Second-Order Syntax in HOPS and in RALF

Wolfram Kahl, Claudia Hattensperger

Department of Computer Science, German Armed Forces University Munich
e-mail: {kahl|claudia}@informatik.unibw-muenchen.de

Abstract HOPS and RALF are two interactive symbol manipulation systems — one for functional programming and program transformation, the other for proving relation algebraic formulae — that both implement interactive application of second-order rewriting rules, HOPS on term graphs and RALF on conventional terms and formulae. Both systems support a larger class of second-order rewriting rules than commonly found in other systems.
In this paper we provide a homogeneous underpinning to second-order syntax and rewriting for easing the transition from terms to term graphs and vice versa, so that aspects that are easier to understand in one view also further understanding in the other view, altogether making a case for bringing second-order syntax more directly to the user interface than usual in most systems today.

1 Introduction

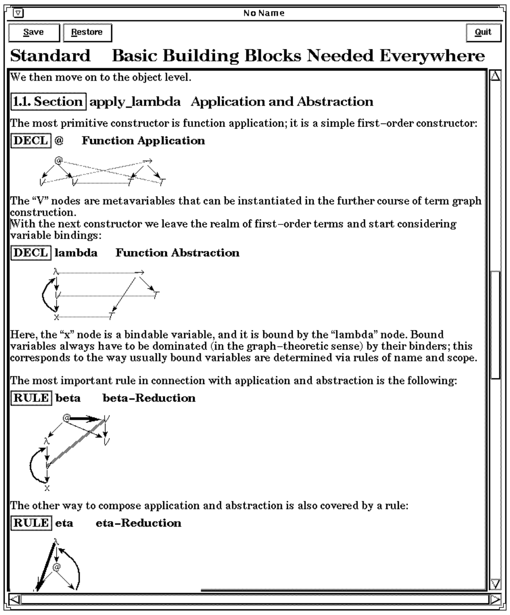
From a small group led by Gunther Schmidt, several interactive symbol manipulation systems have sprung forth, most prominently the **H**igher **O**bject **P**rogramming **S**ystem HOPS[1], a graphically interactive programming and program transformation system [28, 13, 1], and the **R**elation-**AL**gebraic **F**ormula Manipulator RALF [10, 2, 9].

HOPS represents programs as typed directed acyclic term graphs (term DAGs), and from the beginning supported restricted second-order rewriting with zero-ary and unary metavariables, which were, however, not constrained to have bound variables as their successors. As in those early versions this kind of term graph rewriting was not well understood, a theoretical foundation for full second-order term graph rewriting with metavariables of arbitrary arities, cyclic term graphs and more generality than Klop's Combinatory Reduction Systems (CRSs) [20] was laid in [14, 15]. These definitions are used in the latest implementation of HOPS, which however still restricts itself to acyclic graphs.

HOPS is now a literate programming system, presenting modules as documents that may contain nested sections or submodules; these subunits can be folded in the module editor. Interspersed between document text is program content, most prominently consisting of *declarations* of new node labels and of transformation *rules* that may also serve as definitions.

[1] partially supported by the DFG

Declarations are "elementary term graphs" that define the typing at occurrences of the freshly declared node label; for details of the typing system see [17].



The term graph drawing area of declarations, rules and documentation graphs allows direct interaction with the term graphs; graph editing possibilities include construction, instantiation, deletion, manual and automatic transformation; the layout of the term graphs is calculated automatically and may be adjusted manually.

Because of the emphasis on transformation, HOPS is particularly well suited for referentially transparent programming paradigms; its type system places it in the vicinity of Miranda — mechanisms for type classes as in Haskell are under development.

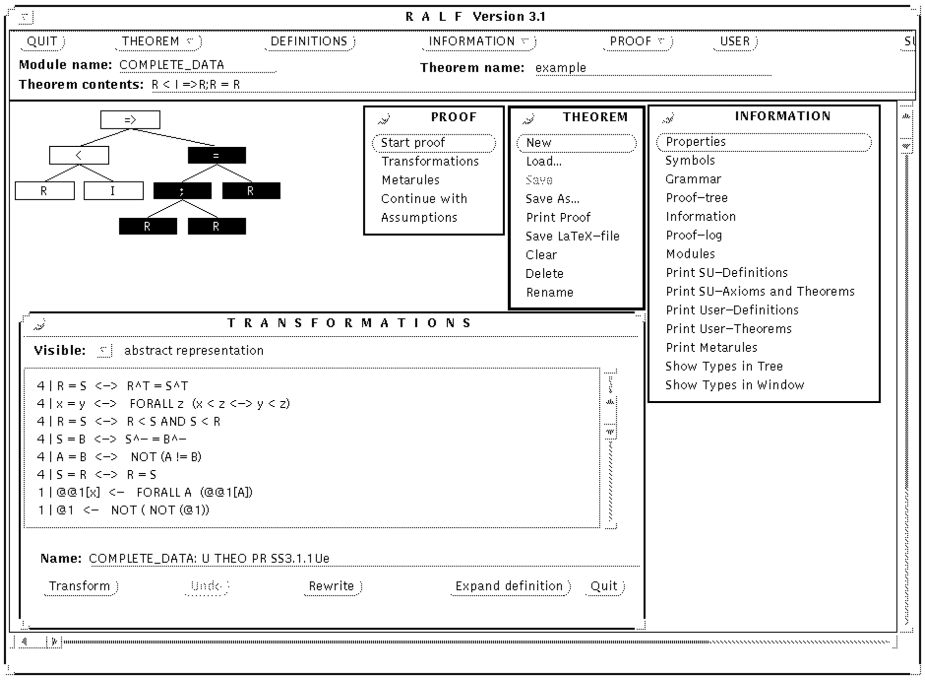
The graphical structure of programs and the transformation capabilities also turn HOPS into a useful tool for visualisation of several aspects of functional programming languages that may help students to overcome some initial barriers by a more intermediate "graphical" understanding of the underlying mechanisms.

A very general term graph attribution mechanism can be used for generating code in conventional programming languages from HOPS modules. Currently we use this to translate HOPS modules to Haskell. The home page of HOPS is at URL: http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/.

RALF is a formula manipulation system and interactive proof checker for relation algebra. Its metalanguage is first-order predicate logic in calculational style, therefore the major tools are substitution of equals for equals, weakening and strengthening. For an introduction to relation algebra see e.g. [23].

In RALF, the proof strategy is top-down reduction: a formula is reduced to axioms of the theory and assumptions. It is based on a calculus allowing not only equivalence preserving rewrite steps as described above, but also weakening or strengthening depending on the context of the expression to be rewritten [9]. This is only a restriction of the applicability of syntactically matching rules for semantic reasons, but has no implications for the rewriting mechanism itself. Metarules offer means to structure the proof, and rewrite rules can be grouped into modules, thus enabling proofs in different theories.

Formulae are represented as trees and the selection of the subexpression to be rewritten is done by mouse click. The system then offers all (relative to a given set of rewrite rules) mathematically correct replacements of the subexpression in a sequence determined by a heuristic.



Proof steps can be saved, reloaded, and undone at any time, and a built-in proof checker tries to recognise the (in-)validity of the current goal. Proof listings in the style of [26] can be printed via LaTeX.

The system is extendable by adding axioms and definitions (in the sense of abbreviations for expressions); axioms and proven theorems are added automatically to the set of rewrite rules.

For better coping with relation algebraic formulae that are embedded into predicate-logical meta-formulae with quantifiers, and also for dealing with binding relation constructors such as arbitrary joins and meets, RALF recently was extended to cope with second-order rules, too.

RALF is available via its home page: URL: http://inf2-www.informatik.unibw-muenchen.de/hattensperger/RALF/html/ralfmanual.html.

In this paper we present an introduction into second-order rewriting that can serve as a foundation for the understanding of second-order rewriting in both systems, but is as homogeneous as possible, thus allowing the reader to gain additional insight into term aspects from the term graph side and vice versa.

After this introductory section, we first present the definitions of second-order terms and second-order term rewriting as a common basis in Sect. 2. Building on this, Sect. 3 investigates the uses of second-order rewriting in RALF. Finally, Sect. 4 is concerned with motivating and defining the second-order term graphs of HOPS.

2 Basic Notions of Second-Order Terms and Rewriting

This section presents basic notions concerning second-order terms, usually called metaterms, together with their rewriting systems, a slight generalisation of Klop's Combinatory Reduction Systems [20] (see [21] for a differently biased gentle introduction). Even though RALF employs a more conventional syntax at its user interface, which is also reflected in Sect. 3, the definitions of this section serve as the fully formal specification of rewriting in RALF, and they also serve as a starting point for motivating our specific definition of term graphs in Sect. 4. We strive to present the definitions in a uniform way and in a few places we consciously sacrifice conformance with the usual way of defining these concepts.

2.1 Metaterms

A general presentation of terms with binders has first been given by Klop in [20] for Combinatory Reduction Systems, to be considered more closely in Sect. 2.2. Klop's aim was a generalisation of Chuch's λ-calculus [6], one calculus containing a certain kind of terms with bound variables. Other kinds of terms with bound variables have long been considered in logics: quantified formulae are "terms" over a certain alphabet in our sense and in the sense of Klop, a point that is emphasised in today's "abstract higher-order syntax".

In terms with bound variables, some constructors must be able to *bind* variables, such as λ in the λ-calculus and $\forall$ and $\exists$ in logics, and other constructors must act as *bound* or, more precisely, *bindable variables*. The rôle of variables in first order term rewriting systems is now taken by so-called *metavariables*. An important difference is that metavariables are not required to have arity zero.

Metaterms are built from an *alphabet* of *constructors*, where a natural number[2] is associated with every constructor, its *arity*. In many term languages the alphabet is partitioned into an enumerable subset containing only zero-ary constructors called *variables* and a finite subset of so-called *function symbols*. To emphasise the contrast with variables, we usually employ the term "*constant constructor*" instead of "function symbol", a name whose motivation comes from semantics, while our concern is only with syntax. Formally we define:

Definition 2.1.1 A **metaterm alphabet** is a tripel $(\mathcal{C}, \mathit{arity}, (\mathcal{K}, \mathcal{B}, \mathcal{V}, \mathcal{M}))$, with an enumerable set $\mathcal{C}$ the elements of which are called **constructors**, the **arity function** $\mathit{arity} : \mathcal{C} \to \mathbb{N}$, $(\mathcal{K}, \mathcal{B}, \mathcal{V}, \mathcal{M})$ is a partitioning of $\mathcal{C}$ where $\mathcal{K}$, the set of **constant constructors**, and $\mathcal{B}$, the set of **binding constructors**, are finite, and where arity restricted to $\mathcal{V}$, the set of **bindable variables**, is the constant function yielding 0 as its value. $\mathcal{M}$ is the set of **metavariables**. $\square$

An only superficial difference to Klop's original definition in [20] is that he considers only one unary binding constructor.[3]

---

[2] The set of natural numbers, $\mathbb{N}$, contains 0 as its smallest element, as usual in computer science.

[3] However, our definition here still does not support the fine-grained distinctions of

Variable binding is achieved by including bindable variables in a special place in metaterm construction:

Definition 2.1.2 The set $\mathcal{MT}_{\mathcal{A}}$ of all **metaterms** over a metaterm alphabet $\mathcal{A} = (\mathcal{C}, \mathit{arity}, (\mathcal{K}, \mathcal{B}, \mathcal{V}, \mathcal{M}))$ is the smallest set such that for every constructor $c : \mathcal{C} - \mathcal{B}$ and every n elements (with $n = \mathit{arity}\ c$, that is, c is an n-ary constructor) $t_1, \ldots, t_n$ of $\mathcal{MT}_{\mathcal{A}}$, the new construct $c(t_1, \ldots, t_n)$ also is an element of $\mathcal{MT}_{\mathcal{A}}$, and for every binding constructor $\beta : \mathcal{B}$, every bindable variable $v : \mathcal{V}$ and every n elements (with $n = \mathit{arity}\ \beta$, that is, β is an n-ary binding constructor) $t_1, \ldots, t_n$ of $\mathcal{MT}_{\mathcal{A}}$, the new construct $\beta v.(t_1, \ldots, t_n)$ also is an element of $\mathcal{MT}_{\mathcal{A}}$. Elements of $\mathcal{MT}_{\mathcal{A}}$ are called *metaterms over* $\mathcal{A}$. A metaterm t over a metaterm alphabet is called **ground** iff it does not contain any metavariables. $\square$

In the CRS literature, what we call a "ground metaterm" is usually just called a term. Instead of "metaterm" we could also say "second-order term", and instead of "ground metaterm" we could also say "term with binding".

For examples let us consider the following metaterm alphabet $\mathcal{A}_\lambda$ for a typical (simply typed) λ-calculus:

$$\begin{aligned}
\mathcal{K}_\lambda &= \{@, \mathsf{pair}, \mathsf{proj1}, \mathsf{proj2}\} \\
\mathcal{B}_\lambda &= \{\lambda\} \\
\mathcal{V}_\lambda &= \{x, y, z, x_0, x_1, x_2, \ldots\} \\
\mathcal{M}_\lambda &= \{\, V_0,\, V_0',\, V_0'', \ldots,\, V_1,\, V_1',\, V_1'', \ldots,\, V_2,\, V_2',\, V_2'', \ldots, \ldots\}
\end{aligned}$$

$\{(@, 2), (\mathsf{pair}, 2), (\mathsf{proj1}, 1), (\mathsf{proj2}, 1), (\lambda, 1)\}$ is contained in arity_λ and for every $i : \mathbb{N}$ we have for the metavariables that

$$i \;=\; \mathit{arity}_\lambda\ V_i \;=\; \mathit{arity}_\lambda\ V_i' \;=\; \mathit{arity}_\lambda\ V_i'' \;=\; \ldots$$

Then $\lambda\, x\, .\, (\lambda\, y\, .\, (\lambda\, z\, .\, (\mathsf{pair}(@(x, z), @(y, z)))))$ is a ground metaterm over $\mathcal{A}_\lambda$. Except for liberal use of parentheses and explicit notation for application, the ground metaterms over $\mathcal{A}_\lambda$ are exactly the terms of the corresponding λ-calculus.

Another alphabet $\mathcal{A}_{\mathsf{PL1}}$ is for first order predicate logic with equality:

$$\begin{aligned}
\mathcal{K}_{\mathsf{PL1}} &= \{a, b, c, \ldots, f_i, g_i, h_i, \ldots, p_i, q_i, r_i, \ldots, =, \mathsf{true}, \mathsf{false}, \neg, \wedge, \vee, \rightarrow\} \\
\mathcal{B}_{\mathsf{PL1}} &= \{\forall, \exists\} \\
\mathcal{V}_{\mathsf{PL1}} &= \{x, y, z, x_0, x_1, x_2, \ldots\} \\
\mathcal{M}_{\mathsf{PL1}} &= \{\, V_0,\, V_0',\, V_0'', \ldots,\, V_1,\, V_1',\, V_1'', \ldots,\, V_2,\, V_2',\, V_2'', \ldots, \ldots\}
\end{aligned}$$

$$\mathit{arity}_{\mathsf{PL1}}\ s = \begin{cases} 0 & \text{if } s \in \{a, b, c, \ldots, \mathsf{true}, \mathsf{false}\} \cup \mathcal{V}_{\mathsf{PL1}}, \\ 1 & \text{if } s \in \{\neg, \forall, \exists\}, \\ 2 & \text{if } s \in \{=, \wedge, \vee, \rightarrow\}, \\ i & \text{if } s \in \{f_i, g_i, h_i, \ldots, p_i, q_i, r_i, \ldots, V_i, V_i', \ldots\}. \end{cases}$$

---

the *binding structures* of Talcott [25], where one binder can bind several variables and differentiation is possible as to in which subterms the respective bound variables may occur.

Now for example $\forall x.(\exists y.(p_7(x,y)))$ is a ground metaterm over $\mathcal{A}_{\mathsf{PL1}}$. Indeed, with the necessary syntactic changes all terms and formulae of first order predicate logic are ground metaterms over $\mathcal{A}_{\mathsf{PL1}}$; in Sect. 3 we shall therefore not distinguish between the conventional predicate logic syntax and the rigorous metaterm syntax of this section.

But, $\forall x.(=(\mathsf{false}, f_1(\wedge(x,y))))$ is a legal ground metaterm over $\mathcal{A}_{\mathsf{PL1}}$, too, although it is pretty nonsensical in first order predicate logic, where it would usually be written $\forall x.\mathsf{false} = f_1(x \wedge y)$. The reason for this is that predicate logic distinguishes between "terms" and "formulae", i.e., is a many (two) sorted formalism. Since we wanted to keep our (meta-)term definition reasonably simple, we have not included respect to sort information. Therefore, these distinctions are outside our scope for the time being.[4]

Another effect we do not take care of at the moment is simultaneous quantification over several variables, as for example in the logic formula $\forall x, y.f_2(x,y) = f_2(y,x)$. Most formalisations of logics consider this kind of notation as a shorthand for separate quantifiers, but it is also possible to accept such multiple bound variables in a formal system, for example within the binding structures of [25] and in the *higher-order abstract syntax* of [8].

An important point of the subterm relation on metaterms is that bound variables *per se* are not subterms of a binding construct. The reason for this is that bound variables *per se* are not considered to be immediate constituents of a binding construct. We can define the sub-metaterm relation directly:

Definition 2.1.3 The **sub-metaterm relation** is the smallest relation between metaterms fulfilling:

i) every metaterm t is a sub-metaterm of itself;
ii) if t_0 is a sub-metaterm of t_i for any $i \in 1, \ldots, n$, then t_0 is a sub-metaterm of $c(t_1, \ldots, t_n)$ for every n-ary constant constructor or metavariable c and a sub-metaterm of $\beta v.(t_1, \ldots, t_n)$ for every n-ary binding constructor β and every bindable variable v. $\square$

In practice we shall often use the word "subterm" instead of "sub-metaterm". Subterms of $\lambda\, x\,.\,(\lambda\, y\,.\,(x))$ are exactly $\lambda\, x\,.\,(\lambda\, y\,.\,(x))$, $\lambda\, y\,.\,(x)$, and x, but not y.

Definition 2.1.4 A *position* is a sequence of natural numbers. Positions can be considered as paths selecting subterms within terms. We define this as follows:

- The subterm of t at $\langle\rangle$, the empty position, is t.
- The subterm of $c(t_1, \ldots, t_n)$ at $i : p$, the position that has i as its first element and p as tail sequence, is the subterm of t_i at p.
- The subterm of $\beta v.(t_1, \ldots, t_n)$ at $i : p$, the position that has i as its first element and p as tail sequence, is the subterm of t_i at p.

Again we intend this to define the smallest relation fulfilling these conditions. $\square$

The subterm of $c(d(a), c(b,b))$ at $\langle 2, 1\rangle$ is b. But it is not possible to say anything about "the subterm of a at $\langle 1, 2\rangle$", for example, and the binding occurrences of bindable variables do not have positions.

[4] See Sect. 3 for a distinction of "terms" and "formulae" in that sense.

We use positions for discussing *freeness* of bindable variables:

Definition 2.1.5 A **variable occurrence** is a pair (p, v) consisting of a position p and a bindable variable v. The set $\mathsf{FVO}(t)$ of **free variable occurrences** of a metaterm t over $\mathcal{A}$ is defined by:

i) if $t = x$ for $x \in \mathcal{V}$ then $\mathsf{FVO}(t) = \{(\langle\rangle, x)\}$,

ii) if $t = c(t_1, \ldots, t_n)$ for metaterms $t_1, \ldots, t_n$ and $c \in \mathcal{K} \cup \mathcal{M}$, then $\mathsf{FVO}(t) = \{i : \{1, \ldots, n\}; (p, x) : \mathsf{FVO}(t_i) \bullet (i : p, x)\}$,

iii) if $t = \beta v(t_1, \ldots, t_n)$ for metaterms $t_1, \ldots, t_n$ and $\beta \in \mathcal{B}$ and $v \in \mathcal{V}$, then $\mathsf{FVO}(t) = \{i : \{1, \ldots, n\}; (p, x) : \mathsf{FVO}(t_i) | v \neq x \bullet (i : p, x)\}$.

A bindable variable v is **free** in t iff it has a free occurrence in t, i.e., there is a position p such that $(p, v) \in \mathsf{FVO}(t)$.

A metaterm is called **closed** if it has no free bindable variables. □

A certain kind of metaterms is especially *well behaved*[5] when we come to matching and rewriting in the next section: the key property being that all metavariable constructs occurring in the metaterm in question are only allowed to have different bound variables as their immediate constituents:

Definition 2.1.6 A metaterm t is a **CRS-pattern** iff it is closed and if for every subterm t_0 of t that is of the shape $m(t_1, \ldots, t_n)$ for a metavariable m, all immediate constituents t_i are different bound variables, i.e., t_0 is of the shape $m(v_1, \ldots, v_n)$, where all v_i are bindable variables (and with the closedness of t therefore bound) and for all $i, j : \mathbb{N}$ with $1 \leq i < j \leq n$ the variables v_i and v_j are different.

A metaterm t is called **linear** if no metavariable occurs more then once in t. □

Examples of linear CRS-patterns in the metaterm alphabet $\mathcal{A}_{\mathsf{PL1}}$ for first order logic are $f_3(V_0, g_1(V_0'))$ with no binding and only zero-ary metavariables, $\forall x.(V_1(x))$ with a unary metavariable, and $\forall x.(\exists y.(V_2(x, y)))$ with a binary metavariable. Counterexamples are $V_1(x)$ which is not closed, $\forall x.(V_1(p_8(x)))$ where the unary metavariable does not have the bound variable as its immediate constituent, and $\forall x.(V_2(x, x))$ where the binary metavariable has twice the same bound variable as its immediate constituents.

2.2 Metaterm Rewriting and Combinatory Reduction Systems

Substitution in metaterms is considerably more complicated than substitution usually; traditionally it is presented in two stages:

Substitutions replace bindable variables with metaterms and are essentially substitutions as known in λ-calculus.

Valuations replace metavariables with "metaterm-segments" and are defined using substitutions.

[5] According to [22] it is decidable whether two CRS-patterns are unifiable; if they are unifiable, a most general unifier can be computed.

As an auxiliary concept we first define a special notion of a "term with one hole":

Definition 2.2.1 A **context** over a metaterm alphabet $(\mathcal{C}, arity, (\mathcal{K}, \mathcal{B}, \mathcal{V}, \mathcal{M}))$ (that by convention does not contain "$\square$" in $\mathcal{C}$) is a term t over the extended metaterm alphabet $(\mathcal{C} \cup \{\square\}, arity \cup \{(\square, 0)\}, (\mathcal{K}, \mathcal{B}, \mathcal{V}, \mathcal{M} \cup \{\square\}))$ where there is exactly one position p such that t contains $\square$ at p. $\square$

This definition is purely technical and says that a context over $\mathcal{A}$ is a term in an alphabet that is generated from $\mathcal{A}$ by adding a new metavariable $\square$, called a "hole", and this variable has to occur exactly once. One might argue, that considering $\square$ to be a bindable variable would be more appropriate, but, as put explained to more detail in [15], it really is a third kind of variable, and for the purposes of this paper we can live with the *ad-hoc* point of view.

Nevertheless, we cannot use metaterm substitution for $\square$, since context substitution in the presence of bound variables must be **allowed** to *capture free variables*, therefore the next definition does not need any renaming mechanism:

Definition 2.2.2 Let t be a term, x a variable, c an n-ary constant constructor or metavariable, β an n-ary binding constructor and $t_1, \ldots, t_n$ terms, then

$$\begin{aligned}
x[t] &= \begin{cases} t & \text{if } x = \square \\ x & \text{otherwise} \end{cases} \\
c(t_1, \ldots, t_n)[t] &= c(t_1[t], \ldots, t_n[t]) \\
\beta x.(t_1, \ldots, t_n)[t] &= \beta x.(t_1[t], \ldots, t_n[t])
\end{aligned}$$

$\square$

Holes are only used to serve as placeholders for subterms that are rewritten, therefore the name "context".

We first present a definition of substitutions:

Definition 2.2.3 A **metaterm substitution** is a finite domain partial function σ from bindable variables to metaterms.

Application of metaterm substitutions is extended to terms in a natural way: Let σ be a substitution, x a variable, c an n-ary constant constructor, β an n-ary binding constructor, m an n-ary metavariable and $t_1, \ldots, t_n$ terms, then

$$\begin{aligned}
\sigma\ x &= \begin{cases} \sigma\ x & \text{if } x \text{ is in the domain of } \sigma \\ x & \text{otherwise} \end{cases} \\
\sigma\ c(t_1, \ldots, t_n) &= c(\sigma\ t_1, \ldots, \sigma\ t_n) \\
\sigma\ \beta x.(t_1, \ldots, t_n) &= \beta z.(\sigma_1\ ([x \backslash z]\ t_1), \ldots, \sigma_1\ ([x \backslash z]\ t_n)) \\
&\quad \text{where } \sigma_1 = \sigma - [x \backslash \sigma\ x] \text{ and } z \text{ is a variable not ocurring} \\
&\quad \text{free in the range of } \sigma_1 \text{ nor, if } z \neq x, \text{ free in any of the } t_i \\
\sigma\ m(t_1, \ldots, t_n) &= m(\sigma\ t_1, \ldots, \sigma\ t_n)
\end{aligned}$$

$\square$

In the case of the binding constructor, if x does not occur in the range of σ_1, we obviously can chose $z = x$ and can thus spare the renaming. We might also have chosen to forbid z to occur in any of the t_i altogether instead of only forbidding it to occur freely there, and as a result the applications of the substitutions $[x \backslash z]$ would never require another renaming.

As usual in any term calculus with bound variables, the result of substitution is now determined uniquely only *up to consistent renaming of bound variables* (called α-conversion in λ-calculi).

Definition 2.2.4 For a given metaterm alphabet $\mathcal{A}$, we define another metaterm alphabet $\mathcal{A}^{\mathbb{N}}$ by adding enumerably many (zero-ary) bindable variables $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3, \ldots$, called **placeholders**, that, without restriction of generality, do not occur in $\mathcal{V}$. □

Definition 2.2.5 A **valuation** over a metaterm alphabet $\mathcal{A}$ is a finite partial function from metavariables, $\mathcal{M}$, into metaterms over $\mathcal{A}^{\mathbb{N}}$, such that placeholders only occur free in the images, and for every n-ary metavariable in the domain of a valuation, at most placeholders $\mathsf{s}_1, \ldots, \mathsf{s}_n$ occur in its image. □

Definition 2.2.6 The **application of a valuation v** to a metaterm t is defined as follows:

i) for a bindable variable $x : \mathcal{V}$ we have $\mathbf{v}\ x = x$,
ii) for an n-ary binder $\beta : \mathcal{B}$, a bindable variable $x : \mathcal{V}$ and metaterms $t_1, \ldots, t_n$ we have $\mathbf{v}\ \beta x.(t_1, \ldots, t_n) = \beta z.(\mathbf{v}\ ([x\backslash z]\ t_1), \ldots, \mathbf{v}\ ([x\backslash z]\ t_n))$, where z is a variable not ocurring free in the range of $\mathbf{v}$ nor, if $z \neq x$, free in any of the t_i,
iii) for an n-ary constructor $c : \mathcal{K} \cup \mathcal{M} - \mathrm{dom}\ \mathbf{v}$ and metaterms $t_1, \ldots, t_n$ we have $\mathbf{v}\ c(t_1, \ldots, t_n) = c(\mathbf{v}\ t_1, \ldots, \mathbf{v}\ t_n)$,
iv) for an n-ary metavariable $m : \mathcal{M}$ in the domain of $\mathbf{v}$ and metaterms $t_1, \ldots, t_n$ we have $\mathbf{v}\ (m(t_1, \ldots, t_n)) = [\mathsf{s}_1\backslash(\mathbf{v}\ t_1), \ldots, \mathsf{s}_n\backslash(\mathbf{v}\ t_n)](\mathbf{v}\ m)$. □

Definition 2.2.7 A **metaterm rewriting rule** or simple **rule** is a pair (l, r) of closed metaterms, in the following always written $\langle l \mapsto r \rangle$.

If l is linear, then $\langle l \mapsto r \rangle$ is a **left-linear** rule.

A rule is **applicable** to a metaterm t at position p, if there is a context t_0 with its $\square$ at p and a valuation $\mathbf{v}$ such that $t = t_0[\mathbf{v}\ l]$.

Let Φ be the set of all such valuations. Then there is (up to α-conversion) exactly one valuation $\mathbf{v}_0 \in \Phi$ with minimal domain since metaterms are finite by definition, which allows to define **rewriting** t **at** p with the rule $\langle l \mapsto r \rangle$ and the **valuation v** as yielding the result $t_0[\mathbf{v}_0\ r]$.

A **metaterm rewriting system** (**MRS**) is then a finite set of metaterm rewriting rules. □

The original definition of Klop stays on the safe side with respect to the choice of the left-hand side:

Definition 2.2.8 A **CRS rule** is a rule $\langle l \mapsto r \rangle$ where l is a CRS pattern and all metavariables occurring in r also occur in l.

A **Combinatory Reduction System** is a finite set of CRS rules. □

For CRS rules, the result of rewriting is determined uniquely without chosing a minimal valuation because of the condition that no metavariable occurring in

r does not occur in l. Linearity on the other hand is a condition that does not influence the definedness of rewriting. The requirement that the left-hand side be a CRS-pattern is useful, because that way the associated matching problem is unique and therefore easy to treat algorithmically.

As an example we give a CRS rule defining β-reduction of λ-calculus:

$$\left\langle @(\lambda\, x\,.\,(V_1(x)),\, V_0) \mapsto V_1(V_0)\right\rangle \ .$$

An application of this rule can be seen in the following figure:

$$\begin{array}{ccc} @(\lambda\, x\,.\,(V_1(x)),\, V_0) \;=\; l & \longrightarrow & r \;=\; V_1(V_0) \\ \downarrow & & \downarrow \\ t = t_0[\mathbf{v}_0\ l] & \longrightarrow & t_0[\mathbf{v}_0\ r] \;=\; t_1 \end{array}$$

With $t_0 = @(\square, c)$ and $\mathbf{v}_0 = [V_0 \backslash @(a, b),\, V_1(\mathsf{s}_1) \backslash \lambda\, y\,.\,(\mathsf{s}_1)]$, this example rewrites $t = @(@(\lambda\, x\,.\,(\lambda\, y\,.\,(x)), @(a, b)), c)$ at $\langle 1\rangle$ to $t_1 = @(\lambda\, y\,.\,(@(a, b)), c)$. This result can again be rewritten using the same rule; the redex is at $\langle\rangle$, so the context is just $\square$, the matching valuation is $[V_0 \backslash c,\, V_1(\mathsf{s}_1) \backslash @(a, b)]$, and the result of the rewriting step is $@(a, b)$.

The CRS rule for η-reduction, $\left\langle \lambda\, x\,.\,(@(V_0, x)) \mapsto V_0\right\rangle$, does not even use a second order metavariable, but nevertheless it is not an ordinary term rewriting rule since the precise definition of metaterm rewriting rule application implies the condition that is usually spelled out, namely that the bound variable x may not occur "in V_0", i.e., in the image of V_0 under the matching valuation.

3 The Use of Second-Order Constructs in RALF

In this section we show an application of second-order metavariables in the metatheory of relation algebra implemented in RALF.

3.1 Why we need second-order constructs

To fix the notation we start with a definition of relation algebra in a style akin to that of [23], but in the notation of [4].

Definition 3.1.1 A (heterogeneous) **relation algebra** $(\mathcal{R}, \sqcup, \sqcap, \overline{}, \breve{}\ , ;)$ is a self-dual category where the homset from A to B is denoted $A \leftrightarrow B$; for two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, we denote their composition with $R;S$, and for an object A of the category, $\mathbb{I} = \mathbb{I}_A$ denotes the *identity relation.*

Furthermore every homset $A \leftrightarrow B$ has to be a complete atomic boolean algebra with *union* "$\sqcup$", *intersection* "$\sqcap$", *negation* "$\overline{}$", *inclusion* "$\sqsubseteq$", *zero element* $\mathbb{L} = \mathbb{L}_{A\leftrightarrow B}$ and *universal element* $\mathbb{T} = \mathbb{T}_{A\leftrightarrow B}$.

The dual arrow $R\breve{}\ : B \leftrightarrow A$ for an arrow (relation) $R : A \leftrightarrow B$ is also called the *transposed* (or *converse*) *relation*, and the *Dedekind rule* $(Q;R \sqcap S) \sqsubseteq (Q \sqcap S;R\breve{}\,);(R \sqcap Q\breve{}\,;S)$ and the *Tarski rule* $R \neq \mathbb{L} \;\leftrightarrow\; \mathbb{T};R;\mathbb{T} = \mathbb{T}$ hold.$\square$

To explain the typical proof-style of relation algebra two example proofs are given. Both proofs were carried out in the system RALF and the proof listings are generated automatically from the internal proof tree. The first proposition states an often used fact about suprema in a complete lattice.

Proposition 3.1.2 Let R be a relation, then

$$\bigsqcup\{y \mid V_1(y)\} \sqsubseteq Z \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z)\ .$$

Proof:

$$\begin{array}{ll}
 & \vdash \bigsqcup\{y \mid V_1(y)\} \sqsubseteq Z \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z) \\
\Longleftarrow & \quad \{\ \ \vdash \underline{\bigsqcup\{x \mid V_1(x)\}} = \overline{\bigsqcap\{y \mid V_1(\overline{y})\}}\quad \} \\
 & \vdash \bigsqcap\{A \mid V_1(\overline{A})\} \sqsubseteq Z \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z) \\
\Longleftarrow & \quad \{\ \ \vdash \overline{R} \sqsubseteq S \;\leftrightarrow\; \overline{S} \sqsubseteq R\quad \} \\
 & \vdash \overline{Z} \sqsubseteq \bigsqcap\{A \mid V_1(\overline{A})\} \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z) \\
\Longleftarrow & \quad \{\ \ \vdash R \sqsubseteq \bigsqcap\{x \mid V_1(x)\} \;\leftrightarrow\; \forall y.(V_1(y) \;\rightarrow\; R \sqsubseteq y)\quad \} \\
 & \vdash \forall B.(V_1(\overline{B}) \;\rightarrow\; \overline{Z} \sqsubseteq B) \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z) \\
\Longleftarrow & \quad \{\ \ \vdash \overline{R} \sqsubseteq S \;\leftrightarrow\; \overline{S} \sqsubseteq R\quad \} \\
 & \vdash \forall B.(V_1(\overline{B}) \;\rightarrow\; \overline{B} \sqsubseteq Z) \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z) \\
 & \quad \{\ \ \vdash \forall x.(V_1(\overline{x})) \;\leftrightarrow\; \forall z.(V_1(z))\quad \}
\end{array}$$

□

In the next proposition, the largest solution to $R;X \sqsubseteq S$ is described by a relational term. In the proof the proposition above is applied.

Proposition 3.1.3 Let R and S be relations, then $\overline{R^{\smile};\overline{S}} = \bigsqcup\{x \mid R;x \sqsubseteq S$. $\mathsf{RRes}(R,S) := \overline{R^{\smile};\overline{S}}$ is called *right residue* of S by R.

Proof:

$$\begin{array}{ll}
 & \vdash \overline{R^{\smile};\overline{S}} = \bigsqcup\{x \mid R;x \sqsubseteq S\} \\
\Longleftarrow & \quad \{\ \ \vdash \underline{R = S} \;\leftrightarrow\; \forall z.(R \sqsubseteq z \;\leftrightarrow\; S \sqsubseteq z)\quad \} \\
 & \vdash \forall A.(\overline{R^{\smile};\overline{S}} \sqsubseteq A \;\leftrightarrow\; \bigsqcup\{x \mid R;x \sqsubseteq S\} \sqsubseteq A) \\
\Longleftarrow & \quad \{\ \ \vdash \underline{\bigsqcup\{y \mid V_1(y)\} \sqsubseteq Z} \;\leftrightarrow\; \forall x.(V_1(x) \;\rightarrow\; x \sqsubseteq Z)\quad \} \\
 & \vdash \forall A.(\overline{R^{\smile};\overline{S}} \sqsubseteq A \;\leftrightarrow\; \forall B.(R;B \sqsubseteq S \;\rightarrow\; B \sqsubseteq A)) \\
\Longleftarrow & \quad \{\ \ \vdash \underline{Q;R \sqsubseteq S} \;\leftrightarrow\; Q^{\smile};\overline{S} \sqsubseteq \overline{R}\quad \} \\
 & \vdash \forall A.(\overline{R^{\smile};\overline{S}} \sqsubseteq A \;\leftrightarrow\; \forall B.(R^{\smile};\overline{S} \sqsubseteq \overline{B} \;\rightarrow\; B \sqsubseteq A)) \\
\Longleftarrow & \quad \{\ \ \vdash \underline{R \sqsubseteq \overline{S}} \;\leftrightarrow\; S \sqsubseteq \overline{R}\quad \} \\
 & \vdash \forall A.(\overline{R^{\smile};\overline{S}} \sqsubseteq A \;\leftrightarrow\; \forall B.(B \sqsubseteq \overline{R^{\smile};\overline{S}} \;\rightarrow\; B \sqsubseteq A)) \\
\Longleftarrow & \quad \{\ \ \vdash \underline{R \sqsubseteq S} \;\leftrightarrow\; \forall z.(\underline{z \sqsubseteq R} \rightarrow z \sqsubseteq S)\quad \} \\
 & \vdash \forall A.(\overline{R^{\smile};\overline{S}} \sqsubseteq A \;\leftrightarrow\; \overline{R^{\smile};\overline{S}} \sqsubseteq A) \\
 & \quad \{\ \ \vdash V_1 \;\leftrightarrow\; V_1\quad \}
\end{array}$$

□

Both examples show the characteristic algebraic proof style: subexpressions are replaced by equivalent expressions. However in relation algebra, a subexpression eventually may also be replaced by a greater or smaller[6] expression depending on the monotonicity properties of the functions (we consider here the logical connectives as functions of type *boolean*^{n} $\mapsto$ *boolean*, $n \in \{0,1,2\}$) the subexpression is an argument of. A calculus for this proof style and a description of a

[6] with respect to the implication or relation ordering.

new class of rewrite systems to implement this calculus is described in [9]. Here however, we want to restrict ourselves to context independent equivalence rules as in the examples above.

In the example proofs above some of the applied theorems are syntactically not terms but metaterms in the sense of Sect. 2.1, as for example

$$\bigsqcup\{x \mid V_1(x)\} = \overline{\bigsqcap\{y \mid V_1(\overline{y})\}} \tag{1}$$

$$R \sqsubseteq \bigsqcap\{x \mid V_1(x)\} \leftrightarrow \forall y.(V_1(y) \rightarrow R \sqsubseteq y) \tag{2}$$

From the term rewriting point of view V_1 can be considered as a unary metavariable and $\bigsqcup$ and $\bigsqcap$ can be considered as binding constructors.[7] These binding constructors themselves can be defined using metavariables and the usual binding constructors of predicate logic $\forall$ and $\exists$:

$$\begin{aligned}
A = \bigsqcup\{x \mid V_1(x)\} &: \leftrightarrow \\
&\forall x.(V_1(x) \rightarrow x \sqsubseteq A) \wedge \forall y.(\forall x.(V_1(x) \rightarrow x \sqsubseteq y) \rightarrow A \sqsubseteq y) \\
A = \bigsqcap\{x \mid V_1(x)\} &: \leftrightarrow \\
&\forall x.(V_1(x) \rightarrow A \sqsubseteq x) \wedge \forall y.(\forall x.(V_1(x) \rightarrow y \sqsubseteq x) \rightarrow y \sqsubseteq A)
\end{aligned}$$

Furthermore, when dealing with quantifiers it is desirable to be able to formulate and apply sentences like

$$\forall x.(V_1(x)) \leftrightarrow \neg\exists x.(\neg V_1(x)) \tag{3}$$

$$(V_0 \rightarrow \forall x.(V_1(x))) \leftrightarrow \forall x.(V_0 \rightarrow V_1(x)) \text{ if } x \text{ is not free in the formula } V_0 \tag{4}$$

In the next section we will show that using MRSs we do not have to consider the sidecondition on equivalence (4) explicitly.

3.2 Relation Algebra as a MRS

We first define the metaterm alphabet $\mathcal{A}_{\mathsf{Rel}}$ for the metatheory of relation algebra and then show how the rewriting formalism of Sect. 2 enables us to drop the explicit mention of the sidecondition on equivalence (4), since it is implied by the second-order matching definition.

$\mathcal{A}_{\mathsf{Rel}}$ is an (two) sorted extension of $\mathcal{A}_{\mathsf{PL1}}$ with subscripts indicating the sort:

$$\begin{aligned}
&\mathcal{K}_{\mathsf{Rel}_{(o,o)}} = \{\sqcap, \sqcup, ;, \breve{}\,, \overline{}, \perp\!\!\!\perp, \mathbb{I}, \top\!\!\!\top, f_0, f_1, \ldots, g_0, g_1 \ldots, \ldots\}\ , \\
&\mathcal{K}_{\mathsf{Rel}_{(o,b)}} = \{\sqsubseteq, =, p_0, p_1, \ldots, q_0, q_1 \ldots, \ldots\}\ , \\
&\mathcal{K}_{\mathsf{Rel}_{(b,b)}} = \{\mathsf{true}, \mathsf{false}, \neg, \wedge, \vee, \leftrightarrow, \rightarrow\}\ ,\ \mathcal{K}_{\mathsf{Rel}} = \mathcal{K}_{\mathsf{Rel}_{(o,o)}} \cup \mathcal{K}_{\mathsf{Rel}_{(o,b)}} \cup \mathcal{K}_{\mathsf{Rel}_{(b,b)}} \\
&\mathcal{B}_{\mathsf{Rel}_{(o)}} = \{\bigsqcup, \bigsqcap\}\ , \qquad \mathcal{B}_{\mathsf{Rel}_{(b)}} = \{\forall, \exists\}\ , \qquad \mathcal{B}_{\mathsf{Rel}} = \mathcal{B}_{\mathsf{Rel}_{(o)}} \cup \mathcal{B}_{\mathsf{Rel}_{(b)}} \\
&\mathcal{V}_{\mathsf{Rel}} = \{x, y, z, x_0, x_1, \ldots\} \\
&\mathcal{M}_{\mathsf{Rel}_{(o,b)}} = \{V_0, V_1, \ldots, W_0, W_1, \ldots, \ldots\}\ ,\ \mathcal{M}_{\mathsf{Rel}_{(o)}} = \{R, S, T, R_0, R_1, \ldots\}\ , \\
&\mathcal{M}_{\mathsf{Rel}} = \mathcal{M}_{\mathsf{Rel}_{(o)}} \cup \mathcal{M}_{\mathsf{Rel}_{(o,b)}}
\end{aligned}$$

---

[7] Otherwise we would have to treat set comprehension seperately as a binding constructor, with $\bigsqcup$ and $\bigsqcap$ as unary constant constructors — however this would have certain technical disadvantages beyond the scope of this paper.

$$arity_{\mathsf{Rel}}\ s = \begin{cases} 0 & \text{if } s \in \{\perp\!\!\!\perp, \mathbb{I}, \mathbb{T}, \mathsf{true}, \mathsf{false}\} \cup \mathcal{V}_{\mathsf{Rel}} \cup \mathcal{M}_{\mathsf{Rel}_{(o)}}\ , \\ 1 & \text{if } s \in \{\bigsqcup, \bigsqcap, \breve{}\ , \overline{}, \neg, \forall, \exists\}\ , \\ 2 & \text{if } s \in \{\sqcap, \sqcup, \sqsubseteq, =, \wedge, \vee, \leftrightarrow, \rightarrow\}\ , \\ i & \text{if } s \in \{f_i, g_i, \ldots, p_i, q_i, \ldots, V_i, W_i, \ldots\}\ . \end{cases}$$

For proper dealing with a logical calculus, the metaterms have to be partitioned into "terms" and "formulae" excluding illegal constructions, as already mentioned in Sect. 2.1:

Definition 3.2.1

- Every $x \in \{\perp\!\!\!\perp, \mathbb{I}, \mathbb{T}, f_0, g_0, \ldots\} \cup \mathcal{V}_{\mathsf{Rel}} \cup \mathcal{M}_{\mathsf{Rel}_{(o)}}$ is a term.
- If $t_1, \ldots, t_n$ are terms and $f_n \in \mathcal{K}_{\mathsf{Rel}_{(o,o)}}$ then $f_n(t_1, \ldots, t_n)$ is a term.
- Every $\phi \in \{\mathsf{true}, \mathsf{false}, V_0, W_0, \ldots\}$ is a formula.
- If $t_1, \ldots, t_n$ are terms and $\phi_n \in \mathcal{K}_{\mathsf{Rel}_{(o,b)}} \cup \{V_n, W_n, \ldots\}$ then $\phi_n(t_1, \ldots, t_n)$ is a formula.
- If $x \in \mathcal{V}_{\mathsf{Rel}}$ and ϕ is a formula then $\forall x.(\phi)$ and $\exists x.(\phi)$ are formulae, and $\bigsqcup\{x \mid \phi\}$ and $\bigsqcap\{x \mid \phi\}$ are terms.

e is a subterm (resp. sub-formula) of E iff e is a sub-metaterm of E and e is a term (resp. formula). Every (sub-)term and every (sub-)formula is an expression (subexpression). □

To ensure well-formedness of expressions after the application of a valuation we have to adapt the definition of substitution and valuation: if x is variable, σ a substitution then σx has to be a term; for every valuation $\mathbf{v}$ restricted on $\mathcal{M}_{\mathsf{Rel}_{(o)}}$ the valuation $\mathbf{v}$ has only terms as values, and restricted on $\mathcal{M}_{\mathsf{Rel}_{(o,b)}}$ only formulae.

In RALF, the application of a theorem is implemented as an application of a metaterm rewrite rule. As we are so far not interested in automated deduction and therefore do not have to deal with termination problems, we will consider every sentence of the shape "formula $\leftrightarrow$ formula" or "term = term" as two rewrite rules. For example sentences (1) and (4) lead to the rewrite rules

$$\left\langle \bigsqcup\{x \mid V_1(x)\} \mapsto \overline{\bigsqcap\{y \mid V_1(\overline{y})\}} \right\rangle \tag{5}$$

$$\left\langle \overline{\bigsqcap\{y \mid V_1(\overline{y})\}} \mapsto \bigsqcup\{x \mid V_1(x)\} \right\rangle \tag{6}$$

$$\left\langle V_0 \rightarrow \forall x.(V_1(x)) \mapsto \forall x.(V_0 \rightarrow V_1(x)) \right\rangle \tag{7}$$

$$\left\langle \forall x.(V_0 \rightarrow V_1(x)) \mapsto V_0 \rightarrow \forall x.(V_1(x)) \right\rangle \tag{8}$$

Notice that the left hand side of rule (6) is not a CRS-pattern, but, as pointed out in Sect. 4.4, matching is still univalent under the slightly more general condindition of "unique encapsulation".

The semantics of the constructors is as usual in relation algebra and first order logic. Formula metavariables correspond to expressions of the shape $\psi(x_1, \ldots, x_n)$ in the meta-language of logic.

The side condition on equivalence (4) can be dropped as there is no valuation mapping V_0 on a formula not fulfilling the side condition. For example, an incorrect rewrite step (let $\longrightarrow_R$ indicate a rewrite step) like

$$\forall x.(x = \mathbb{I} \;\rightarrow\; \mathbb{I} = \mathbb{I} ; x) \qquad \longrightarrow_R \qquad x = \mathbb{I} \;\rightarrow\; \forall x.(\mathbb{I} = \mathbb{I} ; x)$$

is not possible as there is no valuation $\mathbf{v}$ with

$$\mathbf{v}(\forall x.(V_0 \;\rightarrow\; V_1(x))) = \forall x.(x = \mathbb{I} \;\rightarrow\; \mathbb{I} = \mathbb{I} ; x) \ .$$

We can even go beyond deterministic valuations since in an interactive system we can communicate with the user while finding the valuation. An example for a useful logical rule that matches non-deterministically is

$$\big\langle V_1(T) \mapsto \forall x.(V_1(x)) \big\rangle \ .$$

4 Second-Order Term Graph Rewriting in HOPS

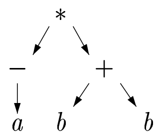
The first few subsections of this section serve to motivate the way we formalise term graphs with bound variables; this formalisation is then reflected very directly in the term graph interaction on the user interface of HOPS.

Terms can be considered as trees, i.e., as a special kind of graphs, in a natural way, which we discuss in 4.1. We choose to introduce our view on variable binding and variable identity while still talking about trees, in order to present the unfamiliar aspects in front of a more familiar background.

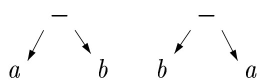
From trees it is only a small step to directed acyclic graphs (DAGs), taken in 4.2; this is formalised in 4.3. Finally we look into a special aspect of unique matching of rules in 4.4.

4.1 Term Trees with Bound Variables

Terms can be considered as labelled trees[8]. For example, the term $(-a) * (b + b)$ can be considered as the labelled tree depicted top the right. Strictly speaking, a, b, $*$, $+$ and $-$ are *node labels* of nodes whose identifiers we do not care about, and the edges are labelled with the ordinal number corresponding to the position of their appearance in the drawing among the outgoing edges of their source node.



This labelling is necessary because otherwise for example the following two trees would be undistinguishable, i.e., isomorphic (as graphs), while the corresponding terms $a - b$ and $b - a$ are clearly different.



The idea behind regarding terms as labelled trees is moving the *structure* of terms into the graph level, while leaving the *contents* in the node labels. With

[8] When we say "tree", we strictly speaking mean "arbourescence" in the graph-theoretic sense, i.e., a directed acyclic graph where every node except one (the root) has exactly one predecessor.

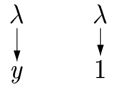
ordinary terms, the only structure to be considered is the immediate constituent relation — or the relation between occurrences of function symbols (*constructors*) and their arguments — together with the argument ordering. The node labels themselves do not reflect structure, but content.

When we move to the metaterms of CRSs or any kind of terms with bound variables, then, first of all, variable binding has to be considered. In the λ-term $t_1 := \lambda x.\, y$, the variable x is not a subterm of t_1; the only nontrivial subterm is y. Therefore, in the tree representation there should be no node representing x.

One way to achieve this was already used by Wadsworth, the inventor of graph reduction [27], who would have represented t_1 by a two-node term graph with the root labelled "λx" and its successor "y".

The problem with this approach is, that the variable binding, which is an essential ingredient of the structure of the λ-term, is not reflected in the structure of the graph, but has been coded into the node labels which originally should only carry structure-independent contents.

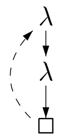
The same effect results from working with de Bruijn indices. Here, the two nodes of the graph representing t_1 would be labelled "λ" and "y" respectively, whereas the tree for the λ-term $\lambda x.\, x$, or $\lambda 1$ in de Bruijn notation, would have "λ" and "1" respectively — again, the variable binding structure is coded into the node labels. In all these approaches, the binding is *derived* from the node labels and other non-primitive structure elements such as reachability.



A *primitive binding concept* cleanly separating structure from contents seems to appear only in the foundations of Bourbaki [3]. There, "*assemblages*" are considered instead of terms, variables are all denoted by the symbol "$\square$", i.e., there are no variable names, and variable binding is denoted by "*liens*", links drawn above the symbol sequence.[9] An example is $\lambda\lambda\square$ for what usually would be $\lambda\, x\, .\, \lambda\, y\, .\, x$.

From an abstract point of view, the "*liens*" of Bourbaki are an additional graph structure superimposed over the tree structure of terms. When moving from terms to term trees, this additional structure induces a *second kind of edges* between the original term tree nodes.

We draw these *binding edges* as dashed, usually curved, edges in our example graphs — only the underlying graph that does not consider binding is still a tree. The above example $\lambda\, x\, .\, \lambda\, y\, .\, x$ then becomes:

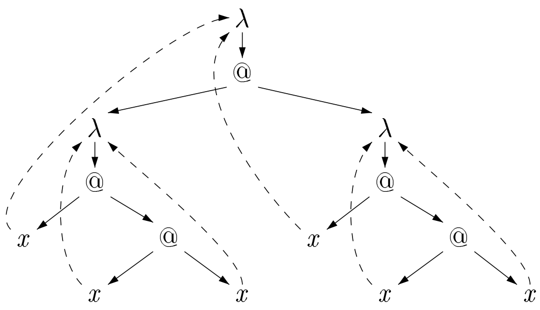


This is the approach we want to follow here, and it means that we move the binding structure explicitly onto the graph level, thus introducing a *primitive binding concept*. We shall, however, use the label "x" instead of "$\square$" for bound variables (at least in the λ-calculus).

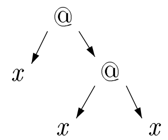
[9] The rôle assigned to the "*lettres*" in the axioms can be interpreted as the view that there are no free variables — "*lettres*" are something essentially different from the variables denoted by the symbol "$\square$".

Trying to represent more λ-terms as enriched term trees like that, one encounters a problem with multiple occurrences of variables. A bound variable that occurs several times is not really a problem, as the enriched tree to the right shows for the standard fixed-point combinator Y. Here, variable nodes bound at the same λ-node belong to the same variable, the same way as in Bourbaki, and, as is generally the case in λ-calculus, different bound variables are bound at different λ-nodes.

$$\lambda f \,.\, (\lambda x \,.\, (f\ (x\ x))\ \lambda x \,.\, (f\ (x\ x)))$$

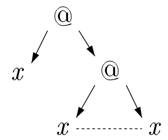


There is, however, corresponding to the absence of unbound occurrences of "□" in Bourbaki, no way to distinguish different free variables — the subterm $(f\ (x\ x))$ of the Y-combinator represented as the subtree to the right obviously has lost some structure that had not been made explicit before, namely the structure of **variable identity**. If we proceed to add that structure to our graphs, too, we advance beyond Bourbaki in the homogeneity and universality of our approach.

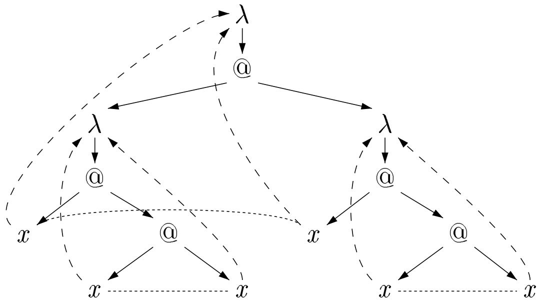


Variable identity obviously has to be a partial equivalence relation on the tree nodes, and has to affect exactly the variable nodes. It can be seen as partitioning the set of all *variable nodes* into equivalence classes that can each be considered as one *variable*.

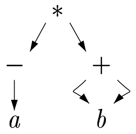
We shall usually not draw the reflexive part of variable identity, which can be deduced from the labels (i.e., every variable node stands in the relation of variable identity with itself), and only a skeleton of its irreflexive part. In the enriched tree corresponding to $(f\ (x\ x))$, the relation of variable identity thus partitions the three variable nodes into two variables corresponding to f and x respectively, the latter consisting of two variable nodes.

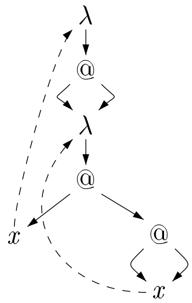


Variable identity should then also be made explicit for bound variables, and our term tree for the fixed-point combinator changes to the following:



4.2 From Trees to DAGs

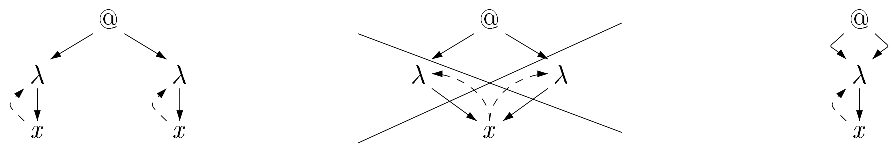




Identification of equal subterms is a means of efficiently representing terms in a data structure. This identification replaces several isomorphic subtrees by only one of them and redirects all edges that pointed to one of those subtrees to the selected copy, thus turning the tree into a *directed acyclic graph* (DAG).

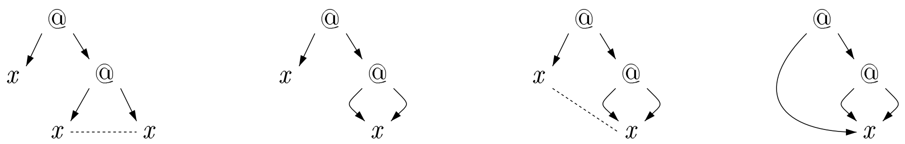
In our introductory example, only identification of the two b nodes is possible, while the fixed-point combinator offers more sharing opportunities. Noticeably, variable identity here becomes trivial again since we always have been able to share different variable nodes.

An important point to be aware of is that bound variables can only be shared together with their binder, making the DAG drawn in the middle illegal:



The graph-theoretic formulation for this condition is that binding has to be univalent and that the binder must always *dominate* any bound variables.

For an analogous reason, different free variables may not be shared, either:



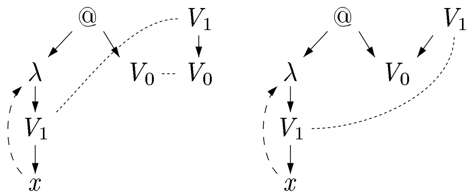
Although in this example all four DAGs are legal, the first two correspond to the term $(f\ (x\ x))$ containing two different variables, while the last two correspond to the term $(x\ (x\ x))$ with only one variable — these are just different terms, so that there is no legal sharing step leading from one to the other.

In all the examples seen so far, sharing could make the variable identity trivial. This ceases to be the case as soon as we start to consider variables that are not zero-ary. The most prominent example of such variables are the metavariables of combinatory reduction systems.

The rule of β-reduction has been presented in 2.2 as

$\langle @(\lambda\, x\, .\, (V_1(x)),\, V_0) \mapsto V_1(V_0) \rangle$

with metavariables V_0 and V_1. We have drawn the β-rule as term "forest" resp. as term DAG to the right. No further sharing is possible, since the two different V_1-nodes have different successors, so variable identity is in general not superfluous in term graphs.



Cyclic term graphs can be used to code recursion, and this has been done for quite some time in implementations of functional programming languages. Recently, the topic has become the subject of closer investigation in the context of rational term rewriting systems [19, 12, 7]. Although in [14, 15, 18] we have considered general term graphs without restriction to acyclicity, HOPS so far only accepts acyclic term graphs, and we shall not consider cycles in this paper.

4.3 Formal Definition of Term DAGs

For a formal treatment of HOPS DAGs, we first define alphabets corresponding to those of Def. 2.1.1. However, we do not introduce a separate class of binders.

Definition 4.3.1 A **term graph alphabet** is a tuple $(\mathcal{N}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$ with

i) $\mathcal{N}$, the set of **node labels**,

ii) $A : \mathcal{N} \to \mathbb{N}$, the **arity function**, [10]

[10] We use parts of the Z-notation [24], e.g. for set comprehensions where Z uses the pattern "{ *signature*| *predicate* • *term*}". E.g., $\{n : \mathbb{N} \mid n < 4 \bullet n^2\} = \{0, 1, 4, 9\}$. If the predicate is constantly true, then Z allows to just write "{ *signature* • *term*}", e.g. $\{x : \mathbb{B} \bullet (x, x)\} = \{(\mathbf{T}, \mathbf{T}), (\mathbf{F}, \mathbf{F})\}$; if the *term* is just the tuple of the variables introduced in the *signature*, then we write "{ *signature*| *predicate*}", e.g. $\{x, y : \mathbb{B} \mid x \neq y\} = \{(\mathbf{T}, \mathbf{F}), (\mathbf{F}, \mathbf{T})\}$. Quantification uses the same patterns, e.g. $\forall x : \mathbb{N} \bullet x + 1 > x$.

The set of relations between two sets A and B is written $A \leftrightarrow B$ and is equal to the power set of the cartesian product: $(A \leftrightarrow B) := \mathbb{P}((A \times B))$. We use the notation of Def. 3.1.1 for these *concrete relations*, too. The set of univalent relations or partial functions from A to B is written $A \nrightarrow B$, and that of total functions or mappings $A \to B$. Application of a function $f : A \nrightarrow B$ to an argument $x : A$ is written "$f\ x$" and is only used if the argument is known to be in the domain of the function: $x \in \operatorname{dom} f$, where $\operatorname{dom} R := \{(x, y) : R \bullet x\}$ for any relation R, and the range of R is $\operatorname{ran} R := \{(x, y) : R \bullet y\}$.

A^* is the set of finite sequences of elements of A; these sequences are considered to be partial functions of type $\mathbb{N} \nrightarrow A$ with contiguous domain which, when nonempty, always includes zero; therefore, if l is a sequence, then $l\ i$ denotes the $(i + 1)$-th element of l. The function $\operatorname{len} : A^* \to \mathbb{N}$ calculates the length of sequences.

When $R : A \leftrightarrow A$ is considered as a **graph**, node $y : A$ is *reachable* from node $x : A$ if and only if $(x, y) \in R^*$. Node r is a *root* if $r \notin \operatorname{ran} R$ and if all nodes $x : A$ are reachable from r. Relation R is *acyclic* if $R^+ \sqsubseteq \overline{\mathbb{I}}$.

iii) $\mathcal{C} : \mathbb{P}(\mathcal{N})$, the set of labels for **constant constructors**,
iv) $\mathcal{B} : \mathbb{P}(\mathcal{N})$, the set of labels for **bindable variables**,
v) $\mathcal{M} : \mathbb{P}(\mathcal{N})$, the set of labels for **metavariables**,
where $\{\mathcal{C}, \mathcal{B}, \mathcal{M}\}$ is a partition of $\mathcal{N}$. □

In the sequel we assume a fixed term graph alphabet $(\mathcal{N}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$.

While conventional term graph definitions only contain node set, node labelling and successor function, we extend this here with explicit variable binding and variable identity, and we have to add conditions that ensure that these faithfully mirror the corresponding variable binding and variable identity concepts on terms. In the result we obtain the expressivity of metaterms without the complexity of the coding schemes using variable names and without the frequent necessity of having to rename variables.

The following definition is an adapted and less general version of that in [15, 18]:

Definition 4.3.2 A **term DAG** is a tuple $G = (\mathcal{V}, M, S, D, B, W)$ with
i) $\mathcal{V}$, the finite **node set**,
ii) $M : \mathcal{V} \to \mathcal{N}$, the **node labelling** function,
iii) $S : \mathcal{V} \to \mathcal{V}^*$, the **successor** function with $M ; A = S ; \mathrm{len}$, i.e., the length of the successor list of each node has to be the arity of its label,
iv) $D : \mathcal{V} \leftrightarrow \mathcal{V}$, the **associated relation**, $D := \{(x, l) : S;\ y : \mathrm{ran}\, l \bullet (x, y)\}$; obviously D is not a primitive component but derived from S; it is listed here for its importance, and we demand that D is acyclic,
v) $B : \mathcal{V} \nrightarrow \mathcal{V}$, the **binding** function, where for $(x, b) \in B$, the *bound variable* x has to have a label in $\mathcal{B}$, the *binder* b has to have a label in $\mathcal{C}$, and b *dominates*[11] x in the graph induced by D.
vi) $W : \mathcal{V} \leftrightarrow \mathcal{V}$ is the **variable identity**, a *partial equivalence relation*[12] defined exactly on *variables*, i.e. on nodes with labels from $\mathcal{B} \cup \mathcal{M}$. The variable identity has to be compatible with the labelling: $W ; M \sqsubseteq M$, and with the binding[13]: $W ; B \sqsubseteq B$. □

Roots in term DAGs are considered wrt. D.

Metaterms according to Def. 2.1.2 that correspond to a rooted term DAG are easily recovered by unfolding recursively along the D-Paths from the root — creation of a unique name for every variable is the easiest means to ensure preservation of the binding and variable identity structure.

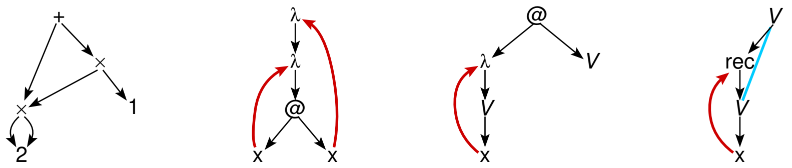
Consider the following examples, taken directly from HOPS, where successor edges are black arrows with their sequence indicated by the left-to-right order

[11] In graph theory, a node b dominates another node x, if for every node a and every path from a to x either b lies on that path or a is reachable from b. Domination therefore implies reachability.

[12] A partial equivalence relation is a symmetric and transitive relation.

[13] This condition, $W ; B \sqsubseteq B$, means that if any node in an equivalence class wrt. W is bound by some binder, then all nodes in that class are bound by the same binder — note the conciseness and elegance of the relational formulation!

of their attachment to their source node; binding edges are drawn as thick, dark grey, usually curved arrows and an irreflexive kernel of variable identity is indicated by thick medium gray lines[14]:



The first two correspond to the terms "$2 \cdot 2 + 2 \cdot 2 \cdot 1$" from arithmetic and "$\lambda x.\lambda f.f\ x$" from λ-calculus. For the last two, let us assume that A and B are metavariables — in HOPS, arity is part of the node label, so that unary and zero-ary metavariables all are drawn with the label V, but according to their arity they should be considered as different labels V_0, $\mathsf{V}_1 : \mathcal{M}$ in the examples. The last two term graphs then correspond to the metaterms "$(\lambda x.B[x])\ A$" (the left-hand side of the rule of β-reduction) and "$B[\mathrm{rec}\, x.B[x]]$", respectively.

Definition 4.3.3 A variable node x is **free below** a node a, if there is a D-path from a to x such that no binder of x lies on that path; if in this constellation x is bound by b, then b **encapsulates** a. □

In conventional term graph formalisms there are no bound variables, and variables corresponding to our metavariables are always zero-ary. Therefore, when considering homomorphisms based on node mappings, the image of such a variable is the whole subgraph starting at the image node of the variable node.

In the case of second-order term graphs however, there are metavariables with successors, and their images have to *stop* before the image nodes of their successors. Therefore we introduce:

Definition 4.3.4 An **interval** in a graph G is a pair $(t, b) : (\mathcal{V} \times (\mathbb{N} \nrightarrow \mathcal{V}))$ consisting of a *top node* t and a finite *lower border* b, which should be considered as a partial node sequence.

The **inner nodes** of the interval (t, b) are those nodes that are D-reachable from t via paths on which there lies no node of $\mathrm{ran}\, b$.

The interval (t, b) is **coherent** if all nodes in the lower border (i.e. all nodes in $\mathrm{ran}\, b$) are D-reachable from t. □

Not every interval is a reasonable candidate for being image of metavariables; conditions corresponding to "no capture of variables" have to be fulfilled. Auxiliary concepts for dealing with this issue are:

Definition 4.3.5 The interval (t, b) is **consistent**, if all nodes encapsulated by inner nodes are inner nodes.

The **encapsulation skeleton** of the interval (t, b) is the set of those inner nodes of (t, b), from which a node in the lower border can be reached via a $(B \sqcup D)$-path. □

[14] For colour media, HOPS uses colours to distinguish the different lines.

The most important use of term graph homomorphisms is to serve as matchings from rule sides into application graphs for transformation or rewriting. In the term context, matching is usually defined as "there exist a context and a (second-order) substitution, such that the result of inserting the substituted rule side into the context is α-equivalent to the application term" — with the definitions of substitution application and insertion into contexts taking care of avoiding "variable capture".

In term graphs, a more direct approach is necessary, and in second-order term graphs the structure that is not there is almost as important as the structure that is there, so the conditions for structure preservation take on an unusual shape, and for avoiding "variable capture" several special conditions are needed.

While in [15] we worked only with total functions and in [18] we went all the way to possibly partial and multivalent relations, here we just present a definition using potentially partial functions. This still allows a reasonably simple treatment of the images of metavariables:

Definition 4.3.6 If for two graphs G_1 and G_2, a partial function $F : \mathcal{V}_1 \nrightarrow \mathcal{V}_2$ is given, then the **image interval** for a metavariable node $m : \mathcal{V}_1$ with $m \in \operatorname{dom} F$ and $M_1\ m \in \mathcal{M}$ is defined to be the interval $(F\ m, (S_1\ m) ; F)$. □

The fact that we only consider acyclic graphs and homomorphisms with rooted domain graphs helps considerably to keep the conditions simple:

Definition 4.3.7 A **fitting** from a rooted term graph G_1 to a term graph G_2 is a function $F : \mathcal{V}_1 \nrightarrow \mathcal{V}_2$ that (we let $F_0 := \{(n_1, n_2) : F \mid M_1\ x_1 \notin \mathcal{M}\}$ be the *constant part* of F)

i) *preserves labels:* $F_0^{\smile} ; M_1 \sqsubseteq M_2$,

ii) *preserves successors:* $\forall (n1, n2) : F_0 \bullet S_1\ n_1 \sqsubseteq (S_2\ n_2) ; F^{\smile}$,

iii) is *coherent* and *consistent:* for every metavariable in $\operatorname{dom} F$ its image interval is coherent and consistent,

iv) *strictly preserves binding:* $F_0^{\smile} ; B_1 = B_2 ; F_0^{\smile}$,

v) *strictly preserves variables:* $W_1 ; F_0 = F_0 ; W_2$,

vi) *respects equally bound variables:* $F_0 ; \underline{W_2 ; F_0^{\smile} \sqcap B_1} ; B_1^{\smile} \sqsubseteq W_1$,

vii) *respects free variables:* $F_0 ; W_2 ; F_0^{\smile} \sqcap \overline{B_1 ;(\mathcal{V}_1 \times \mathcal{V}_1)} \sqsubseteq W_1$ and

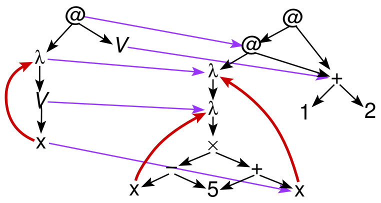
viii) *controls variables:* if for some metavariable node $m : \mathcal{V}_1$ in $\operatorname{dom} F$, a variable node x_2 in the image interval of m is free below $F\ m$, then there is no (variable) node $x_1 : \mathcal{V}_1$ in $\operatorname{dom} F_0$, such that $(F_0\ x_1, x_2) \in W_2$. □

For **linear** term graphs, i.e. where the variable identity restricted to metavariables is contained in the identity relation, this is already the definition of homomorphisms; for non-linear term graphs we have:

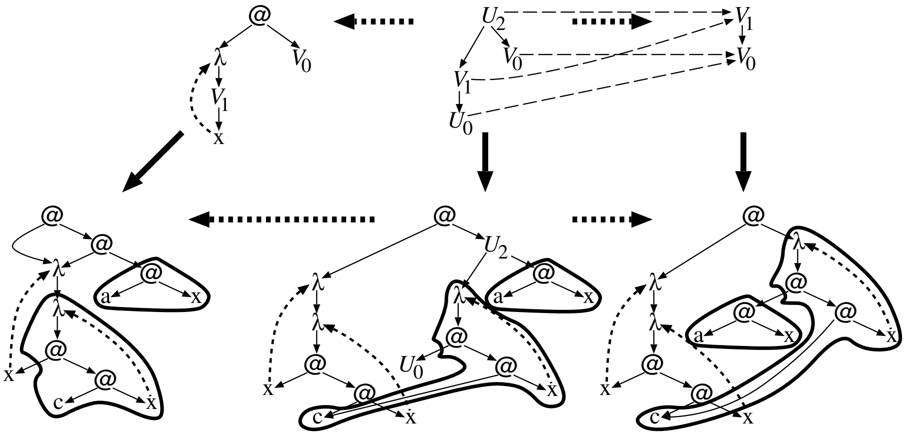
Definition 4.3.8 A **correspondence** between two intervals in a graph is a relation H on the nodeset with exactly the inner nodes of the respective intervals as its domain and range, where H establishes "isomorphism up to sharing" — this needs not be formally defined for the purposes of this paper.

A **homomorphism** is a fitting where for every two related metavariable nodes $(m, n) : W_1$ there is a correspondence between their image intervals. □

Without further restrictions, these homomorphisms are not composable, but in [15] we have shown that this is not necessary for being able to define a sound rewriting concept (see also [16]). As an example homomorphism we show a β-redex, with the thin, dark grey arrows indicating the homomorphism.



These homomorphisms provide us with the matching concept that is appropriate for our second-order term graphs. The term graph rewriting mechanism builds on this homomorphism definition; in [15] it is defined via an extension to the algebraic approach to graph rewriting. A more operational approach to define and understand this rewriting is also possible, but still rather involved, so we omit the details here. We only show an example application of the rule of β-reduction; in the following picture there are six term DAGs: The upper row consists of the left-hand side of the rule, a "gluing graph" and the right-hand side of the rule; the lower row consists of the application graph containing the redex, a "host graph" and the result graph:



The gluing graph and the host graph contain a second kind of metavariables labelled U; these can be replaced via the horizontal "embeddings" in a much less constrained way than the V-variables via the vertical homomorphisms of Def. 4.3.8; the image intervals of these V-variables are indicated in the drawing. The construction of the host graph takes care of eventual duplication of the "non-abstractable parts" [27] of the body of the λ-abstraction.

4.4 Unique Encapsulation

The problem whether there is a matching of a rule's left-hand side into a given application graph, limited to the case where the images of the rule tips are given, is decidable and finitary on finite graph, i.e., it has mostly finitely many solutions.[15] In several contexts, such as automatical reduction, one prefers a

[15] On finite DAGs [11] gives this result.

unitary matching problem, i.e. a matching problem with at most one solution.

Under certain circumstances the matching is determined by the mapping of rule tips into the application graph. An example is the condition imposed on left-hand rule sides in CRSs (see Def. 2.2.8) namely that all successors of every metavariable are different bound variables.

The condition of *unique encapsulation*, that is employed in HOPS, is strictly weaker than that. This condition is aimed at the fact that for uniquely determining the image of a metavariable it suffices (in first approximation) to have a parallel metavariable node encapsulated by some binding node, when the paths from the tip to that binder, from there to the parallel metavariable node in question and from that node to the bound variable are already determined. In earlier versions of HOPS, this still was relatively simple in view of the restriction to at most unary metavariables. In the present implementation of HOPS, as well as in recent extensions to RALF, where metavariables with arbitrarily many successors are supported, we have to formulate a complex recursion over the path structure, and simple encapsulation is not sufficient any more.

The first idea is, that a path from a binder to one of its bound variables (a *binding path*) that passes through only one edge that leaves a metavariable already comes pretty close to fix the image of the end node of that edge.

Second, if of several nodes of one metavariable, one node has one of its leaving edges fixed, then all other nodes of that metavariable also have their corresponding edges fixed because of the correspondences that must exist between the image nodes of all nodes of the metavariable.

Third, an already fixed metavariable edge that lies on a binding path also can help to fix another metavariable on the same path.

The catch mentioned in the first point is that two different successors of a metavariable might be unifiable, or that the terms they represent contain each other. If for every edge-label pair it can be ensured that not for all nodes of a metavariables one of these cases occurs, then the three points from above are sufficient to determine whether all metavariable edges can be fixed.

If this is the case, then we call the graph in question "**uniquely encapsulated**". Under unique encapsulation of a rule's left-hand side, the matching problem becomes unitary.

5 Conclusion

In this paper, we strived to lay a solid foundation for the understanding of general second-order rewriting as used in our two symbolic computation systems RALF and HOPS. Although second-order rewriting in theory is well understood, it is still rarely applied in praxis, and even its most prominent instance — λ-calculus — is even today, sixteen years after the pioneering work of Klop, usually still presented not as a second-order rewriting system, but as a rather special first-order rewriting system with constraints on the occurrences of variables in rule applications. The situation is even more dramatic in logics, where side conditions of the type “where x does not occur free in F” abound. Even the so-called higher-order theorem prover Isabelle does not directly support second-order rules, but uses a mechanism of “provisos” to express “where x does not occur free in F” in every case.

Therefore we propagate an immediate understanding and use of second-order rewriting in interactive symbol manipulation systems as well as in the description of formalisms. The additional level of abstraction can only serve to achieve more compact formulations and more immediate expressions of underlying ideas.

References

1. Arne Bayer and Wolfram Kahl. The **H**igher-**O**bject **P**rogramming **S**ystem “HOPS”. In Buth and Berghammer [5], pages 154–171.
2. Rudolf Berghammer and Claudia Hattensperger. Computer-aided manipulation of relational expressions and formulae using RALF. In Buth and Berghammer [5], pages 62–78.
3. Nicolas Bourbaki. *Éléments de Mathématique, Livre 1: Théorie des Ensembles.* Hermann, 1970.
4. Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors. *Relational Methods in Computer Science.* Advances in Computing. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.
5. Bettina Buth and Rudolf Berghammer, editors. *Systems for Computer-Aided Specification, Development and Verification*, Bericht Nr. 9416. Universität Kiel, 1994.
6. Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, 1941.
7. Andrea Corradini and Frank Drewes. (Cyclic) term graph rewriting is adequate for rational parallel term rewriting. Draft, 1995.
8. Dominic Duggan. Higher-order substitutions. Technical Report CS-93-44, University of Waterloo, Waterloo, Ontario, Canada, October 1993. Revised September 1994.
9. Claudia Hattensperger. *Rechnergestütztes Beweisen in heterogenen Relationenalgebren.* PhD thesis, Fakultät für Informatik, Univ. der Bundeswehr München, 1997. To appear.
10. Claudia Hattensperger, Rudolf Berghammer, and Gunther Schmidt. RALF — a relation-algebraic formula manipulation system and proof checker. notes to a system demonstration. In G. Scollo, editor, *Proc. 3rd Int. Conf. Algebraic Methodology and Software Technology, Enschede, June 21 – June 25*, Berlin-Heidelberg-New York-Tokyo, 1993. Springer. to appear.
11. G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
12. Paola Inverardi and Marisa Venturini Zilli. Rational rewriting. In Igor Prívara, Branislav Rovan, and Peter Ružička, editors, *Mathematical Foundations of Computer Science 1994, 19th International Symposium, MFCS '94, Košice, Slovakia, August 1994, Proceedings*, volume 841 of *LNCS*, pages 433–442. Springer-Verlag, 1994.
13. Wolfram Kahl. Can functional programming be liberated from the applicative style? In Bjørn Pehrson and Imre Simon, editors, *Technology and Foundations, Information Processing '94, Proceedings of the IFIP 13th World Computer Congress,*

Hamburg, Germany, 28 August – 2 September 1994, Volume I, volume A-51 of *IFIP Transactions*, pages 330–335. IFIP, North-Holland, 1994.

14. Wolfram Kahl. Kategorien von Termgraphen mit gebundenen Variablen. Technischer Bericht 9503, Fakultät für Informatik, Universität der Bundeswehr München, September 1995. 191 pages. URL: http://diogenes.informatik.unibw-muenchen.de:8080/kahl/PAPERS/Kahl-1995a_TGVcats.ps.gz.
15. Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag Wissenschaft, München, 1996. ISBN 3-931327-60-4; also doctoral dissertation at Fakultät für Informatik, Universität der Bundeswehr München.
16. Wolfram Kahl. A fibred approach to rewriting — how the duality between adding and deleting cooperates with the difference between matching and rewriting. Technical Report 9702, Fakultät für Informatik, Universität der Bundeswehr München, D-85577 Neubiberg, May 1997.
17. Wolfram Kahl. A framework for user-friendly declarative typing systems. Technical Report 9704, Fakultät für Informatik, Universität der Bundeswehr München, D-85577 Neubiberg, December 1997.
18. Wolfram Kahl. Relational treatment of term graphs with bound variables. *Journal of the IGPL*, 1998. To appear. URL: http://diogenes.informatik.unibw-muenchen.de:8080/kahl/PAPERS/Kahl-1998a_reltgv.ps.gz.
19. J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. *The Adequacy of Term Graph Rewriting for Simulating Term Rewriting*, chapter 12, pages 157–170. John Wiley, 1993.
20. Jan Willem Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhD thesis.
21. Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical computer science*, 121(1/2):279–, December 1993.
22. Dale Miller. A logic programming language with lambda abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
23. Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoretical Computer Science. Springer Verlag, 1993.
24. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.
25. Carolyn L. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112:68–81, 1993. Short version in: Second International Conference on Algebraic Methodology and Software Technology, AMAST91.
26. Antonetta J.M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *Lect. Notes in Comput. Sci.* Springer, 1990.
27. Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. D.Phil. thesis, Oxford University, September 1971.
28. Hans Zierer, Gunther Schmidt, and Rudolf Berghammer. An interactive graphical manipulation system for higher objects based on relational algebra. In Gottfried Tinhofer and Gunther Schmidt, editors, *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 246, pages 68–81, Bernried, Starnberger See, June 1986. Springer-Verlag.