonoverlapping and left-linear.

Actually, what we have defined now are full CRSs, with unrestricted term formation. We conclude this section with a more precise definition of sub-CRSs.

Definition 12.9. (1) Let $(R, \rightarrow_R)$ be a CRS as defined above. Let T be a subset of $\mathtt{Terms}(R)$, which is closed under $\rightarrow_R$. Then $(T, \rightarrow_R | T)$, where $\rightarrow_R | T$ is the restriction of $\rightarrow_R$ to T, is a *substructure* of $(R, \rightarrow_R)$.

(2) If $(R, \rightarrow_R)$ is orthogonal, so are its substructures.

13. Confluence proof à la Aczel and superdevelopments

In this section we will sketch a short proof of the fact that all orthogonal CRSs are confluent and we will briefly discuss the notion of superdevelopment. For full proofs see [42].

13.1. Confluence

The proof of confluence for orthogonal CRSs proceeds along the lines of the proof by Aczel of confluence for orthogonal contraction schemes, which form a subclass of CRSs [2]. The proof strategy in Aczel's proof is the same as in the proof of confluence of λ-calculus with β-reduction by Tait and Martin-Löf. This strategy is also employed in several other proofs [38, 46]. The idea is roughly as follows. A relation $\geqslant$ on terms is defined such that its transitive closure equals reduction. For this relation the diamond property is proved. A binary relation $\rhd$ satisfies the diamond property if, whenever $a \rhd b$ and $a \rhd c$, there exists a d such that $b \rhd d$ and $c \rhd d$. Having proved the diamond property for $\geqslant$, confluence of the reduction relation follows immediately.

The method of Aczel's proof is the same as in the proof by Tait and Martin-Löf. The difference is due to the relation on terms that is defined. If we write $\geqslant$ for Aczel's relation and $\mapsto_1$ for Tait and Martin-Löf's relation, we have that $\mapsto_1$ implies $\geqslant$, but not necessarily vice versa. For the proof of confluence for orthogonal CRSs, a relation like Aczel's, denoted as $\geqslant$, is used.

Definition 13.1. The relation $\geqslant$ on $\mathtt{Terms}$ is defined as follows:

(1) $x \geqslant x$ for every variable x;

(2) if $s \geqslant t$ then $[x]s \geqslant [x]t$ for every variable x;

(3) if $s_1 \geqslant t_1, \ldots, s_n \geqslant t_n$ then $F(s_1, \ldots, s_n) \geqslant F(t_1, \ldots, t_n)$ for every n-ary function symbol F;

(4) if $s_1 \geqslant t_1, \ldots, s_n \geqslant t_n$ and $F(t_1, \ldots, t_n) = \sigma(\alpha)$ for some reduction rule $\alpha \rightarrow \beta$ and valuation σ, then $F(s_1, \ldots, s_n) \geqslant \sigma(\beta)$.

The first three clauses of the definition state that $\geqslant$ is a reflexive relation that is closed under term formation. The fourth clause expresses that $s \geqslant t$ if s reduces to t by a parallel "inside-out" reduction, where redexes that are "created upwards" may be contracted. Note that in this clause $F(s_1, \ldots, s_n)$ is not necessarily a redex. Here lies the difference with the relation $\mapsto_1$. Consider, for example, the following term rewriting system:

$$F(B) \to C,$$

$$A \to B.$$

Then we have $F(A) \geqslant C$ but not $F(A) \mapsto_1 C$.

In general, the fourth clause can be depicted as follows:

$$\begin{array}{ccc} F(s_1, \ldots, & s_n) & \\ \rotatebox{90}{$\geqslant$} \quad & \rotatebox{90}{$\geqslant$} & \searrow\!\!\geqslant \\ \sigma(\alpha) = F(t_1, \ldots, & t_n) & \longrightarrow \sigma(\beta). \end{array}$$

The next proposition states that $\geqslant$ is indeed a useful relation for which to prove the diamond property.

Proposition 13.2. *The transitive closure of* $\geqslant$ *equals reduction.*

The crucial step in proving the diamond property for $\geqslant$ is proving that $\geqslant$ satisfies a property named "coherence". This notion was originally introduced by Aczel [2].

Definition 13.3. A binary relation $\rhd$ on `Terms` is said to be *coherent* with respect to reduction if the following holds: if $F(a_1, \ldots, a_n) = \sigma(\alpha)$ for some reduction rule $\alpha \to \beta$ and valuation σ, and $a_1 \rhd b_1, \ldots, a_n \rhd b_n$, then we have for some valuation τ that $F(b_1, \ldots, b_n) = \tau(\alpha)$ with $\sigma(\beta) \rhd \tau(\beta)$.

Coherence can be depicted as follows:

$$\begin{array}{cccc} F(a_1, & \ldots, & a_n) \to & a \\ \triangledown & & \triangledown & \triangledown \\ F(b_1, & \ldots, & b_n) \to & b. \end{array}$$

It is now a matter of routine to prove coherence of $\geqslant$ with respect to reduction.

Lemma 13.4. *The relation* $\geqslant$ *is coherent with respect to reduction.*

If coherence for the relation $\geqslant$ has been established, the diamond property of $\geqslant$ can be proved by induction.

Theorem 13.5. *The relation* $\geqslant$ *satisfies the diamond property.*

Proof. Suppose $a \geqslant b$ and $a \geqslant c$. By induction on the derivation of $a \geqslant b$ it can be proved that a d exists such that $a \geqslant d$ and $b \geqslant d$. $\square$

Confluence of orthogonal CRSs is now a direct consequence of this theorem.

Corollary 13.6. *All orthogonal CRSs are confluent.*

13.2. Superdevelopments

Besides the proof by Tait and Martin-Löf for confluence of λ-calculus with β-reduction there are other proofs, one of which proceeds by proving first that all developments are finite. A *development* is a reduction sequence in which only descendants of redexes that are present in the initial term may be contracted. Redexes that are created along the way are not allowed to be contracted. Both confluence proofs are related in the following way: $M \mapsto_1 N$ if and only if a (complete) development $M \twoheadrightarrow N$ exists (see [5]).

A natural question now is whether reduction sequences corresponding exactly to the relation $\geqslant$ can be characterized, and if so, whether they are always finite. For the case of λ-calculus, it turns out that reduction sequences corresponding to $\geqslant$ can be characterized by a more liberal notion of development, called a *superdevelopment.* This is done by defining a set of labelled λ-terms Λ_1 and labelled β-reduction $\rightarrow_{\beta_1}$ on them. The difference between developments and superdevelopments in λ-calculus can be understood by considering the different ways in which β-redexes can be created. This has been studied by Lévy [32]. The following possibilities are distinguished (written in the usual notation for λ-calculus):

(1) $((\lambda x.\lambda y.M)N)P \rightarrow_\beta (\lambda y.M[x:=N])P$;

(2) $(\lambda x.x)(\lambda y.M)N \rightarrow_\beta (\lambda y.M)N$;

(3) $(\lambda x.C[xM])(\lambda y.N) \rightarrow_\beta C'[(\lambda y.N)M']$, where C' and M' stand for C and M, respectively, in which all free occurrences of x have been replaced by $\lambda y.N$.

In a development, no created redexes at all may be contracted. In a superdevelopment, created redexes of the first two kinds may be contracted. Note that, if we think of a λ-term as a tree built from application- and λ-nodes, the redexes in the first two cases are "created upwards". In the last case, on the other hand, the redex is not created upwards, and may not be contracted in a superdevelopment.

It is proved in [42] that (complete) superdevelopments correspond exactly to the relation $\geqslant$ and moreover that all superdevelopments are finite. The result that all superdevelopments are finite illustrates that all infinite β-reduction sequences in λ-calculus are due to the third way of redex creation; indeed redex creation, e.g. in the reduction sequence of $(\lambda x.xx)(\lambda x.xx)$, happens in this way. The first two kinds of creating redexes are "innocent" and may be contracted in a superdevelopment.

We will now define the set of labelled λ-terms and labelled β-reduction on them. Application nodes are written explicitly, but abstraction terms as usual. Lambda's will be labelled by a label from a countably infinite set of labels I, and application nodes be labelled by a subset of I.

Definition 13.7. The set Λ_1 of labelled λ-terms is defined as the smallest set such that

(1) $x \in \Lambda_1$ for every variable x,

(2) if $M \in \Lambda_1$ and $i \in I$, then $\lambda_i x.M \in \Lambda_1$,

(3) if $M, N \in \Lambda_1$ and $X \subset I$, then $@^X(M, N) \in \Lambda_1$.

The reduction rule β_1 on Λ_1 is defined as

$$@^X(\lambda_i x.Z(x), Z') \to_{\beta_1} Z(Z') \quad \text{if } i \in X.$$

As usual in λ-calculus we adopt the variable convention, i.e. all bound variables in a statement are supposed to be different from the free ones. Note that the set of labelled λ-terms with labelled β-reduction is in fact an orthogonal CRS.

The idea of a superdevelopment is that only β-redexes are contracted if the application node "knows" the λ already, or more formally, if the λ occurs in the scope of the application node in the initial term. Now β_1 reduction is used to formalize this idea. An expression $@^X(\lambda_i x.M, N)$ is a β_1-redex if $i \in X$. Reduction steps of a term that are allowed according to the notion of superdevelopments we have in mind are β_1-reduction steps if the term is labelled such that the label of an application node contains no more than the labels of λ's in its scope. We will call a labelled λ-term *good* if it satisfies this condition on the labels.

For example, $@^{\{2\}}(@^{\{1\}}(\lambda_1 x.\lambda_2 y.xy, z), u)$ is a good term but $@^{\{1\}}(\lambda_1 x.@^{\{2\}}(x, y), \lambda_2 y.y)$ is not good.

All reducts of a good term are good, intuitively because β_1-reduction cannot push a λ outside the scope of an application node in which it occurred originally.

Now we can define superdevelopments. If $M \in \Lambda_1$ is a good term such that all λ's occurring in M have a different label, and $M \twoheadrightarrow_{\beta_1} N$ is a β_1-reduction, then this reduction sequence is a *superdevelopment* after erasing all labels.

The following results are proved in [42].

Theorem 13.8 (Finite superdevelopments). *If a λ-term M is labelled such that all λ's have a different label then all its β_1-reductions are finite.*

Theorem 13.9. *$M \geqslant N$ if and only if there exists a β_1-reduction sequence to β_1-normal form $M' \twoheadrightarrow_{\beta_1} N'$ such that M', N' yield M, N after erasing labels.*

14. Related work

There have been several approaches to formulating a general framework for term rewriting, including first-order term rewriting and lambda calculi. Without attempting to give a complete historical survey of such approaches, we mention some of the most noteworthy, referring for a more elaborate discussion to [30] or to the original references.

One of the first extended formats consists of Hindley's $\lambda(a)$-reductions. They combine λ-calculus with orthogonal TRSs, thus containing all orthogonal λ-TRSs. In fact they contain more than λ-TRSs, since right-hand sides of rules may include λ-terms. They also contain Church's δ-rules (see Example A.1).

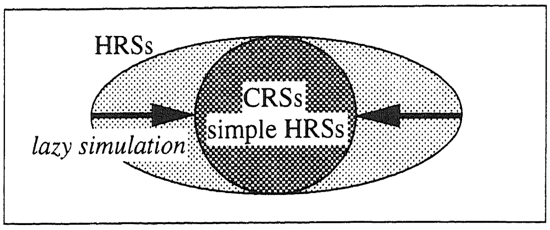

Fig. 3.

The fundamental idea leading to the present framework of CRSs was formulated by Aczel [2], who devised "contraction schemes". They do not support arbitrary complex pattern matching as in first-order TRSs, but apart from that they introduce variable-binding as in the present CRSs.

Wolfram [48] describes a general notion of higher-order rewriting. This is the starting point for a recent formulation of higher-order rewriting that is given by Nipkow [37] in his higher-order rewrite systems (HRSs). The metalanguage employed for HRSs is the simply typed λ-calculus, facilitating the definition of substitution. For a comparison of CRSs and HRSs, see [39]. It turns out that both formats are roughly co-extensive, and have the same expressive power. This is a satisfactory state of affairs to us, since it hints at the possibility that the formulation of CRSs and HRSs, in spite of the apparent differences in their actual definition, has hit upon a canonical framework for higher-order rewriting. (This does not mean that there are not several desirable extensions of the present CRS/HRS format; see our list of possible extensions in Section 15.) In Fig. 3 the relation between HRSs and CRSs is indicated. For a large class of HRSs that we have called "simple HRSs", including λ-calculus and TRSs, we have an exact correspondence between CRSs and HRSs, modulo notational differences. That is, there are direct translations between terms in CRS-format and in HRS-format that preserve one-step reduction in both directions. The "surplus-HRSs" do not really add expressive power: they can be simulated by CRSs, but less directly. Namely, one step in the CRS corresponds to one step in the HRS, but one step in the HRS will correspond to several steps in the CRS. Roughly, there is an analogy with the relation of λ-calculus to $\lambda\sigma$-calculus or λ-calculus with explicit substitution: in the latter one β-step is simulated by several steps (see [1]). Thus, we can say that CRSs have a more "explicit" substitution mechanism than HRSs. This can be considered as either an advantage or a disadvantage, depending on one's point of view or needs. In the figure we have referred to the more explicit (i.e. "slower") way of CRSs for evaluating substitutions as "lazy simulation".

The format of higher-order rewriting developed by [28, 29] is equivalent to that of CRSs but the set-up is closer that of λ-calculus and of first-order logic.

Extensions of λ-calculus by means of conditions were studied in [45, 46]. These "conditional λ-calculi" comprise many CRSs; in a personal communication we have learned that a slight generalization of the conditions leads to the whole class of CRSs (in fact, even a somewhat larger class).

In summary, there seems to be a convergence of several proposals for notions of higher-order rewriting.

15. Concluding remarks and questions

We have presented the framework for higher-order rewriting as first fully described in [30], where Aczel's original idea was extended with general pattern-matching as in first-order TRSs. In the present introduction we have given a more precise exposition than in [30] of the substitution mechanism that is involved, and we have also sketched a confluence proof (recently obtained by [42], but also present in the work of Nipkow and Takahashi) adapting Aczel's original one to the present framework.

The phrase "higher order" may need an explanation. It is meant as a contrast to the usual "first order" format of term rewriting. Here the word "first order" has a precise meaning: terms are rewritten that are from a first order language (one that features in first order predicate logic). The phrase "higher order" has a less well-defined meaning yet we feel that it is the right terminology, the more so because our CRS format turns out to be quite close to, and even in some sense co-extensive with, the higher-order rewrite systems introduced by Nipkow [37]. The word higher order there has a well-defined meaning, as that framework employs variables and operators of higher type, types being as in simply typed λ-calculus (see our previous section for a comparison). Some confusion is likely to arise in view of the wide-spread usage in the functional language community of the term 'higher-order' when dealing with an applicative system such as Combinatory Logic (CL), the idea there being that operators need not be provided with all their intended arguments (CL can be viewed as having 'varyadic' operators), so that an operator with an incomplete list of arguments yields another operator, i.e. the first operator is of 'higher order'. However, usage of the term 'higher order' in this connection seems questionable to us, because CL is nothing more than an ordinary first-order term rewriting system! In view of the comparison with the HRSs as in the previous section, showing the tight connection, we feel quite confident that the present higher-order rewrite format, whether it be in the actual form of CRSs or that of HRSs, has hit upon a canonical framework. In either case, CRSs and HRSs have advantages and disadvantages in their presentation: the substitution mechanism of HRSs may be simpler, but it presupposes knowledge of simply typed λ-calculus and long $\beta\eta$-normal forms; CRS-rules can be written down without being concerned with the need for "meta-typing" them, but they have a more intricate substitution mechanism. Also, the distinction in CRSs between variables $x, y, z, \ldots$ and metavariables $Z, Z(x), \ldots$, as opposed to the uniform treatment in HRSs, may be viewed as both an advantage (since they play different roles) and a disadvantage (since it proliferates the notion of variable). We will now mention some directions of research aimed at enhancing the applicability of CRSs that we are currently pursuing.

(a) Inclusion of commutative/associative operators. A very useful extension of the confluence result for orthogonal CRSs will be to establish confluence in the presence of commutative/associative operators. Several axiomatisations arising in process algebra will profit from such an extension.

(b) Inclusion of free variable rules, as in π-calculus. At present, we have required in a CRS reduction rule $s \to t$ that t and s are closed metaterms. That is, they may contain metavariables, of course, but not free variables. Actually, this is not forced upon us, and we may consider rules containing free variables. A proviso is necessary: free variables contained in the right-hand side t must also occur in the left-hand side s. The importance of this extension is that free variable rules occur in rewrite systems associated with π-calculus. To maintain orthogonality, and hence confluence, it must be required that in a system containing free variable rules only variables can be substituted for these free variables. (This requirement is met in π-calculus.) As an example, consider λ-calculus extended with the free variable rule $xx \to I$. By considering the reducts of $(\lambda x.xx)M$ it is clear that confluence is lost.

(c) Relaxing the orthogonality condition to weak orthogonality. This seems a difficult question.[1] However, when weak orthogonality is restricted so that critical pairs only arise from "overlay's", i.e. by overlap at the root, then the proof as outlined in Section 13 is still valid.

(d) Settling our claim that CRSs are more expressive than λ-TRSs. This will require a detailed analysis, as indicated in Section 8 ("proof idea").

(e) Ground confluence vs confluence vs metaconfluence. Above, we only established confluence for terms, not metaterms. A stronger confluence result can be obtained at once, however, admitting metavariables; for the moment let us call this "metaconfluence". For nonorthogonal systems, however, the notions separate.

(f) Developing a model theory (semantics) for CRSs [49, 3]. Whereas for first-order TRSs there is a good model theory given by the usual notion of algebra, no analogous concept is available when bound variables are present. For λ-calculus it is already nontrivial to formulate suitable notions of a model.

(g) Describing some recently studied typed λ-calculi as CRSs; likewise for some recently proposed calculi aiming to combine processes and λ-calculus, such as π-calculus. More and more typed λ-calculi are emerging at present; likewise for calculi such as π-calculus. It will be profitable to show that they are in fact orthogonal CRSs. Then the uniform confluence proof can be applied.

(h) Developing versions of CRSs with "explicit substitution", analogous to the $\lambda\sigma$-calculi for λ-calculus [1].

(i) As pointed out in [38] there is a need to extend the notion of CRSs (and of HRSs) in such a way that metavariables in left-hand sides of rewrite rules may require

[1] (Added in proof.) Recently, the second and third authors have solved this question positively: weakly orthogonal CRSs are confluent.

their arguments to be instances of patterns. An example is

$$F([x]Z(\text{cons}(\text{zero},x)))\to G([x]Z(x))$$

for constructors cons and zero. This rule strips away the head "zero" of a "cons" throughout the instantiation of Z at appropriate places. At present such rules do not fit in the scope of CRSs or HRSs.

Appendix: extended examples

We conclude with four larger examples. The first two are extensions of pure λ-calculus; the second one is in fact a λ-TRS. The third one is a two-sorted labelled version of λ-calculus, and the last example is a presentation of system F in the CRS format. All four are orthogonal CRSs.

A.1. λ-calculus with δ-rules of Church

This is an extension of λ-calculus with a constant δ and a possibly infinite set of rules of the form

$$\delta M_1 \dots M_n \to N,$$

where the M_i $(i=1,\dots,n)$ and N are closed terms and the M_i are moreover in $\beta\delta$-normal form, i.e. contain no β-redex and no subterm as in the left-hand side of a δ-rule. To ensure nonoverlapping there should, moreover, not be two left-hand sides of different δ-rules of the form $\delta M_1 \dots M_n$ and $\delta M_1 \dots M_m$, with $m \geqslant n$. (So, every left-hand side of a δ-rule is a normal form with respect to the other δ-rules.) Thus we obtain an orthogonal CRS.

A.2. λ-calculus with pairing, definition by cases, and iterator

From Aczel [2]. Note that this is an example of a definable extension of λ-calculus

Pairing	$D_0(DZ_1Z_2)\to Z_1$
	$D_1(DZ_2Z_2)\to Z_2$
Definition by cases	$R_nQ_1Z_1\dots Z_n\to Z_1$
	$\vdots$
	$R_nQ_nZ_1\dots Z_n\to Z_n$
Iterator	$J0Z_1Z_2\to Z_2$
	$J(SZ_0)Z_1Z_2\to Z_1(JZ_0Z_1Z_2)$
Beta	$(\lambda([x]Z(x)))Z'\to Z(Z')$

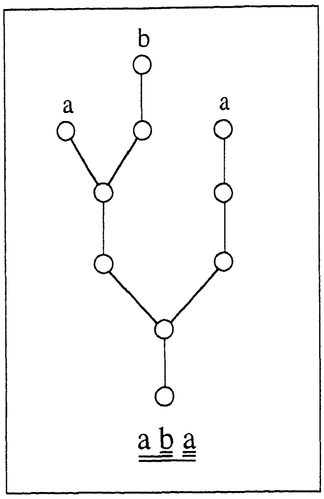

Fig. 4.

A.3. Lévy's λ-calculus

This is a labelled λ-calculus, called λ^{L}, where the labels ("Lévy-labels") keep track of much of the history of a reduction. It is an extremely useful tool in giving precise definitions of notions such as descendants, equivalence of reductions, etc. It was introduced in [32]; a simplified version is in [30].

Lévy-labels are unary–binary trees with end-nodes labelled by $a, b, c, \ldots$ (see the example). More precisely, the set L of Lévy-labels is generated from some atomic labels $a, b, c, \ldots$ by concatenation and underlining, as follows.

(1) $a, b, c, \ldots \in L$ (atomic labels),

(2) if $\alpha, \beta \in L$, then $\alpha\beta \in L$ (concatenation),

(3) if $\alpha \in L$, then $\underline{\alpha} \in L$ (underlining).

Terms of λ^{L} are generated as follows:

(1) $x, y, z, \ldots \in \mathtt{Terms}(\lambda^{\mathrm{L}})$

(2) if $M \in \mathtt{Terms}(\lambda^{\mathrm{L}})$, then $NM \in \mathtt{Terms}(\lambda^{\mathrm{L}})$

(3) if $M \in \mathtt{Terms}(\lambda^{\mathrm{L}})$ and $\alpha \in L$, then $M^{\alpha} \in \mathtt{Terms}(\lambda^{\mathrm{L}})$

So labelled λ-terms may be partially labelled, or not at all. Labelled β-reduction is defined by

$$(\lambda x.Z(x))^{\alpha} Z' \to (Z(Z'^{\underline{\alpha}}))^{\underline{\alpha}}.$$

Here we identify iterated labels with their concatenation: $(M^{\alpha})^{\beta} = M^{\alpha\beta}$. The label α in the redex $(\lambda x.Z(x))^{\alpha} Z'$ is called the *degree* of that redex. An important feature of λ^{L} is that, during a reduction, descendants of a redex keep the same degree, while created redexes have a degree higher than that of the creator redex. (The height of a label is the height of the tree corresponding to it, as suggested in the example of Fig. 4.)

Example A.1. $((\lambda x.(x^a z)^b)^c(\lambda y.yy)^d)^e \to ((\lambda y.yy)^{d\underline{c}a} z)^{b\underline{c}e}$. Note that the redex which is created in the right-hand side of this step has indeed a higher degree ($d\underline{c}a$) than that of the creator redex in the left-hand side (c).

Remark. The identification $(M^\alpha)^\beta = M^{\alpha\beta}$ is here entirely innocent, but a closer look reveals that in fact this entails a little nastiness; namely, the introduction of an ambiguous rewrite rule. Let us write $\mathrm{lab}(Z, \alpha)$ for Z^α and $\mathrm{conc}(\alpha, \beta)$ for $\alpha\beta$. Then the identification amounts to employing the rewrite rule

$$\mathrm{lab}(\mathrm{lab}(Z, \alpha), \beta) \to \mathrm{lab}(Z, \mathrm{conc}(\alpha, \beta)),$$

which is self-overlapping: $\mathrm{lab}(\underline{\mathrm{lab}(Z, \alpha)}, \beta)$.

Yet we can present λ^{L} as a (two-sorted) orthogonal CRS, without "cheating", by having infinitely many labelled β-rules, as follows:

$$(\ldots((\lambda x.Z(x))^{\alpha_1})^{\cdots\alpha_n})Z' \to (Z(Z'^{\underline{\alpha_1 \ldots \alpha_n}}))^{\underline{\alpha_1} \cdots \alpha_n}.$$

A.4. Second-order polymorphic λ-calculus

In this example we consider second-order polymorphic λ-calculus (or polymorphic typed λ-calculus, or second-order typed λ-calculus, or system F, or $\lambda 2$) based on the presentation in [17]. We will show that it is an orthogonal CRS when only β-reduction (both for term application and type application) is considered, and a weakly orthogonal CRS when η-reduction (for terms and types) is also taken into account. In the first case we immediately have confluence by invoking the confluence proof for orthogonal CRSs. The same holds for the second case, due to the recent confluence proof for weakly orthogonal CRSs as mentioned earlier.

For treatments of second-order polymorphic λ-calculus, we refer, e.g., to [23, several articles in Ch. 2], [7, 43, 17].

The basic intuition is as follows. In simply typed λ-calculus there is, e.g., an identity function $\lambda x : \sigma . x$ for each type σ. Polymorphic λ-calculus is an extension of typed λ-calculus in the sense that type abstraction is possible, so that all the $\lambda x : \sigma . x$ can be taken together to form one *second-order identity function* $\Lambda t.(\lambda x : t.x)$ which specializes to a particular identity function after feeding it a type σ:

$$(\Lambda t.(\lambda x : t.x))\sigma \to \lambda x : \sigma . x$$

Here t is a type variable, and Λt is type abstraction, written with a big lambda to distinguish it from abstraction on the object level, λx. In the sequel we will employ a syntax somewhat different from that used in this example.

Definition A.2. $\mathtt{Var}$ is a set of (term) *variables* $x_1, x_2, \ldots$, usually written as $x, y, z, \ldots$ $\mathtt{Tvar}$ is a set of *type variables* $t_1, t_2, \ldots$, usually written as $t, s, \ldots$ B is a set of *base types* (ground types). The set T of *types* is defined inductively as follows:

(a) base types and type variables are types,

(b) if $\sigma, \tau \in T$, then $\sigma \to \tau \in T$,

(c) if $t \in \mathtt{Tvar}$ and $\sigma \in T$, then $\forall t.\sigma \in T$.

Definitions of free and bound type variable occurrences and of closed type expressions are as usual. Likewise notions of renaming bound type variables (α-conversion) are as usual. For a precise treatment of these issues see [17].

We assume the presence of a set $\mathtt{S}$ of *constant symbols* c, each with its own type, written $\mathtt{type}(c)$, which is required to be a closed type.

As in [17] we introduce "raw" terms, i.e. terms that are not yet subject to a typing discipline.

Definition A.3. The set of *polymorphic raw terms*, $P\Lambda$, is defined as follows:
(a) $c \in P\Lambda$, $x \in P\Lambda$ for all constants $c \in \mathtt{S}$ and $x \in \mathtt{Var}$,
(b) if $M, N \in P\Lambda$, then $(MN) \in P\Lambda$,
(c) if $x \in \mathtt{Var}$, $\sigma \in T$ and $M \in P\Lambda$, then $\lambda x : \sigma . M \in P\Lambda$,
(d) if $\sigma \in T$ and $M \in P\Lambda$, then $(M\sigma) \in P\Lambda$,
(e) if $t \in \mathtt{Tvar}$ and $M \in P\Lambda$, then $(\Lambda t . M) \in P\Lambda$.

We will now state the reduction rules on $P\Lambda$ as in [17]:

$$(\lambda x : \sigma . M) N \to_\beta M[x := N] \quad (\beta\text{-reduction rule}),$$

$$\lambda x : \sigma . Mx \to_\eta M \quad (\eta\text{-reduction rule}),$$

$$(\Lambda t . M)\tau \to_{\tau\beta} M[t := \tau] \quad (\text{type } \beta\text{-reduction rule}),$$

$$\Lambda t . Mt \to_{\tau\eta} M \quad (\text{type } \eta\text{-reduction rule}).$$

Note that the raw terms are very raw indeed: not only are they not subject to the type discipline that will be introduced below, but the sorts (terms versus types) are also mixed up: $(\lambda x : \sigma . M)\tau$ as well as $(\Lambda t . M)N$ are raw terms.

Let us rewrite this in CRS format. As introduced above, CRSs are single-sorted, and we wish to maintain that property. We therefore start with a set of proto-terms even more "raw" than the ones above. At first, types and terms will not be distinguished.

Definition A.4. (*Proto-terms for polymorphic second-order λ-calculus*).
(a) The alphabet of proto-$\lambda 2$ consists of:
- variables $x, y, \ldots$,
- 0-ary and unary metavariables $Z, Z(x), \ldots$,
- constants $b, b', \ldots$ (called "base types"),
- constants $c, c', \ldots$ (called "term constants"),
- binary function symbols $\to$, :, @,
- unary function symbols $\lambda, \Lambda, \forall$,
- an abstraction operator $[\_]\_$.

(b) Terms and metaterms are defined from this alphabet as usual for CRSs.

(c) The rewrite rules of proto-$\lambda 2$ are:

$$@(\lambda([x]:(Z'',Z(x))),Z')\to Z(Z') \quad (\beta\text{-rule}),$$

$$@(\Lambda([x]Z(x)),Z')\to Z(Z') \quad (\text{type } \beta\text{-rule}),$$

$$\lambda([x]:(Z'',@(Z,x)))\to Z \quad (\eta\text{-rule}),$$

$$\Lambda([x]@(Z,x))\to Z \quad (\text{type } \eta\text{-rule}).$$

Proto-$\lambda 2$ with only the β-rules is clearly an orthogonal CRS, and hence confluent. With β- and η-rules there is a harmful overlap causing nonconfluence [17]. Proto-$\lambda 2$ is an extension of pure λ-calculus with respect to the set of terms, not rules. It contains many garbage terms but also intended terms, coding the polymorphic terms we are aiming for. The term $\lambda([x]:(N,M))$ will stand for $\lambda x:N.M$; the N here will later turn out to be of sort "type".

We will now describe how the set of proto-terms (i.e. terms of proto-$\lambda 2$) is restricted to the set of polymorphically typable terms as intended. We note that in taking this restricted subset we are free to use every device: the format of CRSs has no bearing on that. We start with singling out a subset of the proto-terms called "types". These are defined as follows:

(a) variables $x, y, z, \ldots$ are types,
(b) base types $b, b', \ldots$ are types,
(c) if σ, τ are types, then $(\sigma\to\tau)$ is a type,
(d) if x is a variable and σ a type, then $\forall([x]\sigma)$ is a type.

Only the first clause needs comment. All variables are called types because we do not distinguish type variables versus term variables as we wish to stay in a single-sorted framework. This will not cause problems: type- and term variables can be used interchangeably; it is their relative position that will determine what they actually are in a term.

A type assignment is a finite set of the form

$$x_1:\sigma_1,\ldots,x_n:\sigma_n,$$

where the x_i are pairwise different variables, and the σ_j are types not containing any of the x_i freely (in order not to confuse the roles of the x_i as term variables and of the variables free in some σ_j as type variables). We also suppose that a fixed assignment of closed types to the constants $c, c', \ldots$ is given; notation: type(c), etc.

A typing judgement is an expression of the form

$$\Delta \rhd M:\sigma$$

where Δ is a type assignment, σ a type, and M a proto-term. Typing judgements are derived by the following proof system.

Axioms:

$$\Delta \rhd c:\mathtt{type}(c)$$

$$\Delta, x:\sigma \rhd x:\sigma$$

Inference rules:

$$\frac{\Delta \rhd M : \sigma \rightarrow \tau \quad \Delta \rhd N : \sigma}{\Delta \rhd @(M, N) : \tau}$$

$$\frac{\Delta, x : \sigma \rhd M : \tau}{\Delta \rhd \lambda([x] : (\sigma, M)) : \sigma \rightarrow \tau}$$

$$\frac{\Delta \rhd M : \forall([t]\sigma(t))}{\Delta \rhd @(M, \tau) : \sigma(\tau)}$$

$$\frac{\Delta \rhd M : \sigma}{\Delta \rhd \Lambda([t]M) : \forall([t]\sigma)}$$

In the last inference rule there is the following proviso: if Δ contains a $x : \sigma$ such that x is free in M and τ is free in σ, then the rule may not be applied.

If a typing judgement $\Delta \rhd M : \sigma$ can be derived using this inference system, we write

$$\vdash \Delta \rhd M : \sigma$$

and say that M type-checks with type σ under type-assignment Δ. A proto-term M is called *typable* if there are Δ, σ such that $\vdash \Delta \rhd M : \sigma$. We now restrict the set of proto-terms to the set of typable proto-terms, and we claim that (with the same rewrite rules as above) this yields a sub-CRS of proto-λ2. The statement of this claim is known as the *subject-reduction property*. This is Lemma 5.2 in [17], although here for a larger set of proto-terms than the raw terms there; the proof is according to [17]: tedious but not difficult. The sub-CRS of typable proto-terms is the intended one: polymorphic second-order λ-calculus. With only the β-rules it is orthogonal; with β- and η-rules it is weakly orthogonal.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien and J.-J. Lévy, Explicit substitutions, in: *Proc. ACM Conf. on Principles of Programming Languages*, San Francisco (1990) 31–46.

[2] P. Aczel, A general Church–Rosser theorem. Technical Report, Univ. of Manchester, Manchester, 1978.

[3] P.B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof* (Academic Press, New York 1986).

[4] H.P. Barendregt, Pairing without conventional restraints, *Z. Math. Logik Grundlag. Math.* **20** (1974) 289–306.

[5] H.P. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, 2nd edn. (North-Holland, Amsterdam, 1984).

[6] H.P. Barendregt, Functional programming and lambda-calculus, in: J. van Leeuwen, ed., *Formal Methods and Semantics, Handbook of Theoretical Computer Science, Vol. B*, Ch. 7 (Elsevier, Amsterdam, 1990) 321–364.

[7] H.P. Barendregt, Typed lambda-calculi, in: S. Abramsky, D. Gabbay and T. Maibaum, eds, *Handbook of Logic in Computer Science, Vol. I* (Oxford Univ. Press, Oxford, 1992) 117–309.

[8] A. Berarducci and C. Böhm, A self-interpreter of λ-calculus having a normal form. Technical Report, Università di Aquila, 1992. Rapporto Technico 16, Dip. di Matematica Pura ed Applicata.

[9] G. Berry, Séquentialité de l'évaluation formelle des λ-expressions, in: *Proc. 3ième Colloque Internat. sur la Programmation*, Paris, Dunod (mars 1978).

[10] V. Breazu-Tannen, Combining algebra and higher-order types, in: *Proc. 3rd annual IEEE Symposium on Logic in Computer Science*, Edinburgh (1988) 80–90.

[11] V. Breazu-Tannen and J. Gallier, Polymorphic rewriting conserves algebraic strong normalization and confluence, in: *Proc. 16th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science Vol. 372. (Springer, Berlin, 1989) 137–150.

[12] A. Church, *The Calculi of Lambda Conversion, Vol. 6: Annals of Mathematics Studies* (Princeton Univ. Press, Princeton, NJ, 1941).

[13] A. Church, *Introduction to Mathematical Logic* (Princeton Univ. Press, Princeton, NJ, 1956).

[14] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science (Pitman, London, 1986).

[15] J.W. de Bakker, *Mathematical Theory of Program Correctness* (Prentice-Hall, Englewood Cliffs, NJ, 1980).

[16] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen, ed., *Formal Methods and Semantics, Handbook of Theoretical Computer Science, Vol. B*, Ch. 6 (Elsevier, Amsterdam, 1990) 243–320.

[17] J. Gallier, On Girard's 'Candidats de reductibilité', in P. Odifreddi, ed., *Logic and Computer Science*, (Academic Press, New York, 1990) 123–203, **32**, *APIC Stud. Data Processing*.

[18] R.O. Gandy, Proofs of strong normalization, in J.P. Seldin and J.R. Hindley, eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (Academic Press, New York, 1980) 457–477.

[19] J.-Y. Girard, *Proof Theory and Logical Complexity, Vol. I* (Bibliopolis, Napoli, 1987).

[20] J.R. Hindley, The equivalence of complete reductions, *Trans. Amer. Math. Soc.* **229** (1977) 227–248.

[21] J.R. Hindley, Standard and normal reductions, *Trans. Amer. Math. Soc.* **241** (1978) 253–271.

[22] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-Calculus, Vol. 1: London Mathematical Society Student Texts* (Cambridge Univ. Press, Cambridge, 1986).

[23] G. Huet, ed., *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming (Addison-Wesley, Reading, MA, 1990).

[24] S. Kahrs, *λ-rewriting*. Ph.D. Thesis, Universität Bremen, 1991.

[25] S. Kahrs, Compilation of combinatory reduction systems, Univ. of Edinburgh, Edinburgh, 1992.

[26] J.W. Klop and R.C. de Vrijer, Unique normal forms for lambda calculus with surjective pairing, *Inform. and Comput.* **80** (2) (1989) 97–113.

[27] J.R. Kennaway, Sequential evaluation strategies for parallel-or and related systems, *Ann. Pure Appl. Logic* **43** (1989) 31–56.

[28] Z. Khasidashvili, Expression reduction systems, Technical Report, I. Vekua Institute of Applied Mathematics, Univ. of Tblisi, Georgia, 1990.

[29] Z. Khasidashvili, Church-Rosser theorem in orthogonal combinatory reduction systems. INRIA Rocquencourt Report No. 1825, 1992.

[30] J.W. Klop, Combinatory reduction systems, Ph.D. Thesis, Mathematical Centre Tracts Nr. 127. CWI, Amsterdam, 1980.

[31] J.W. Klop, Term rewriting systems, in: S. Abramsky, D. Gabbay and T. Maibaum, eds., *Handbook of Logic in Computer Science, Volume II* (Oxford Univ. Press, Oxford, 1992) 1–116.

[32] J.-J. Lévy, An algebraic interpretation of the $\lambda\beta$K-calculus and a labelled λ-calculus, in: C. Böhm, ed., *Proc. Rome Conf. λ-Calculus and Computer Science Theory*, Lecture Notes in Computer Science, Vol. 37 (Springer, Berlin, 1975) 147–165.

[33] A. Middeldorp, Modular properties of term rewriting systems, Ph.D. Thesis, Vrije Universiteit, Amsterdam, 1990.

[34] R. Milner, A complete inference system for a class of regular behaviours. *J. Comput. System Sci.* **28** (1984) 439–466.

[35] F. Müller, Confluence of the lambda-calculus with left-linear algebraic rewriting, *Inform. Process. Lett.* **41** (1992) 293–299.

[36] R.P. Nederpelt, Strong normalization for a typed lambda-calculus with lambda structured types, Ph.D. Thesis, Technische Universiteit Eindhoven, 1973.

[37] T. Nipkow, Higher-order critical pairs, in: *Proc. 6th Ann IEEE Symp. on Logic in Computer Science* (1991) 342–349.

[38] T. Nipkow, Orthogonal higher-order rewrite systems are confluent, in: M. Bezem and J.F. Groote, eds., *Proc. Internat. Conf. on Typed Lambda Calculi and Applications*, Utrecht, Lecture Notes in Computer Science, Vol. 664 (Springer, Berlin, 1993) 306–317.

[39] V. van Oostrom and F. van Raamsdonk, Comparing CRSs and HRSs (in preparation).

[40] G.D. Plotkin, LCF as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–257.

[41] D. Prawitz, Ideas and results in proof theory, in: J.E. Fenstad, ed., *Proc. 2nd Scandinavian Logic Symposium* (North-Holland, Amsterdam, 1971) 235–307.

[42] F. van Raamsdonk, Confluence and superdevelopments, in: C. Kirchner, ed., *Proc. 5th Internat. Conf. on Rewrite Techniques and Applications*, Lecture Notes in Computer Science, Vol. 690 (Springer, Berlin, 1993) 168–182.

[43] A. Scedrov, A guide to polymorphic types, in: P. Odifreddi, ed., *Logic and Computer Science* (Academic Press, New York, 1990) 123–203, **32** *APIC Stud. Data Processing.*

[44] S. Stenlund, *Combinators, λ-terms and Proof Theory* (Reidel, Dordrecht, 1972).

[45] M. Takahashi, Parallel reductions in λ-calculus, *J. Symbolic Comput.* **7** (1989) 113–123. Revised version as Report C-103, April 1992, Tokyo Institute of Technology.

[46] M. Takahashi, λ-calculi with conditional rules, in: M. Bezem and J.F. Groote, eds., *Proc. Internat. Conf. on Typed Lambda Calculi and Applications*, Utrecht, Lecture Notes in Computer Science, Vol. 664 (Springer, Berlin, 1993) 406–417.

[47] C. Talcott, A theory of binding structures and applications to rewriting. Technical Report, Stanford Univ., Stanford. 1991.

[48] D.A. Wolfram, Rewriting, and equational unification: the higher-order cases, in: R.V. Book, ed. *Proc. 4th Internat. Conf. on Rewriting Techniques and Applications* (Springer, Berlin, 1991) 25–37.

[49] D.A. Wolfram, *The Clausal Theory of Types* (Cambridge Univ. Press, Cambridge, 1993).