A Verification of Extensible Record Types

Andreas V. Hense

Gert Smolka

Fachbereich 14

Universität des Saarlandes

Technischer Bericht Nr. A 03/92

LIMITED DISTRIBUTION NOTICE
This report has been submitted for publication and will probably be copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the publisher, its distribution prior to publication should be limited to peer communication and specific requests.

A Verification of Extensible Record Types

Andreas V. Hense

Universität des Saarlandes

Im Stadtwald 15

6600 Saarbrücken 11, Germany

hense@cs.uni-sb.de

Gert Smolka

Universität des Saarlandes and

German Research Center for Artificial Intelligence (DFKI)

Stuhlsatzenhausweg 3

6600 Saarbrücken 11, Germany

smolka@dfki.uni-sb.de

May 5, 1992

Abstract

In the strive for more flexible type checkers, Rémy and Wand proposed flexible record types in order to avoid the complications involved with subtyping.

We formalize their approach in the framework of order-sorted logic. **R**, the considered language of records, is so simple that the type inference algorithm can be expressed as a constraint solver in first-order logic. We specify expressions and types, give an effective method for finding a typing, and state soundness results. **R** can be the basis of type inference for object-oriented programming languages.

keywords: records, order-sorted logic, rational tree unification, term rewriting, type inference, recursive types, object-oriented programming

1 Motivation

Languages with records and subtypes are currently receiving much attention [1, 2, 4, 3, 5, 20, 29]. This intensive research activity is motivated by the endeavor to increase the flexibility and reliability of programming languages at the same time. Objects of object-oriented languages can be modeled as records, and subtyping is a notion occurring in this context. Subtyping, in those languages, roughly amounts to saying that an object (= record) having more methods (= components) than necessary can always "do the job". Closely linked to the notion of 'object' is an internal state. It is however problematic to mix a general subtyping notion with imperative features [4]. This may have been the incentive behind Rémy's [24] and Wand's [30, 31] "subtyping without subtypes" by extensible record types. A variable which requires an a-component will have a type that is specific about this a-component, but is extensible in the sense that an object having the required a-component and further components as well also fits the type.

Wand presents a type inference algorithm for a record language and an informal semantics of extensible record types. A correctness proof of this algorithm cannot be performed without a formalization of extensible record types. We formalize these types in first-order logic. As far as we know, this is the first formalization of a type inference system entirely based on first-order logic. We are aware that this is only possible because of the extreme simplicity of the considered language.

The considered language, **R**, is the result of repeated "onion peeling": object-oriented programming languages can be translated into a λ-calculus with records and imperative features [12]. The imperative features can be stripped off. If we also strip off the "λ-part", we get **R**. Summing up, it can be said that **R** is the kernel that allows the description of the object-oriented programming language O'SMALL [13]. The language **R** is extremely simple: only a stepwise construction of records and the selection of components can be expressed. Although **R** will be specified later, we give its abstract syntax here:

$$e \quad ::= \quad x \mid [\,] \mid e \cdot a[e] \mid e.a \ .$$

There are variables, an empty record, extension of a record by one field ("record cons"), and record selection. If a record is extended more than once by the same field, the rightmost extension wins. Variables are introduced for later extensions of **R**. We will see that, in this framework, we are able to infer types for variables. The language contains no recursion; consequently, there are no non-terminating calculations (every term has a normal form). It is more restricted than Wand's language, because it does not contain general record concatenation. The latter is responsible for the loss of the principal type property [15, 31]. The motive for our considerations is O'SMALL. There, method definitions in classes are explained by fieldwise record extension. Classes are no first-class values (i.e. they cannot be the result of functions etc.) and, thus, method names are known beforehand. There are two points where our record language is insecure: (1) The left-hand side of an extension should be a record. (2) A record from which a component is selected should contain that component. To separate secure from insecure expressions, we define a language of types and give an effective procedure to test whether an expression has a typing.

The characteristics of the type system are *flexible record types* and *infinite types*. When typing the expression $(x \cdot b[5]).a$, we get $[a : \alpha, \ldots]$ as a flexible record type for x. This means that x must be a record with an a-component and perhaps further components. The type of the whole expression is α.

In object-oriented languages, recursive types occur even in situations where one would not expect them in "ordinary" languages. The use of the pseudo variable `self` [11, 13] is the reason for this. We need more than finite trees for representing types. If finite trees had been sufficient, we would have taken the initial model and definition 5.3 would have become much simpler. What we need instead, are finitely representable infinite trees. They are called *rational trees* [22, 24] or *regular trees*.

Infinite types are necessary because of the peculiarities of the object-oriented programming style. The example $(x.a)x$ can be read as "send the message a to x with the argument x". This would be the object-oriented way of writing the addition $x + x$. The type inferred for x is $\alpha = [a : \alpha \rightarrow \beta, \ldots]$, an infinite type.

Section 2 advertises the advantages of the order-sorted approach for type inference systems. Sections 4 and 5 specify expressions and types, and state confluence and termination results for the corresponding rewrite systems. Expressions and types are related to each other in section 6, and the notion of a typing is defined. An effective method for finding a typing is presented in section 7. Correctness and completeness of the effective method as well as soundness of the typing rules are proved. Finally, in section 8, a "lazy" rewriting systems for **R** is presented and compared to the "eager" system of section 4.

Warning! The words 'sort' and 'type' are both used and have different meanings. The expressions of the record language are defined in the framework of order-sorted logic and, thus, every expression has a sort. The types of the expressions are defined in the same framework and, thus, every type (term) has a sort. Expressions will be related to types. An expression related to a type is said to have that type. The languages of expressions and of types are not mixed. In other words, there are no type declarations in expressions; the types are inferred.

2 The order-sorted approach

Many-sorted logic is the basis for algebraic specifications [10, 9, 23], rewriting techniques [18, 19], and unification theory [17, 27]. Its results extend to order-sorted logic [28, 26] under certain conditions. In many-sorted logic, the sorts are completely unrelated, while in the order-sorted case, there is a subsort relationship. When it comes to the description of the semantics of programming languages, the subsort relationship is extremely useful: in an untyped language, the denotations of all terms belong to one set. There is no separation into a number of unrelated sets. Consequently, the many-sorted approach is not helpful for structuring this problem. Even in an untyped language, there is usually a distinction between terms denoting errors and terms denoting proper values. It certainly makes sense to see the set of proper values as a subset, or a subsort, of all denotations.

A specification of a language in order-sorted logic can be turned into an executable operational semantics by reading the equations as rewrite rules and showing that the resulting rewrite system is confluent. As stated already, we have two languages: one is the language of expressions, **R**, and the other is the language of types, **RT**. Each language specification can be turned into an executable operational semantics. To have sophisticated "calculations" on types is not common but, here, we have to model the so-called *padding* [31] for extensible record types. When an extensible record type is unified with another record type, the missing components are padded in such that both have the same set of labels.

What we have explained so far could have been done with only a slight loss of structure using many-sorted logic instead of order-sorted logic. The advantage of the latter becomes clearer in the soundness proofs. There, we must show that terms having a type cannot produce an error when they are evaluated. Thanks to a property of the rewrite rules (they are sort-decreasing), the soundness proof reduces to showing that the term itself, or its normal form, is in the subsort of proper values.

Terms for expressions and terms for types τ will be constructed successively. They will be related by the typing relation $: \tau$. Typings are then first-order formulas over the typing relation. The type inference algorithm is formulated as a constraint solver on these formulas.

3 Mathematical preliminaries

Note The hurried reader may skip this section and return to it if he wants to know more details in subsequent sections.

3.1 Order-sorted logic

The definitions follow the notation of [28].

Syntax We use lower case bold roman font words for *sort symbols*, e.g. **p**, **sp**, **f**. *Function symbols* are declared with their arity. If the arity is zero, we call them *constant symbols*. We use x, y, z for variables. Every variable x has a *sort*, $sort(x)$, which is a sort symbol. A *subsort declaration* has the form $\mathbf{r} < \mathbf{s}$, where $\mathbf{r}$ and $\mathbf{s}$ are sort symbols. A *function declaration* has the form

$$f _ \ldots _ : \mathbf{s}_1 \times \ldots \times \mathbf{s}_n \rightarrow \mathbf{s}$$

where n is the arity of f. Function symbols do not have to be written in prefix notation; they may appear in "mixfix".

A *signature* Σ is a set of subsort and function declarations. The *subsort order* $\mathbf{r} \leq_\Sigma \mathbf{s}$ of Σ is the least partial order $\leq_\Sigma$ on the sort symbols of Σ so that $\mathbf{r} \leq_\Sigma \mathbf{s}$ if $\mathbf{r} < \mathbf{s} \in \Sigma$. The subsort order is extended componentwise to strings of sort symbols. If there is no danger of confusion, the index Σ is omitted.

Let Σ be a signature. A Σ-term of sort s is either a variable x so that $sort(x) \leq_\Sigma$ s, or it has the form $f(s_1, \ldots, s_n)$ and there is a declaration

$$(f _ \ldots _ : \mathrm{s}_1 \times \ldots \times \mathrm{s}_n \to \mathrm{r}) \in \Sigma$$

such that $\mathrm{r} \leq_\Sigma \mathrm{s}$ and, for $i \in \{1, \ldots, n\}$, s_i is a Σ-term of sort s_i.

A Σ-equation is an ordered pair of Σ-terms written as $s \doteq t$. A Σ-term is called *ground* if it contains no variables. $\mathcal{V}s$ is used for the set of variables occurring in the term s. A Σ-substitution is a function from Σ-terms to Σ-terms such that

- if s is a Σ-term of sort s, then θs is a Σ-term of sort s,
- $\theta f(s_1, \ldots, s_n) = f(\theta s_1, \ldots, \theta s_n)$,
- $\mathcal{D}\theta := \{x \mid \theta x \neq x\}$ is finite.

Substitution [7] of a variable v by term e in term e' is denoted by $[e/v]e'$.

$$[e_2/v_2]([e_1/v_1]e)$$

is abbreviated to

$$[e_1/v_1, e_2/v_2]e \ .$$

These substitutions avoid name clashes: bound variables (as in λ-terms) are appropriately renamed.

A *specification* $\mathcal{S} = (\Sigma, \mathcal{E})$ consists of a signature Σ and a set $\mathcal{E}$ of Σ-equations, called *axioms* of $\mathcal{S}$. Σ or $\mathcal{E}$ may be countably infinite.

Semantics Let Σ be a signature. A Σ-*algebra* $\mathcal{A}$ consists of *denotations* $\mathrm{s}^\mathcal{A}$ and $f^\mathcal{A}$ for the sort and function symbols of Σ such that:

- $\mathrm{s}^\mathcal{A}$ is a set.
- If $(\mathrm{r} < \mathrm{s}) \in \Sigma$, then $\mathrm{r}^\mathcal{A} \subset \mathrm{s}^\mathcal{A}$.
- $C_\mathcal{A} := \bigcup\{\mathrm{s}^\mathcal{A} \mid \mathrm{s}$ is a sort symbol of $\Sigma\}$ is called the *carrier* of $\mathcal{A}$.
- $f^\mathcal{A}$ is a mapping $D_f^\mathcal{A} \to C_\mathcal{A}$ whose domain $D_f^\mathcal{A}$ is a subset of $C_\mathcal{A}^n$,where n is the arity of f.
- If $(f _ \ldots _ : \mathrm{s}_1 \times \ldots \times \mathrm{s}_n \to \mathrm{s}) \in \Sigma$ and $a_i \in \mathrm{s}_i^\mathcal{A}$ $(i \in \{1, \ldots, n\})$, then $(a_1, \ldots, a_n) \in D_f^\mathcal{A}$ and $f^\mathcal{A}(a_1, \ldots, a_n) \in \mathrm{s}^\mathcal{A}$.

$C^n_{\mathcal{A}}$ denotes the cartesian product $C_{\mathcal{A}} \times \ldots \times C_{\mathcal{A}}$ with n factors. A function symbol has only one denotation, although it may have more than one declaration in the signature.

Let $\mathcal{A}$ be a Σ-algebra and V a set of Σ-variables. A Σ-*assignment* is a mapping $\alpha : V \rightarrow C_{\mathcal{A}}$ such that $\alpha(x) \in (sort(x))^{\mathcal{A}}$ for all variables $x \in V$. Given a Σ-assignment α and a Σ-term s, the *denotation* $[\![s]\!]_\alpha$ of s in $\mathcal{A}$ under α is defined as follows:

$$\begin{aligned} [\![x]\!]_\alpha &= \alpha(x), \\ [\![f(s_1, \ldots, s_n)]\!]_\alpha &= f^{\mathcal{A}}([\![s_1]\!]_\alpha, \ldots, [\![s_n]\!]_\alpha) \ . \end{aligned}$$

Validity of Σ-equations in a Σ-algebra $\mathcal{A}$ is defined as follows:

$$\mathcal{A} \models s \doteq t :\Leftrightarrow \forall \Sigma\text{-assignment } \alpha([\![s]\!]_\alpha = [\![t]\!]_\alpha) \ .$$

If $\mathcal{A} \models s \doteq t$, we say that $s \doteq t$ is *valid* in $\mathcal{A}$ or that $\mathcal{A}$ *satisfies* $s \doteq t$.

Let $\mathcal{S} = (\Sigma, \mathcal{E})$ be a specification and $\mathcal{A}$ a Σ-algebra. $\mathcal{A}$ is a *model* of $\mathcal{S}$ if $\mathcal{A}$ satisfies every equation of $\mathcal{E}$.

Connectives and quantifiers are interpreted as usual [16]: let $\mathcal{T}$ be an interpretation and α a Σ-assignment. $(\phi)^{\mathcal{T}}$ are the solutions of ϕ in $\mathcal{T}$, i.e. the set of all Σ-assignments.

$$\begin{aligned} (\phi \wedge \psi)^{\mathcal{T}} &= (\phi)^{\mathcal{T}} \cap (\psi)^{\mathcal{T}} \\ (s = t)^{\mathcal{T}} &= \{\alpha \mid [\![s]\!]_\alpha = [\![t]\!]_\alpha\} \\ (\exists v(\phi))^{\mathcal{T}} &= \{\alpha \mid \exists d \in \mathcal{T}([d/v]\alpha \in (\phi)^{\mathcal{T}})\} \end{aligned}$$

Definition 3.1 We say that ψ is a *consequence* of ϕ (written $\phi \models \psi$) if and only if every interpretation which is a model of ϕ is also a model of ψ.

We use the symbol $\models$ for both the satisfaction relation ($\mathcal{A} \models \phi$) and for the *consequence relation* ($\phi \models \psi$). The symbol preceding "$\models$" determines the meaning.

Definition 3.2 Let ϕ be a constraint of logical connectives and existential quantifiers. The *prenex normal form* [22] of ϕ is $\exists v_1, \ldots, v_m(\phi')$, where $v_1, \ldots, v_m$ are all existentially quantified variables of ϕ, and ϕ' contains no more existential quantors.

3.2 Order-sorted rewriting

The definitions follow the notation of [28]. The notations and results of term rewriting [18, 19] are generalized to order-sorted logic.

Definition 3.3 An (algebraic) Σ-*rewrite rule* $s \rightarrow t$ is a Σ-equation $s \doteq t$ such that s is not a variable, and every variable occurring in the right-hand side t occurs in the left-hand side s. An *(algebraic) rewriting system* is a specification $\mathcal{R} = (\Sigma, \mathcal{E})$ such that every equation in $\mathcal{E}$ is an (algebraic) rewrite rule.

The notion of algebraic rewriting rules [8] is used to differentiate these rules from the β-reduction rule of the λ-calculus. There is also the notion of *applicative* term rewriting systems [21] which are algebraic rewriting systems containing a special binary operator called *application*. This operator is usually syntactically invisible and left-associative.

The following notation and theorems come from [18]. Let $\rightarrow$ be a binary relation on some set. Then, $\xrightarrow{*}$ denotes the reflexive and transitive closure of $\rightarrow$, and $\xleftrightarrow{*}$ denotes the reflexive, symmetric, and transitive closure of $\rightarrow$. We write $s \downarrow t$ (read "s and t *converge*") if there is an r such that $s \xrightarrow{*} r$ and $t \xrightarrow{*} r$. The relation $\rightarrow$ is called *locally confluent* if $r \rightarrow s$ and $r \rightarrow t$ implies $s \downarrow t$. The relation $\rightarrow$ is called *confluent* if $r \xrightarrow{*} s$ and $r \xrightarrow{*} t$ implies $s \downarrow t$. The relation $\rightarrow$ is called *terminating* if there are no infinite chains $s_1 \rightarrow s_2 \rightarrow \ldots$. An element s is called *normal* if there is no t such that $s \rightarrow t$. An element t is called a *normal form* of s if $s \xrightarrow{*} t$ and t is normal.

Proposition 3.4 Let $\rightarrow$ be a confluent relation. Then no element has more than one normal form. If $\rightarrow$ is confluent and terminating, then every element has exactly one normal form.

Theorem 3.5 Let $\rightarrow$ be a confluent relation. Then $s \xleftrightarrow{*} t$ if and only if $s \downarrow t$.

Theorem 3.6 A relation is confluent if it is locally confluent and terminating.

The following definitions come from [28]. A syntactical Σ-object, O', is a *variant* of a Σ-object O if O' is obtainable from O by consistent variable renaming, i.e. there exist Σ-substitutions θ and ψ such that $O' = \theta O$ and $O = \psi O'$.

Definition 3.7 An *overlap* of a rewrite system $\mathcal{R}$ is a triple $(s \rightarrow t, \pi, s' \rightarrow t')$ such that

- $s \rightarrow t$ and $s' \rightarrow t'$ are variable disjoint variants of rules of $\mathcal{R}$, and π is a position of s such that s/π is not a variable,
- if $s/\pi = s$, then $s \rightarrow t$ is not a variant of $s' \rightarrow t'$,
- there exists an $\mathcal{R}$-substitution θ such that $(\theta s)/\pi = \theta s'$.

An overlap $(s \rightarrow t, \pi, s' \rightarrow t')$ is called a *variant* of an overlap $(u \rightarrow v, \pi, u' \rightarrow v')$ if $u \rightarrow v$ is a variant of $s \rightarrow t$ and $u' \rightarrow v'$ a variant of $s' \rightarrow t'$.

Definition 3.8 A *critical pair* of an overlap $(s \rightarrow t, \pi, s' \rightarrow t')$ of $\mathcal{R}$ is a pair $(\theta t, \theta(s[\pi \leftarrow t']))$ such that $(\theta s)/\pi = \theta s'$, $\theta(s[\pi \leftarrow t'])$ is an $\mathcal{R}$-term, and θ is an $\mathcal{R}$-substitution. A pair (s, t) is called $\mathcal{R}$-critical if (s, t) is a critical pair of an overlap of $\mathcal{R}$. We say that a pair (s, t) *converges* in $\mathcal{R}$ if $s \downarrow_{\mathcal{R}} t$.

Theorem 3.9 Let $\mathcal{R}$ be a sort decreasing rewriting system. Then $\mathcal{R}$ is locally confluent if and only if all critical pairs of $\mathcal{R}$ converge.

3.3 Notation

An *abbreviating notation for records* is used for better readability:

$$(\ldots(([\,]\cdot a_1[s_1])\cdot a_2[s_2])\ldots)\cdot a_n[s_n]$$

is written as

$$[a_1 \mapsto s_1, a_2 \mapsto s_2, \ldots, a_n \mapsto s_n]\ .$$

For record types, we use colons instead of arrows:

$$(\ldots(([\,]\cdot a_1[\tau_1])\cdot a_2[\tau_2])\ldots)\cdot a_n[\tau_n]$$

becomes

$$[a_1 : \tau_1, a_2 : \tau_2, \ldots, a_n : \tau_n]\ .$$

Furthermore, $\rho_{\overline{a,b}} \cdot \rho_a \cdot b[\text{int}]$ is written as $[b : \text{int}, \ldots]$ or $[b : \text{int}, \mathcal{R}]$. The ρ_a does not appear explicitly in this abbreviating notation. The three dots or the row variable [31] $\mathcal{R}$ stand for the extensible part. Three dots are used for an anonymous row variable.

For convenience, we use a *vector notation*. Let $_ * _$ be a binary relation symbol, $v_1, \ldots, v_n$ pairwise disjoint variables, and $t_1, \ldots, t_n$ terms. We abbreviate the conjunction $v_1 * t_1 \wedge \ldots \wedge v_n * t_n$ to $\vec{v} * \vec{t}$.

4 Expressions

We specify a simple record language in the framework of order-sorted logic following the notation of [28]. The syntax differs slightly from the abstract syntax in the introduction because we aim at a close correspondence between expressions and types. The equations can be turned into confluent rewrite rules, thus yielding an operational semantics.

Definition 4.1 (Signature R) We define the signature $\mathbf{R}$ of records. Let $\mathbf{a}$ be the sort of all expressions ranged over by variables x, y, z. Let $\mathbf{p}$ be the sort of proper values ranged over by variables p, q. Let $\mathbf{c}$ be the sort of other values ranged over by variables c, d. Let $\mathbf{r}$ be the sort of record values ranged over by variables r, and $\mathbf{s}$ the sort of single record values ranged over by variables s. The latter are record values with exactly one component. The sorts are ordered as follows (see also figure 1): $\mathbf{s} < \mathbf{r} < \mathbf{p} < \mathbf{a}, \mathbf{c} < \mathbf{p}$. a and b stand for labels from a countable set of labels. Thus, $a[_]$ stands for one instance of a countable number of functions. We assume a total order on the set of labels.

$$[\,] \ :\ \mathbf{r}$$
$$a[_] \ :\ \mathbf{a} \rightarrow \mathbf{a}$$

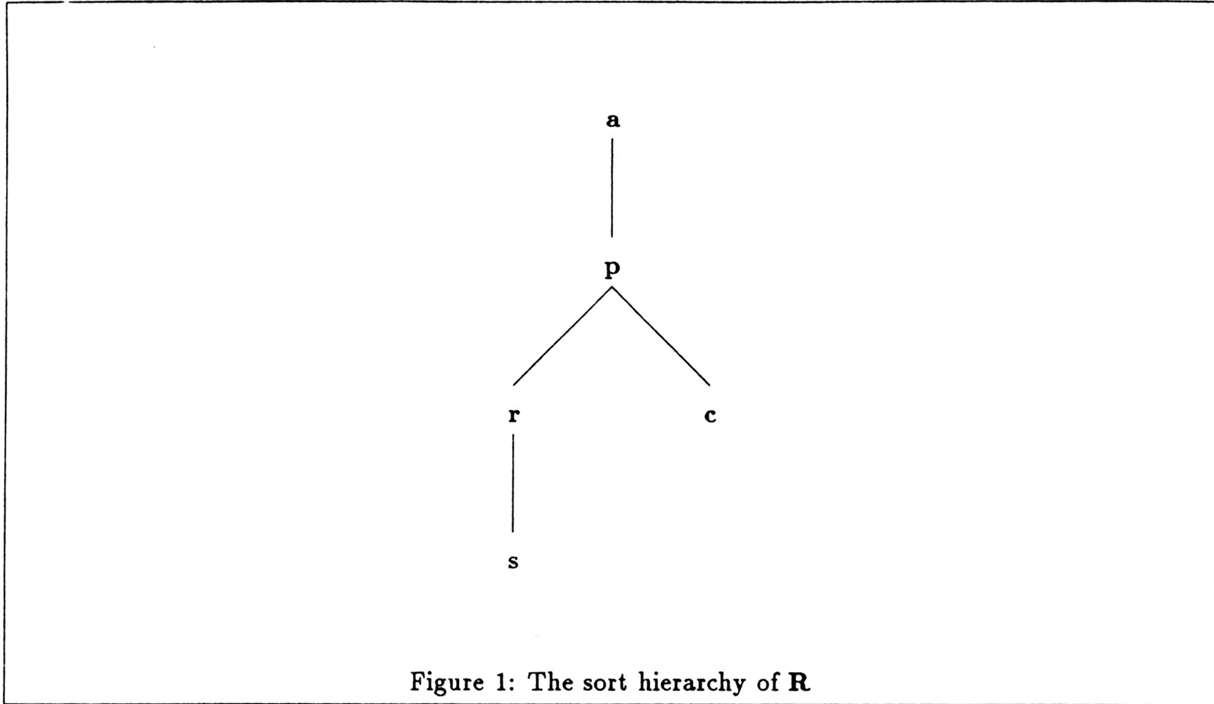

Figure 1: The sort hierarchy of **R**

$$\begin{aligned} a[_] &: \mathbf{p} \rightarrow \mathbf{s} \\ _.a &: \mathbf{a} \rightarrow \mathbf{a} \\ _\cdot_ &: \mathbf{a} \times \mathbf{a} \rightarrow \mathbf{a} \\ _\cdot_ &: \mathbf{r} \times \mathbf{s} \rightarrow \mathbf{r} \end{aligned}$$

$[\,]$ stands for the empty record, $a[_]$ builds a singleton record with the field a, $_.a$ is the selection of label a, and $_\cdot_$ is a "right-preferential record cons", i.e. exactly one component can be added to a record overwriting existing components with the same label.

The semantics of the operations is given by models fulfilling equations. The following equation schemes stand for a countable number of equations (*cf.* remarks on labels above). The conditions in equations (3), (4), and (9) are conditions for the existence of an equation, *not* conditional equations.

Definition 4.2 (R-equations)

variables: $p, q \;:\; \mathbf{p}$

$r, s \;:\; \mathbf{r}$

For every label a

$$a[p] \cdot a[q] \doteq a[q]. \tag{1}$$

For every label a

$$(r \cdot a[p]) \cdot a[q] \doteq r \cdot a[q]. \tag{2}$$

For every label a, for every label b, and $a < b$

$$b[q] \cdot a[p] \doteq a[p] \cdot b[q]. \tag{3}$$

For every label a, for every label b, and $a < b$

$$(r \cdot b[q]) \cdot a[p] \doteq (r \cdot a[p]) \cdot b[q]. \tag{4}$$

$$[\,] \cdot s \doteq s. \tag{5}$$

For every label a

$$a[p].a \doteq p. \tag{6}$$

For every label a

$$(r \cdot a[p]).a \doteq p. \tag{7}$$

For every label a, for every label b, and $a \neq b$

$$a[p].b \doteq [\,].b. \tag{8}$$

For every label a, for every label b, and $a \neq b$

$$(r \cdot a[p]).b \doteq r.b. \tag{9}$$

Equations (1) and (2) represent right-preferential overwriting: if a label occurs more than once in a record, the value of the rightmost occurrence wins. Equations (3) and (4) put the labels into their assumed total order (e.g. lexicographical order for strings). Equation (5) states that the empty record is the left-neutral element of concatenation. Equations (6), (7), (8), and (9) realize selection of a component. Rule (8) is the product of a completion process: it has been added to the other rules to make the system confluent. The equations are formulated in a strict way. We will be using them (oriented to the right) as rewrite rules. In equation (1), p is disregarded in the rewriting process. Nevertheless, the sort of p implies that it is a proper value. The term

$$a[a[[\,]].a] \cdot a[[\,]]$$

cannot be rewritten with rule (1), because $a[[\,]].a$ is not in $\mathbf{p}$. Only after rewriting it to

$$a[[\,]] \cdot a[[\,]],$$

using rule (6), does rule (1) become applicable. The signature, **R**, does not contain other values than records. Only a sort and corresponding variables are provided for other values.

Since we do not intend to restrict ourselves to the initial model, we will define the theory of **R**-terms by stating explicitly which axioms are given.

Definition 4.3 The *theory*, $\mathcal{T}_{\mathbf{R}}$, of **R**-*terms* is given by equations (1) through (9) and the axiom scheme

variables: $x, y \quad : \quad \mathbf{a}$

For every label a, for every label b, and $a \neq b$

$$a[x] \doteq b[y] \quad \rightarrow \quad \bot\,. \tag{10}$$

Theorem 4.4 The rewrite system obtained by orienting the equations of definition 4.2 to the right is sort-decreasing, terminating, and confluent.

Corollary 4.5 (4.4) Every **R**-term has a (unique) normal form in the rewriting system of theorem 4.4.

All functions with a sort less or equal to **p** in the codomain are *constructors*. In definition 4.1, all functions, except the selection, are therefore constructors. If the normal form of a term contains a selection, it represents an error element. Its sort is **a**, but not **p**. Record concatenation is another source of error elements: if the normal form of the right-hand side is not in **s**, the result will not be in **p**.

5 Types

So far, we have defined terms and their operational semantics by an order-sorted specification and the transformation of the equations to a confluent rewrite system. Now, we will define type terms in the same way. In many "ordinary" type systems, two types are different if they are syntactically different; e.g. int $\rightarrow$ (int $\rightarrow$ int) is different from int $\rightarrow$ int. In our type system, syntactically different type terms may denote the same type: e.g. the types $a[\text{int}]\cdot a[\text{int}]$ and $a[\text{int}]$ are the same. This is why, in definition 5.1, there will be equations on type expressions – something which is a little unusual.

Before going into details, let us give an intuitive idea about flexible record types. Consider the function $\lambda x.(x.a)$. We use additional parentheses to emphasize that the first dot is the one of λ-abstraction and the second, the one of record selection. Variables x, y, z are also used for λ-bound variables. A record $[a \mapsto 5]$ could be its argument, and the result of the application $(\lambda x.(x.a))[a \mapsto 5]$ would be 5. To infer the type $[a : \alpha] \rightarrow \alpha$ for this function would be quite inflexible since the function can also have arguments that contain further components; e.g. $(\lambda x.(x.a))[a \mapsto 5, b \mapsto 7] = 5$. We therefore infer a flexible record type which looks like this in the abbreviated notation: $[a : \alpha, \ldots] \rightarrow \alpha$.

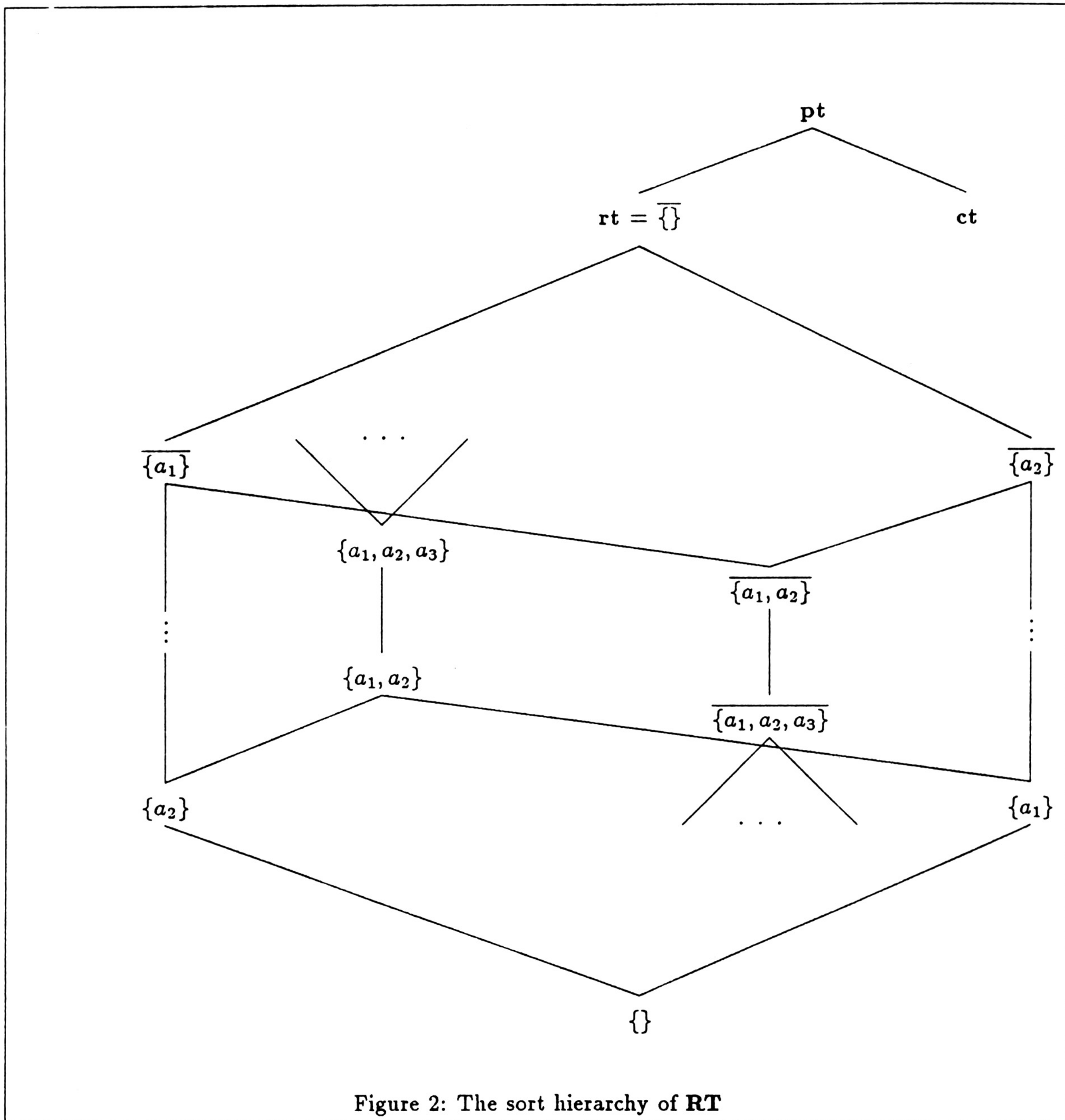

Figure 2: The sort hierarchy of **RT**

The ellipsis stands for a record extension ranging over all labels except a. This idea is formalized in definition 5.1.

The types for the simple record language (definition 4.1) are specified in the same framework as the values, i.e. order-sorted logic. The sort system is more elaborate than the one of expressions. The type language is essentially the language of expressions without selection.

Definition 5.1 We define the signature **RT** of types for records. Let **pt** be the sort of all *types* ranged over by variables α, β. Let **ct** be the sort of types for other values ranged over by variable γ. The set of record sorts is an infinite lattice containing finite and cofinite sorts. Variables ρ, ranging over terms having record sorts, are indexed with finite or cofinite sets. A finite sort is denoted by its set A or – more explicitly – by $\{a_1, \ldots, a_n\}$, where $n \geq 0$ and the a_is are pairwise disjoint labels. Finite sorts are ranged over by variables $\rho_{a_1,\ldots,a_n}$. A cofinite sort is denoted by its set $\overline{A}$ or – more explicitly – by $\overline{\{a_1, \ldots, a_n\}}$, where $n \geq 0$. Cofinite sorts are ranged over by variables $\rho_{\overline{a_1,\ldots,a_n}}$. The sort $\overline{\{\}}$ is the top element of the lattice, and is also called **rt**. For variables ranging over **rt**, the index may be omitted, and we just write ρ. The finite sort A stands for the type of a record with the labels contained in A. The cofinite sort $\overline{A}$ stands for the type of a record extension ranging over all labels, except those contained in A.

$\mathbf{rt}$	$<$	$\mathbf{pt}$		
A	$<$	$\mathbf{rt}$		
$\overline{A}$	$<$	$\mathbf{rt}$		
A	$<$	B	if and only if	$A \subset B$
$\overline{A}$	$<$	$\overline{B}$	if and only if	$A \supset B$
A	$<$	$\overline{B}$	if and only if	$A \cap B = \emptyset$

Figure 2 shows the sort hierarchy of type terms. The lattice part below **rt** has a complex structure which is infinitely branching upwards as well as downwards.

$[\,]$	:	$\{\}$
$a[_]$	:	$\mathbf{pt} \rightarrow \{a\}$
$_ \cdot _$	:	$\mathbf{rt} \times \{a\} \rightarrow \mathbf{rt}$
$_ \cdot _$	:	$A \times \{a\} \rightarrow A \cup \{a\}$
$_ \cdot _$	:	$\overline{A \uplus \{a\}} \times \{a\} \rightarrow \overline{A}$

Definition 5.2 (RT-equations) In the following equation schemes, variables with distinct indices are distinct. In order to understand the rules, the sorts of the variables described above are extremely important.

variables: α : **pt**
ρ : **rt**
ρ_a : {a}
ρ_b : {b}

For every label a

$$\rho_a \cdot a[\alpha] \doteq a[\alpha]. \tag{11}$$

For every label a

$$(\rho \cdot \rho_a) \cdot a[\alpha] \doteq \rho \cdot a[\alpha]. \tag{12}$$

For every label a, for every label b, and $a < b$

$$\rho_b \cdot \rho_a \doteq \rho_a \cdot \rho_b. \tag{13}$$

For every label a, for every label b, and $a < b$

$$(\rho \cdot \rho_b) \cdot \rho_a \doteq (\rho \cdot \rho_a) \cdot \rho_b. \tag{14}$$

$$[\,] \cdot \rho \doteq \rho. \tag{15}$$

$$\rho \cdot [\,] \doteq \rho. \tag{16}$$

Equations (11) and (12) represent right-preferential overwriting. Equations (13) and (14) order the labels. Equation (15) states again that the empty record type is the left-neutral element of concatenation. The only "new" equation, compared to the equations for values, is the one for the right-neutrality (16). We need this equation, because the type of the empty record is in the subsort of the types of records having at most one component. This contrasts to the sort hierarchy for values.

As in the definition of $\mathcal{T}_{\mathbf{R}}$, we want to admit more than the initial model. For our types, this is essential because, in particular, we want to admit rational trees. Thus, the next definition explicitly states the axioms for **RT**-terms.

Definition 5.3 The *theory of* **RT**-*terms*, $\mathcal{T}_{\mathbf{RT}}$, is given by equations (11) through (16) and the following axiom schemes:

variables: α, β : **pt**
ρ, ρ' : **rt**
ρ_a, ρ'_a : $\{a\}$
$\rho_{\overline{a}}$: $\overline{\{a\}}$
$\rho_{\overline{b}}$: $\overline{\{b\}}$
$\rho_{\overline{a,b}}, \rho'_{\overline{a,b}}$: $\overline{\{a, b\}}$

For every label a

$$\rho_{\bar{a}} \cdot \rho_a \doteq \rho'_{\bar{a}} \cdot \rho'_a \quad \rightarrow \quad \rho_{\bar{a}} \doteq \rho'_{\bar{a}} \ \wedge \ \rho_a \doteq \rho'_a. \tag{17}$$

For every label a

$$\rho_{\bar{a}} \doteq \rho \cdot \rho_a \quad \rightarrow \quad \rho_{\bar{a}} \doteq \rho \ \wedge \ \rho_a \doteq [\,]. \tag{18}$$

For every label a

$$[\,] \doteq \rho \cdot a[\alpha] \quad \rightarrow \quad \bot\,. \tag{19}$$

For every label a, for every label b, and $a \neq b$

$$\rho_{\overline{a,b}} \cdot a[\alpha] \doteq \rho'_{\overline{a,b}} \cdot b[\beta] \quad \rightarrow \quad \bot\,. \tag{20}$$

For every label a

$$\rho \cdot a[\alpha] \doteq \rho' \cdot a[\beta] \quad \rightarrow \quad \alpha \doteq \beta. \tag{21}$$

The last axiom scheme states that record types are decomposable:

variables: ρ : **rt**

$\rho_{\overline{a_1,\ldots,a_n}}$: $\overline{\{a_1,\ldots,a_n\}}$

ρ_{a_i} : $\{a_i\}$ $(1 \leq i \leq n)$

For all label sets $A = \{a_1,\ldots,a_n\}$ and all ρ, there exist $\rho_{\overline{a_1,\ldots,a_n}}, \rho_{a_1},\ldots,\rho_{a_n}$ such that

$$\rho \ \doteq \ \rho_{\overline{a_1,\ldots,a_n}} \cdot \rho_{a_1} \cdot \ldots \cdot \rho_{a_n}\,. \tag{22}$$

Theorem 5.4 The rewrite system obtained by orienting the equations of definition 5.2 to the right is sort-decreasing, terminating, and confluent.

Corollary 5.5 (5.4) Every term of definition 5.1 has a (unique) normal form.

6 Typings

So far, we have defined expressions and types by the theories $\mathcal{T}_{\mathbf{R}}$ and $\mathcal{T}_{\mathbf{RT}}$. Expressions and types will be related by the typing relation. Axioms for the typing relation will be stated, and our theory will be the union of all theories presented.

Definition 6.1 The *typing relation* relates terms of sorts **a** and **pt**. It is denoted by a colon. The following axiom schemes are expressed in such a way that they "decompose a term" when applied left to right:

variables: x, y : **a**

α : **pt**

ρ, ρ' : **rt**

For every label a

$$x \cdot a[y] : \rho \quad \leftrightarrow \quad \exists \rho', \alpha(\rho \doteq \rho' \cdot a[\alpha] \ \wedge \ x : \rho' \wedge y : \alpha). \tag{23}$$

For every label a

$$a[x] : \rho \quad \leftrightarrow \quad \exists \alpha(\rho \doteq a[\alpha] \wedge x : \alpha). \tag{24}$$

$$[] : \rho \quad \leftrightarrow \quad \rho \doteq []. \tag{25}$$

For every label a

$$x.a : \alpha \quad \leftrightarrow \quad \exists \rho(x : \rho \cdot a[\alpha]). \tag{26}$$

The next axiom states that types are disjoint:

variables: x : **a**

α, β : **pt**

$$x : \alpha \wedge x : \beta \quad \rightarrow \quad \alpha \doteq \beta. \tag{27}$$

The *theory* $\mathcal{T}_{\mathrm{tr}}$ *of the typing relation* is given by axioms schemes (23) through (27).

In the rule for concatenation (23), we de not know whether the type of e contains a component with label a. However, the result is guaranteed to have such a component, and any previous component with label a is overwritten. In the rule for selection (26), the selected component must be in the record. Note that we have defined the typing relation for a restricted language. The second component of concatenation (23) has an explicit label. In the introduction, it was mentioned that the motive for this investigation is the object-oriented language O'SMALL. O'SMALL does not have classes as first-class values and, therefore, this restriction can be imposed. With this restriction, we have principal types, whereas Wand [30] does not have them.

The following theory will be used in the sequel.

Definition 6.2 We define the *theory* $\mathcal{T}$ as the union of the theories above,

$$\mathcal{T} \quad := \quad \mathcal{T}_{\mathbf{R}} \cup \mathcal{T}_{\mathbf{RT}} \cup \mathcal{T}_{\mathrm{tr}} \,.$$

With this theory, we are able to define semantically what we mean by "a term has a type". The corresponding formal concept of this notion is the typing.

Definition 6.3 A *type environment* Γ has the form

$$\Gamma = \exists \alpha_1, \ldots, \alpha_n(\vec{x} : \vec{\sigma} \wedge \vec{\beta} \doteq \vec{\tau}),$$

such that exactly one type variable α and the value variables $\vec{x}$ are free.

A type environment contains types for term variables and a substitution $\vec{\beta} \doteq \vec{\tau}$. The substitution is used to represent regular trees. The only free type variable represents the type of the considered term. This will become clear in an example.

In the sequel, we will drop the index $\mathcal{T}$ and write $\models$ instead of $\models_{\mathcal{T}}$ since there is no danger of confusion.

In the following definition, we will see when a type environment can be called a typing for a term.

Definition 6.4 We say that a term e has a *typing* Γ if there is a type environment Γ, such that

$$\Gamma \models e : \alpha,$$

where α is the only free type variable of Γ.

Definition 6.5 Let Γ and Γ' be two typings. Γ is *more general than* Γ' if and only if $\Gamma' \models \Gamma$.

The *"more-general-than"* relation is a preorder.

Definition 6.6 A typing Γ is called *prinicipal* for an expression e if and only if $\Gamma' \models \Gamma$ for every typing Γ' of e.

Proposition 6.7 Rewriting preserves types. Let $\Gamma := \vec{x} : \vec{\sigma} \wedge \vec{\beta} \doteq \vec{\tau}$ and $\Gamma' := \vec{x} : \vec{\sigma'} \wedge \vec{\beta} \doteq \vec{\tau'}$, and let $\Gamma \xrightarrow{*} \Gamma'$ mean that $\vec{\sigma} \xrightarrow{*} \vec{\sigma'}$ and $\vec{\tau} \xrightarrow{*} \vec{\tau'}$. Then the following implication holds.

$$\left.\begin{array}{rcl} \Gamma & \models & e : \tau \\ \Gamma & \xrightarrow{*} & \Gamma' \\ e & \xrightarrow{*} & e' \\ \tau & \xrightarrow{*} & \tau' \end{array}\right\} \Gamma' \models e' : \tau' \; .$$

Lemma 6.8 If a term has a typing, then all its subterms have typings.

Theorem 6.9 Well typed terms do not go wrong: for all ground e, if e has a typing, then the normal form of e is of sort **p**.

Theorem 6.10 Every ground term of sort **p** has a typing.

Corollary 6.11 The normal form of a ground term in the restricted language is of sort **p** if and only if it has a typing.

7 Finding a typing

After having defined the typing relation, we will now give an effective method for deciding whether a term has a typing.

Definition 7.1 We define $\mathcal{W}_r$, an algorithm for finding a typing for a term. The input is $e : \alpha$, a term and a type variable. The output is either a principal typing or failure.

Step 1 Use axioms (23) through (26) to decompose the expression e recursively until only variables occur on the left-hand side of the colon.

Step 2 Apply the rule

variables: x : **a**

$$\frac{x : \sigma \wedge x : \tau}{x : \sigma \wedge \sigma \doteq \tau}$$

until there is at most one membership for every variable.

Step 3 Let $\{a_1, \ldots, a_n\}$ be the set of labels occurring in e. For every variable ρ in the constraints resulting from steps 1 and 2 add

variables: $\rho_{\overline{a_1,\ldots,a_n}}$: $\overline{\{a_1,\ldots,a_n\}}$
ρ_{a_i} : $\{a_i\}$ $(1 \leq i \leq n)$
ρ : **rt**

$$\exists \rho_{\overline{a_1,\ldots,a_n}}, \rho_{a_1}, \ldots, \rho_{a_n}(\rho \doteq \rho_{\overline{a_1,\ldots,a_n}} \cdot \rho_{a_1} \cdot \ldots \cdot \rho_{a_n})$$

and replace ρ in all the other equations by the right-hand side of the equation above.

Step 4 Transform to prenex normal form. We then get:

$$\exists \alpha_1, \ldots, \alpha_m(x_1 : \sigma_1 \wedge \ldots \wedge x_n : \sigma_n \wedge P \wedge E)$$

where P contains the conjunction of the new equations from step 3, and E does not contain any of the old ρ's.

Step 5 Transform E using the following rules. The rules are a specialization of rational tree unification [6] to record type terms. Note that variables for type terms and variables in type terms are sometimes indexed by a sort, and that variables having different sort indices are different.

variables: α, β : **pt**
ρ_b : $\{b\}$

$$\frac{\phi \wedge \sigma \doteq \sigma}{\phi} \tag{28}$$

If $\alpha \neq \beta$ and $\alpha \in \mathcal{V}\phi$ then

$$\frac{\phi \wedge \alpha \doteq \beta}{([\beta/\alpha]\phi) \wedge \alpha \doteq \beta} \tag{29}$$

If $|\sigma| \geq 2$ then

$$\frac{\phi \wedge \sigma \doteq \alpha}{\phi \wedge \alpha \doteq \sigma} \tag{30}$$

If $1 < |\sigma| \leq |\tau|$ then

$$\frac{\phi \wedge \alpha \doteq \sigma \wedge \alpha \doteq \tau}{\phi \wedge \alpha \doteq \sigma \wedge \sigma \doteq \tau} \tag{31}$$

For every label a

$$\frac{\phi \wedge \sigma_{\overline{a}} \cdot \sigma_a \doteq \tau_{\overline{a}} \cdot \tau_a}{\phi \wedge \sigma_{\overline{a}} \doteq \tau_{\overline{a}} \wedge \sigma_a \doteq \tau_a} \tag{32}$$

For every label a, for every label b, and $a < b$

$$\frac{\phi \wedge \sigma_{\overline{b}} \cdot \sigma_a \doteq \tau \cdot \rho_b}{[[\,]/\rho_b](\phi \wedge \sigma_{\overline{b}} \cdot \sigma_a \doteq \tau) \wedge \rho_b \doteq [\,]} \tag{33}$$

For every label a, for every label b, and $a < b$

$$\frac{\phi \wedge \sigma \cdot \rho_b \doteq \tau_{\overline{b}} \cdot \sigma_a}{[[\,]/\rho_b](\phi \wedge \sigma \doteq \tau_{\overline{b}} \cdot \sigma_a) \wedge \rho_b \doteq [\,]} \tag{34}$$

For every label a

$$\frac{\phi \wedge a[\sigma] \doteq a[\tau]}{\phi \wedge \sigma \doteq \tau} \tag{35}$$

For every label a, for every label b, and $a \neq b$

$$\frac{\phi \wedge \rho_{\overline{a,b}} \cdot a[\alpha] \doteq \rho'_{\overline{a,b}} \cdot b[\beta]}{\bot} \tag{36}$$

For every label a

$$\frac{\phi \wedge [\,] \doteq \rho \cdot a[\alpha]}{\bot} \tag{37}$$

Step 6 Garbage collection. Eliminate all existentially quantified variables that have been successfully "eliminated" by rule (29) in step 5.

It is assumed that after each application of a rule in step 5, the type terms are transformed into normal form by the rewriting rules obtained from definition 5.2. This is important for justifying

rules (33) and (34). After step 1, the constraint contains exactly one variable that is not quantified. This variable stands for the type of the whole term e. After step 2, occurrences of the same variable must have the same type. Step 3 does all the padding at once. An intuitive reason, for why padding is possible, is that the ρ_{a_i} can all be instantiated to the empty record type. Given a term together with all its constraints, it makes all labels occurring in the term explicit in the record types of the constraints. Thus, we can do without padding rules in step 5. The proof of theorem 7.2 is much simpler when the padding rules are not mixed with the unification rules. Step 4 moves all quantifiers to the outside. This step prepares for step 5, where we can deal with constraints that do not contain quantifiers.

For a discussion of rules (28), (29), (30), and (31), refer to [6]. The canonically defined term size is written with vertical bars in rules (30) and (31). Rules (32) through (35) decompose record types. Rule (32) analyses a record type with equal components. We always consider record types in normal form with respect to the rules of definition 5.2 and thus, in rule (33), we know that a b-component cannot be contained in the type of the left-hand side. It must therefore be empty on the right-hand side too. Similarly for rule (34). Rule (35) analyzes single fields. Rules (36) and (37) represent failure.

Before we state some properties of the algorithm $\mathcal{W}_r$, we will give two examples of its working. The first example shows flexible record types. We assume that we have integers as other values. The input to the algorithm is

$$(x \cdot b[5]).a : \alpha.$$

step 1 $(x \cdot b[5]).a : \alpha.$

Transform with rule (26) yielding

$$\exists \rho (x \cdot b[5] : \rho \cdot a[\alpha]).$$

Transform with rule (23) yielding

$$\exists \rho, \rho', \beta(\rho \cdot a[\alpha] \doteq \rho' \cdot b[\beta] \wedge x : \rho' \wedge 5 : \beta).$$

Transform with an additional rule for integers yielding

$$\exists \rho, \rho', \beta(\rho \cdot a[\alpha] \doteq \rho' \cdot b[\beta] \wedge x : \rho' \wedge \beta \doteq \mathbf{int}).$$

step 2 Nothing has to be done.

step 3

$$\begin{aligned} &\exists \rho_{\overline{a,b}}, \rho_a, \rho_b, \rho'_{\overline{a,b}}, \rho'_a, \rho'_b, \rho, \rho', \beta(\\ &\quad \rho \doteq \rho_{\overline{a,b}} \cdot \rho_a \cdot \rho_b \wedge \rho' \doteq \rho'_{\overline{a,b}} \cdot \rho'_a \cdot \rho'_b \wedge \\ &\quad \rho_{\overline{a,b}} \cdot \rho_a \cdot \rho_b \cdot a[\alpha] \doteq \rho'_{\overline{a,b}} \cdot \rho'_a \cdot \rho'_b \cdot b[\beta] \wedge \\ &\quad x : \rho'_{\overline{a,b}} \cdot \rho'_a \cdot \rho'_b \wedge \beta \doteq \mathbf{int}). \end{aligned}$$

step 4 Has already been done on the way.

step 5 $\rho_{\overline{a,b}} \cdot a[\alpha] \cdot \rho_b \doteq \rho'_{\overline{a,b}} \cdot \rho'_a \cdot b[\beta] \wedge x : \rho'_{\overline{a,b}} \cdot \rho'_a \cdot \rho'_b \wedge \beta \doteq \mathbf{int}.$

Apply rule (29) and rule (32) twice yielding

$$\rho_{\overline{a,b}} \doteq \rho'_{\overline{a,b}} \wedge a[\alpha] \doteq \rho'_a \wedge \rho_b \doteq b[\mathbf{int}] \wedge x : \rho'_{\overline{a,b}} \cdot \rho'_a \cdot \rho'_b \wedge \beta \doteq \mathbf{int}.$$

Apply rule (30) and rule (29) yielding

$$x : \rho'_{\overline{a,b}} \cdot a[\alpha] \cdot \rho'_b \wedge \beta \doteq \mathbf{int} \wedge \rho_{\overline{a,b}} \doteq \rho'_{\overline{a,b}} \wedge \rho'_a \doteq a[\alpha] \wedge \rho_b \doteq b[\mathbf{int}].$$

step 6 Eliminating all existentially quantified variables that have been "eliminated" yields

$$x : \rho'_{\overline{a,b}} \cdot a[\alpha] \cdot \rho'_b.$$

We have found a typing. The type of the whole expression is α, and the type of x is $[a : \alpha, \ldots]$. The a-component of x must have type α, and x may have more components.

The second example will demonstrate what purpose the substitution in the type environment serves. To obtain a type that is not representable by a finite tree, we have to temporarily assume that the signature **R** is extended by some equality operator

$$. = . : \mathbf{a} \times \mathbf{a} \to \mathbf{a},$$

that **RT** is extended by a new sort **bool**, and that the typing relation is extended by the axiom scheme

$$x = y : \alpha \quad \leftrightarrow \quad \exists \rho (x : \rho \wedge y : \rho \wedge \alpha \doteq \mathbf{bool}) \ . \tag{38}$$

We will now show how the algorithm $\mathcal{W}_r$ finds a principal typing for the expression $x = a[x]$. The input to the algorithm is

$$x = a[x] : \alpha.$$

step 1 $x = a[x] : \alpha.$

Transform with rule (38) yielding

$$\exists \rho (x : \rho \wedge a[x] : \rho \wedge \alpha \doteq \mathbf{bool}).$$

Transform with rule (24) yielding

$$\exists \rho, \beta (x : \rho \wedge \rho \doteq a[\beta] \wedge x : \beta \wedge \alpha \doteq \mathbf{bool}).$$

step 2 $\exists \rho, \beta (x : \rho \wedge \rho \doteq a[\beta] \wedge \beta \doteq \rho \wedge \alpha \doteq \mathbf{bool}).$

step 3 $\exists \rho_{\overline{a}}, \rho_a, \rho, \beta (\rho \doteq \rho_{\overline{a}} \cdot \rho_a \wedge x : \rho_{\overline{a}} \cdot \rho_a \wedge \rho_{\overline{a}} \cdot \rho_a \doteq a[\beta] \wedge \beta \doteq \rho_{\overline{a}} \cdot \rho_a \wedge \alpha \doteq \mathbf{bool}).$

step 4 Has already been done on the way.

step 5 $x : \rho_{\overline{a}} \cdot \rho_a \wedge \rho_{\overline{a}} \cdot \rho_a \doteq a[\beta] \wedge \beta \doteq \rho_{\overline{a}} \cdot \rho_a \wedge \alpha \doteq \mathbf{bool}.$

Apply rule (29) yielding

$$x : \rho_{\overline{a}} \cdot \rho_a \wedge \rho_{\overline{a}} \cdot \rho_a \doteq a[\rho_{\overline{a}} \cdot \rho_a] \wedge \beta \doteq \rho_{\overline{a}} \cdot \rho_a \wedge \alpha \doteq \mathbf{bool}.$$

Apply rule (32) yielding

$$x : \rho_{\overline{a}} \cdot \rho_a \wedge \rho_{\overline{a}} \doteq [\,] \wedge \rho_a \doteq a[\rho_{\overline{a}} \cdot \rho_a] \wedge \beta \doteq \rho_{\overline{a}} \cdot \rho_a \wedge \alpha \doteq \mathbf{bool}.$$

Apply rule (29) yielding

$$x : \rho_a \wedge \rho_{\overline{a}} \doteq [\,] \wedge \rho_a \doteq a[\rho_a] \wedge \beta \doteq \rho_a \wedge \alpha \doteq \mathbf{bool}.$$

step 6 Eliminating all existentially quantified variables that have been "eliminated" yields

$$x : \rho_a \wedge \rho_a \doteq a[\rho_a] \wedge \alpha \doteq \mathbf{bool}.$$

We have found a typing. The type of the whole expression is **bool**, and x has the infinite type $a[a[a[a[a[\ldots]]]]]]$.

Theorem 7.2 The algorithm $\mathcal{W}_r$ terminates.

Step 1 eliminates all terms that are not variables. It uses the axioms of the typing relation.

Proposition 7.3 Steps 2 through 5 leave the set of solutions invariant.

Theorem 7.4 The algorithm $\mathcal{W}_r$ finds a typing if it exists.

Let us discuss the consequences of theorems 7.2 and 7.4. When the algorithm terminates, there is no more applicable rule in step 5. Then, the constraints either have a resolved form, which we have called typing, or failure is the result. In a typing, we have a type for every variable of the term, a type for the term itself, and a substitution. The substitution finitely represents rational trees (recursive types).

Corollary 7.5 (7.3) The typing Γ, found by the algorithm $\mathcal{W}_r$, is a principal typing.

8 A lazy rewrite system

We have seen that the equations of definition 4.2 yield strict rewrite rules when they are oriented to the right. This is a result of the sorts of variables used in the equations. The idea of well-typedness is strict in the literature. We have defined well-typedness semantically and, thus, the semantics of values had to be in accordance with it. That is, we had no choice for the equations: they had to be strict.

In this section, we present lazy rewrite rules. They are much more permissive with respect to the order of rewriting steps. In many cases, they allow for more efficient evaluation since not all of the subterms have to be evaluated. We will show that, for well-typed terms, the strict and the lazy rules "do the same thing".

Definition 8.1 We define a lazy rewrite system for **R**-terms by the following rules:

variables: $x, y, z \quad : \quad \mathbf{a}$

For every label a

$$a[x] \cdot a[y] \quad \dot{=} \quad a[y]. \tag{39}$$

For every label a

$$(x \cdot a[y]) \cdot a[z] \quad \dot{=} \quad x \cdot a[z]. \tag{40}$$

For every label a, for every label b, and $a < b$

$$b[y] \cdot a[x] \quad \dot{=} \quad a[x] \cdot b[y]. \tag{41}$$

For every label a, for every label b, and $a < b$

$$(z \cdot b[y]) \cdot a[x] \quad \dot{=} \quad (z \cdot a[x]) \cdot b[y]. \tag{42}$$

$$[\,] \cdot x \quad \dot{=} \quad x. \tag{43}$$

For every label a

$$a[x].a \quad \dot{=} \quad x. \tag{44}$$

For every label a

$$(x \cdot a[y]).a \quad \dot{=} \quad y. \tag{45}$$

For every label a, for every label b, and $a \neq b$

$$a[x].b \doteq [\,].b. \tag{46}$$

For every label a, for every label b, $a \neq b$

$$(x \cdot a[y]).b \doteq x.b. \tag{47}$$

Theorem 8.2 The rewrite system of definition 8.1 is sort-decreasing, terminating, and confluent.

Corollary 8.3 (8.2) Every **R**-term has a (unique) normal form in the rewriting system of definition 8.1.

The following proposition about the preservation of types during the rewrite process is not as strong as theorem 6.7, and we have to show more things since we are not replacing equals by equals anymore.

Proposition 8.4 Value rewriting preserves types. Let e be an **R**-term with typing $\Gamma \models e : \tau$ and $e \rightarrow_l e'$, where $\rightarrow_l$ denotes the rewrite relation of definition 8.1. Then $\Gamma \models e' : \tau$.

In the following two examples, the set of solutions increases as the term is rewritten.

- Here, we have a ground term that becomes typable:

$$\begin{array}{ccc} ((b[1] \cdot a[c[2].d]).b : \alpha)^{\mathcal{T}} & \overset{(3),(7)}{\longrightarrow} & (1 : \alpha)^{\mathcal{T}} \\ \| & & \| \\ \emptyset & \Rightarrow & \{\text{int}\} \end{array}$$

- The second one is a non-ground term and the set of solutions increases.

$$\begin{array}{ccc} ((a[x+1] \cdot a[x].b : \alpha)^{\mathcal{T}} & \overset{(1)}{\rightarrow} & (a[x] : \alpha)^{\mathcal{T}} \\ \| & & \| \\ \{a[\text{int}]\} & \Rightarrow & \{a[\alpha] \mid \alpha \text{ type}\} \end{array}$$

Finally, the next theorem says that the strict rewrite rules ($\rightarrow_s$) resulting from definition 4.2 and the lazy rewrite rules ($\rightarrow_l$) from definition 8.1 have the same behavior on well-typed terms.

Theorem 8.5 Let e be an **R**-term with $\Gamma \models e : \tau$, and n, an irreducible **R**-term of sort **p**. Then

$$e \overset{*}{\rightarrow}_s n \quad \text{if and only if} \quad e \overset{*}{\rightarrow}_l n\,.$$

9 Conclusion

We have formalized type soundness using order-sorted rewriting systems. This results in simple but sometimes lengthy proofs. The proofs had to be omitted for the sake of brevity. They will be contained in the first author's forthcoming Ph.D. thesis [14]. Our language has fieldwise record extension, but no general concatenation. In this respect it resembles the language of Rémy. Whereas Rémy [25] uses *record algebras* (*algébres touffues*) that deal with infinite tuples, we use algebras with infinitely many constructors, and the tuples are finite. We describe the padding algorithm of Wand [31] dealing with infinite label sets. The padding in our description of the algorithm is done all at once in step 3.

Acknowledgements We thank Christian Fecht and Reinhold Heckmann for careful draft reading. Harald Ganzinger suggested lexicographic and multiset orders for termination proofs.

References

[1] R. Amadio and L.Cardelli. Subtyping recursive types. Technical Report 62, Systems Research Center, Palo Alto, Cal., Aug. 1990.

[2] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Logic in Computer Science*, pages 112–128, 1989.

[3] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming and Computer Architecture*, pages 273–280. ACM, 1989.

[4] L. Cardelli. Structural subtyping and the notion of power type. In *Symposium on Principles of Programming Languages*, pages 70–79. ACM, Jan. 1988.

[5] L. Cardelli and J. C. Mitchell. Operations on records. *Lecture Notes in Computer Science*, 389:75–81, 1989. extended abstract.

[6] A. Colmerauer. *Logic Programming*, chapter Prolog and Infinite Trees, pages 231–251. Academic Press, 1982.

[7] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[8] D. J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. In R. V. Book, editor, *Rewriting Techniques and Applications, 4th RTA-91, LNCS 488*, pages 37–48. Springer, 1991.

[9] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 1: Equations and Initial Semantics of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[10] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Data Structuring*, volume IV of *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[11] A. Goldberg and D. Robson. *Smalltalk-80: the Language*. Addison-Wesley, 1989.

[12] A. V. Hense. Type inference for O'small. Technical Report A 06/91, Universität des Saarlandes, Fachbereich 14, Oct. 1991.

[13] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, Sept. 1991.

[14] A. V. Hense. *Polymorphic type inference for object-oriented programming languages*. PhD thesis, Universität des Saarlandes, Fachbereich 14, D-6600 Saarbrücken, 1992. forthcoming.

[15] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

[16] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG-Report 53, IBM Deutschland, Oct. 1988. to appear in Journal of Logic Programming.

[17] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2,...,ω*. PhD thesis, Université Paris 7, 1976. Thèse de doctorat d'état.

[18] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[19] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[20] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Symposium on Lisp and Functional Programming*, 1988.

[21] J. W. Klop. Term rewriting systems: a tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 32, 1987.

[22] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Logic in Computer Science*, Edinburgh, 1988.

[23] J. Meseguer and J. Goguen. *Algebraic Methods in Semantics*, chapter Initiality, Induction, and Computability. Cambridge University Press, 1985.

[24] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages*, pages 77–88. ACM, 1989.

[25] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Languages Fonctionnels*. PhD thesis, Université Paris 7, Feb. 1990.

[26] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* PhD thesis, Universität Kaiserslautern, Germany, Apr. 1988.

[27] J. Siekmann. Unification theory. In *European Conference on Artificial Intelligence*, pages vi–xxxv, Brighton Centre, England, 1986.

[28] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. *Order-Sorted Equational Computation*, volume 2 of *Resolution of Equations in Algebraic Structures*, chapter 10, pages 297–367. Academic Press, 1989.

[29] R. Stansifer. Type inference with subtypes. In *Symposium on Principles of Programming Languages*, pages 88–97. ACM, Jan. 1988.

[30] M. Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science*, pages 92–97, 1989.

[31] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.