Types for REWERSE reasoning and query languages I3-D4

Horatiu Cirstea, Emmanuel Coquery, Wlodzimierz Drabent, Francois Fages, Claude Kirchner, Luigi Liquori, Benjamin Wack, Artur Wilk

I3-D4

Types for REWERSE reasoning and query languages

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Warsaw,Linköping/I3-D4/D/PU/a1
Responsible editors:	W. Drabent and A.Wilk
Reviewers:	Pierre Deransart, Hans Jürgen Ohlbach
Contributing participants:	Linköping, Nancy, Paris, Warsaw
Contributing workpackages:	I3
Contractual date of deliverable:	28 February 2005
Actual submission date:	04 March 2005

Abstract

This report presents proposals for a type system for a subset of REWERSE languages. We study two approaches to such a type system, which are based on descriptive and prescriptive typing. As an example rule language we use XML query language Xcerpt.

Keyword List
constraints, rewrite calculus, rule based language, type system, type inference, polymorphic typing, descriptive typing, prescriptive typing

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Types for REWERSE reasoning and query languages

Horatiu Cirstea[1] and Emmanuel Coquery[2] and Włodzimierz Drabent[3] and François Fages[2] and Claude Kirchner[1] and Luigi Liquori[5] and Benjamin Wack[1] and Artur Wilk[4]

[1] LORIA: CNRS, INRIA, Universities of Nancy
615, rue du Jardin botanique,
54600 Villers-lès-Nancy Cedex, France

[2] INRIA
France

[3] Institute of Computer Science, Polish Academy of Sciences,
Warsaw, Poland

[4] Department of Computer and Information Science,
Linköping University
S-581 83 Linköping, Sweden

[5] INRIA Sophia Antipolis
2004, route des Lucioles - BP 93
06902 Sophia Antipolis, France

04 March 2005

Abstract

This report presents proposals for a type system for a subset of REWERSE languages. We study two approaches to such a type system, which are based on descriptive and prescriptive typing. As an example rule language we use XML query language Xcerpt.

Keyword List
constraints, rewrite calculus, rule based language, type system, type inference, polymorphic typing, descriptive typing, prescriptive typing

Contents

1 Introduction

This report presents proposals for type systems for subsets of REWERSE languages. It is structured, following the approach already used in [CCD+04], by first considering descriptive types and then prescriptive ones.

Section 2 presents a descriptive typing approach to XML query language Xcerpt. The role of descriptive typing is to provide approximations of the semantics of programs. We propose a formalism to define sets of data terms which is a generalization of tree automata. The defined sets roughly correspond to sets of documents definable by means of XML schema languages, like DTD and XML Schema. The main contribution is an algorithm for computing the type of possible results of an Xcerpt rule, given the type of the database. Two versions of the algorithm are presented, one is simpler and more efficient while the other provides more accurate results. The algorithm can be used to automatically check correctness of Xcerpt programs with respect to specification given by means of types, and to compute (approximations of) the sets of results of non-recursive Xcerpt programs.

Prescriptive typing addresses the issue of the composition. By providing types for function and predicate symbols, and in turn modules, one expresses the syntactic categories to which they can be applied. By checking the consistency of a program w.r.t. these types, a prescriptive type system therefore provide a discipline to compose them correctly. As the Rewriting calculus provides a foundational framework to express in detail the operational semantics of rewrite based languages, we show in Section 3, how the calculus could be used to express Xcerpt constructions. Then, we introduce a polymorphic type system for the Rewriting calculus. This provides the capability to propose polymorphic type system for Xcerpt, via the expression of Xcerpt programs as Rewriting calculus terms. As several rule languages for manipulating semi-structured data have their semantics inherited from constraint logic programming, our prescriptive type system TCLP for constraint logic languages provides a good basis for a type system for these rule languages. Section 4 explains how the ideas of prescriptive typing for constraint logic languages can be adapted for REWERSE rule languages such as the XML query language Xcerpt. First, we present TCLP and recall some of its properties. Then we explain how it can be adapted to rule languages for querying and transforming semi-structured data. This is illustrated through a prescriptive type system for XML query language Xcerpt. We also discuss some type checking issues for this type system.

2 Descriptive typing for Xcerpt

2.1 Introduction

In this section we present a descriptive typing approach to XML query language Xcerpt [BS02b, BS02c, BS02a, BBSW03]. As explained in the previous paper [CCD+04] types in descriptive typing are approximations of program semantics. The considered programming language is untyped and typing does not influence its semantics. In this way one can combine advantages of typed and untyped programming languages. Descriptive types can be used as specifications for programs and for XML databases. In the context of descriptive typing, type checking means (automatic) verification whether a program is correct w.r.t. its specification; type inference means computing (approximation of) the semantics of a given program. To make type checking and inference possible a restriction to some class of recursive sets is necessary, together with a fixed formalism of defining sets.

An objective of this work is to develop analysis techniques for rules used in web applications. A main intended application is locating errors in (programs consisting of) rules. The rules we deal with can be seen as transformers of sets of XML documents. Query language Xcerpt has been chosen as a representative example of a rule language.

XML (eXtensible Markup Language) has become a dominant standard for data encoding and exchange on the Internet. It has been designed to create more structured and adaptable documents and document systems. Sets of documents, often called types, can be specified using various schema languages, like DTD [Ext], XML Schema [Fe01], or RELAX NG [CM01]. Applications which deal with many different DTD's or XML Schemas require mechanisms for comparing such specifications; in other words to compare types. This includes comparing types given by different schema languages. For this purpose a common view of them is necessary.

As XML data are essentially tree structured, a natural approach is to view XML documents as trees (or, equivalently, terms), and types as sets of trees. So we need a formalism to describe decidable sets of trees. It should be able to describe sets corresponding to those specified by major schema languages for XML. Our intended application requires that basic operations on sets expressed in the formalism (like intersection and checks for membership, emptiness and inclusion) are decidable and efficient algorithms for them exist. A well known such formalism is tree automata [CDG+99] (or tree grammars, which are just another view of tree automata). However tree automata deal with terms where each symbol has a fixed arity. This is not compatible with XML, where the number of elements between a given pair of a start-tag and end-tag is not fixed. One can adjust the view of XML data to the tree automata formalism, by representing sequences of arbitrary length as lists (this means terms built using two symbols of fixed arities 2 and 0). In this way n children of a tree node can be replaced by one child, which is a list of length n. Such an approach is used in [HVP00]. We follow here another approach – extending the tree automata formalism.

As abstraction of XML data we employ data terms. Data terms can be seen as mixed trees, which are labelled trees where children of a node are either linearly ordered or unordered. Our formalism for defining sets of data terms combines tree grammars with regular expressions. The latter are used to describe the possible sequences (or sets) of children of a single node in a tree. Similar formalism is used in [MLMK03], the novelty of our approach is that we deal with mixed trees.

There exist various rule languages related to XML documents (like RuleML [BTW01] or Xcerpt). Usually rules are (intended to be) applied to documents of a certain type. An obvious question arises about the set of possible results of such a rule (or of a set of rules). One would like to express the type of rule results in terms of the types of documents to which the rule is applied. A variant of this question is checking whether the rule is type correct – one requires that any result of the rule is of certain type and wants to prove (or disprove) this fact. Ability to perform such checks automatically, or to compute the type of results, is instrumental for discovering errors in the rules. Experience with programming languages shows how crucial static typing has been for quick discovering of certain kinds of errors in programs and thus for improving efficiency of programmers and quality of programs. On the other hand, experience with untyped programming languages, like Prolog, shows how lack of typing makes many simple errors difficult to discover.

In this chapter we present descriptive typing for (a large subset of) XML query language Xcerpt [BS02b, BS02c, BS02a, BBSW03]. Xcerpt stems from logic programming. It uses patterns instead of paths to navigate the database. The mechanism of matching a pattern against a database resembles unification. We present a method of computing the type of results for an Xcerpt program, given a type of the database. To simplify the presentation, our method is introduced for programs consisting of a single rule of a rather restricted form. Abandoning this restriction is however discussed informally. The method applies to checking of type correctness of arbitrary programs and to finding the result type for non recursive programs. It also subsumes checking whether a given data term is a member of a given type. A former version of this chapter appeared as [WD03]; a main contribution of the current paper is a more precise version of the algorithm.

The chapter is organized as follows. The next section introduces data terms and their correspondence to XML data. Section 2.3 presents the formalism of type definitions. Section 2.4 discusses certain restrictions on type definitions, their purpose is to obtain simpler and more efficient algorithms. The following section discusses algorithms for basic operations on types. Section 2.6 presents Xcerpt and introduces the algorithm for computing query answer types.

2.2 Modelling XML Data

We model XML data using a formalism of data terms similar to that defined in [BS02b]. Data terms can be seen as mixed trees which are labelled trees where children of a node are either linearly ordered or unordered. This is related to existence of two basic concepts in XML: *tags* which are nodes of an ordered tree and *attributes* that attach attribute-value mappings to nodes of a tree. These mappings are represented as unordered trees. Unordered children of a node may also be used to abstract from the order of elements, when this order is inessential. We assume that there is no syntactic difference between XML tag names and attribute names and they both are labels of nodes in our mixed trees (and symbols of our data terms). The infinite alphabet of labels will be denoted by $\mathcal{L}$.

A content of an element is a sequence of other elements or **basic constants**. Basic constants are basic values such as attribute values and all "free" data appearing in an XML document – all data that is between start and end tag except XML elements. Basic constants occur as strings in XML documents but they can play a role of data of other types depending on an adequate definition in DTD (or other schema languages) e.g. IDREF, CDATA,.... The set of basic constants will be denoted by $\mathcal{B}$. In our notation we will enclose all basic constants in quotation marks "".

XML documents are represented as *data terms.*

Definition 2.2.1 *A* **data term** *is an expression defined inductively as follows:*

- *Any basic constant is a data term,*
- *If l is a label and $t_1, \ldots, t_n$ are $n \geq 0$ data terms, then $l[t_1 \cdots t_n]$ and $l\{t_1 \cdots t_n\}$ are data terms.*

The linear ordering of children of the node with label l is denoted by enclosing them by brackets $[\,]$, while unordered children are enclosed by braces $\{\}$.

A *subterm* of a data term t is defined inductively: t is a subterm of t, and any subterm of t_i $(1 \leq i \leq n)$ is a subterm of $l'[t_1 \cdots t_n]$ and of $l'\{t_1 \cdots t_n\}$.

To show how XML elements are represented by data terms, consider an XML element

$$E = \text{<}tag\ attr_1{=}value_1 \cdots attr_k{=}value_k\text{>}E_1 \cdots E_n\text{</}tag\text{>},$$

$(k \geq 0, n \geq 0)$ where each E_i (for $i = 1, \ldots, n$) is an element or the text occurring between two elements or before the first element or after the last element. E is represented as a data term $tag[attributes\ child_1 \cdots child_n]$, where the data terms $child_1, \ldots, child_n$ represent $E_1, \ldots, E_n$, and the data term

$$attributes = \&\{attr_1[value_1] \cdots attr_k[value_k]\}$$

represents the attributes of E. If E has no attributes then $attributes$ is the data term $\&\{\,\}$, which will be usually abbreviated as $\&$. Subterms representing attributes are not ordered and this is denoted by enclosing them by braces.

Example 2.2.1 *This is an XML element and the corresponding data term.*

`<CD price="9.90" year="1985">`	*CD*[&{*price*["9.90"] *year*["1985"]}
`    Empire Burlesque`	"Empire Burlesque"
`    <subtitle></subtitle>`	*subtitle*[&]
`    <artist>Bob Dylan</artist>`	*artist*[& "Bob Dylan"]
`    <country>USA</country>`	*country*[& "USA"]
`</CD>`	]

The *root* of a data term t, denoted *root(t)*, is defined as follows . If t is of the form $l[t_1 \cdots t_n]$ or $l\{t_1 \cdots t_n\}$ then $root(t) = l$; for t being a basic constant we assume that $root(t) = \$$.

2.3 Type Definitions

Here we introduce a formalism for specifying a class of decidable sets of data terms representing XML documents. It is a certain simplification of the formalism of [BDM04]. First we specify a set of **type names** $\mathcal{T} = \mathcal{C} \cup \mathcal{S} \cup \mathcal{V}$ which consist of

- **type constants** from the alphabet $\mathcal{C}$
- **special type names** from the alphabet $\mathcal{S}$
- **type variables** from the alphabet $\mathcal{V}$

We associate each type name T with a set $[\![T]\!]$ (the *type denoted by* T) of data terms which are allowed values assigned to T. For T being a type constant or a special type name, the elements of $[\![T]\!]$ are basic constants.

Type constants corresponds to an XML schema language base types. The set of type constants is fixed and finite. In our examples we will use a type constant $\#$ assuming that $[\![\#]\!]$ is the set of non empty strings of characters. This is similar to `#PCDATA` in DTD.

For a special type name T the corresponding set $[\![T]\!]$ is a finite set of basic constants $\{c_1, \ldots, c_m\}$ $(m \geq 0)$. This set is specified by a rule of the form $T \to c_1|\ldots|c_m$. In our notation, type constants and special type names are sequences of letters beginning with character $\#$.

Each type variable T is associated with a set of data terms $[\![T]\!]$ which is specified in a way similar to that of [BDM04] and described below. First we introduce some auxiliary notions. The empty string will be denoted by ϵ. A *regular expression* over an alphabet Σ is ε, ϕ, any $a \in \Sigma$ and any $r_1 r_2$, $r_1|r_2$ and r_1^*, where r_1, r_2 are regular expressions. A language $L(r)$ of strings over Σ is assigned to each regular expression r in the standard way (see e.g. [HU79]). In particular, $L(\phi) = \emptyset$, $L(\varepsilon) = \{\epsilon\}$ and $L(r_1|r_2) = L(r_1) \cup L(r_2)$.

Definition 2.3.1 *A* **regular type expression** *is a regular expression over the alphabet of type names $\mathcal{T}$. We abbreviate a regular expression $r^n|r^{n+1}|\cdots|r^m$, where $n \leq m$, as $r(n:m)$, $r^n r^*$ as $r(n:\infty)$, rr^* as r^+, and $r(0:1)$ as $r^?$. A regular type expression of the form*

$$W_1 \cdots W_k$$

where $k \geq 0$, each W_i is $T_i(n_{i,1} : n_{i,2})$, $0 \leq n_{i,1} \leq n_{i,2} \leq \infty$ for $i = 1, \ldots, k$, and $T_1, \ldots, T_k$ are distinct type names, will be called a **multiplicity list**.

Multiplicity lists will be used to specify multisets of type names.

Definition 2.3.2 *A* **type definition for type variables** *$T_1, ..., T_n$ is a set of rules $\{R_1, ..., R_n\}$ where each rule R_i $(i = 1, ..., n)$ is of the form*

$$T_i \to G_i,$$

$T_1, ..., T_n$ are distinct, and each G_i is an expression of the form $l_i[r_i]$ or $l_i\{q_i\}$ where l_i is a label, r_i is a regular type expression over $\{T_1, \ldots, T_n\} \cup \mathcal{C} \cup \mathcal{S}$, and q_i is a multiplicity list over $\{T_1, \ldots, T_n\} \cup \mathcal{C} \cup \mathcal{S}$.

A type definition for type variables together with a set of rules defining special type names will be called a **type definition**. A rule of the form $T \to G$ (occurring in a type definition D) will be called the *rule for* T (in D). We require that for any special type name S the definition contains at most one rule for S.

Example 2.3.1 *Consider type definition D:*

$$\begin{array}{l}
Cd \to cd[Title\ Artist^+\ \#Category^?] \\
Title \to title[\#\ Subtitle^?] \\
Subtitle \to subtitle[\#] \\
Artist \to artist[\#] \\
\#Category \to pop \mid rock \mid classic
\end{array}$$

D contains a rule for each of type variables: Cd, Title, Subtitle, Artist *and a rule for special type name $\#Category$. Labels occurring in D are:* cd, title, subtitle, artist, *and* pop, rock, classic *are basic constants.*

Type definitions are a kind of grammars, they define sets by means of derivations over data patterns.

Definition 2.3.3 *A* **data pattern** *is inductively defined as follows*

- *a type name and a basic constant are data patterns,*
- *if* $d_1, \ldots, d_n$ $(n \geq 0)$ *are data patterns and* l *is a label then* $l[d_1 \cdots d_n]$ *and* $l\{d_1 \cdots d_n\}$ *are data patterns.*

Thus, data terms are data patterns, but not necessarily vice versa, since a data pattern may include type names in place of data terms. Given a type definition D we use it to define a rewrite relation $\rightarrow_D$ on data patterns.

Definition 2.3.4 *Let* d, d' *be data patterns.* $d \rightarrow_D d'$ *iff one of the following holds:*

1. d' *is obtained from* d *by replacing an occurrence of a type variable* T *in* d *by* $l[s]$*, for some rule* $T \rightarrow l[r]$ *in* D *and some* $s \in L(r)$ *(so* s *is a string of type names).*
2. d' *is obtained from* d *by replacing an occurrence of a type variable* T *in* d *by* $l\{s\}$*, for some rule* $T \rightarrow l\{r\}$ *in* D *and a permutation* s *of some* $s_0 \in L(r)$*.*
3. d' *is obtained from* d *by replacing an occurrence of a type constant* C *by a basic constant in* $[\![C]\!]$*.*
4. *There exists in* D *a rule* $S \rightarrow c_1 | \ldots | c_m$ *for a special type name* S*, and* d' *is obtained from* d *by replacing an occurrence of* S *by one of the basic constants* $c_1, \ldots, c_m$*.*

Example 2.3.2 *For the type definition* D *from the previous example it holds:* $Cd \rightarrow_D cd[Title\ Artist\ \#Category] \rightarrow_D^* cd[title[\#]\ artist[\#]\ \text{"pop"}] \rightarrow_D^* cd[title[\text{"Stop"}]\ artist[\text{"Sam Brown"}]\ \text{"pop"}]$*.*

Iterating the rewriting steps we may eventually arrive at a data term. This gives a semantics for type definitions.

Definition 2.3.5 *Let* D *be a type definition. The* type $[\![T]\!]_D$ *associated with a type name* T *by* D *is the set of the data terms that can be obtained from* T

$$[\![T]\!]_D = \{\, t \mid T \rightarrow_D^* t \text{ and } t \text{ is a data term} \}$$

Notice that if T is a type constant then $[\![T]\!]_D = [\![T]\!]$. If it is clear from the context which type definition is considered, we will often omit the subscript in the notation $[\![\ ]\!]_D$ and similar ones.

2.4 Proper Type Definitions

For our analysis of Xcerpt rules we need algorithms computing intersection of sets defined by type definitions, and performing emptiness and inclusion checks for such sets. To obtain efficient algorithms we impose certain restrictions on type definitions. They are discussed in this section.

Consider a type definition D. If $T \rightarrow G$ is the rule for a type variable T in D, where G is of the form $l[r]$ or $l\{q\}$, then l will be called the **label** of T (in D) and denoted $label_D(T) = l$.

For T being a type constant or a special type name we define $label_D(T) = \$$. So if $d \in [\![T]\!]$ then $root(d) = label(T)$.

We assume that alphabet of labels $\mathcal{L} \cup \{\$\}$ is totally ordered by a relation $\leq$; we call this ordering *alphabetic ordering.* A multiplicity list $W_1 \dots W_k$, where each $W_i = T_i(n_{i,1} : n_{i,2})$ and T_i is a type name, is **sorted** w.r.t D if $label_D(T_1) \leq \dots \leq label_D(T_k)$. For practical reasons we assume that the multiplicity lists occurring in our type definitions are sorted.

We say that a type definition D is **proper**, if for each regular expression r in D all distinct type names occurring in r have different labels. Thus given a term $l[c_1 \dots c_n]$ and a rule $T \to l[r] \in D$ or a term $l\{c_1 \dots c_n\}$ and a rule $T \to l\{r\} \in D$ for each c_i, the root of c_i determines at most one type name S such that S occurs in r and $label_D(S) = root(c_i) = l_i$. Such type name S will be denoted $type_D(l_i, r)$. If any type occurring in r does not have label l_i we assume that $type_D(l_i, r) = NULL$. We use $types_D(r)$ to denote the set of all type names occurring in the regular expression r.

Notice that, for a proper type definition D, at most one type constant or special type name occurs in any regular expression of D since all type constants and special type names have the same label \$.

Restriction to proper type definitions results in simpler and more efficient algorithms. Unless stated otherwise, we assume that the considered type definitions are proper. The class of proper type definitions, when restricted to ordered terms (i.e. without $\{\}$), is essentially the same as single-type tree grammars of [MLMK03]. Dealing only with proper definitions seems reasonable, as the sets defined by main XML schema languages (DTD and XML Schema) can be expressed by such definitions [MLMK03].

Example 2.4.1 *Type definition* $D_1 = \{A{\to}a[A|B|C], B{\to}b[D], C{\to}b[\#], D{\to}c[\#]\}$ *is not proper because type names* B, C *have the same label* b *and occur in one regular expression. In contrast,* $D_2 = \{A{\to}a[A|B|D], B{\to}b[CD], C{\to}b[\#], D{\to}c[\#]\}$ *is proper and e.g.* $type_{D_2}(b, A|B|D) = B$ *and* $type_{D_2}(b, CD) = C$.

Our algorithms employ inclusion and equality checks for languages described by given regular expressions, and computing intersection of such languages. This can be done by transforming regular expressions to deterministic finite automata (DFA's) and using standard efficient algorithms for DFA's.

In the general case the number of states in a DFA may be exponentially greater than the length of the corresponding regular expression [HU79]. Notice that the XML definition [Ext] requires (Section 3.2.1) that content models specified by regular expressions in element type declarations of a DTD are *deterministic* in the sense of Appendix E of [Ext]. It seems that the formal meaning of this requirement is that the regular type expressions are 1-unambiguous in a sense of [BKW98]. For such regular expressions a corresponding DFA can be constructed in linear time.

2.5 Operations on Types

In this section we describe algorithms computing basic operations on types: check for emptiness, intersection, and check for inclusion.

2.5.1 Emptiness Check

We show how to check if a type defined by a type definition is empty. In what follows we assume that the regular expressions in type definitions do not have useless symbols. A type name T is *useless* in a regular expression r if no string in $L(r)$ contains T. (If r contains a useless symbol then the regular expression ϕ occurs in r.)

A type name T in a type definition D will be called **nullable** if no data terms can be derived from T. In other words, $[\![T]\!]_D = \emptyset$ iff T is nullable in D.

To find nullable symbols in a type definition D we mark type names in D in the following way. First we mark all type constants and all special type names (that do not denote $\emptyset$). Then we mark each unmarked type variable T_i in D with the rule for T_i of the form $T_i \to l[r_i]$ or of the form $T_i \to l\{r_i\}$ such that there exists a sequence of marked type names $S_1 \dots S_m \in L(r_i)$ ($m \geq 0$). We repeat the second step until an iteration which does not change anything. The type names which are unmarked in D are nullable.

Example 2.5.10 *Let us use the algorithm to find nullable type names in a type definition* $D = \{\, A{\to}a[AB],\ B{\to}b[B^*]\,\}$. *The initial step does not mark any type names. In the second step we mark* B *because* $\epsilon \in L(B^*)$. *In the next iteration we cannot mark any other type names and the algorithm stops. Since* A *is unmarked, it is nullable.*

2.5.2 Intersection of Types

Here we explain a way of obtaining the intersection of two types. Let D_1, D_2 be type definitions (possibly $D_1 = D_2$). We construct a type definition D describing intersections of types defined by D_1, D_2. For each pair of type variables S_1, S_2 from, respectively, D_1, D_2 we introduce a new type variable $S_1 \dot\cap S_2$. D will satisfy $[\![S_1 \dot\cap S_2]\!]_D = [\![S_1]\!]_{D_1} \cap [\![S_2]\!]_{D_2}$.

We assume that for each pair S_1, S_2 of type constants there is a type constant $S_1 \dot\cap S_2$ such that $[\![S_1 \dot\cap S_2]\!] = [\![S_1]\!] \cap [\![S_2]\!]$. For each pair S_1, S_2, where one element is a type constant and the other is a special type name or both are special type names, we introduce a new special type name $S_1 \dot\cap S_2$.

The type definition D is the smallest set of rules such that if S_1, S_2 are type variables and D_1, D_2 contain, respectively, rules of the form $S_1 \to l[r_1]$ and $S_2 \to l[r_2]$ or of the form $S_1 \to l\{r_1\}$ and $S_2 \to l\{r_2\}$ then

- let, for $i = 1, 2$, s_i be the regular expression r_i with every type name U replaced by $label_{D_i}(U)$,
- let s be a regular expression such that $L(s) = L(s_1) \cap L(s_2)$, if the parentheses in the rules for S_1, S_2 are $\{\}$ then we require that s is a sorted multiplicity list (like s_1, s_2 are),
- for each label l occurring in s let $S_{1,l}, S_{2,l}$ be the type names such that $type_{D_1}(l, r_1) = S_{1,l}$, $type_{D_2}(l, r_2) = S_{2,l}$,
- let r be s with every label l replaced by $S_{1,l} \dot\cap S_{2,l}$,
- if the rules for S_1, S_2 are of the form $S_1 \to l[r_1]$ and $S_2 \to l[r_2]$ then D contains the rule $S_{1,l} \dot\cap S_{2,l} \to l[r]$;

  if the rules for S_1, S_2 are of the form $S_1 \to l\{r_1\}$ and $S_2 \to l\{r_2\}$ then D contains the rule $S_{1,l} \dot\cap S_{2,l} \to l\{r\}$.

If S_1 and S_2 are type variables and D_1, D_2 contain, respectively, rules of the form $S_1 \to l[r_1]$ and $S_2 \to l\{r_2\}$ or of the form $S_1 \to l\{r_1\}$ and $S_2 \to l[r_2]$ then D contains the rule $S_1 \dot\cap S_2 \to l[\phi]$.

If S_1, S_2 are special type names, or one of them is a special type name and the other is a constant type then D contains the rule $S_1 \dot{\cap} S_2 \rightarrow c_1 | \ldots | c_n$, where $[\![S_1]\!]_{D_1} \cap [\![S_2]\!]_{D_2} = \{c_1, \ldots, c_n\}$.

From the description above and from Definition 2.3.4 (of the derivation relation $\rightarrow_D$) it follows that

$$
\begin{array}{lll}
T \dot{\cap} U \rightarrow_D l[T_1 \dot{\cap} U_1 \cdots T_n \dot{\cap} U_n] & \text{iff} & T \rightarrow_{D_1} l[T_1 \cdots T_n], \\
& & U \rightarrow_{D_2} l[U_1 \cdots U_n], \text{ and} \\
& & label_{D_1}(T_i) = label_{D_2}(U_i) \text{ for } i = 1, \ldots, n,
\end{array}
$$

for any $n \geq 0$ and type variables $T, U, T_1, \ldots, T_n, U_1, \ldots, U_n$. A similar fact holds for rules with $\{\}$. Thus $T \dot{\cap} U \rightarrow_D^* t$ iff $T \rightarrow_{D_1}^* t$ and $U \rightarrow_{D_2}^* t$ (for any data term t); this implies $[\![T \dot{\cap} U]\!]_D = [\![T]\!]_{D_1} \cap [\![U]\!]_{D_2}$. If definitions D_1 and D_2 are proper then D is proper.

Example 2.5.11 *Consider type definitions:* $D = \{\, A \rightarrow l[B|C],\ B \rightarrow l[A^+],\ C \rightarrow m[\,]\,\}$ *and* $D' = \{\, A' \rightarrow l[A'^*|C'],\ C' \rightarrow m[C'^*]\,\}$. *We construct a type definition* D'' *which defines type* $A \dot{\cap} A'$ *being the intersection of types* A *and* A' *(*$[\![A \dot{\cap} A']\!]_{D''} = [\![A]\!]_D \cap [\![A']\!]_{D'}$*).* $D'' = \{\, A \dot{\cap} A' \rightarrow l[B \dot{\cap} A' | C \dot{\cap} C'],\ B \dot{\cap} A' \rightarrow l[(A \dot{\cap} A')^+],\ C \dot{\cap} C' \rightarrow m[\,]\,\}$. *Example 2.5.12 will show that* $[\![A]\!]_D \subseteq [\![A']\!]_{D'}$ *and that is why* $[\![A \dot{\cap} A']\!]_{D''} = [\![A]\!]_D$.

2.5.3 Inclusion Subtyping

The algorithm presented here is based on the approach taken in [BDM04]. The slight difference comes from the fact that our formalism is more specific for Xcerpt. In our approach we do not use *label language* but instead we assume that every type constant has the same label \$.

Let T_1, T_2 be type names defined in type definitions D_1, D_2, respectively. T_1 is an *inclusion subtype* of T_2 iff $[\![T_1]\!]_{D_1} \subseteq [\![T_2]\!]_{D_2}$. We present an algorithm which checks this fact. It is not required that D_1 is proper.

The first part of the algorithm constructs a set $C(T_1, T_2)$ of pairs of types to be compared. It is the smallest set such that

- if $label(T_1) = label(T_2)$ then $(T_1, T_2) \in C(T_1, T_2)$,
- if
  - $(T_1', T_2') \in C(T_1, T_2)$,
  - D_1, D_2 contain, respectively, rules $T_1' \rightarrow l[r_1]$ and $T_2' \rightarrow l[r_2]$, or $T_1' \rightarrow l\{r_1\}$ and $T_2' \rightarrow l\{r_2\}$ (with the same label l), and
  - type names T_1'', T_2'' occur respectively in r_1, r_2, and $label_{D_1}(T_1'') = label_{D_2}(T_2'')$

  then $(T_1'', T_2'') \in C(T_1, T_2)$. As D_2 is proper, for every T_1'' in r_1, there exists at most one T_2'' in r_2 satisfying this condition.

The second part of the algorithm checks whether $[\![T_1']\!] \subseteq [\![T_2']\!]$ for each $(T_1', T_2') \in C(T_1, T_2)$:

IF $C(T_1, T_2) = \emptyset$ THEN return false
ELSE for each $(T_1', T_2') \in C(T_1, T_2)$ do the following:
    IF T_1', T_2' are special type names or type constants
        THEN check whether $[\![T_1']\!] \subseteq [\![T_2']\!]$ and return the result
    Let $T_1' \to l[r_1]$ and $T_2' \to l[r_2]$, or $T_1' \to l\{r_1\}$ and $T_2' \to l\{r_2\}$
        be rules of D_1, D_2, respectively
    Let s_1 and s_2 be the regular expressions over labels
        corresponding to r_1 and r_2
    Check whether $L(s_1) \subseteq L(s_2)$
  IF for all pairs from $C(T_1, T_2)$ the answer is true THEN return true
  ELSE return false

The algorithm employs a check if $[\![T_1']\!] \subseteq [\![T_2']\!]$, where each of T_1', T_2' is either a special type name or a type constant. This check is based on recorded information about inclusion of the sets defined by type constants and about which constants are members of these sets.

If the algorithm returns *true* then $[\![T_1]\!]_{D_1} \subseteq [\![T_2]\!]_{D_2}$. If it returns *false* and D_1 has no nullable symbols (i.e. $[\![T]\!]_{D_1} \neq \emptyset$ for each type name T in D_1) then $[\![T_1]\!]_{D_1} \not\subseteq [\![T_2]\!]_{D_2}$. The main fact used in the proof of this property is that a positive answer of the algorithm means the following. For any (S, U), $(S_1, U_1), \ldots, (S_n, U_n) \in C(T_1, T_2)$ if $S \to_{D_1} l[S_1 \cdots S_n]$ then $U \to_{D_2} l[U_1 \cdots U_n]$. A similar fact holds for terms with $\{\}$ (remember that in this case the regular expressions in the applied rules are sorted multiplicity lists).

Example 2.5.12 *Consider the type definitions from the Example 2.5.11:* $D = \{\, A{\to}l[B|C],\ B{\to}l[A^+],\ C{\to}m[\,]\,\}$ *and* $D' = \{\, A'{\to}l[A'^*|C'],\ C'{\to}m[C'^*]\,\}$. *To check whether* $[\![A]\!]_D \subseteq [\![A']\!]_{D'}$*, first we construct set* $C(A, A')$ *which is* $\{(A, A'), (B, A'), (C, C')\}$. *Then the second part of the algorithm checks if* $L(l|m) \subseteq L(l^*|m)$, $L(l^+) \subseteq L(l^*|m)$ *and* $L(\epsilon) \subseteq L(m^*)$. *Since all the checks give positive results, we conclude that* $[\![A]\!]_D \subseteq [\![A']\!]_{D'}$.

Notice that for a proper D_2 and 1-unambiguous regular expressions [BKW98] in D_1, D_2 the algorithm is polynomial. In the general case a polynomial algorithm does not exist, as inclusion for a less general formalism of tree automata is EXPTIME-complete [CDG+99].

2.6 Typing of Xcerpt Query Results

In this section we first introduce XML query language Xcerpt. Then we discuss objectives of computing types of query results and present an algorithm.

2.6.1 Xcerpt – Introduction

Xcerpt is a rule-based query and transformation language for XML (see [BS02a, BS02b, BS02c, BBSW03, FBS+04]). It employs patterns instead of paths to query XML and semistructured data. This approach stems from logic programming. A query term is matched against a data term from a database. A successful matching results in binding the variables in the query term to certain subterms of the data term. This operation is called simulation unification.

We consider here a somehow simplified version of Xcerpt. We focused on core Xcerpt features to make our algorithms simpler and better understandable. The main difference is that our data terms represent trees while in full Xcerpt terms are used to represent graphs (by adding unique identifiers to some tree nodes and introducing nodes which are references to these identifiers). Other neglected Xcerpt features in respect to the Xcerpt version described in [SB04] are:

functions and aggregations, non-pattern conditions, optional subterms, position specifications, negation, regular expressions, *all* and *some* constructs, and label variables in query terms.

We assume that a database is a data term or a multiset of data terms. There are two other kinds of terms in Xcerpt: query terms and construct terms. A **construct term** is a data term possibly with some subterms replaced by variables. We define query terms later on. Any data term is a construct term, and any construct term is a query term. The role of query terms is to be matched against a database. Construct terms are used in constructing data terms which are query results. Queries in Xcerpt are (sets of) rules; the premise of a rule is a query term and the conclusion of a rule is a construct term.

Definition 2.6.1 **Query terms** *are inductively defined as follows:*

- *Any basic constant is a query term.*
- *A variable X is a query term.*
- *If q is a query term, then $desc\, q$ is a query term.*
- *If X is a variable and q is a query term, then $X \leadsto q$ is a query term.*
- *If l is a label and $q_1 \dots q_n$ $(n \geq 0)$ are query terms, then $l[q_1 \dots q_n]$, $l\{q_1 \dots q_n\}$, $l[[q_1 \dots q_n]]$ and $l\{\{q_1 \dots q_n\}\}$ are query terms (called* rooted *query terms).*

For a rooted query term $q = l\alpha q_1, \dots, q_n\beta$, where $\alpha\beta$ are parentheses $[\,], [[\,]], \{\}$ or $\{\{\}\}$, $root(q) = l$ and $q_1, \dots, q_n$ are the child subterms *of q. If q is a basic constant then $root(q) = \$$.*

To informally explain the role of query terms, consider a query term $q = l\alpha q_1 \dots q_m\beta$ and a data term $d = l'\alpha' d_1 \dots d_n\beta'$, where $\alpha, \beta, \alpha', \beta'$ are parentheses. In order to q match d it is necessary that $l = l'$. Moreover the child subterms $q_1, \dots, q_m$ of q should match certain child subterms of d. Single parentheses in d ($[\,]$ or $\{\}$) mean that $m = n$ and each q_i should match some (distinct) d_j. Double parentheses mean that $m \leq n$ and $q_1, \dots, q_m$ are matched against some m terms out of $d_1, \dots, d_n$. Curly brackets ($\{\}$ or $\{\{\}\}$) in q mean that the order of the child subterms in d does not matter; square brackets in q mean that $q_1, \dots, q_m$ should match (a subsequence of) $d_1, \dots, d_n$ in the same order.

A variable matches any data term, $desc\, q$ matches a data term d whenever q matches some subterm of d. A query term $X \leadsto q$ matches any data term matched by q. A side effect of a query term X or $X \leadsto q$ matching a data term d is that variable X obtains a value d.

Now we formally define which query terms match which data terms and what are the resulting assignments of data terms to variables. We do not follow the original definition of simulation unification. Instead we define a notion of answer substitution for a query term q and a data term d. As usually, by a ***substitution*** (of data terms for variables) we mean a set $\theta = \{\, X_1/d_1, \dots, X_n/d_n \,\}$, where $X_1, \dots, X_n$ are distinct variables and $d_1, \dots, d_n$ are data terms; its domain $dom(\theta)$ is $\{X_1, \dots, X_n\}$, its application to a (query) term is defined in a standard way.

Definition 2.6.2 *A substitution θ is an* answer substitution *(shortly, an* ***answer****) for a query term q and a data term d if q and d are of one of the forms below and the corresponding condition holds. (In what follows $m, n \geq 0$, X is a variable, l is a label, $q, q_1, \dots$ are query terms, and $d, d_1, \dots$ data terms; set notation is used for multisets, for instance $\{d, d\}$ and $\{d\}$ are different multisets).*

q	d	condition on q and d
c	c	c is a basic constant
$l[q_1 \cdots q_n]$	$l[d_1 \cdots d_n]$	θ is an answer for q_i and d_i, for each $i = 1, \ldots, n$
$l[[q_1 \cdots q_m]]$	$l[d_1 \cdots d_n]$	for some subsequence $d_{i_1}, \ldots, d_{i_m}$ of $d_1, \ldots, d_n$ (i.e. $0 < i_1 < \ldots < i_m \leq n$) θ is an answer for q_j and d_{i_j}, for each $j = 1, \ldots, m$,
$l\{q_1 \cdots q_n\}$	$l\{d_1 \cdots d_n\}$ or $l[d_1 \cdots d_n]$	for some permutation $d_{i_1}, \ldots, d_{i_n}$ of $d_1, \ldots, d_n$ (i.e. $\{d_{i_1}, \ldots, d_{i_n}\} = \{d_1, \ldots, d_n\}$) θ is an answer for q_j and d_{i_j} for each $j = 1, \ldots, m$,
$l\{\{q_1 \cdots q_m\}\}$	$l\{d_1 \cdots d_n\}$ or $l[d_1 \cdots d_n]$	for some $\{d_{i_1}, \ldots, d_{i_m}\} \subseteq \{d_1, \ldots, d_n\}$ θ is an answer for q_j and d_{i_j} for each $j = 1, \ldots, m$,
X	d	$\theta X = d$
$X \leadsto q$	d	$\theta X = d$ and θ is an answer for q and d
$desc\, q$	d	θ is an answer for q and some subterm d' of d

We say that q matches d *if there exists an answer for* q, d.

Thus if q is a rooted query term (or a basic constant) and $root(q) \neq root(d)$ then no answer for q, d exists. If $q = d$ then any θ is an answer for q, d. A query $l\{\{\}\}$ matches any data term with the label l. If θ, θ' are substitutions and $\theta \subseteq \theta'$ then if θ is an answer for q, d then θ' is an answer for q, d. If a variable X occurs in a query term q then queries $X \leadsto q$ and $X \leadsto desc\, q$ match no data term, provided that $q \neq X$ and q is not of the form $desc \cdots desc\, X$.

Example 2.6.1 *Query term* $q_1 = a[\, c\{\{d[\,]\, "e"\}\}\, f[[g[\,]\, h\{"i"\}]]\,]$ *matches data terms* $a[\, c\{"e"\, d[\,]\, g[\,]\}\, f[g[\,]\, l[\,]\, h["i"]\,]\,]$ *and* $a[\, c[d[\,]\, g[\,]\, "e"]\, f[g[\,]\, h["i"]\,]\,]$. *In contrast, data terms* $f[h["i"]\, g[\,]\,]$ *and* $f\{g[\,]\, h["i"]\}$ *are not matched by* $f[[g[\,]\, h\{"i"\}]]$. *Query term* $q_2 = desc\; w\{\{\}\}$ *matches data terms* $a[b\{w[\,]\}]$ *and* $w\{"s"\}$. *Query term* $q_2 = a[[\, X_1 \leadsto c[[d\{\}]]\, X_2\, "p"\,]]$ *matches* $a["s"\, c[d\{\}\, "r"]\, h\{j[\,]\}\, "p"]$, *with an answer which binds* X_1 *to* $c[d\{\}\, "r"]$ *and* X_2 *to* $h\{j[\,]\}$.

Each answer for a query term q binds all the variables of the query to some data terms. For any such answer θ' (for q and d) there exists an answer $\theta \subseteq \theta'$ (for q and d) binding exactly these variables. We will call such answers *non redundant.* Out of Definition 2.6.2 one can derive an algorithm which produces non redundant answers for a given q and d. Construction of the algorithm is rather simple, we skip the details. Non redundant answers are actually those of interest; we consider a more general class of answers to simplify Definition 2.6.2.

An Xcerpt program is a set of construct-query rules. We restrict ourselves to a simple kind of rules and to programs consisting of a single rule.

Definition 2.6.3 *A* **construct-query rule** *(shortly,* query rule *or* query*) is an expression of the form* $t \leftarrow q$*, where* t *is a construct term,* q *is a query term and every variable occurring in*

t also occurs in q. t will be sometimes called the head *and q the* body *of the rule. If θ is an answer for q and a data term d then $t\theta$ is a* **result** *for query $t \leftarrow q$ and d.*

Each result of a query rule is a data term, as an answer for a query term binds all the variables of the rule to data terms.

Example 2.6.2 *Consider a database:*

$$\begin{array}{ll} \textit{catalogue}[& \textit{cd}[\,\textit{title}[\,\textit{"Empire Burlesque"}]\,\textit{artist}[\,\textit{"Bob Dylan"}]\,\textit{year}[\,\textit{"1985"}]\,] \\ & \textit{cd}[\,\textit{title}[\,\textit{"Hide your heart"}]\,\textit{artist}[\,\textit{"Bonnie Tyler"}]\,\textit{year}[\,\textit{"1988"}]\,] \\ & \textit{cd}[\,\textit{title}[\,\textit{"Stop"}]\,\textit{artist}[\,\textit{"Sam Brown"}]\,\textit{year}[\,\textit{"1988"}]\,]\,] \end{array}$$

Here is a rule which extracts titles and artists for the CD's issued in 1988 and presents the results in a changed form (title as name and artist as author). TITLE and ARTIST are variables.

$$\begin{array}{l} \textit{result}[\,\textit{name}[\textit{TITLE}]\,\textit{author}[\textit{ARTIST}]\,] \leftarrow \\ \qquad \textit{catalogue}\{\{\,\textit{cd}\{\textit{title}[\textit{TITLE}]\,\textit{artist}[\textit{ARTIST}]\,\textit{year}[\,\textit{"1988"}]\,\}\}\} \end{array}$$

The results returned by the rule are:

$$\begin{array}{l} \textit{result}[\,\textit{name}[\,\textit{"Hide your heart"}]\,\textit{author}[\,\textit{"Bonnie Tyler"}]\,] \\ \textit{result}[\,\textit{name}[\,\textit{"Stop"}]\,\textit{author}[\,\textit{"Sam Brown"}]\,] \end{array}$$

2.6.2 Reasoning about Types of Xcerpt Query Results

In this section we study the relation between types of databases and types of query results. Assume that the only information available about the database is that it is a data term (or a set of data terms) of a given type $[\![T_{\mathrm{DB}}]\!]$. One may want to know what query results are possible for such database. We show how to compute (a superset of) the set of such results. The set will be expressed as a type, specified by a type definition. We will usually call it the query result type.

Computing the query result type may serve some additional purposes. 1. If this type is empty, then the query will never give an answer for a data term from $[\![T_{\mathrm{DB}}]\!]$. An algorithm checking this property is obtained by combining computing query result type with checking emptiness of a type. 2. If some specification of the intended type of results exists, one may check if the query is correct w.r.t. the specification, by checking whether the computed type of the results is included in the specified one. 3. If we use a data term d as the body of the query, then computing the result type is also a check whether $d \in [\![T_{\mathrm{DB}}]\!]$. Namely $d \in [\![T_{\mathrm{DB}}]\!]$ iff the result type is not empty. 4. The algorithm computing the query result type produces as a side effect the types of the variables of the queries. For each variable from the query it gives a set containing every value that can be assigned to the variable (when querying a data term from type $[\![T_{\mathrm{DB}}]\!]$). This provides additional information about the behaviour of the query. We may consider specifications of the types of the query variables. A query is correct w.r.t. such a specification if for every variable the computed type is a subset of the specified type.

Example 2.6.3 *Consider the type definition D from Example 2.3.1 and a construct-query rule Q:*

$$\begin{array}{l} \textit{result}[\,\textit{name}[\textit{TITLE}]\,\textit{author}[\textit{ARTIST}]\,] \leftarrow \\ \qquad \textit{cd}\{\{\,\textit{TITLE}\ \textit{ARTIST}{\leadsto}\textit{artist}\{\{\}\}\ \textit{"rock"}\}\} \end{array}$$

The intention of the rule is to collect titles and authors of all the CD's of the rock category. When the query term of the rule is matched against a database of type Cd, the variables TITLE, ARTIST are bound to data terms of types, respectively, Title, Artist or Artist, Artist. As the variable TITLE is intended to take values only of type Title, the query is incorrect w.r.t. our expectations. The type Result of the query result can be described by the following type definition $D' = D \cup \{ Result{\rightarrow}result[Name\ Author],\ Name{\rightarrow}name[Title|Artist],\ Author{\rightarrow}author[Artist] \}$.

In what follows we assume a fixed proper type definition D (describing the type of the database).

To represent a set of answers (for a query term and a set of data terms) we will use a mapping $m: V \to \mathcal{E}$, where V is the set of variables occurring in the considered query rule and $\mathcal{E}$ is a set of expressions. $\mathcal{E}$ contains 0, 1, the type names from D, and expressions of the form $T_1 \cap T_2$, where $T_1, T_2 \in \mathcal{E}$. Each expression E from $\mathcal{E}$ denotes a set $[\![E]\!]$ of data terms. $[\![1]\!]$ denotes the set of all data terms, $[\![0]\!] = \emptyset$, $[\![T]\!] = [\![T]\!]_D$ for any type name T, and $[\![T_1 \cap T_2]\!] = [\![T_1]\!] \cap [\![T_2]\!]$. The set of substitutions corresponding to a mapping $m: V \to \mathcal{E}$ is

$$substitutions_D(m) = \{ \theta \mid \forall_{X \in V}\ \theta X \in [\![m(X)]\!] \}.$$

(So if $\theta \in substitutions_D(m)$ then $V \subseteq dom(\theta)$ and if $\theta \subseteq \theta'$ then $\theta' \in substitutions_D(m)$.)

We define $\bot, \top: V \to \mathcal{E}$ by $\bot(X) = 0$ and $\top(X) = 1$ for every $X \in V$. For $Y_1, \ldots, Y_k \in V$, $T_1, \ldots, T_k \in \mathcal{E}$, mapping $[Y_1 \mapsto T_1, \ldots, Y_k \mapsto T_k]: V \to \mathcal{E}$ is defined as

$$[Y_1 \mapsto T_1, \ldots, Y_k \mapsto T_k](X) = \begin{cases} T_i & \text{if } X = Y_i \\ 1 & \text{otherwise.} \end{cases}$$

We will not distinguish between expressions $T \cap 1$ and T, and between $T \cap 0$ and 0 (where $T \in \mathcal{E}$). For any $m_1, m_2: V \to \mathcal{E}$ we introduce $m_1 \cap m_2: V \to \mathcal{E}$ such that

$$(m_1 \cap m_2)(X) = m_1(X) \cap m_2(X).$$

Notice that $m \cap \bot = \bot$ and $m \cap \top = m$ for any $m: V \to \mathcal{E}$.

For a particular query term there may be many possible assignments of types for variables. That is why we will use sets of mappings from $V \to \mathcal{E}$. For such sets M_1 and M_2 we define:

$$\begin{array}{rcl} M_1 \sqcap M_2 & = & \{m_1 \cap m_2 \mid m_1 \in M_1, m_2 \in M_2\} \\ M_1 \sqcup M_2 & = & M_1 \cup M_2 \end{array}$$

Hence $M \sqcap \{\bot\} = \{\bot\}$, $M \sqcap \{\top\} = M$, for any set of mappings M. We will not distinguish between $M \sqcup \{\bot\}$ and M, and between $M \sqcup \{\top\}$ and $\{\top\}$.

2.6.3 Computing Approximated Set of Answers for a Query Term.

A first step of computing the types of results of query rules is computing the set of answers for a given query term q and the data terms from a given $[\![T]\!]_D$. We begin presentation of our algorithm from its auxiliary procedure, called *restrict_language*.

The input for *restrict_language* are a regular type expression r_S, parentheses $\alpha\beta$ and a type variable T. As it will be explained later, r_S is related to a query term $l\alpha q_1, \ldots, q_n\beta$ and all the strings in $L(r_S)$ are of the same length n. Below we assume that r is a regular type expression occurring in the rule for T in D. The procedure returns a regular language $L' \subseteq L(r_S)$. The strings $T_1 \cdots T_n$ that belong to L' satisfy certain conditions depending on the kind of parentheses $\alpha\beta$:

- if $\alpha\beta$ are single or double brackets, and the rule for T in D contains braces then $L' = \emptyset$,
- if $\alpha\beta$ are single brackets, and the rule for T in D contains brackets then $T_1 \cdots T_n \in L(r)$,
- if $\alpha\beta$ are double brackets, and the rule for T in D contains brackets then $T_1 \cdots T_n$ is a subsequence of a string from $L(r)$,
- if $\alpha\beta$ are single braces then a permutation of $T_1 \cdots T_n$ belongs to $L(r)$,
- if $\alpha\beta$ are double braces then a permutation of $T_1 \cdots T_n$ is a subsequence of a string from $L(r)$.

$T_1 \cdots T_n \in L'$ means that applying query $l\alpha q_1, \ldots, q_n\beta$ to data terms from $[\![T]\!]$ results in applying q_i to data terms from $[\![T_i]\!]$ (for $i = 1, \ldots, n$).

restrict_language$(r_S, \alpha\beta, T)$:
  let r be the regular expression in the rule for T
  let s be r with every type name U replaced by $U|\epsilon$
  IF the rule for T in D is of the form $T \to l\{r\}$ and ($\alpha\beta = [\,]$ or $\alpha\beta = [[\,]]$) THEN
    return $\emptyset$
  IF $\alpha\beta = [\,]$ THEN return $L(r) \cap L(r_S)$
  IF $\alpha\beta = [[\,]]$ THEN return $L(s) \cap L(r_S)$
  IF $\alpha\beta = \{\}$ THEN return $perm(L(r)) \cap L(r_S)$
  IF $\alpha\beta = \{\{\}\}$ THEN return $perm(L(s)) \cap L(r_S)$

Here $perm(L)$ stands for the language of permutations of the strings from a language L.

The procedure employs some operations on regular languages. One of them is intersection of such languages which can be computed in standard way by construction of a product automaton. The other operation computing intersection of a regular language $L(r_S)$ and a language $perm(L(r))$ containing all permutations of words of some other regular language is more complex. To compute it we use the fact that the regular expression r_S has a special form. Let $T_1, \ldots, T_m$ be type names occurring in r and let r_{All} be the regular expression $T_1|\ldots|T_m$. The regular expression r_S has the form $S_1 S_2 \cdots S_n$, where each S_i is a type name occurring in r or it is the regular expression r_{All}. Let $U_1 \cdots U_k$ be $S_1 \cdots S_n$ with all r_{All} removed.

To compute $perm(L(r)) \cap L(r_S)$ we first construct an automaton for the language $L(r)$. Then, treating it as a graph, we find all paths of a length n leading from a start state to some final state, containing transitions with labels $U_1, \ldots, U_k$ in an arbitrary order. To do that we can use a standard algorithm e.g. breadth first search. For each such path consider the multiset $\{U_1, \ldots, U_k, V_1, \ldots, V_{n-k}\}$ of the labels of the path, and the multiset $\{V_1, \ldots, V_{n-k}\}$. Let W be the set of such $(n-k)$-element multisets corresponding to the obtained paths. Then $perm(L(r)) \cap L(r_S)$ is the set of all strings $S'_1 \cdots S'_n$, where $S'_1 \cdots S'_n$ is obtained from $S_1 \cdots S_n$ by replacing the i-th occurrence of r_{All} by V_i, for some $\{V_1, \ldots, V_{n-k}\} \in W$. (So for each multiset $\{V_1, \ldots, V_{n-k}\} \in W$ and each permutation $V'_1, \ldots, V'_{n-k}$ of $V_1, \ldots, V_{n-k}$ there exists one string of the form $\cdots V'_1 \cdots V'_{n-k} \cdots$ in $perm(L(r)) \cap L(r_S)$. Such strings for a given $\{V_1, \ldots, V_{n-k}\} \in W$ can be described by a DFA with $O(k \cdot 2^{n-k})$ states, where each state records the set of the already consumed symbols from $\{V_1, \ldots, V_{n-k}\}$ and the number of consumed symbols from $\{U_1, \ldots, U_k\}$.)

The algorithm is inefficient and is applicable only to cases where n and k are small. It seems that mainly such cases occur in practice. For a general case one can resort to an approximate algorithm, for instance returning $\emptyset$ if the above-mentioned set of paths is found to be empty and $L(r_S)$ otherwise.

Now we are ready to present an algorithm which computes the set of answers for a given query term q and the data terms from a given type $[\![T]\!]_D$ of a database.

$match(q, T)$:
IF q is a variable X THEN
return $\{[X \mapsto T]\}$
IF q is of the form $X \leadsto q'$
return $\{[X \mapsto T]\} \sqcap match(q', T)$
IF q is of the form $desc\, q'$ THEN
IF T is a type constant or a special type name THEN
return $match(q', T)$
let r be the regular type expression in the rule for T in D
return $match(q', T) \sqcup \bigsqcup_{T' \in types(r)} match(q, T')$
(Now q is a rooted query term or a basic constant).
IF $root(q) \neq label(T)$ THEN return $\emptyset$
IF T is a type constant or a special type name THEN
IF q is a basic value in $[\![T]\!]$ THEN return $\{\top\}$ ELSE return $\emptyset$
let $q = l\alpha q_1 \cdots q_n \beta$ $(n \geq 0)$,
let r be the regular type expression in the rule for T in D
let $\{S_1, \ldots, S_m\}$ be the set $types(r)$
let r_{All} be regular type expression $S_1 | \ldots | S_m$
let r_S be regular type expression $r_1 r_2 \ldots r_n$ where

$$r_i = \begin{cases} type(root(q_i), r) & \text{if } q_i \text{ is a rooted query term} \\ & \text{or a basic constant,} \\ r_{All} & \text{otherwise} \end{cases}$$

return $\{\, m_1 \cap \ldots \cap m_n \mid T_1 \ldots T_n \in restrict\_language(r_S, \alpha\beta, T),$
$m_1 \in match(q_1, T_1), \ldots, m_n \in match(q_n, T_n) \,\}$

Notice that each mapping $m \in match(q, T)$ has a property that $m(X)$ is neither 1 nor 0 for any variable X occurring in q, and $m(X) = 1$ for any X not occurring in q. (It is however possible that $m(X) = T_1 \cap T_2$, where $[\![T_1]\!] \cap [\![T_2]\!] = \emptyset$.)

The set of mappings $match(q, T)$ produced by the algorithm describes the possible answers for q. If q does not contain $\leadsto$ then the description is exact.

Proposition 2.6.1 *Let q be a query term and $S = \bigcup_{m \in match(q,T)} substitutions_D(m)$. If θ is an answer for q and a data term $d \in [\![T]\!]_D$ then $\theta \in S$. If q does not contain $\leadsto$ then each $\theta \in S$ is an answer for q and some $d \in [\![T]\!]$.*

The values of the mappings from $M = match(q, T)$ may be expressions of the form $T_1 \cap \ldots \cap T_n$, where each T_i is a type name. Consider the set W_M of all such expressions

$$W_M = \left\{ T_1 \cap \ldots \cap T_n \;\middle|\; \begin{array}{l} T_1 \cap \ldots \cap T_n = m(X),\ m \in M,\ X \in V \\ n > 1, \text{ each } T_i \text{ is a type name} \end{array} \right\}.$$

For any expression $E \in W_M$, $[\![E]\!]$ is the intersection of types defined by D. Using the algorithm from 2.5.2 we can construct a type definition D_M such that for each $E \in W_M$ there exists a type variable T_E for which $[\![T_E]\!]_{D_M} = [\![E]\!]$. Moreover, $[\![T]\!]_{D_M} = [\![T]\!]_D$ for all type variables occurring in D (hence for those occurring in M). If D is proper then D_M is proper.

So without lack of generality we can assume that $match(q, T)$ returns a set of mappings M such that $m(X)$ is a type name, for each $m \in M$ and for each variable X occurring in q.

The complexity of the procedure $match(q, T)$ is bad for query terms of a certain structure. The worst case is when a query term q is of the form $l\{\{q_1 \cdots q_n\}\}$ and the number of unrooted query terms among $q_1, \ldots, q_n$ is big. In this case the practical usage of the algorithm may be impossible. The complexity increases also when the number of the children of q or the number of different types occurring in the regular expression for T grows.

Example 2.6.4 *Consider a type definition* $D = \{T \to l\{T_1\, T_2^*\}, T_1 \to a[\#], T_2 \to b[\#]\}$ *and a query term* $q = l\{\{X \rightsquigarrow b["s"]\ Y\}\}$. *We execute* $match(q, T)$. *In the first run of the procedure we call function* $restrict\_language((T_1|T_2)(T_1|T_2), \{\{\}\}, T)$ *which returns a language* $L = \{T_1T_1, T_1T_2, T_2T_1, T_2T_2\}$. *Then for each element of the language* L *we call match for relevant query terms and types:*

- $match(X \rightsquigarrow b["s"], T_1)$ *and receive* $\emptyset$
- $match(Y, T_1)$ *and obtain* $\{[Y \to T_1]\}$
- $match(X \rightsquigarrow b["s"], T_2)$ *and obtain* $\{[X \to T_2]\}$
- $match(Y, T_2)$ *and obtain* $\{[Y \to T_2]\}$

Now we consider only two elements of L *(*T_2T_1 *and* T_2T_2*) for which we get not empty mappings. As a result for* $match(q, T)$ *we get a set of mappings* $\{[X \to T_2, Y \to T_1], [X \to T_2, Y \to T_2]\}$. *The received result is not exact. The mappings show that* X *may be bound to data terms of type* T_2. *In fact* X *can be bound only to such data terms of* T_2 *which have* $"s"$ *inside.*

2.6.4 Computing Better Approximations of Set of Answers for a Query Term

The previous section describes an algorithm which computes an approximation of the set of answers for a query term. Here we provide an algorithm computing more precise approximations of such sets. In the previous algorithm we only check if a query term matches data terms of a given type. To have more exact results for a query $X \rightsquigarrow q'$ we must know which data terms of a given type are matched by the query q', and describe the set of such terms by a type definition. The computed set of answers is exact in a case when a query term does not contain multiple occurrences of a variable. The sets of answers for such query terms are not expressible by regular sets. The algorithm is much more complex than the previous one as it requires constructing new types which are subsets of the types defined by a given type definition.

As before we start the presentation from an auxiliary procedure $restrict\_language_E$. Its input arguments are: a regular expression r, a number of child query terms n and parentheses $\alpha\beta$. Below we assume that $T_1, \ldots, T_m$ are the type names occurring in r. The procedure returns a regular language L' over the alphabet $\{T_1, \ldots, T_m, U_{11}, \ldots, U_{n1}, \ldots, U_{1m}, \ldots, U_{nm}\}$. Each symbol U_{ij} will represent the set of data terms from the type $[\![T_j]\!]$ matched by a query q_i.

Let h be a homomorphism such that $h(U_{ij}) = T_j$ and $h(T_i) = T_i$. The language L' is the biggest set satisfying the following conditions:

- $h(L') \subseteq L(r)$

- for each $w \in L'$ and for each $i = 1, \ldots, n$ there is exactly one symbol U_{ij} in w

-
  - if $\alpha\beta = [\,]$ then
    L' contains only words of the form $U_{1j_1} \cdots U_{nj_n}$ ($j_i \in \{1, \ldots, m\}$),
  - if $\alpha\beta = \{\,\}$ then
    L' contains only permutations of words of the form $U_{1j_1} \cdots U_{nj_n}$ ($j_i \in \{1, \ldots, m\}$),
  - if $\alpha\beta = [[\,]]$ then
    every word of L' has a subsequence of the form $U_{1j_1} \cdots U_{nj_n}$,
  - if $\alpha\beta = \{\{\}\}$ then
    every word of L' has a sub-sequence which is a permutation of a word of the form $U_{1j_1} \cdots U_{nj_n}$.

$restrict\_language_E(r, n, \alpha\beta)$:
let r_{All} be the regular expression $T_1|\ldots|T_m$ where $\{T_1, \ldots, T_m\} = types(r)$
let r' be the regular expression r with every type name T_i replaced by $T_i|U_{1i}|\ldots|U_{ni}$
let s be the regular expression $(U_{11}|\ldots|U_{1m}) \ldots (U_{n1}|\ldots|U_{nm})$
let s' be the regular expression $r^*_{All}(U_{11}|\ldots|U_{1m})r^*_{All} \ldots r^*_{All}(U_{n1}|\ldots|U_{nm})r^*_{All}$
IF $\alpha\beta = [\,]$ THEN return $L(r') \cap L(s)$
IF $\alpha\beta = [[\,]]$ THEN return $L(r') \cap L(s')$
IF $\alpha\beta = \{\}$ THEN return $L(r') \cap perm(L(s))$
IF $\alpha\beta = \{\{\}\}$ THEN return $L(r') \cap perm(L(s'))$

The most complex operation employed by the algorithm above is a computation of a language $L(r') \cap perm(L(s'))$. The intersection of regular languages can be computed in a standard way by construction of a product automaton. Both languages $L(r')$ and $perm(L(s'))$ are regular. An automaton for $L(r')$ can be obtained in a standard way; notice that we may use an NFA, avoiding expensive construction of a DFA. We describe how to build a DFA for the language $perm(L(s'))$. We use here the fact that the regular expression s' is of a special form: $r^*_{All}(U_{11}|\ldots|U_{1m})r^*_{All} \ldots r^*_{All}(U_{n1}|\ldots|U_{nm})r^*_{All}$. The states of the automaton defining $perm(L(s'))$ are subsets of $\{1, \ldots, n\}$, there is also a garbage state *error*. The automaton is in a state $S \subseteq \{1, \ldots, n\}$ when S is the set of indices i of those symbols U_{ij} that have been already read. The initial state of the automaton is $\emptyset$ and the final state is $\{1, \ldots, n\}$. If a symbol U_{ij} is read in a state S then:

- if $i \in S$ then the next state is *error*,
- otherwise we move to state $S \cup \{i\}$.

If a symbol from $\{T_1, \ldots, T_n\}$ is read, the state is not changed.

The function $match_E(q, T)$ presented below returns a set of pairs (m_k, U_k), where m_k is a mapping from variables (occurring in q) to types and $[\![U_k]\!]$ is a set of data terms from $[\![T]\!]$ which are matched by q, resulting in answers from $substitutions(m_k)$. Let q be of the form $l\alpha q_1 \cdots q_n\beta$, the rule for T be of the form $T \to l\alpha' r\beta'$ and $T_1, \ldots, T_m$ be the type names occurring in r. For every (m_k, U_k) returned by $match(q, T)$, the rule for U_k will be of the form $U_k \to l\alpha' r'\beta'$ where r' is a regular expression over the alphabet $\{T_1, \ldots, T_m, U_{11}, \ldots, U_{nm}\}$. Each U_{ij} is a type name denoting a set of those data terms of type T_j which are matched by q_i.

Every string of $L(r')$ is obtained from a string from $L(r)$ by replacing some occurrences of type names T_j by U_{ij}. The function $restrict\_language_E$ suggests which query terms should be matched against which type names. As there are many possible such associations on each level of query term q, $match_E(q,T)$ returns not a single pair (m_k, U_k) but a set of such pairs. The union of all such types U_k is the set of those data terms from $[\![T]\!]$ which are matched by q.

For the algorithm below we assume that there exists a type definition D where the type T is defined. During the execution of the algorithm new types are being created and the type definition D is being extended with rules defining the new types. We assume that procedure $define(U \rightarrow ...)$ adds a rule $U \rightarrow ...$ to the type definition, and that U is a new type name, not occurring elsewhere.

$match_E(q,T)$:
  IF q is a variable X THEN
    return $\{([X \mapsto T], T)\}$
  IF q is of the form $X \leadsto q'$ THEN
    return $\{(m_i \cap [X \mapsto U_i], U_i) \mid (m_i, U_i) \in match_E(q', T)\}$
  IF q is of the form $desc\, q'$ THEN
    IF T is a type constant or a special type name THEN
      return $match_E(q', T)$
    IF the rule for T is of the form $T \rightarrow l[r]$
      let $\{T_1, \ldots, T_m\} = types(r)$
      let r' be a regular expression such that $L(r') = restrict\_language_E(r, 1, \{\{\}\})$
      return $match_E(q', T) \cup \{\, (m_j, U) \mid j \in \{1, \ldots, m\},\ (m_j, T_{1j}) \in match_E(q, T_j),$
        $define(U \rightarrow l\alpha' r'' \beta')$, where r'' is r' with the type name U_{1j} replaced by T_{1j},
        and each U_{1k} where $k \neq j$ replaced by ϕ }
    IF the rule for T is of the form $T \rightarrow l\{r\}$
      let r be a multiplicity list $T_1(l_1 : u_1) \cdots T_m(l_m : u_m)$
      (we may assume that $u_j > 0$, for $j = 1, \ldots, m$)
      return $match_E(q', T) \cup$
        $\{\, (m_j, U) \mid j \in \{1, \ldots, m\},\ (m_j, U_j) \in match_E(q, T_j),$
        $define(U \rightarrow l\{r''\})$, where r'' is a multiplicity list
        $U_j\, T_1(l_1 : u_1) \cdots T_j(max(l_j - 1, 0) : u_j - 1) \cdots T_m(l_m : u_m)\,\}$
  (Now q is a rooted query term or a basic constant).
  IF $root(q) \neq label(T)$ THEN return $\emptyset$
  IF T is a type constant or a special type name THEN
    IF c is a basic constant in $[\![T]\!]$ THEN
      $define(T' \rightarrow c)$
      return $\{(\top, T')\}$
    ELSE return $\emptyset$

  let $q = l\alpha q_1 \cdots q_n \beta$ $(n \geq 0)$,
  IF the rule for T is of the form $T \rightarrow l[r]$ THEN
    let $\{T_1, \ldots, T_m\} = types(r)$
    let r' be a regular expression such that $L(r') = restrict\_language_E(r, n, \alpha\beta)$
    return $\{\, (m_{1j_1} \cap \ldots \cap m_{nj_n}, U) \mid$ for each $i = 1, \ldots, n,\ j_i \in \{1, \ldots, m\},$
      $(m_{ij_i}, T_{ij_i}) \in match_E(q_i, T_{j_i}),$
      $define(U \rightarrow l[r''])$, where r'' is r' with
        the type names $U_{1j_1}, \ldots, U_{nj_n}$ replaced with $T_{1j_1}, \ldots, T_{nj_n}$, and
        all type names U_{kl} $(k = 1, \ldots, n,\ l = 1 \ldots m)$ other than $U_{1j_1}, \ldots, U_{nj_n}$
        replaced with ϕ }

IF the rule for T is of the form $T \to l\{r\}$ THEN
IF $\alpha\beta = [\,]$ or $\alpha\beta = [[\,]]$ THEN
return $\emptyset$
IF $\alpha\beta = \{\}$ THEN
return $\{\, (m_{1j_1} \cap \ldots \cap m_{nj_n}, U) \mid$ for each $i = 1, \ldots, n,\ j_i \in \{1, \ldots, m\}$,
$(m_{ij_i}, U_{ij_i}) \in match_E(q_i, T_{j_i})$, $T_{j_1} \ldots T_{j_n} \in perm(L(r))$,
$define(U \to l\{U_{1j_1} \ldots U_{nj_n}\})\}$
IF $\alpha\beta = \{\{\}\}$ THEN
let r be a multiplicity list $T_1(l_1 : u_1) \ldots T_m(l_m : u_m)$
return $\{\, (m_{1j_1} \cap \ldots \cap m_{nj_n}, U) \mid$ for each $i = 1, \ldots, n,\ j_i \in \{1, \ldots, m\}$,
$(m_{ij_i}, U_{ij_i}) \in match_E(q_i, T_{j_i})$,
y_j (for $j = 1, \ldots, m$) is the number of occurrences of j in $j_1 \ldots j_n$, and $y_j \leq u_j$,
$define(U \to l\{r''\})$, where r'' is a multiplicity list
$U_{1j_1} \ldots U_{nj_n} T_1(max(l_1 - y_1, 0) : u_1 - y_1) \ldots T_m(max(l_m - y_m, 0) : u_m - y_m)\,\}$

The algorithm has the following property.

Proposition 2.6.2 *If θ is an answer for a query term q and a data term $d \in [\![T]\!]$ then there exists $(m, T') \in match_E(q, T)$ such that $\theta \in substitutions_D(m)$ and $d \in [\![T']\!]$.*

The reverse implication holds if q does not contain multiple occurrences of a variable.

In general the algorithm may produce a huge number of new types and that is why its usage is limited to queries of a certain structure. We are currently investigating the possibility to make it more efficient at the cost of producing less accurate answers. We want to obtain an algorithm that gives better approximations of answers than the previous algorithm and can be practically used.

Example 2.6.5 *Recall example 2.6.4. We considered there a type definition $D = \{\, T \to l\{T_1\, T_2^*\},\ T_1 \to a[\#],\ T_2 \to b[\#]\,\}$ and a query term $q = l\{\{X \rightsquigarrow b[\text{"}s\text{"}]\ Y\}\}$. We execute $match_E(q, T)$. In the first run of the procedure we try to match query terms $q_1 = X \rightsquigarrow b[\text{"}s\text{"}]$ and $q_2 = Y$ with all types that occur in the multiplicity list $T_1 T_2^*$ (it is an abbreviation for $T_1(1:1)\, T_2(0:\infty)$). As a result of matching a query term q_i with a type T_j we obtain type U_{ij}. In the following steps we call*

- *$match_E(q_1, T_1)$ and receive $\emptyset$ (this implies $[\![U_{11}]\!] = \emptyset$),*
- *$match_E(q_2, T_1)$, and obtain $\{([Y \mapsto T_1], T_1)\}$, which means $U_{21} = T_1$,*
- *$match_E(q_1, T_2)$ and obtain $\{([X \mapsto U_{12}], U_{12})\}$ with new rules $\{U_{12} \to b[\#_1],\ \#_1 \to \text{"}s\text{"}\}$ added to D,*
- *$match_E(q_2, T_2)$ and obtain $\{([Y \mapsto T_2], T_2)\}$, which means $U_{22} = T_2$.*

Now we consider all possible associations of query terms q_1, q_2 with types T_1, T_2; this means considering the set of pairs $\{\, U_{11}, U_{21};\ U_{11}, U_{22};\ U_{12}, U_{21};\ U_{12}, U_{22}\,\}$. The pairs containing U_{11} can be skipped, as $[\![U_{11}]\!] = \emptyset$. For each remaining pair we construct a new type (adding new rules to D):

- *For the pair U_{12}, U_{21}, $y_1 = 1$, $y_2 = 1$ and we construct a new multiplicity list $r'' = U_{12} U_{21} T_1(0:0)\, T_2(0:\infty)$. As $U_{21} = T_1$, we replace U_{21} by T_1 in r'' and eventually obtain a rule $U_1'' \to l\{T_1 U_{12} T_2^*\}$.*

- *For U_{12}, U_{22} we have $y_1 = 0,\ y_2 = 2$, and $r'' = U_{12}U_{22}T_1(1:1)\,T_2(0:\infty)$. As $U_{22} = T_2$, we replace U_{22} by T_2 obtaining $r'' = U_{12}T_2T_1(1:1)\,T_2(0:\infty) = U_{12}T_1\,T_2^+$. Eventually we obtain a rule $U_2'' \rightarrow l\{T_1U_{12}T_2^+\}$.*

As a result for $match_E(q,T)$ we get a set of pairs $\{\,([X{\rightarrow}U_{12}, Y{\rightarrow}T_1], U_1''), ([X{\rightarrow}U_{12}, Y{\rightarrow}T_2], U_2'')\,\}$ and a new type definition $D' = D \cup \{\, U_{12} \rightarrow b[\#_1],\ \#_1 \rightarrow "s",\ U_1'' \rightarrow l\{T_1U_{12}T_2^*\},\ U_2'' \rightarrow l\{T_1U_{12}T_2^+\}\,\}$.

2.6.5 Computing the Type of Query Results.

In what follows we consider the first version of the algorithm computing the answers for a query term (cf. Section 2.6.3).

Given a proper type definition D and a set $match(q,T)$ of mappings describing answers to a query term q, the set of results for a query $t \leftarrow q$ and data terms from $[\![T]\!]_D$ is a subset of

$$R = \bigcup_{m \in match(q,T)} R(m) \qquad \text{where } R(m) = \{\, t\theta \mid \theta \in substitutions_D(m)\,\}$$

(by Proposition 2.6.1). If q does not contain $\rightsquigarrow$ and there is only one occurrence of each variable in t then R is the set of results.[1]

We first show how to compute $R(m)$. We construct a type definition with a type name T_u for each subterm u of the query head t. If u is a variable X then T_X is $m(X)$. The type names T_u corresponding to the basic constants occurring in t are new distinct special type names. For the remaining subterms of t the corresponding variables of t are new distinct type variables. We construct a set of rules

$$\begin{array}{rl} rules(t,m) = & \{\, T_c \rightarrow c \ \mid\ c \text{ is a basic constant and a subterm of } t\,\} \\ \cup & \{\, T_u \rightarrow l\alpha T_{u_1} \cdots T_{u_n}\beta \mid u = l\alpha u_1 \cdots u_n\beta \text{ is a subterm of } t\,\}. \end{array}$$

Type definition $D_m = D \cup rules(t,m)$ describes $R(m)$:

$$[\![T_t]\!]_{D_m} = \{\, t\theta \mid \theta \in substitutions(m)\,\}$$

Let us find out whether D_m is proper. For each subterm u of t consider a corresponding label. If u is a variable then the corresponding label is $label_D(T_u) = label_D(m(u))$. Otherwise it is $root(u)$. The type definition D_m is proper iff for each subterm u of t the labels corresponding to distinct child subterms of u are distinct. (Repeated occurrences of the same child subterm are allowed.)

Computing $rules(t,m)$ for each $m \in match(q,T)$ completes our algorithm. The union R of the sets $[\![T_t]\!]_{D \cup rules(t,m)}$ contains all the results for query $t \leftarrow q$ and any database which is a data term (or a set of data terms) from $[\![T]\!]$.

If t is a variable then R may contain data terms with distinct roots; such a set is not a type in our sense (i.e. is not $[\![T]\!]_D$ for any type definition D). If t is not a variable then one may express R by a single type definition, possibly non proper. Assume that for no type name there exist rules in two distinct sets $rules(t,m)$ (in other words, all the newly introduced type names are distinct). If t is a basic constant then R is defined by an obvious definition $\{T_R \rightarrow t\}$ or $\emptyset$.

[1] If there is a multiple occurrence of a variable in a construct term t it means that in the query result each occurrence of the variable must be replaced with the same value. It is not sufficient for the values to be of the same type. Such a set of results is not regular.

So assume that t is a term $l\alpha t_1 \cdots t_n \beta$. For each $m \in match(q, T)$, let T_t^m be the type variable corresponding to t in $rules(t, m)$ and let r_m be the regular expression in the rule for T_t^m. Let T_R be a new type variable and r be the union of the regular expressions r_m, for $m \in match(q, T)$. For the type definition

$$D' = \{ T_R \to l\alpha r\beta \} \cup D \cup \bigcup_{m \in match(q,T)} rules(t, m)$$

we have $R = [\![T_R]\!]_{D'}$.

In general, the type definition D' is not proper. It may be impossible to describe R by a proper type definition. Instead one may consider constructing a proper type definition defining a superset of the given set. This topic is however outside of the scope of this chapter.

Example 2.6.6 *Consider the type Cd from Example 2.3.1 and the construct-query rule Q from Example 2.6.3. We want to use the algorithm of Section 2.6.3 to obtain the type of the results for construct-query rule Q and a database from* $[\![Cd]\!]$. *First, we call* $match(cd\{\{\, TITLE\ ARTIST \rightsquigarrow artist\{\{\}\}\ \text{"rock"}\}\}, Cd)$ *which results in a set of mappings* $\{\, [TITLE \to Artist, ARTIST \to Artist], [TITLE \to Title, ARTIST \to Artist]\,\}$. *Then, for each obtained mapping we construct a new type definition as described above. Finally, we get two type definitions:*

$$D' = D \cup \{Result \to result[Name\ Author],\ Name \to name[Artist],\ Author \to author[Artist]\},$$
$$D'' = D \cup \{Result \to result[Name\ Author],\ Name \to name[Title],\ Author \to author[Artist]\}.$$

Thus every query result is a member of $[\![Result]\!]_{D'} \cup [\![Result]\!]_{D''}$. *The latter set is equal to that described by the proper type definition of Example 2.6.3.*

2.6.6 Analysis of Xcerpt Programs.

It is easy to generalize the method presented in this section to query rules containing more than one query term. The method applies also to Xcerpt programs containing many query rules, provided they are not recursive and the constructed type definitions are proper. If a query term q from a rule R_2 is matched against the results of a query rule R_1 then the algorithm applied to R_1 gives a type definition which is an input to the algorithm applied to R_2. The algorithm requires that the type definition is proper. It is however sufficient that each D_m, treated separately, is proper (for a condition under which this happens, see above). The algorithm for R_2 can be executed repetitively, for each D_m as an input. Each run of the algorithm produces some description of a result set, the union of these sets is the set of results of query rule R_2.

Applying this idea to a recursive set of rules may result in a non terminating sequence of applications of the algorithm. Here one needs an approach similar to abstract interpretation or set constraints solving. (For related work in the area of logic programming see e.g. [DMP02] and references therein.) However our approach can still be used to check correctness of recursive sets of rules w.r.t. type specifications. Consider a set P of Xcerpt rules and a specification S describing a set of allowed database terms and sets of allowed query results. A sufficient condition for correctness of P w.r.t. S is that each rule of P applied to allowed data terms produces an allowed result. This is an inductive proof method, similar to those used for partial correctness of programs. (For such a method for logic programs see [DM01] and references therein.) If specification S is given by a proper type definition then the sufficient condition can be checked by means of algorithms described in this paper. For each rule of P one can compute the set of results, using the algorithm described above; it is not necessary that the obtained type definition is proper. Then, using the algorithm of Section 2.5.3, one can check if the computed set is included in that given by S.

Example 2.6.7 *Consider Example 2.6.6 and assume that, according to the specification, the set of allowed query results is $[\![Result]\!]_{D''}$. The set of query results $R = [\![Result]\!]_{D'} \cup [\![Result]\!]_{D''}$ obtained in Example 2.6.6 is not a subset of $[\![Result]\!]_{D''}$. (The algorithm of Section 2.5.3 shows that $[\![Result]\!]_{D'} \not\subseteq [\![Result]\!]_{D''}$.) So the sufficient condition for correctness described above is not satisfied. Actually, any member of R is an answer to the considered query, so the query is indeed incorrect w.r.t. the given specification.*

2.7 Future work

We need a better understanding of the complexity of the presented algorithms from the practical point of view. We mean here application of descriptive typing to locating errors in rule programs. The main questions is, in which cases the computational costs become unacceptable. The related question is, in which cases we need the better accuracy provided by the second, less efficient version of *match*. We need both theoretical considerations and practical experiments, which would lead to a design of a practical algorithm, which would be a useful compromise between accuracy and efficiency. For the experiments we need a prototype implementation of the more precise algorithm.

Another subject is to study how much the restriction to proper type definitions may be relaxed. (A slightly more general class of definitions is considered in [BDM04].) Also, we should learn how serious the restriction is from the practical point of view.

The presented work should be generalized to the omitted features of Xcerpt. Some of them, e.g. label variables or optional subterms, seem to be easy to be dealt with. Dealing with others, e.g. negation or terms representing graphs, may be more problematic and will be an object of our investigation.

An interesting topic is comparison between the descriptive typing approach of this section and the prescriptive approach of Section 4. The comparison may possibly be formalized by describing (what is computed by) our algorithms by means of rules similar to those used in prescriptive type systems.

2.8 Conclusions

We introduced an abstraction of XML data by data terms and a formalism of type definitions to specify sets of data terms. To simplify our algorithms, we restrict this formalism to proper type definitions. The restriction seems acceptable, as the sets defined by main XML schema languages (DTD and XML Schema) can be expressed by proper type definitions. (Here we neglect some special features of these languages, like non context-free conditions of DTD on uniqueness of identifiers). Our algorithms are more efficient when the regular expressions in the type definitions are 1-unambiguous in a sense of [BKW98]. Restriction to such regular expressions seems not unnatural; for instance the regular expressions in DTD are required to be 1-unambiguous.

The main contribution of this work is an algorithm for computing (approximations of) the sets of results of Xcerpt rules, given (approximations of) the sets of databases. This makes it possible to prove correctness of Xcerpt programs w.r.t. specifications expressed by type definitions, and to compute approximations of the sets of results of non recursive Xcerpt programs. The algorithm is presented in two versions, one giving more precise results, the other one more efficient.

We have chosen Xcerpt as an example rule language; we expect that the ideas of this paper are applicable to other rule languages used in web applications.

The work on a prototype implementation of the algorithms is in progress. The less precise algorithm has been implemented as an additional module in Xcerpt prototype. We plan to implement the other version of the algorithm and also to extend the prototype to be able to handle all constructs used in Xcerpt.

3 Prescriptive type inference for rewrite-based languages

3.1 Introduction

A promising line of research unifying the logic paradigm with the functional paradigm is that of *rewrite-based* languages [MOM02] (Elan [The04d], Maude [The04c], ASF+SDF [The04a], OBJ* [FN96, Gog04], ...). Although these languages are less used than object-oriented languages (Java [Sun04], C# [Mic04], ...), they can also serve as (formal) common intermediate languages for implementing compilers for rewrite-based, functional, object-oriented, logic, and other "high-level" modern languages.

One of the main advantages of the rewrite-based languages is *pattern-matching* which allows one to discriminate between alternatives. Each pattern is associated with an action; once an instance of a pattern is recognized, the corresponding term is rewritten to a new one. Another advantage of rewrite-based languages (in contrast to ML or Haskell) is the ability to handle non-determinism by means of a collection of results: pattern-matching needs not to be exclusive, *i.e.* multiple branches can be taken simultaneously. An empty collection of results represents a matching failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection.

Useful applications of pattern-matching lie in the field of pattern recognition, and strings/trees manipulation. It has also been widely used in functional and logic programming, for instance in ML [MTHM97, The03a], Haskell [The04b], Scheme [The04e], or Prolog [The03b]. However, in all these applications, pattern-matching is considered as a convenient mechanism for expressing complex requirements about the function's argument, rather than a basis for an *ad hoc* paradigm of computation; we argue that the computational behavior of a calculus can be deeply influenced by the presence of pattern-matching.

3.1.1 The Rewriting-calculus

One of the most commonly used models of computation, the Lambda-calculus, uses only trivial pattern-matching. This calculus has recently been extended, initially for programming concerns, either by introducing patterns in Lambda-calculi [Pey87, vO90], or by introducing matching and rewrite rules in functional languages. More concerned with extending logics, Stehr has studied a Calculus of Constructions enhanced with rewriting logic [Ste02].

The *Rewriting-calculus* [CKL01b, CLW04] is a foundational framework integrating matching, rewriting and functions in a uniform way. Its abstraction mechanism is based on the rewrite rule formation: in a term of the form $P \twoheadrightarrow A$, one abstracts over the pattern P.

If an abstraction $P \twoheadrightarrow A$ is applied to the term B, then the evaluation mechanism is based on the instantiation (in A) of the free-variables present in P with the appropriate subterms of B. Indeed, this instantiation is achieved by matching P against B. One of the advantages of matching is that it can be customized with elaborated theories, in particular equational ones.

As a foundational calculus, the Rewriting-calculus is a non-trivial generalization of the Lambda-calculus, since we get the Lambda-calculus back if every pattern P is a variable. The rewrite relation generated by a set of rewrite rules, i.e. what is usually called "term rewriting" can also be conveniently modelled in the Rewriting-calculus [CLW04]. In particular, the notions of *rule application* and *result* (basic ingredients of Term Rewriting Systems) become explicit.

In the Rewriting-calculus, a rewrite rule is a first-class citizen, which can be created, manipulated and modified by the calculus itself. The abilities to manipulate rules and to define evaluation strategies represent the basic methods in rewrite-based languages. These strategies can be *implicit* as in ASF+SDF [vDHK96], *local* as in OBJ* and Maude, or *user defined* as in Elan and Maude [CM02]. Previous papers [Cir00, CKL01a, CLW04] showed that the Rewriting-calculus can be used as a core engine calculus for rewrite-based languages such as Elan and Maude.

3.1.2 A foundational framework for Web Reasoning

What makes the Rewriting-calculus appealing for reasoning on the web is precisely its foundational features that allow us to represent the atomic actions (i.e. rules) and the chaining of these actions (i.e. what we called above strategies) in order to achieve a global goal like, for example, transforming semi-structured data, extracting informations or inferring new ones. As the matching mechanism of the calculus is parameterized by a theory, this allows us for example to express in a precise way how the semi-structured data should be matched.

To exemplify these features, let us consider again the simple but useful example 2.6.2 on page 13.

In this example, the data base considered is the term

$$\begin{array}{rll} DB = & catalogue[& cd[\,title[\text{"Empire Burlesque"}]\,artist[\text{"Bob Dylan"}]\,year[\text{"1985"}]\,] \\ & & cd[\,title[\text{"Hide your heart"}]\,artist[\text{"Bonnie Tyler"}]\,year[\text{"1988"}]\,] \\ & & cd[\,title[\text{"Stop"}]\,artist[\text{"Sam Brown"}]\,year[\text{"1988"}]\,]\,] \end{array}$$

A rule extracting titles and artists for the CD's issued in 1988 and presenting the results in a changed form (title as name and artist as author) is expresses, using an Xcerpt like syntax where $TITLE$ and $ARTIST$ are variables, by

$$\begin{array}{l} result[\,name[TITLE]\;author[ARTIST]\,] \;\leftarrow \\ \qquad catalogue\{\{\;cd\{\,title[TITLE]\;artist[ARTIST]\;year[\text{"1988"}]\,\}\}\} \end{array}$$

Notice in particular that in this example, the syntax used in the rule states that matching should be done using an associative and commutative like equational theory.

In the Rewriting calculus, the database can be seen as an algebraic ρ-term:

$$\begin{array}{rll} DBrho = & catalogue(& cd(title(\text{"Empire Burlesque"}), artist(\text{"Bob Dylan"}), year(\text{"1985"})), \\ & & cd(title(\text{"Hide your heart"}), artist(\text{"Bonnie Tyler"}), year(\text{"1988"})), \\ & & cd(title(\text{"Stop"}), artist(\text{"Sam Brown"}), year(\text{"1988"}))) \end{array}$$

and the request is expressed by the following expression, called a ρ-term (rewrite rule):

$$\begin{array}{rl} cd88Rule = & catalogue_{\{\}}(cd_{\{\}}(title(TITLE), artist(ARTIST), year(\text{"1988"})),\;\; SubCat) \\ & \twoheadrightarrow \\ & result(name(TITLE), author(ARTIST)) \end{array}$$

The $catalogue_{\{\}}$ and $cd_{\{\}}$ symbols indicate that the matching against the corresponding symbols in the database are done modulo the equational theory that we denote on purpose $\mathcal{TH}(\{\})$ and which roughly corresponds to an associative and commutative theory with neutral element. The $SubCal$ variable is an *extension variable*, classical in associative and commutative rewriting [PS81, JK86, KK99]. When no subscript is used a syntactic matching is used.

As we formally detail it below, the evaluation rules of the calculus yield as result for the application of the request to the database, i.e. for the ρ-term $(cd88Rule\ DBrho)$, the ρ-term:

$$\begin{array}{l} result(name(\text{"Hide your heart"}), author(\text{"Bonnie Tyler"})) \\ ; \\ result(name(\text{"Stop"}), author(\text{"Sam Brown"})) \end{array}$$

where the " ;" operator states that a "list" of elementary result is obtained.

On this simple example, the Rewriting calculus allows us to express in a fully explicit way all the components that should be acting to solve a query: the rules, the data base, the matching theory in use. We can also see that the result is a first class entity of the calculus. This allows us in particular to chain application of rules as can be shown on more elaborated examples.

If we consider an associative and commutative ; operator and we apply the following ρ-term

$$\begin{array}{rl} authorsRule = & (X \,;\, L \\ & \twoheadrightarrow \\ & (result(name(TITLE), author(ARTIST)) \twoheadrightarrow singer(ARTIST))\ X) \end{array}$$

to the previously obtained result then we obtain as result the term

$$\begin{array}{l} singer(\text{"Bonnie Tyler"}) \\ ; \\ singer(\text{"Sam Brown"}) \end{array}$$

The variable X can be instantiated by the matching algorithm either by a term of the form $result(\ldots)$ or by a structure of this kind of terms. In the former case, the rule in the right hand side can be applied successfully and the result is the corresponding singer. For the latter case, the matching fails and the failure is eliminated as shown in the next sections. Since the matching algorithm yields several solutions, the final result is a structure corresponding to all the possible instantiations of X.

As for many other frameworks, what is open today is to understand whether the Rewriting calculus is able to model the main features of languages like Xcerpt. This needs in particular to make explicit the matching theory used at run time in such languages, see for example [DPR98, Sch04].

As detailed in [CCD$^+$04], rewrite based languages have been equipped with various prescriptive type systems. We study here a powerful such polymorphic one.

3.1.3 Typed Rewriting calculi

Static analysis via a type system (inherited from the Lambda-calculus) enforces a safer programming discipline. In [LW05], a Rho-calculus *à la* Church (called RhoF) featuring second-order polymorphic types was proposed, together with a corresponding type inference system *à la* Curry (uRhoF). A simple type inference was provided for RhoF, but only undecidability of typing was proved for uRhoF. In this work, we characterize the reasons for this undecidability, and we define a proper subset $\mathsf{uRhoF}_{\ll}$ with an inference algorithm.

In Subsection 2.2, we present the syntax of the calculus and its small-step semantics. In Subsection 2.3, we introduce the fully typed second-order rewriting calculus *à la* Church RhoF: types of the bound variables are specified in the term, making type reconstruction and verification quite straightforward. The calculus enjoys subject reduction, and type uniqueness. In Subsection 2.4, we present the calculus *à la* Curry uRhoF: type information is not given in the term, and the type system is not fully syntax-directed, thus enforcing a flexible polymorphic type discipline. The calculus enjoys subject reduction, but as it is well-known for the λ-calculus, type inference is undecidable. In Subsection 2.5, we give the type inference algorithms, and prove their correctness, completeness and principality.

Syntactic Cat.			**Abstract Syntax**
K	$\in$	$\mathcal{K}inds$	$K \ ::= \ *$
$\tau.\iota$	$\in$	$\mathcal{T}ype$	$\tau \ ::= \ \iota \mid \alpha \mid \tau \twoheadrightarrow \tau \mid \forall\alpha.\tau$
$\Gamma.\Delta$	$\in$	$\mathcal{C}ontext$	$\Delta \ ::= \ \emptyset \mid \Delta, \alpha{:}K \mid \Delta, f{:}\tau \mid \Delta, X{:}\tau$
$P.Q$	$\in$	$\mathcal{P}attern$	$P \ ::= \ \mathsf{stk} \mid \alpha \mid X \mid f(\overline{P})$ *(all vars occur only once in any P)*
$A.B.f$	$\in$	$\mathcal{T}erm$	$A \ ::= \ \mathsf{stk} \mid f \mid X \mid P \twoheadrightarrow_\Delta A \mid [P \ll_\Delta A]A \mid A\ A \mid A\ \tau \mid A\,;A$

Figure 1: Syntax of RhoF

3.2 The System RhoF

This section stems from [LW05]. We detail the syntax and the semantics of RhoF, and we give some examples.

3.2.1 Syntax

Notational Conventions. We consider the meta-symbols "_ $\twoheadrightarrow$ _" (function- and type-abstraction), and "[_ $\ll$ _]" (delayed matching constraint), and "_ ; _" (structure operator). The application operator is denoted by concatenation.

We assume that the application operator associates to the left, while the other operators associate to the right. The priority of the application is higher than that of "[$\ll$]" which is higher than that of "$\twoheadrightarrow$" which is, in turn, of higher priority than the " ;". The symbol τ ranges over the set $\mathcal{T}ype$ of types, the symbol ι ranges over the set $\mathcal{T}ype_K$ of type constants ($\mathcal{T}ype_K \subseteq \mathcal{T}ype$), the symbols α, β range over the set $\mathcal{T}ype_V$ of type-variables ($\mathcal{T}ype_V \subseteq \mathcal{T}ype$), the symbols $A, B, C, \ldots, U, V, W$ range over the set $\mathcal{T}erm$ of (un)typed terms, the symbols $X, Y, Z, \ldots$ range over the set $\mathcal{V}ar$ of term variables ($\mathcal{V}ar \subseteq \mathcal{T}erm$), the symbols $a, b, c, \ldots, f, g, h, \ldots$ range over a set $\mathcal{T}erm_K$ of term constants ($\mathcal{T}erm_K \subseteq \mathcal{T}erm$). The symbols P, Q range over the set $\mathcal{P}attern$ of patterns, ($\mathcal{V}ar \subseteq \mathcal{P}attern \subseteq \mathcal{T}erm$). The symbols θ, ϕ, ψ range over substitutions. Finally, the symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}$ range over $\mathcal{T}ype \cup \mathcal{T}erm$. We denote $\overline{A}$ for $A_1 \cdots A_n$, for $n \geq 0$. The application of a constant, say f, to a term A will be usually denoted by $f(A)$, following the algebraic folklore; this convention can be currified in order to denote a function taking multiple arguments, *e.g.* $f(\overline{A}) \triangleq f(A_1, \cdots, A_n) \triangleq f\, A_1 \cdots A_n$.

Syntax (Figure 1). The *types* are as one would expect from a polymorphic type system (*i.e.* type-variables can be bound in types through the $\forall$ binder). The *patterns* are algebraic terms (*i.e.* terms constructed only with variables, constants and applications) which can be used as left-hand sides of the rewriting rules; the set of patterns is obviously included in the set of terms. The well-known linearity restriction [vO90] is needed to keep the small-step semantics confluent. A typed *rewriting rule* of the form $P \twoheadrightarrow_\Delta A$ abstracting over the free-variables of P is a first-class citizen of the calculus. The context Δ records the type of the free-variables of P (bound in A). When a pattern is a variable, we write $X \twoheadrightarrow_\tau A$, instead of $X \twoheadrightarrow_{(X:\tau)} A$ (by a little abuse of notation). An *application* is implicitly denoted by concatenation; note that *"terms can be applied to types"*. The *delayed matching constraint* $[P \ll_\Delta A]B$ can be seen as the term B with its free-variables (declared in Δ) constrained by the matching between P and A. The symbol stk is the special constant representing all the delayed matching constraints whose matching problem is unsolvable. A *structure* is a collection of terms that can be seen either as a set of rewriting rules or as a set of results.

3.2.2 Free-Variables and Substitutions.

Definition 3.2.1 (Free-variables Fv)

$$
\begin{array}{lll@{\qquad}lll}
\mathsf{Fv}(f) & \triangleq & \emptyset & \mathsf{Fv}(P \twoheadrightarrow_\Delta A) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(\Delta) \setminus \mathsf{Fv}(P) \\
\mathsf{Fv}(\mathsf{stk}) & \triangleq & \emptyset & \mathsf{Fv}([P \ll_\Delta \mathcal{A}]B) & \triangleq & \mathsf{Fv}((P \twoheadrightarrow_\Delta B)\ \mathcal{A}) \\
\mathsf{Fv}(X) & \triangleq & \{X\} & \mathsf{Fv}(A\,; B/A\ B) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(B) \\
\mathsf{Fv}(\alpha) & \triangleq & \{\alpha\} & \mathsf{Fv}(A\ \tau) & \triangleq & \mathsf{Fv}(A) \cup \mathsf{Fv}(\tau) \\
 & & & \mathsf{Fv}(\tau_1 \twoheadrightarrow \tau_2) & \triangleq & \mathsf{Fv}(\tau_1) \cup \mathsf{Fv}(\tau_2)
\end{array}
$$

As usual, we work modulo *α-conversion* and we adopt Barendregt's *"hygiene-convention"* [Bar84], *i.e.* free- and bound-variables have different names. This allows us to define substitutions quite straightforwardly, since it avoids problems like variable capture.

Definition 3.2.2 (Substitutions)
A substitution θ is a mapping from the set of term variables (resp. type variables) to the set of terms (resp. types). A finite substitution θ has the form $\{A_1/X_1 \ldots A_m/X_m\}$, or $\{\tau_1/\alpha_1 \ldots \tau_m/\alpha_m\}$, and its domain $\mathsf{Dom}(\theta)$ denotes $\{X_1, \ldots, X_m\}$, resp. $\{\alpha_1, \ldots, \alpha_m\}$. The application of a substitution θ to a term A (resp. type τ), denoted by $A\theta$ (resp. $\tau\theta$), is defined as follows:

$$
\begin{array}{lll@{\qquad}lll}
f\theta & \triangleq & f & (P \twoheadrightarrow_\Delta A)\theta & \triangleq & P \twoheadrightarrow_\Delta A\theta \\
\mathsf{stk}\theta & \triangleq & \mathsf{stk} & ([P \ll_\Delta \mathcal{A}]B)\theta & \triangleq & [P \ll_\Delta \mathcal{A}\theta]B\theta \\
(A\ \tau)\theta & \triangleq & (A\theta)\ (\tau\theta) & (A\,; B/A\ B)\theta & \triangleq & A\theta\,; B\theta/A\theta\ B\theta \\
X_i\theta & \triangleq & \begin{cases} A_i & \text{if} X_i \in \mathsf{Dom}(\theta) \\ X_i & \text{otherwise} \end{cases} & \alpha_i\theta & \triangleq & \begin{cases} \tau_i & \text{if } \alpha_i \in \mathsf{Dom}(\theta) \\ \alpha_i & \text{otherwise} \end{cases} \\
\iota\theta & \triangleq & \iota & (\tau_1 \twoheadrightarrow \tau_2)\theta & \triangleq & \tau_1\theta \twoheadrightarrow \tau_2\theta
\end{array}
$$

3.2.3 Matching Equations, Theories and Term Approximations.

The core mechanism of the Rewriting-calculus is pattern-matching. When a delayed matching constraint is evaluated, then a corresponding matching problem has to be solved. We use a theory for the Rho-calculus (introduced in [CLW04]) that handles uniformly matching failures and eliminates them when not significant for the computation. We define rules for handling this kind of terms and we show how they are integrated in the calculus. The classical notions of matching equations and matching solutions are defined as usual.

Definition 3.2.3 (Matching)
Given a theory $\mathbb{T}$

1. *A matching equation is a problem $\mathsf{T} \triangleq P \prec\!\!\!\ll_{\mathbb{T}} A$ where P is a pattern and A is a term;*

2. *A substitution θ is a solution of the matching equation* T *if* $P\theta \stackrel{\mathbb{T}}{=} A$.

Different theories and the corresponding pattern-matching problems can be formally defined and solved, for example as explained in [CKL01a]. If the equation $P \prec\!\!\prec_{\mathbb{T}} A$ has a unique solution, we denote it by $\theta_{(P \prec\!\!\prec_{\mathbb{T}} A)}$.

We define a superposition relation $\sqsubseteq \; : \mathcal{P}attern \times \mathcal{T}erm$ between patterns and terms whose aim is to characterize a broad class of matching equations that are *potentially* solvable. If $P \sqsubseteq A$ we say that "*P does potentially superpose with A*" and, by negation, if $P \not\sqsubseteq A$ then "P surely does not superpose with A".

Definition 3.2.4 (Superposition)

1. *The relation of superposition $P \sqsubseteq A$ is defined according to the structure of P as follows:*

$$
\begin{array}{llll}
f & \sqsubseteq & f & \\
\mathsf{stk} & \sqsubseteq & \mathsf{stk} & \\
X & \sqsubseteq & A & (\forall A) \\
\alpha & \sqsubseteq & \tau & (\forall \tau)
\end{array}
\qquad
\begin{array}{l}
f(\overline{P}) \;\sqsubseteq\; A \;\; \text{if } A \equiv f(\overline{B}) \wedge \overline{P} \sqsubseteq \overline{B} \\
P \;\sqsubseteq\; A \;\; \text{if } A \equiv \begin{cases} X \vee (A_1\,;A_2) \vee A\;\tau\;\vee \\ (A_1\;A_2 \wedge A_1 \notin \mathcal{P}attern) \vee \\ ([Q \ll_\Delta A_1]A_2 \wedge Q \sqsubseteq A_1 \wedge P \sqsubseteq A_2)\;(\forall P) \end{cases}
\end{array}
$$

2. *If $P \sqsubseteq A$ is not satisfied we write $P \not\sqsubseteq A$.*

Starting from the superposition relation, we define a reduction relation that eliminates from a term all the definitively stuck subterms, *i.e.* all the delayed matching constraints whose matching problem is unsolvable independently of subsequent instantiations and reductions.

Definition 3.2.5 (Stuck Theory, $\mathbb{T}_{\mathsf{stk}}$)
The relation $\rightharpoonup_{\mathsf{stk}}$ is defined by the following rules:

$$
\begin{array}{rcll}
[P \ll_\Delta A]B & \rightharpoonup_{\mathsf{stk}} & \mathsf{stk} & \quad \text{if } P \not\sqsubseteq A \\
\mathsf{stk}\,;A & \rightharpoonup_{\mathsf{stk}} & A & \\
A\,;\mathsf{stk} & \rightharpoonup_{\mathsf{stk}} & A & \\
\mathsf{stk}\;A & \rightharpoonup_{\mathsf{stk}} & \mathsf{stk} &
\end{array}
$$

We denote by $\mapsto_{\mathsf{stk}}$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\rightarrow_{\mathsf{stk}}$. The symmetric and transitive closure of $\mapsto\!\!\!\rightarrow_{\mathsf{stk}}$ is denoted by $\stackrel{\mathsf{stk}}{=}$. Let $\mathbb{T}_{\mathsf{stk}}$ be the theory associated to the congruence $\stackrel{\mathsf{stk}}{=}$. Matching equations in the theory $\mathbb{T}_{\mathsf{stk}}$ are denoted $P \prec\!\!\prec_{\mathsf{stk}} A$.

As mentioned previously, these rules are used to propagate or eliminate the definitively stuck terms.

3.2.4 The Polymorphic Rewriting-calculus, RhoF

Figure 2 shows the reduction rules of RhoF parameterized by the theory $\mathbb{T}_{\mathsf{stk}}$ (recall the symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}$ range over $\mathcal{T}ype \cup \mathcal{T}erm$).

Let us quickly explain the top-level rules:

$$
\begin{array}{rcl}
(P \twoheadrightarrow_{\Delta} A)\ \mathcal{B} & \to_{\rho} & [P \ll_{\Delta} \mathcal{B}]A \\
[P \ll_{\Delta} \mathcal{B}]A & \to_{\sigma} & A\theta_{(P \prec\!\!\prec_{\mathsf{stk}} \mathcal{B})} \\
(A\,;B)\ C & \to_{\delta} & A\ C\,;B\ C \\
A & \to_{\mathsf{stk}} & B
\end{array}
$$

Figure 2: Top-level Rules of RhoF

(ρ) this rule triggers the application of an abstraction to a term, but does not immediately try to solve the associated matching equation.

(σ) this rule is applied if and only if the matching equation $P \prec\!\!\prec_{\mathsf{stk}} \mathcal{B}$ has at least one solution: in this case the matching solutions are computed and applied to the term A. If there is more than one match, a structure collecting all the different results is obtained when the rule is applied. If there is no solution, this rule does not apply and thus, the term that is on the left-hand side represents a matching failure. As we shall see, further reductions or instantiations are likely to modify $\mathcal{B}$ so that the equation has a solution and the rule can be triggered.

(δ) this rule distributes structures on the left-hand side of the application. This gives the possibility, for example, to apply in parallel two distinct pattern-abstractions A and B to a term C.

(stk) pushes into the operational semantics the rewriting rules that are particular to the theory adopted in the calculus; in our case the above defined $\mathbb{T}_{\mathsf{stk}}$-theory.

We denote by $\mapsto_{\rho\sigma\delta}$ the contextual closure induced by these rules. Its reflexive and transitive closure is denoted by $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$. The symmetric and transitive closure of $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ is denoted by $=_{\rho\sigma\delta}$. Notice that these relations are parameterized by the adopted theory $\mathbb{T}_{\mathsf{stk}}$. We denote by $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ the relation $\mapsto_{\mathsf{stk}} \cup \mapsto_{\rho\sigma\delta}$. For $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$, the following holds.

Theorem 3.2.1 (Church Rosser for RhoF [CLW04])
The relation $\mapsto^{\mathsf{stk}}_{\rho\sigma\delta}$ is confluent.

3.3 The Polymorphic Type System RhoF

Types can be used as predicates for terms of Rho-calculus. Terms can be directly *decorated* with types and then every closed term comes directly with a unique, intrinsic type. In the *fully typed* approach of [LW05], a type judgement will be denoted by the symbol $\vdash_{\mathsf{T}}$ (for Typed terms). A *typed system* is a set of rules for proving judgements of the shape $\Gamma \vdash_{\mathsf{T}} A : \tau$, where A is a typed term, τ is a type, and Γ is a context. The meaning of such a judgement is: the term A has type τ under the context Γ, and Γ records the types of the free-variables of Γ and τ. Figures 3, and 4 presents the kinding/typing rules of RhoF, which are directly inspired by the Girard System F [Gir86]. More precisely, the system proves judgement of the shape:

$$\Gamma \vdash_{\mathsf{T}} ok \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} \tau : * \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} P : \tau \quad \text{and} \quad \Gamma \vdash_{\mathsf{T}} A : \tau$$

We discuss only the typing rules for well-formed terms and patterns, the other typing rules being standard.

Well-formed Contexts

$$\frac{}{\emptyset \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Empty}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \mathit{ok} \quad \alpha \notin \mathsf{Dom}(\Gamma)}{\Gamma, \alpha{:}* \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Var}^{\forall})$$

$$\frac{\Gamma \vdash_{\mathsf{T}} \mathit{ok} \quad \iota \notin \mathsf{Dom}(\Gamma)}{\Gamma, \iota{:}* \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Const}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \mathit{ok} \quad \Gamma \vdash_{\mathsf{T}} \tau : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\tau \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Var})$$

Well-kinded Types

$$\frac{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{T}} \mathit{ok}}{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{T}} \iota : *} \quad (\mathit{Type\cdot Const}) \qquad \frac{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{T}} \mathit{ok}}{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{T}} \alpha : *} \quad (\mathit{Type\cdot Var})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{T}} \tau : *}{\Gamma \vdash_{\mathsf{T}} \forall \alpha.\tau : *} \quad (\mathit{Type\cdot Poly}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : *}{\Gamma \vdash_{\mathsf{T}} \tau_1 \twoheadrightarrow \tau_2 : *} \quad (\mathit{Type\cdot Arrow})$$

$$\frac{\Gamma \vdash_{\mathsf{T}} \mathit{ok} \quad \iota \notin \mathsf{Dom}(\Gamma)}{\Gamma, \iota{:}* \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Const}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \mathit{ok} \quad \Gamma \vdash_{\mathsf{T}} \tau : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\tau \vdash_{\mathsf{T}} \mathit{ok}} \quad (\mathit{Ctx\cdot Var})$$

Well-kinded Types

$$\frac{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{T}} \mathit{ok}}{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{T}} \iota : *} \quad (\mathit{Type\cdot Const}) \qquad \frac{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{T}} \mathit{ok}}{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{T}} \alpha : *} \quad (\mathit{Type\cdot Var})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{T}} \tau : *}{\Gamma \vdash_{\mathsf{T}} \forall \alpha.\tau : *} \quad (\mathit{Type\cdot Poly}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : *}{\Gamma \vdash_{\mathsf{T}} \tau_1 \twoheadrightarrow \tau_2 : *} \quad (\mathit{Type\cdot Arrow})$$

Figure 3: The Kind System for RhoF

- $(\mathit{Term\cdot Var})(\mathit{Term\cdot Const})$**:** As usual, the context determines the type of variables. It cannot contain two declarations for the same variable (or constant);
- $(\mathit{Term\cdot Stuck})$**:** Since stk can appear in any structure, its type can be virtually anything but ***falsum***, ***i.e.*** $\bot \triangleq \forall\alpha.\alpha$;
- $(\mathit{Term\cdot Abs}^{\twoheadrightarrow})$**:** For the left-hand side of the arrow-type, we use the type of the pattern P; this rule allows one to hide some type information in a pattern containing applications (*e.g.* τ_2 disappears in the final type of $f(X)$ in the judgement $f{:}\tau_2 \twoheadrightarrow \tau_1, X{:}\tau_2 \vdash f(X) : \tau_1$). The context Δ gives the types of the free-variables of P. The type system ensures that the solutions of the corresponding matching equations are well-typed;
- $(\mathit{Term\cdot Appl}^{\twoheadrightarrow})$**:** We directly exploit the information given in the type of the function, statically checking that the given argument has the expected type τ_1;

Well-formed Terms and Patterns

$$\frac{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} X : \tau} \ (Term{\cdot}Var) \qquad \frac{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} ok}{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{T}} f : \tau} \ (Term{\cdot}Const)$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{T}} A : \tau_1 \to \tau_2 \\ \Gamma \vdash_{\mathsf{T}} B : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{T}} A\ B : \tau_2} \ (Term{\cdot}Appl^{\to}) \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau_1 \quad \mathsf{Fv}(P) = \mathsf{Dom}(\Delta) \\ \Gamma, \Delta \vdash_{\mathsf{T}} A : \tau_2 \quad \Gamma, \Delta \vdash_{\mathsf{T}} \tau_1 \to \tau_2 : *\end{array}}{\Gamma \vdash_{\mathsf{T}} P \to_{\Delta} A : \tau_1 \to \tau_2} \ (Term{\cdot}Abs^{\to})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{T}} A : \tau}{\Gamma \vdash_{\mathsf{T}} \alpha \to_{*} A : \forall\alpha.\tau} \ (Term{\cdot}Abs^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{T}} A : \forall\alpha.\tau_1 \quad \Gamma \vdash_{\mathsf{T}} \tau_2 : *}{\Gamma \vdash_{\mathsf{T}} A\ \tau_2 : \tau_1\{\tau_2/\alpha\}} \ (Term{\cdot}Appl^{\forall})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{T}} A : \tau \\ \Gamma \vdash_{\mathsf{T}} B : \tau\end{array}}{\Gamma \vdash_{\mathsf{T}} A ; B : \tau} \ (Term{\cdot}Struct) \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau_1 \quad \mathsf{Fv}(P) = \mathsf{Dom}(\Delta) \\ \Gamma, \Delta \vdash_{\mathsf{T}} A : \tau_2 \quad \Gamma \vdash_{\mathsf{T}} B : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{T}} [P \ll_{\Delta} B]A : \tau_2} \ (Term{\cdot}Match^{\to})$$

$$\frac{\Gamma \vdash_{\mathsf{T}} \tau : * \quad \tau \neq \bot}{\Gamma \vdash_{\mathsf{T}} \mathsf{stk} : \tau} \ (Term{\cdot}Stuck) \qquad \frac{\Gamma \vdash_{\mathsf{T}} \tau_2 : * \quad \Gamma, \alpha{:}* \vdash_{\mathsf{T}} A : \tau_1}{\Gamma \vdash_{\mathsf{T}} [\alpha \ll_{*} \tau_2]A : \tau_1\{\tau_2/\alpha\}} \ (Term{\cdot}Match^{\forall})$$

Figure 4: The Type System for RhoF

- $(Term{\cdot}Abs^{\forall})$**:** The rationale is: $\alpha \to_{*} A \simeq \alpha \to_{(\alpha:*)} A$. Abstraction on type-variables makes the polymorphic mechanism available at the user-level: note that a trivial pattern is used in polymorphic-abstraction.
- $(Term{\cdot}Appl^{\forall})$**:** The rationale is: all free occurrences of α in τ_1 are substituted with τ_2. Any well-formed type τ_2 is suitable, which makes the typing fully polymorphic.
- $(Term{\cdot}Struct)$**:** This rule states that all the members of a structure have the same type. This is important when considering structures as a collection of results; if a function can return different results, then we would at least expect them to have the same type;
- $(Term{\cdot}Match^{\to})(Term{\cdot}Match^{\forall})$**:** The first rule states that the constraint $[P \ll_{\Delta} \mathcal{B}]A$ gets the same type as $(P \to_{\Delta} A)\ \mathcal{B}$. This is sound since $(P \to_{\Delta} A)\ \mathcal{B} \to_{\rho} [P \ll_{\Delta} \mathcal{B}]A$. The second rule instantiates α with τ_2.

Example 3.3.1 (Some derivable typing judgements [Bar92]-inspired)
Let $\Gamma \triangleq \iota{:}*, f{:}\iota \to \iota, a{:}\iota$. *The following judgements are derivable:*

$\emptyset$	$\vdash_{\mathsf{T}}$	$\bot \equiv \forall\alpha.\alpha : *$	*Second-order definition of falsum*
$\emptyset$	$\vdash_{\mathsf{T}}$	$\alpha \to_{*} X \to_{\bot} (X\ \alpha) : \forall\alpha.(\bot \to \alpha)$	*Ex falso sequitur quodlibet*[2]
$\emptyset$	$\vdash_{\mathsf{T}}$	$\beta \to_{*} Y \to_{\beta} X : \forall\beta.(\beta \to \beta)$	*Polymorphic identity*
Γ	$\vdash_{\mathsf{T}}$	$(\gamma \to_{*} f(Z) \to_{(Z:\gamma)} Z)\ \iota\ f(a) : \iota$	*Polymorphic instantiation-application*

[2]Anything follows from a false judgement: the subject of this judgement is its proof.

3.3.1 Metatheory of RhoF

The type system ensures that arguments of a function have the same types as the corresponding formal parameters. The rule ($Term{\cdot}Appl$) only checks that the pattern expected by a function and the argument (considered as a single term) have the same type. The shape of pattern is essential to guarantee the soundness of the type system: the more expressive the patterns are, the more non-sense can follow.

Remark 3.3.1 (Spoofer [BCKL03]) *If we allow variables as the head symbol of a pattern (called "active variables"), then we can write the following counterexample. In the context* $\Gamma \triangleq X{:}\tau_1 \twoheadrightarrow \tau_2, Y{:}\tau_1, f{:}\tau_3 \twoheadrightarrow \tau_2, a{:}\tau_3$, *the pattern*

$$\frac{\Gamma \vdash_{\mathsf{T}} X : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma \vdash_{\mathsf{T}} Y : \tau_1}{\Gamma \vdash_{\mathsf{T}} X(Y) : \tau_2} \quad \textit{and the term} \quad \frac{\Gamma \vdash_{\mathsf{T}} f : \tau_3 \twoheadrightarrow \tau_2 \quad \Gamma \vdash_{\mathsf{T}} a : \tau_3}{\Gamma \vdash_{\mathsf{T}} f(a) : \tau_2}$$

have a common type, but the solution of the matching problem $X(Y) \prec\!\!\prec f(a)$ *instantiates* X *and* Y *with terms not having the expected type (i.e. subject reduction is lost); if* $\Gamma \triangleq f{:}\tau_3 \twoheadrightarrow \tau_2, a{:}\tau_3$ *and* $\Delta \triangleq X{:}\tau_1 \twoheadrightarrow \tau_2, Y{:}\tau_1$, *then* $\Gamma \vdash_{\mathsf{T}} (X(Y) \twoheadrightarrow_{\Delta} Y)f(a) : \tau_1$ *but* $\Gamma \vdash_{\mathsf{T}} a : \tau_3$.

All the metaproperties presented below for RhoF are adapted from the classical properties of the Girard's Lambda-calculus.

Lemma 3.3.1 (Substitution Lemma)

1. *If* $\Gamma, \Delta \vdash_{\mathsf{T}} P : \tau$ *and* $\Gamma \vdash_{\mathsf{T}} B : \tau$ *and* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$, *are such that* $P \prec\!\!\prec_{\mathsf{stk}} B$ *has a solution* θ, *then for all* $X \in \mathsf{Fv}(P)$, *there exists* σ *such that* $\Gamma, \Delta \vdash_{\mathsf{T}} X : \sigma$ *and* $\Gamma \vdash_{\mathsf{T}} X\theta : \sigma$.

2. *If* $\Gamma, \Delta \vdash_{\mathsf{T}} A : \tau$, *then for any well-typed substitution* θ *such that* $\mathsf{Dom}(\theta) = \mathsf{Dom}(\Delta)$, *we have* $\Gamma \vdash_{\mathsf{T}} A\theta : \tau$.

Theorem 3.3.1 (Subject Reduction for RhoF)
If $\Gamma \vdash_{\mathsf{T}} A : \tau$ *and* $A \mapsto^{\mathsf{stk}}_{\rho\delta} B$, *then* $\Gamma \vdash_{\mathsf{T}} B : \tau$.

Proof. *By an induction on the derivation of* $\Gamma \vdash_{\mathsf{T}} A : \tau$. □

Theorem 3.3.2 (Type Uniqueness for $\rho^{\mathsf{Fix}}_{\twoheadrightarrow}$)
If $\Gamma \vdash_{\mathsf{T}} A : \tau_1$ *and* $\Gamma \vdash_{\mathsf{T}} A : \tau_2$, *and* $\mathsf{stk} \notin A$, *then* $\tau_1 \equiv \tau_2$.

Proof. *By an easy induction on the structure of* A. □

The next example shows that termination is not guaranteed for typable terms in RhoF.

Example 3.3.2 (Non Termination of Typable Terms [CKLW02])
If $\Gamma \vdash_{\mathsf{T}} A : \tau$ *then* A *can diverge. Take* $\Gamma \triangleq f{:}(\iota \twoheadrightarrow \iota) \twoheadrightarrow \iota$, *and* $\Delta \triangleq X{:}\iota$, *and* $A \triangleq \omega\ f(\omega)$ *with* $\omega \triangleq f(X) \twoheadrightarrow_{\Delta} X\ f(X)$. *Therefore,* $\Gamma \vdash_{\mathsf{T}} \omega\ f(\omega) : \iota$, *but* $\omega\ f(\omega) \mapsto\!\!\!\!\twoheadrightarrow^{\mathsf{stk}}_{\rho\delta} \ldots$ *This negative result proves that conjecture* (i) *of Exercise at pp. 14 of [CKL02] was false. Notice that* $\omega f(\omega)$ *is typable without using the second-order features of RhoF.*

3.3.2 Typing the representation of Xcerpt queries

To show the flexibility of this polymorphic type system, let us take back the example query 2.6.2:

$$
\begin{array}{l}
(catalogue\{\{\ cd\{title[TITLE]\ artist[ARTIST]\ year[\text{"1988"}]\,\},\ SubCat\}\} \\
\twoheadrightarrow \\
result[\ name[TITLE]\ author[ARTIST]\,])
\end{array}
$$

It is possible to assign it a simple type, by declaring the following signature:

$$
\begin{array}{rcl}
title & : & string \twoheadrightarrow tl \\
artist & : & string \twoheadrightarrow art \\
year & : & int \twoheadrightarrow yr \\
cd & : & tl \twoheadrightarrow art \twoheadrightarrow yr \twoheadrightarrow entry \\
\{\{\ \}\} & : & entry \twoheadrightarrow entry \twoheadrightarrow entry \\
catalogue & : & entry \twoheadrightarrow ctl \\
name & : & string \twoheadrightarrow nm \\
author & : & string \twoheadrightarrow aut \\
result & : & nm \twoheadrightarrow aut \twoheadrightarrow res
\end{array}
$$

The query is then assigned type $ctl \twoheadrightarrow res$ (which is correct: it takes a catalogue and returns a result). During type checking, the bound variables $TITLE$, $ARTIST$ and $SubCat$ are found to be of types respectively $string$, $string$ and $entry$. Thus, this type checking mechanism allows a basic account of correctness.

However, it remains quite limited and can not scale up for real size examples. Supposing for instance in our query we have two instances of $\{\{\ \}\}$ used on distinct data types, we will not be able to give a proper signature. The previous signature can then be modified using type variables:

$$\{\{\ \}\}\ :\ \forall\alpha.(\alpha \twoheadrightarrow \alpha \twoheadrightarrow \alpha)$$

This way, a structure built using $\{\{\ \}\}$ can contain any kind of elements, provided they all have the same type. Still, this setting is not really convenient since in RhoF the instantiation of type variables has to be explicitly specified in the terms. For instance, in the previous query, we would have to annotate the curly brackets with the type of the elements they contain: $\{\{\ \}\}_{entry}$ in order to force α to be $entry$. The next subsection provides a mechanism for implicit instantiation of such type variables.

3.4 The Polymorphic Type Inference uRhoF

In the previous subsection, we studied terms of the Rho-calculus decorated with types. In this ***fully typed*** approach, every closed term comes directly with a unique, intrinsic type. In this subsection, we discuss another way of giving types to terms of the Rho-calculus: the ***type assignment*** approach introduced by Curry [Cur34] for the Theory of Combinators, and then modified by Curry and Feys [CF58, CHS72]. The judgements have the shape $\Gamma \vdash_{\mathsf{U}} U : \tau$, where U is a term of the (untyped) Rho-calculus, τ is a type, and Γ is the context that assigns types to the free-variables of U and τ.

Syntactic Cat.			**Abstract Syntax**
As for RhoF			As for RhoF
$U.V.f$	$\in$	$\mathcal{T}erm$	$U.V \ ::= \ \mathsf{stk} \mid f \mid X \mid P \twoheadrightarrow U \mid [P \ll U]U \mid U\ U \mid U\,;U$

Figure 5: Syntax of uRhoF

In this approach (called *à la* Curry by Barendregt), types are viewed as *predicates* (*properties*) of terms, and each closed term can be assigned either none or infinitely many types. Those systems are called *type assignment systems.* When we look at the Rho-calculus as a kernel calculus underneath a pattern-matching based programming language, this approach corresponds to Elan, or Maude, or OBJ*, or ASF+SDF, or Haskell, or ML-like languages, where the user can write programs in a completely untyped language, and types are automatically inferred at compilation-time. Type inference can be also intended as the construction of an abstract interpretation of the program, that can be used as a correctness criterion.

For the Lambda-calculus, in [Cur34, Lei83, GR88], it was observed that some of the type assignment systems already known in the literature can also be obtained from a fully typed system by means of an *erasing function* that erases type information from terms in a typed system. In particular, the Curry type assignment system ($F1$) [Cur34] can be obtained from $\Lambda \twoheadrightarrow$, the polymorphic type assignment system ($F2$) [Lei83] from $\Lambda 2$, and the higher-order type assignment system ($F\omega$) [GR88] from the higher-order λ-calculus $\Lambda\omega$.

Let $\mathcal{D}er_{\mathsf{T}}$ be a typed derivation, and $(\!| - |\!)$ be the erasing function. By applying $(\!| - |\!)$ to the "subject" of every judgement in $\mathcal{D}er_{\mathsf{T}}$, we obtain a valid type assignment derivation $\mathcal{D}er_{\mathsf{U}}$ with the same structure of the typed one. *Vice versa*, every type assignment derivation can be viewed as the result of an application of $(\!| - |\!)$ to a typed one. In particular, the erasing function $(\!| - |\!)$ induces an *isomorphism* between every typed system and the corresponding type assignment system.

Definition 3.4.1 *The Erasing Function.*

$$\begin{array}{lcllcll}
(\!| \mathsf{stk} |\!) & \triangleq & \mathsf{stk} & (\!| A\ \tau |\!) & \triangleq & (\!| A |\!) & \\
(\!| f |\!) & \triangleq & f & (\!| \alpha \twoheadrightarrow_{*} A |\!) & \triangleq & (\!| A |\!) & \\
(\!| X |\!) & \triangleq & X & (\!| [\alpha \ll_{*} \tau]B |\!) & \triangleq & (\!| B |\!) & \\
(\!| A\ B |\!) & \triangleq & (\!| A |\!)\ (\!| B |\!) & (\!| P \twoheadrightarrow_{\Delta} A |\!) & \triangleq & P \twoheadrightarrow (\!| A |\!) & P \not\equiv \alpha \\
(\!| A\,;B |\!) & \triangleq & (\!| A |\!)\,;(\!| B |\!) & (\!| [P \ll_{\Delta} A]B |\!) & \triangleq & [P \ll (\!| A |\!)](\!| B |\!) & P \not\equiv \alpha
\end{array}$$

This definition can easily be extended to derivations.

Syntax (Figure 5). One can easily see that the syntax is obtained by simply "hiding" the types from the user. Type abstraction ($\alpha \twoheadrightarrow_{*} A$) and type application ($A\ \tau$) are no longer necessary since the polymorphism is fully implicit. As in ML, a term can be seen as an untyped one, but the typing machinery is called before accepting such a term.

Typing Rules (Figures 6 and 7). A primitive polymorphic type assignment system was sketched in [CKL02] (without any metatheory). It proves judgement of the shape:

$$\Gamma \vdash_{\mathsf{U}} ok \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} \tau : * \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} P : \tau \quad \text{and} \quad \Gamma \vdash_{\mathsf{U}} U : \tau$$

Well-formed Contexts

$$\frac{}{\emptyset \vdash_{\mathsf{U}} ok} \ (Ctx{\cdot}Empty) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \alpha \notin \mathsf{Dom}(\Gamma)}{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} ok} \ (Ctx{\cdot}Var^{\forall})$$

$$\frac{\Gamma \vdash_{\mathsf{U}} ok \quad \iota \notin \mathsf{Dom}(\Gamma)}{\Gamma, \iota{:}* \vdash_{\mathsf{U}} ok} \ (Ctx{\cdot}Const) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \Gamma \vdash_{\mathsf{U}} \tau : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\tau \vdash_{\mathsf{U}} ok} \ (Ctx{\cdot}Var)$$

Well-kinded Types

$$\frac{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, \iota{:}*, \Gamma_2 \vdash_{\mathsf{U}} \iota : *} \ (Type{\cdot}Const) \qquad \frac{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, \alpha{:}*, \Gamma_2 \vdash_{\mathsf{U}} \alpha : *} \ (Type{\cdot}Var)$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} \tau : *}{\Gamma \vdash_{\mathsf{U}} \forall \alpha.\tau : *} \ (Type{\cdot}Poly) \qquad \frac{\Gamma \vdash_{\mathsf{U}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{U}} \tau_2 : *}{\Gamma \vdash_{\mathsf{U}} \tau_1 \twoheadrightarrow \tau_2 : *} \ (Type{\cdot}Arrow)$$

Figure 6: The Kind Assignment System for uRhoF

We discuss only the typing rules for well-formed terms and patterns which differ from the the corresponding typed ones.

- $(Term{\cdot}Abs^{\twoheadrightarrow})$**:** The domain of Δ is given by the free-variables of P, *i.e.* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$.
- $(Term{\cdot}Abs^{\forall})$**:** This rule is not syntax directed; the classical side-condition about the freshness of α is enforced by the well-formedness of the context in the premises.
- $(Term{\cdot}Appl^{\forall})$**:** This rule is not syntax directed; the type τ_2 is guessed.
- $(Term{\cdot}Match^{\twoheadrightarrow})$**:** The context Δ is built from the free-variables of P, *i.e.* $\mathsf{Dom}(\Delta) = \mathsf{Fv}(P)$.

All the metaproperties presented below for uRhoF are adapted from system $F2$ of Leivant and for RhoF.

Lemma 3.4.1 (Substitution of type variables in uRhoF)
If $\Gamma \vdash_{\mathsf{U}} U : \tau$ *and* $\mathsf{Dom}(\theta) \subseteq \mathcal{T}ype_V$ *and* $\mathsf{CoDom}(\theta) \subseteq \mathsf{Dom}(\Gamma)$ *then* $\Gamma\theta \vdash_{\mathsf{U}} U : \tau\theta$.

Theorem 3.4.1 (Subject Reduction for uRhoF)
If $\Gamma \vdash_{\mathsf{U}} U : \tau$ *and* $U \mapsto^{\mathsf{stk}}_{\rho\sigma\delta} V$, *then* $\Gamma \vdash_{\mathsf{U}} V : \tau$.

Proof. *By an induction on the derivation of* $\Gamma \vdash_{\mathsf{U}} U : \tau$. □

Since uRhoF is essentially the counterpart of $F2$ of Leivant, and since Rho-calculus is a conservative extension of Lambda-calculus, it follows that type inference problem is undecidable.

Theorem 3.4.2 (Undecidability of Type Inference for uRhoF)
For a closed U *such that* $\mathsf{stk} \notin U$, *the following problem is undecidable:*

Well-formed Terms and Patterns

$$\frac{\Gamma \vdash_{\mathsf{U}} \tau : * \quad \tau \neq \bot}{\Gamma \vdash_{\mathsf{U}} \mathsf{stk} : \tau} \; (Term{\cdot}Stuck)$$

$$\frac{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, X{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} X : \tau} \; (Term{\cdot}Var) \qquad \frac{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} ok}{\Gamma_1, f{:}\tau, \Gamma_2 \vdash_{\mathsf{U}} f : \tau} \; (Term{\cdot}Const)$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{U}} U : \tau_1 \to \tau_2 \\ \Gamma \vdash_{\mathsf{U}} V : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{U}} U\ V : \tau_2} \; (Term{\cdot}Appl^{\to}) \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \;\; \mathsf{Dom}(\Delta) = \mathsf{Fv}(P) \\ \Gamma, \Delta \vdash_{\mathsf{U}} U : \tau_2 \;\; \Gamma, \Delta \vdash_{\mathsf{U}} \tau_1 \to \tau_2 : *\end{array}}{\Gamma \vdash_{\mathsf{U}} P \to U : \tau_1 \to \tau_2} \; (Term{\cdot}Abs^{\to})$$

$$\frac{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} U : \tau}{\Gamma \vdash_{\mathsf{U}} U : \forall\alpha.\tau} \; (Term{\cdot}Abs^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} U : \forall\alpha.\tau_1 \quad \Gamma \vdash_{\mathsf{U}} \tau_2 : *}{\Gamma \vdash_{\mathsf{U}} U : \tau_1\{\tau_2/\alpha\}} \; (Term{\cdot}App^{\forall})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{U}} U : \tau \\ \Gamma \vdash_{\mathsf{U}} V : \tau\end{array}}{\Gamma \vdash_{\mathsf{U}} U \,;V : \tau} \; (Term{\cdot}Struct) \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \;\; \mathsf{Dom}(\Delta) = \mathsf{Fv}(P) \\ \Gamma, \Delta \vdash_{\mathsf{U}} U : \tau_2 \;\; \Gamma \vdash_{\mathsf{U}} V : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{U}} [P \ll V]U : \tau_2} \; (Term{\cdot}Match^{\to})$$

Figure 7: The Type Assignment System for uRhoF

- *Type Inference: given* Γ *(gives meaning to constants), is there a type* τ *such that* $\Gamma \vdash_{\mathsf{U}} U : \tau$ *?*

Proof. *It follows* a fortiori *from the well known result of Wells [Wel99].* □

3.4.1 Typing the representation of Xcerpt queries

Again in this system we can assign a polymorphic type $\forall\alpha.(\alpha \to \alpha \to \alpha)$ to constructors such as {{ }} but we do not need to state which type will instantiate α. The query

$$\begin{array}{l}(\mathit{catalogue}\{\{\ \mathit{cd}\{\mathit{title}[\mathit{TITLE}]\ \mathit{artist}[\mathit{ARTIST}]\ \mathit{year}[\text{"1988"}]\,\},\ \ \mathit{SubCat}\}\} \\ \to \\ \mathit{result}[\,\mathit{name}[\mathit{TITLE}]\ \mathit{author}[\mathit{ARTIST}]\,])\ \ \mathit{DB}\end{array}$$

can then be assigned the type $ctl \to res$ without having to annotate it. The instantiation of the type variable α to $entry$ is inferred, but the constructor {{ }} remains fully polymorphic for further use in another query.

A perhaps even more interesting use of such a type system is the definition of polymorphic functions (*i. e.* in the context of the semantic web, polymorphic queries). Suppose we want to maintain a database containing any kind of objects, with the only assumption that a colour is assigned to these objects. The constructors of such a database will be:

$$\begin{array}{rcl} colour & : & string \to clr \\ coloured\_object & : & \forall\alpha.(\alpha \to clr \to clrd(\alpha)) \\ [\,] & : & \forall\alpha.(\alpha \to \alpha \to \alpha) \\ coloured\_catalogue & : & clrd(\alpha) \to clrd\_ctl(\alpha) \end{array}$$

Then for instance, if $cat : animal$ and $mouse : animal$, the database

$$\begin{array}{ll} coloured\_catalogue[& coloured\_object[cat, colour[\text{“}black''], \\ & coloured\_object[mouse, colour[\text{“}white'']]] \end{array}$$

is correct and has type $clrd\_ctl(animal)$.

For querying, we will use the constructors

$$\begin{array}{rcl} \{\{\ \}\} & : & \forall\alpha.(\alpha \to \alpha \to \alpha) \\ result & : & \forall\alpha.(\alpha \to res(\alpha)) \end{array}$$

The following query selects every black object:

$$\begin{array}{l} coloured\_catalogue\{\{coloured\_object[OBJ, colour[\text{“}black'']], SubCat\}\} \\ \to \\ result[OBJ] \end{array}$$

It is accepted by our type system and can be assigned the type $\forall\alpha.\big(clrd\_ctl(\alpha) \to res(\alpha)\big)$. It denotes that its argument can be a coloured catalogue of any type of objects, and ensures that the results of the query all have the same type as the elements of the catalogue.

The next subsection deals with automated type inference for such polymorphic queries.

3.5 Type inference

This section recalls the results from [LW05] and extends them with a full description of $\mathsf{uRhoF}_{\ll}$, a decidable fragment of uRhoF. A type inference algorithm is given and proved sound, correct and principal.

3.5.1 Decidability of typing for RhoF

Theorem 3.5.1 (Decidability of Typing for RhoF)
For a closed A such that $\mathsf{stk} \notin A$, the following problems are decidable:

1. *Type Reconstruction: is there a type τ such that $\emptyset \vdash_{\mathsf{T}} A : \tau$?*

2. *Type Checking: for a given τ, is it true that $\emptyset \vdash_{\mathsf{T}} A : \tau$?*

Proof.

1. *We give the sketch of a recursive algorithm (Figure 8) for building τ (or returning* false *if it does not exist).*

2. *We use the previous algorithm for type reconstruction (Figure 9). By uniqueness of typing, $\Gamma \vdash_{\mathsf{T}} A : \tau$ if and only if τ is equivalent to the type found for A.*

□

$$
\begin{array}{lllll}
\mathsf{Type}^2(A;\Gamma) & \triangleq & \multicolumn{3}{l}{\mathsf{match}\ A\ \mathsf{with}} \\
& & \alpha & \Rightarrow & * \\
& & & & \mathsf{if}\ \alpha{:}* \in \Gamma \\
& & X/f & \Rightarrow & \tau \\
& & & & \mathsf{if}\ X/f{:}\tau \in \Gamma \\
& & A_1 \mathrel{;} A_2 & \Rightarrow & \mathsf{Type}^2(A_1;\Gamma) \\
& & & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \mathsf{Type}^2(A_2;\Gamma) \\
& & P \twoheadrightarrow_\Delta A_1 & \Rightarrow & \mathsf{Type}^2(P;\Gamma,\Delta) \twoheadrightarrow \mathsf{Type}^2(A_1;\Gamma,\Delta) \\
& & & & \mathsf{if}\ \mathsf{Type}^2(P;\Gamma,\Delta) \neq \mathsf{false}\ \neq \mathsf{Type}^2(A_1;\Gamma,\Delta) \\
& & & & \mathsf{and}\ P \not\equiv \alpha \\
& & [P \ll_\Delta A_1]A_2 & \Rightarrow & \mathsf{Type}^2(A_2;\Gamma,\Delta) \\
& & & & \mathsf{if}\ \mathsf{Type}^2(P;\Gamma,\Delta) = \mathsf{Type}^2(A_1;\Gamma,\Delta) \neq \mathsf{false} \\
& & & & \mathsf{and}\ P \not\equiv \alpha \\
& & A_1\ A_2 & \Rightarrow & \tau_2 \\
& & & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \tau_1 \twoheadrightarrow \tau_2\ \mathsf{and}\ \mathsf{Type}^2(A_2;\Gamma) = \tau_1 \\
& & \alpha \twoheadrightarrow_* A_1 & \Rightarrow & \forall\alpha.\mathsf{Type}^2(A_1;\Gamma,\alpha{:}*) \\
& & & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma,\alpha{:}*) \neq \mathsf{false} \\
& & [\alpha \ll_* \tau]A_1 & \Rightarrow & \mathsf{Type}^2(A_1;\Gamma,\alpha{:}*)\{\tau/\alpha\} \\
& & A_1\ \tau & \Rightarrow & \tau_1\{\tau/\alpha\} \\
& & & & \mathsf{if}\ \mathsf{Type}^2(A_1;\Gamma) = \forall\alpha.\tau_1 \\
& & \_ & \Rightarrow & \mathsf{false}
\end{array}
$$

Figure 8: The Algorithm Type^2

$$\mathsf{Typecheck}^2(A;\Gamma;\tau) \quad \triangleq \quad \mathsf{if}\ \mathsf{Type}^2(A;\Gamma) = \tau\ \mathsf{then\ true\ else\ false}$$

Figure 9: The Algorithm $\mathsf{Typecheck}^2$

3.5.2 uRhoF$_\ll$: a decidable fragment of uRhoF

It is well-known that the type assignment system uRhoF is undecidable; let us recall why.

A first reason for this is the vast amount of types available: quantification can occur anywhere in a type. We need to restrict polymorphism to well known "type schemes" of the form $\forall\overline{\alpha}.\tau$, where τ is a first-order type, *i.e.* a monomorphic-type. As example, $\forall\alpha.\alpha \twoheadrightarrow \alpha \ \simeq\ \{\tau \twoheadrightarrow \tau \mid \tau \in \mathcal{T}ype^{\twoheadrightarrow}\}$ is the type-scheme for polymorphic identity. Type-schemas are equal modulo α-conversion. We define simultaneous instantiations of type-schemas, via a relation (denoted by $\leq$) as follows:

$$\tau_1 \leq \tau_2 \text{ iff } \tau_2 \triangleq \forall\overline{\alpha}.\tau_3 \text{ and } \tau_1 \triangleq \tau_3\{\overline{\tau}/\overline{\alpha}\} \text{ for suitable } \overline{\tau}.$$

The other main problem (more peculiar to our calculus) is the ability to define any number of constants with a given type, without really considering them as constructors. Thus, in uRhoF$_\ll$, a constant can have a type $f : \forall\alpha.(\alpha \to \iota)$ where the parameter α does not appear explicitly in the rightmost type ι. Then when typing

$$[f(X) \ll f(Y \twoheadrightarrow Y)](X1)$$

the pattern $f(X)$ gets type ι, where the type of $Y \twoheadrightarrow Y$ is forgotten. Then it is impossible to infer correctly the type of X: a standard algorithm would suggest the most general type $\forall\beta(int \to \beta)$, so the type computed for the expression above can be anything.

This "lax" typing discipline for constant leads to undecidability of typing: indeed, the inference algorithm could be easily patched to deal with the example above, but the problem is that the pattern could be replaced by a variable Z and the matching against $f(X)$ can then be arbitrarily nested in the body of the delayed matching constraint. Thus, we need to enrich the rightmost term of the type of f with all the type variables appearing in the whole type. The resulting type system is given in Figures 10 and 11. The only rules that needs to be commented are:

- Formation of admissible type schemes follow some strict rules: every bound variable has to appear in the rightmost type, hence the context is divided in two parts in order to keep track of these bound variables;
- $(Type{\cdot}Poly)$: all type scheme are built at once;
- $(Term{\cdot}Var)$, and $(Term{\cdot}Const)$: the type of a variable/constant is a type instance of its type-scheme;
- $(Term{\cdot}Abst^{\twoheadrightarrow})$: the context Δ has to be inferred, but it can assign only types (not type-schemes) to the variables of P. It corresponds to the behavior of the typing rule for functional abstraction `fun` $x \to a$ in ML.
- $(Term{\cdot}Match)$: this rule performs a restricted form of polymorphic type inference. Again the context Δ used to type P assigns only types to the free variables of P, but when typing U the corresponding type-schemes can be used. It is an enhanced version of the ML let featuring matching.

Figure 12 shows a simple type derivation in uRhoF$_\ll$for the problematic term shown at the beginning of this subsubsection. We see that the pattern $f(X)$ is assigned a type $\iota_1(\beta \twoheadrightarrow \beta)$, ensuring that X has type $\beta \twoheadrightarrow \beta$. Then generalization gives it type $\forall\beta.(\beta \twoheadrightarrow \beta)$ when typing the body, which ensures that any type of x is an instance of $\beta \twoheadrightarrow \beta$.

The next subsubsection presents an algorithm (called $\mathsf{W}^{\ll}$) that gives a solution to the above problem.

3.5.3 The Algorithm $\mathsf{W}^{\ll}$

We customize the algorithm W of Damas-Milner [DM82] (see also the Caml notes of F. Pottier [Pot]). We present an algorithm $\mathsf{W}^{\ll}$that takes as input a uRhoF$_\ll$-term U, an environment Γ, and a set of "fresh" type variables $\mathcal{V}ar$, and (1) checks if it can be well-typed, and (2) infers a *principal typing* τ for U in Γ, such that:

1. The judgement $\Gamma \vdash_{\mathsf{U}} U : \tau$ is derivable

Abstract Syntax

τ	$::=$	$\alpha \mid \iota_{\overline{\alpha}} \mid \tau \mathrel{-\!\triangleright} \tau$	Poly Types
σ	$::=$	$\forall \overline{\alpha}.\tau$	Poly Type Schemes
$U.V$	$::=$	As for uRhoF	Poly Terms

Well-formed Contexts

$$\frac{}{\emptyset \vdash_{\mathsf{U}} ok}\ (Ctx{\cdot}Empty) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \Gamma \vdash_{\mathsf{U}} \sigma : * \quad X \notin \mathsf{Dom}(\Gamma)}{\Gamma, X{:}\sigma \vdash_{\mathsf{U}} ok}\ (Ctx{\cdot}Var)$$

Well-kinded Type Schemes

$$\frac{\Gamma, \overline{\alpha{:}*} \vdash_{\mathsf{U}} ok}{\Gamma \vdash_{\mathsf{U}} \iota_{\overline{\alpha}} : *}\ (TypeScheme{\cdot}Const) \qquad \frac{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} ok}{\Gamma \vdash_{\mathsf{U}} \alpha : *}\ (TypeScheme{\cdot}Var)$$

$$\frac{\alpha \in \mathsf{Lab}(\sigma) \quad \Gamma, \alpha{:}* \vdash_{\mathsf{U}} \sigma : *}{\Gamma \vdash_{\mathsf{U}} \forall \alpha.\sigma : *}\ (TypeScheme^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} \tau_1 : * \quad \Gamma \vdash_{\mathsf{U}} \tau_2 : *}{\Gamma \vdash_{\mathsf{U}} \tau_1 \mathrel{-\!\triangleright} \tau_2 : *}\ (TypeScheme^{\mathrel{-\!\triangleright}})$$

$\mathsf{Lab}(\iota_{\overline{\alpha}}) = \overline{\alpha}$ $\qquad$ $\mathsf{Lab}(\tau_1 \mathrel{-\!\triangleright} \tau_2) = \mathsf{Lab}(\tau_2)$

$\mathsf{Lab}(\alpha) = \{\alpha\}$ $\qquad$ $\mathsf{Lab}(\forall \alpha.\sigma) = \mathsf{Lab}(\sigma) \setminus \alpha$

Figure 10: Types of $\mathsf{uRhoF}_{\ll}$

2. If $\Gamma \vdash_{\mathsf{U}} U : \tau'$, then there exists a substitution θ, such that $\Gamma\theta \vdash_{\mathsf{U}} U : \tau\theta$.

Definition 3.5.1 (The Algorithm $\mathsf{W}^{\ll}$)
The algorithm $\mathsf{W}^{\ll}$ is given in Figure 13. It uses the classical algorithm mgu*, that is the unification algorithm between first-order terms of [Rob65] (hence omitted).*

Definition 3.5.2 (Independence) 1. *A substitution θ is independent from a set of type-variables $\mathcal{V}ar$, written $\theta \not< \mathcal{V}ar$, if $\mathsf{Dom}(\theta) \cap \mathcal{V}ar = \emptyset$ and $\mathsf{CoDom}(\sigma) \cap \mathcal{V}ar = \emptyset$.*

2. *Two substitutions θ_1 and θ_2, are equals out of $\mathcal{V}ar$, written $\theta_1 \overset{\mathcal{V}ar}{=}_{\not<} \theta_2$, if $\alpha\theta_1 = \alpha\theta_2$, for all $\alpha \notin \mathcal{V}ar$.*

Theorem 3.5.2 (Soundness of $\mathsf{W}^{\ll}$)
If $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) = (\tau; \theta; \mathcal{V}ar')$, then $\Gamma\theta \vdash_{\mathsf{U}} U : \tau$.

Proof. *By induction on the structure of U. We treat the cases of variables, abstraction, application and matching constraints; constants are treated similarly to variables and structures similarly to application.*

- *if $U = X$ then $\tau = \mathsf{Inst}(\Gamma(X); \mathcal{V}ar)$ and $\theta = \theta_{\mathsf{ID}}$. The function* Inst *ensures that $\tau \leq \Gamma(X)$ so $\Gamma \vdash_{\mathsf{U}} X : \tau$.*

Well-formed Terms and Patterns

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{U}} \tau : * \\ \Gamma_1, X{:}\sigma, \Gamma_2 \vdash_{\mathsf{U}} ok \quad\quad \sigma \leq \tau\end{array}}{\Gamma_1, X{:}\sigma, \Gamma_2 \vdash_{\mathsf{U}} X : \tau}\;(Term{\cdot}Var)$$

$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathsf{U}} V : \tau_1 & \\ \Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 & \mathsf{Bv}(\mathsf{CoDom}(\Delta)) = \emptyset \\ \Gamma, \mathsf{Gen}(\Delta;\Gamma) \vdash_{\mathsf{U}} U : \tau_2 & \mathsf{Dom}(\Delta) = \mathsf{Fv}(P)\end{array}}{\Gamma \vdash_{\mathsf{U}} [P \ll V].U : \tau_2}\;(Term{\cdot}Match)$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{U}} \tau : * \\ \Gamma_1, f{:}\sigma, \Gamma_2 \vdash_{\mathsf{U}} ok \quad\quad \sigma \leq \tau\end{array}}{\Gamma_1, f{:}\sigma, \Gamma_2 \vdash_{\mathsf{U}} f : \tau}\;(Term{\cdot}Const)$$

$$\frac{\begin{array}{ll}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 & \mathsf{Bv}(\mathsf{CoDom}(\Delta)) = \emptyset \\ \Gamma, \Delta \vdash_{\mathsf{U}} U : \tau_2 & \mathsf{Dom}(\Delta) = \mathsf{Fv}(P)\end{array}}{\Gamma \vdash_{\mathsf{U}} P \rightarrowtriangle U : \tau_1 \rightarrowtriangle \tau_2}\;(Term{\cdot}Abst^{\rightarrowtriangle})$$

Remove $(Term{\cdot}Abst^{\forall})$ and $(Term{\cdot}Appl^{\forall})$

$\mathsf{Gen}(\tau;\Gamma) \quad \triangleq \quad \forall\overline{\alpha}.\tau$ where $\overline{\alpha} = \mathsf{Fv}(\tau) \setminus \mathsf{Fv}(\Gamma)$
and Gen is pointwise extended to contexts.

Figure 11: Terms of $\mathsf{uRhoF}_{\ll}$

$$\frac{\dfrac{(*) \quad \dfrac{\dfrac{\dfrac{}{\beta \leq \beta}}{\Gamma, Y{:}\beta \vdash_{\mathsf{U}} Y : \beta}}{\Gamma \vdash_{\mathsf{U}} Y \rightarrowtriangle Y : \beta \rightarrowtriangle \beta}}{\Gamma \vdash_{\mathsf{U}} f(Y \rightarrowtriangle Y) : \iota_1(\beta \rightarrowtriangle \beta)} \quad \dfrac{(*) \quad \dfrac{\beta \rightarrowtriangle \beta \leq \beta \rightarrowtriangle \beta}{\Gamma, X : \beta \rightarrowtriangle \beta \vdash_{\mathsf{U}} X : \beta \rightarrowtriangle \beta}}{\Gamma, X : \beta \rightarrowtriangle \beta \vdash_{\mathsf{U}} f(X) : \iota_1(\beta \rightarrowtriangle \beta)} \quad \dfrac{\dfrac{\iota \rightarrowtriangle \iota \leq \forall\beta.(\beta \rightarrowtriangle \beta)}{\Gamma, \Delta \vdash_{\mathsf{U}} X : \iota \rightarrowtriangle \iota} \quad \dfrac{\iota \leq \iota}{\Gamma, \Delta \vdash_{\mathsf{U}} 1 : \iota}}{\Gamma, \Delta \vdash_{\mathsf{U}} X\ 1 : \iota}}{\Gamma \vdash_{\mathsf{U}} [\ f(X) \ll f(Y \rightarrowtriangle Y)]\ (X\ 1) : \iota}$$

where $\Gamma \triangleq 1{:}\iota, f{:}\forall\alpha.(\alpha \rightarrowtriangle \iota_1(\alpha))$, and $\Delta \triangleq X{:}\forall\beta.(\beta \rightarrowtriangle \beta)$.

and $(*)$ is $\dfrac{(\beta \rightarrowtriangle \beta) \rightarrowtriangle \iota_1(\beta \rightarrowtriangle \beta) \leq \forall\alpha.(\alpha \rightarrowtriangle \iota_1(\alpha))}{\Gamma \vdash_{\mathsf{U}} f : (\beta \rightarrowtriangle \beta) \rightarrowtriangle \iota_1(\beta \rightarrowtriangle \beta)}$

Figure 12: A Simple Type Derivation in $\mathsf{uRhoF}_{\ll}$.

$$\boxed{\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) = (\tau; \theta; \mathcal{V}ar')}$$

$$
\begin{array}{rcl}
\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) & \triangleq & \textsf{match } U \textsf{ with} \\[4pt]
f & \Rightarrow & \textsf{if } f \in \mathsf{Dom}(\Gamma) \textsf{ then} \\
 & & \textsf{take } (\tau; \mathcal{V}ar') = \mathsf{Inst}(\Gamma(f); \mathcal{V}ar) \textsf{ and } \theta = \theta_{\text{ID}} \\[4pt]
X & \Rightarrow & \textsf{if } X \in \mathsf{Dom}(\Gamma) \textsf{ then} \\
 & & \textsf{take } (\tau; \mathcal{V}ar') = \mathsf{Inst}(\Gamma(X); \mathcal{V}ar) \textsf{ and } \theta = \theta_{\text{ID}} \\[4pt]
U_1 \,; U_2 & \Rightarrow & \textsf{let } (\tau_1; \theta_1; \mathcal{V}ar_1) = \mathsf{W}^{\ll}(\Gamma; U_1; \mathcal{V}ar) \textsf{ in} \\
 & & \textsf{let } (\tau_2; \theta_2; \mathcal{V}ar_2) = \mathsf{W}^{\ll}(\Gamma\theta_1; U_2; \mathcal{V}ar_1) \textsf{ in} \\
 & & \textsf{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \textsf{ in} \\
 & & \textsf{take } \tau = \tau_2\phi \textsf{ and } \theta = \phi \circ \theta_2 \circ \theta_1 \textsf{ and } \mathcal{V}ar' = \mathcal{V}ar_2 \\[4pt]
P \twoheadrightarrow U_1 & \Rightarrow & \textsf{let } \overline{X} = \mathsf{Fv}(P) \textsf{ and } \overline{\alpha_X} \in \mathcal{V}ar_1 \textsf{ in} \\
 & & \textsf{let } (\tau_1; \theta_1; \mathcal{V}ar_1) = \mathsf{W}^{\ll}(\Gamma, \overline{X{:}\alpha_X}; P; \mathcal{V}ar \setminus \{\overline{\alpha_X}\}) \textsf{ in} \\
 & & \textsf{let } (\tau_2; \theta_2; \mathcal{V}ar_2) = \mathsf{W}^{\ll}(\Gamma\theta_1, \overline{X{:}\alpha_X\theta_1}; U_1; \mathcal{V}ar_1) \textsf{ in} \\
 & & \textsf{take } \tau = \tau_1\theta_2 \twoheadrightarrow \tau_2 \textsf{ and } \theta = \theta_2 \circ \theta_1 \textsf{ and } \mathcal{V}ar' = \mathcal{V}ar_2 \\[4pt]
U_1\ U_2 & \Rightarrow & \textsf{let } (\tau_1; \theta_1; \mathcal{V}ar_1) = \mathsf{W}^{\ll}(\Gamma; U_1; \mathcal{V}ar) \textsf{ in} \\
 & & \textsf{let } (\tau_2; \theta_2; \mathcal{V}ar_2) = \mathsf{W}^{\ll}(\Gamma\theta_1; U_2; \mathcal{V}ar_1) \textsf{ in} \\
 & & \textsf{let } \alpha \in \mathcal{V}ar_2 \textsf{ in} \\
 & & \textsf{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2 \twoheadrightarrow \alpha) \textsf{ in} \\
 & & \textsf{take } \tau = \alpha\phi \textsf{ and } \theta = \phi \circ \theta_2 \circ \theta_1 \textsf{ and } \mathcal{V}ar' = \mathcal{V}ar_2 \setminus \{\alpha\} \\[4pt]
[P \ll U_2].U_1 & \Rightarrow & \textsf{let } (\tau_1; \theta_1; \mathcal{V}ar_1) = \mathsf{W}^{\ll}(\Gamma; U_2; \mathcal{V}ar) \textsf{ in} \\
 & & \textsf{let } \overline{X} = \mathsf{Fv}(P) \textsf{ and } \overline{\alpha_X} \in \mathcal{V}ar_1 \textsf{ in} \\
 & & \textsf{let } (\tau_2; \theta_2; \mathcal{V}ar_2) = \mathsf{W}^{\ll}(\Gamma\theta_1, \overline{X{:}\alpha_X}; P; \mathcal{V}ar_1 \setminus \{\overline{\alpha_X}\}) \textsf{ in} \\
 & & \textsf{let } \phi = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \textsf{ in} \\
 & & \textsf{let } (\tau_3; \theta_3; \mathcal{V}ar_3) = \mathsf{W}^{\ll}(\Gamma\theta_1\theta_2\phi, \overline{X{:}\mathsf{Gen}(\alpha_X\theta_2\phi; \Gamma\theta_1\theta_2\phi)}; U_1; \mathcal{V}ar_2) \textsf{ in} \\
 & & \textsf{take } \tau = \tau_3 \textsf{ and } \theta = \theta_3 \circ \phi \circ \theta_2 \circ \theta_1 \textsf{ and } \mathcal{V}ar' = \mathcal{V}ar_3 \\[4pt]
\_ & \Rightarrow & \textsf{false}
\end{array}
$$

$$\mathsf{Inst}(\forall\overline{\alpha}.\tau; \mathcal{V}ar) \triangleq (\tau\{\overline{\beta/\alpha}\}; \mathcal{V}ar \setminus \{\overline{\beta}\}) \quad \text{where } \overline{\beta} \text{ are distinct fresh variables taken in } \mathcal{V}ar$$

Figure 13: The Algorithm $\mathsf{W}^{\ll}$.

- *if* $U = P \twoheadrightarrow U_1$ *then* $\tau = \tau_1\theta_2 \twoheadrightarrow \tau_2$ *and* $\theta = \theta_2 \circ \theta_1$ *where* τ_1, θ_1, τ_2 *and* θ_2 *are obtained by recursive calls to* $\mathsf{W}^{\ll}$ *for the subterms* P *and* U_1.

  Let us take $\Delta = \overline{X : \alpha_X}$. *By induction hypothesis we have* $\Gamma\theta_1, \Delta\theta_1 \vdash_{\mathsf{U}} P : \tau_1$ *and* $\Gamma\theta, \Delta\theta \vdash_{\mathsf{U}} U_1 : \tau_2$. *By type variables substitution (lemma 3.4.1), we also have* $\Gamma\theta, \Delta\theta \vdash_{\mathsf{U}} P : \tau_1\theta_2$. *Thus we can indeed derive* $\Gamma\theta \vdash_{\mathsf{U}} P \twoheadrightarrow U_1 : \tau_1\theta_2 \twoheadrightarrow \tau_2$.

- *if* $U = U_1\ U_2$ *then* $\tau = \alpha\phi$ *and* $\theta = \phi \circ \theta_2 \circ \theta_1$ *where* $\tau_1\theta_2\phi = \tau_2\phi \twoheadrightarrow \alpha\phi$ *and* τ_1, θ_1, τ_2 *and* θ_2 *are obtained by recursive calls to* $\mathsf{W}^{\ll}$ *on the subterms* U_1 *and* U_2.

  By induction hypothesis we have $\Gamma\theta_1 \vdash_{\mathsf{U}} U_1 : \tau_1$ *and* $\Gamma\theta_1\theta_2 \vdash_{\mathsf{U}} U_2 : \tau_2$. *By type variables substitution (lemma 3.4.1), we have* $\Gamma\theta \vdash_{\mathsf{U}} U_1 : \tau_1\theta_2\phi$ *and* $\Gamma\theta \vdash_{\mathsf{U}} U_2 : \tau_2\phi$. *Since we know that* $\tau_1\theta_2\phi = \tau_2\phi \twoheadrightarrow \alpha\phi$, *we can indeed derive* $\Gamma\theta \vdash_{\mathsf{U}} U_1\ U_2 : \alpha\phi$.

- *if* $U = [P \ll U_2]U_1$ *then* $\tau = \tau_3$ *and* $\theta = \theta_3 \circ \phi$ *where* $\tau_1\theta_2\phi = \tau_2\phi$ *and* $\tau_1, \theta_1, \tau_2, \theta_2, \tau_3$ *and* θ_3 *are obtained by recursive calls to* $\mathsf{W}^{\ll}$ *on the subterms* U_2, P *and* U_1.

  Let us take $\Delta = \overline{X : \alpha_X}$. *By induction hypothesis we have* $\Gamma\theta_1 \vdash_{\mathsf{U}} U_2 : \tau_1$ *and* $\Gamma\theta_1\theta_2, \Delta\theta_2 \vdash_{\mathsf{U}} P : \tau_2$ *and* $\Gamma\theta_1\theta_2\phi\theta_3, \mathsf{Gen}(\Delta\theta_2\phi; \Gamma\theta_1\theta_2\phi)\theta_3 \vdash_{\mathsf{U}} U_1 : \tau_3$. *It is easy to see that*

  $\mathsf{Gen}(\Delta\theta_2\phi; \Gamma\theta_1\theta_2\phi)\theta_3 = \mathsf{Gen}(\Delta\theta_2\phi\theta_3; \Gamma\theta_1\theta_2\phi\theta_3)$ *since* $\theta_3 \not< \mathsf{Dom}(\Gamma\theta_1\theta_2\phi)$.

  By type variables substitution (lemma 3.4.1), we have $\Gamma\theta \vdash_{\mathsf{U}} U_2 : \tau_1\theta_2\phi\theta_3$ *and* $\Gamma\theta, \Delta\theta_2\phi\theta_3 \vdash_{\mathsf{U}} P : \tau_2\phi\theta_3$. *Since we know that* $\tau_1\theta_2\phi = \tau_2\phi$, *we can indeed derive* $\Gamma\theta \vdash_{\mathsf{U}} [P \ll U_2]U_1 : \tau$.

  □

Theorem 3.5.3 (Completeness and Principality of $\mathsf{W}^{\ll}$)
For all $\mathcal{V}ar$ *and* Γ, *such that* $\mathcal{V}ar \cap \mathsf{CoDom}(\Gamma) = \emptyset$, *if* $\Gamma\phi \vdash_{\mathsf{U}} U : \tau'$, *then:*

1. $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) \neq \mathsf{false}$;
2. $\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) = (\tau; \theta; \mathcal{V}ar')$, *for some* τ *and* θ *and* $\mathcal{V}ar'$;
3. $\tau' = \tau\psi$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not<} \psi \circ \theta$, *for some* ψ.

Proof. *By induction on the structure of* U. *We treat the cases of variables, abstraction, application and matching constraints; constants are treated similarly to variables and structures similarly to application.*

- *if* $U = X$ *then* $\Gamma(X)\phi \leq \tau'$, *which by definition means that* $\Gamma(X) = \forall\overline{\alpha}.\tau_1$ *and* $\tau' = \tau_1\phi\{\overline{\tau}/\overline{\alpha}\}$. *In this case,* $\mathsf{W}^{\ll}$ *never fails and* $\tau = \mathsf{Inst}(\Gamma(X); \mathcal{V}ar) = \tau_1\{\overline{\beta}/\overline{\alpha}\}$ *and* $\theta = \theta_{\mathsf{ID}}$. *Thus, with* $\psi = \{\overline{\tau}/\overline{\beta}\}$ *over* $\overline{\beta} = \mathcal{V}ar \setminus \mathcal{V}ar'$ *and* $\psi = \phi$ *otherwise, we have* $\tau' = \tau_1\phi\{\overline{\tau}/\overline{\alpha}\} = \tau_1\{\overline{\beta}/\overline{\alpha}\}\phi\{\overline{\tau}/\overline{\beta}\} = \tau\psi$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not<} \psi$.

- *if* $U = P \twoheadrightarrow U_1$ *then* $\tau' = \tau'_1 \twoheadrightarrow \tau'_2$ *where* $\Gamma\phi, \Delta \vdash_{\mathsf{U}} P : \tau'_1$ *and* $\Gamma\phi, \Delta \vdash_{\mathsf{U}} U_1 : \tau'_2$ *for a certain* Δ *assigning types to the free variables of* P. *By suitable renaming of the bound variables in* U, *we can always assume* $\Delta = \overline{X : \alpha_X}$ *and consider an extended* ϕ *so that in fact we typed* P *and* U_1 *in the context* $\Gamma\phi, \Delta\phi$.

  By induction hypothesis, $\mathsf{W}^{\ll}(\Gamma, \Delta; P; \mathcal{V}ar \setminus \{\overline{\alpha_X}\}) = (\tau_1; \theta_1; \mathcal{V}ar_1)$ *such that* $\tau'_1 = \tau_1\psi_1$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not<} \psi_1 \circ \theta_1$ *for a certain* ψ_1.

  In particular $(\Gamma, \Delta)\phi = (\Gamma, \Delta)\theta_1\psi_1$ *so by induction hypothesis* $\mathsf{W}^{\ll}(\Gamma\theta_1, \Delta\theta_1; U_1; \mathcal{V}ar_1) = (\tau_2; \theta_2; \mathcal{V}ar_2)$ *such that* $\tau'_2 = \tau_2\psi_2$ *and* $\psi_1 \overset{\mathcal{V}ar}{=}_{\not<} \psi_2 \circ \theta_2$ *for a certain* ψ_2.

Thus, $\mathsf{W}^{\ll}(\Gamma; P \rightarrowtriangle U_1; \mathcal{V}ar)$ *does not fail, and it returns* $(\tau_1\theta_2 \rightarrowtriangle \tau_2; \theta_2 \circ \theta_1; \mathcal{V}ar_2)$*. Thus, with* $\psi = \psi_2$*, we have* $\tau\psi = (\tau_1\theta_2 \rightarrowtriangle \tau_2)\psi_2 = \tau_1\psi_1 \rightarrowtriangle \tau_2\psi_2 = \tau_1' \rightarrowtriangle \tau_2' = \tau'$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_1 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_2 \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi \circ \theta_2 \circ \theta_1$.

- *if* $U = U_1\ U_2$ *then there is some* τ_1' *such that* $\Gamma\phi \vdash_{\mathsf{U}} U_1 : \tau_1' \rightarrowtriangle \tau'$ *and* $\Gamma\phi \vdash_{\mathsf{U}} U_2 : \tau_1'$.

  By induction hypothesis, $\mathsf{W}^{\ll}(\Gamma; U_1; \mathcal{V}ar) = (\tau_1; \theta_1; \mathcal{V}ar_1)$ *such that* $\tau_1' \rightarrowtriangle \tau' = \tau_1\psi_1$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_1 \circ \theta_1$ *for some* ψ_1.

  In particular $\Gamma\phi = \Gamma\theta_1\psi_1$ *so by induction hypothesis,* $\mathsf{W}^{\ll}(\Gamma\theta_1; U_2; \mathcal{V}ar_1) = (\tau_2; \theta_2; \mathcal{V}ar_2)$ *such that* $\tau_1' = \tau_2\psi_2$ *and* $\psi_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_2 \circ \theta_2$ *for some* ψ_2.

  Then $\psi_3 = \psi_2 \circ \{\alpha/\tau'\}$ *is a unifier for* $\tau_1\theta_2 = \tau_2 \rightarrowtriangle \alpha$ *: we have indeed* $\tau_1\theta_2\psi_3 = \tau_1\theta_2\psi_2 = \tau_1\psi_1 = \tau_1' \rightarrowtriangle \tau' = \tau_2\psi_2 \rightarrowtriangle \alpha\{\alpha/\tau'\} = (\tau_2 \rightarrowtriangle \alpha)\psi_3$*. Thus the most general unifier is some* ψ_4 *such that* $\psi_3 = \psi \circ \psi_4$*. We can now conclude: indeed* $\tau' = \alpha\psi_3 = \alpha\psi_4\psi = \tau\psi$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_1 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_2 \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_3 \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi \circ \psi_4 \circ \theta_2 \circ \theta_1\psi \circ \theta$.

- *if* $U = [P \ll U_2]U_1$ *then there is some* τ_1' *and* $\Delta = \overline{X : \alpha_X}$ *such that* $\Gamma\phi \vdash_{\mathsf{U}} U_2 : \tau_1'$ *and* $\Gamma\phi, \Delta\phi \vdash_{\mathsf{U}} P : \tau_1'$ *and* $\Gamma\phi, \mathsf{Gen}(\Delta\phi; \Gamma) \vdash_{\mathsf{U}} U_1 : \tau_2$ *(where* ϕ *has been suitably extended on* $\mathsf{Dom}(\Delta)$*).*

  By induction hypothesis $\mathsf{W}^{\ll}(\Gamma; U_2; \mathcal{V}ar) = (\tau_1; \theta_1; \mathcal{V}ar_1)$ *with* $\tau_1' = \tau_1\psi_1$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_1 \circ \theta_1$ *for some* ψ_1.

  In particular $\mathsf{Dom}(\theta_1) \cap \mathsf{Dom}(\Delta) = \emptyset$ *and so* $\Gamma\phi, \Delta\phi = (\Gamma\theta_1, \Delta)\psi_1$*. By induction hypothesis,* $\mathsf{W}^{\ll}(\Gamma\theta_1, \Delta; P; \mathcal{V}ar_1 \setminus \{\overline{\alpha_X}\}) = (\tau_2; \theta_2; \mathcal{V}ar_2)$ *with* $\tau_1' = \tau_2\psi_2$ *and* $\psi_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_2 \circ \theta_2$ *for some* ψ_2.

  Then we have $\tau_1\theta_2\psi_2 = \tau_1\psi_1 = \tau_1' = \tau_2\psi_2$*, thus the equation* $\tau_1\theta_2 = \tau_2$ *admits a most general unifier* μ *such that* $\psi_2 = \psi_3 \circ \mu$ *for some* ψ_3.

  So far we have seen that $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_1 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_2 \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_3 \circ \mu \circ \theta_2 \circ \theta_1$*. Thus,* $\Gamma\phi, \Delta\phi = (\Gamma\theta_1\theta_2\mu, \Delta\theta_2\mu)\psi_3$ *so by induction hypothesis* $\mathsf{W}^{\ll}$ *succeeds on* U_1 *and returns* $(\tau_3; \theta_3; \mathcal{V}ar_3)$ *such that* $\tau' = \tau_3\psi_4$ *and* $\psi_3 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_4 \circ \theta_3$ *for some* ψ_4.

  Finally, the algorithm returns $\tau = \tau_3$ *so indeed* $\tau' = \tau\psi_4$ *and* $\phi \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_3 \circ \mu \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_4 \circ \theta_3 \circ \mu \circ \theta_2 \circ \theta_1 \overset{\mathcal{V}ar}{=}_{\not\ll} \psi_4 \circ \theta$.

$\square$

Theorem 3.5.4 (Decidability of Type Inference for uRhoF$_{\ll}$)
For a closed term U *such that* $\mathsf{stk} \notin U$*, the following problems are decidable:*

1. *Type Inference: given* Γ *(gives meaning to constants), is there a* τ *such that* $\Gamma \vdash_{\mathsf{U}} U : \tau$ *?*

2. *Type Checking: given* Γ *and* τ'*, does the judgement* $\Gamma \vdash_{\mathsf{U}} U : \tau'$ *hold ?*

Proof.

1. *By soundness and completeness, we have* $\exists\tau,\ \Gamma \vdash_{\mathsf{U}} U : \tau \iff \mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) \neq \mathsf{false}$

2. By soundness and principality, this judgement is equivalent to

$$\mathsf{W}^{\ll}(\Gamma; U; \mathcal{V}ar) = (\tau; \theta; \mathcal{V}ar) \ \wedge \ \tau' = \tau\psi \text{ with } \mathsf{Dom}(\psi) \subseteq \mathcal{V}ar$$

Since matching is decidable in the language of types, this problem is decidable.

□

Notice that all the examples we gave previously belong to the sublanguage $\mathsf{uRhoF}_{\ll}$. We conjecture that any "reasonable" query (*i. e.* the ones that are really used in semantic web programs) can be represented using only this restricted form of polymorphism.

3.6 Related Work and Conclusions

In this work we presented two systems, the Fully-typed Polymorphic Rho-calculus (RhoF) and the Type Inference Polymorphic Rho-calculus (uRhoF): both systems enjoy subject reduction of typable terms. RhoF also enjoys the decidability of type checking and of type reconstruction.

Because of the decidability of type-checking in RhoF, customizing an existing rewriting-language with polymorphic-types seems an interesting alternative to validate code without limiting code expressiveness. From the point of view of type inference, the main motivation, in introducing uRhoF, is to find an easy way to validate code of many existing lines of rewrite-based algorithms via static analysis. Finally, we have studied a variant of uRhoF (called $\mathsf{uRhoF}_{\ll}$) featuring a restricted form of polymorphism *à la* Damas-Milner-Tofte and customized the well-known algorithm W of Damas-Milner [DM82].

4 Prescriptive typing: from CLP to Xcerpt

4.1 Introduction

One of main issues addressed by prescriptive typing is composition. By providing types to the signature of function and predicate symbols, one expresses the syntactic categories to which a function, a predicate, and in turn a complete module, is supposed to be applied. Prescriptive types are therefore an integral part of the programs and modules and constitute a discipline to compose them correctly.

The idea of specifying types for XML data is already well established and there exist standardized ways of declaring types such as DTDs [Ext] and XML Schema [Fe01]. Thus, a prescriptive type system for rule languages manipulating XML data seems to be the next natural step. Moreover, composition of sets, or modules, of rules also becomes import as the Web grows in complexity. For instance, one may use a set of modules to extract data from different repositories and then use another module to extract useful information from this data. A prescriptive type system would ensure that the composition of these modules is possible, and produces data that is correct w.r.t. a given specification.

Several rule languages for querying semi-structured data, such as Xcerpt [BS02a] or RuleML [BTW01] have their semantics inherited from the semantics of (constraint) logic languages. For example, Xcerpt rules are close to predicate clauses, with a head and a body. From this point of view, our prescriptive type system for CLP [FC01, Coq] provides a good basis for a type system for theses rule languages.

In this section, we explain the type system TCLP and its properties and show how the ideas of prescriptive typing of constraint logic programming (CLP) languages can be adapted to rule languages for querying and transforming semi-structured data. This is illustrated through the description of a type system for the Xcerpt language. At this occasion, we discuss some of the differences between terms manipulated by CLP programs and terms representing semi-structured data from the typing point of view.

The rest of this section is organized as follows. Section 4.2 describes the type algebra and section 4.3 presents the type system for CLP. In sections 4.4 and 4.5 we recall the theorems for expressing the consistency of the type system w.r.t. the execution model. In section 4.6, we discuss the adaptation of prescriptive type system for CLP to languages manipulating semi-structured data and derive a prescriptive type system for Xcerpt. In section 4.7 we discuss some issues about type checking and section 4.8 concludes.

4.2 Type Structure

4.2.1 Preliminaries

We define here the algebraic structure of quasi-lattices that we will use as a very general framework for representing types with complex subtype relationships.

Let $(E, \leq)$ be a partially ordered set. For a non empty subset S of E, we note $\downarrow S = \{x \in E | \forall y \in S\ x \leq y\}$ the set of lower bounds of S and $\uparrow S = \{x \in E | \forall y \in S\ y \leq x\}$ the set of upper bounds of S. For the empty set, $\downarrow\emptyset = \emptyset$ and $\uparrow\emptyset = \emptyset$. We note $\sqcap S$ (resp. $\sqcup S$) the greatest lower bound (resp. least upper bound) of S whenever it exists. A lower quasi-lattice (resp. upper quasi-lattice) is a partially ordered set where any finite subset having a lower (resp. upper) bound has a greatest lower bound (resp. a least upper bound). A quasi-lattice is an upper and a lower quasi-lattice.

Definition 4.2.1 (Complete quasi-lattice) *A partially ordered set is a* complete quasi-lattice *(in the sense of sets) if for all non empty subsets $S \subseteq E$, $\sqcap S$ exists whenever $\downarrow S \neq \emptyset$ and $\sqcup S$ exists whenever $\uparrow S \neq \emptyset$.*

4.2.2 Types

For typing CLP languages, we are interested in type languages allowing subtyping relations between type constructors of different arities, like $list(\alpha) \leq term$ for instance. This allows, for example, to type the application of metaprogramming predicates to homogeneous lists, by viewing these lists as terms. In general, such subtyping relations specify subtyping relations between specific arguments of the type constructors. For instance, by writing $k_1(\alpha, \beta) \leq k_2(\beta)$, we specify that types built with k_1 are subtypes of the ones built with k_2 when the second argument of k_1 and the argument of k_2 correspond, the first argument of k_1 being forgotten in the subtype relationship.

From a formal point of view, it is simpler (and more general) to express the relationship between arguments by working with a structure of labelled terms. In the formalism of Pottier [Pot00], each argument of a constructor is indicated by a type label instead of a position. Moreover, positive and negative type labels are distinguished in order to express the covariance or the contravariance of arguments *w.r.t.*the subtyping relation. Note that this notion of type label is distinct from the notion of label presented in section 2. In the following, when it is clear from the context, we will refer to "type labels" simply as "labels".

So let $\mathcal{L}^+$ and $\mathcal{L}^-$ be two disjoint countable sets of *labels*, we note $\mathcal{L} = \mathcal{L}^+ \uplus \mathcal{L}^-$. Let $(\mathcal{K}, \leq_\mathcal{K})$ be a complete quasi-lattice of type constructors. Let a be the *arity* function defined from $\mathcal{K}$ into the finite parts of $\mathcal{L}$. We denote by a^+ (resp. a^-) the function which associates the positive (resp. negative) labels to a constructor. We assume that there is at least one type constructor with an empty arity, k_0.

Definition 4.2.2 $(\mathcal{K}, \leq_\mathcal{K}, \mathcal{L}^+, \mathcal{L}^-, a)$ *is a* signature *if:*

1. *for all* $k_1 \leq_\mathcal{K} k_2 \leq_\mathcal{K} k_3$, $a(k_1) \cap a(k_3) \subseteq a(k_2)$.
2. *for all* $S \subseteq \mathcal{K}$, *if* $\sqcap S$ *exists, then* $a(\sqcap S) \subseteq \bigcup_{s \in S} a(s)$.
3. *for all* $S \subseteq \mathcal{K}$, *if* $\sqcup S$ *exists, then* $a(\sqcup S) \subseteq \bigcup_{s \in S} a(s)$.
4. *for all* $k_1 \leq_\mathcal{K} k_2$, *there exists* k *s.t.* $k_1 \leq_\mathcal{K} k \leq_\mathcal{K} k_2$ *and* $a(k) = a(k_1) \cap a(k_2)$.

Conditions 1, 2, 3 express the coherence of labels *w.r.t.*the order relation and are similar to the ones found in [Pot00] for lattices. Condition 4 is specific to quasi-lattices, its purpose is to forbid signatures like $k_1(\alpha) \leq_\mathcal{K} k_2(\beta)$ which do not induce a quasi-lattice structure for types. For example, if k_3 and k_4 are not comparable, then $k_2(k_3)$ and $k_2(k_4)$ have common lower bounds, like $k_1(k_3)$ and $k_1(k_4)$, but don't have a greatest common lower bound.

For a signature $(\mathcal{K}, \leq_\mathcal{K}, \mathcal{L}^+, \mathcal{L}^-, a)$, we note $\mathcal{L}^*$ the set of finite strings of labels, ϵ the empty string, "." the string concatenation and $|w|$ the length of w. We are interested in (possibly infinite) types formed upon $\mathcal{K}$, where the positions of subterms are defined by strings of labels.

Definition 4.2.3 *Let $(\mathcal{K}, \leq_\mathcal{K}, \mathcal{L}^+, \mathcal{L}^-, a)$ be a signature. A (possibly infinite)* type *is a partial mapping τ from $\mathcal{L}^*$ into $\mathcal{K}$ such that:*

1. *Its domain is prefix closed:* $\forall w = w_1.w_2 \in dom(\tau), w_1 \in dom(\tau)$.

2. $\epsilon \in dom(\tau)$.

3. *For all positions* $w \in dom(\tau)$, *for all* $l \in \mathcal{L}$, $w.l \in dom(\tau)$ *if and only if* $l \in a(\tau(w))$.

We note $\mathcal{T}(\mathcal{S})$ the set of (possibly infinite) types built upon the signature $\mathcal{S}$. In the following, we assume a fixed signature $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, a)$ and we note $\mathcal{T} = \mathcal{T}(\mathcal{S})$ the set of types built upon $\mathcal{S}$. We note τ/w the type $\tau' : v \mapsto \tau(w.v)$, that is the subterm occurring at position w in τ.

4.2.3 Subtyping ordering

The subtyping relation $\leq$ is defined over types, as the intersection of a sequence $(\leq_n)$ of preorders over types defined by:

- $\leq_0 = \mathcal{T} \times \mathcal{T}$
- $\tau \leq_{n+1} \tau'$ if $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$ and for all labels $l \in a(\tau(\epsilon)) \cap a(\tau'(\epsilon))$:
  - either $l \in \mathcal{L}^+$ and $\tau/l \leq_n \tau'/l$
  - or $l \in \mathcal{L}^-$ and $\tau'/l \leq_n \tau/l$
- $\leq = \bigcap_{n \in \mathbb{N}} \leq_n$

Proposition 4.2.1 $\leq$ *is an order over* $\mathcal{T}$.

Proposition 4.2.2 *Let* $\tau_1, \tau_2 \in \mathcal{T}$. $\tau_1 \leq \tau_2$ *if and only if* $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon)$ *and for all labels* $l \in a(\tau_1(\epsilon)) \cap a(\tau_2(\epsilon))$*:*

- *either* $l \in \mathcal{L}^+$ *and* $\tau_1/l \leq \tau_2/l$
- *or* $l \in \mathcal{L}^-$ *and* $\tau_2/l \leq \tau_1/l$

Theorem 4.2.1 *[CF03]* $(\mathcal{T}, \leq)$ *is a complete quasi-lattice.*

4.2.4 Subtyping constraints

Let $\mathcal{W}$ be a countable set of *type variables*, or *parameters*, noted $\alpha, \beta, \ldots$. Types with variables are defined as the set, noted $\mathcal{T}_{\mathcal{W}}$, of (possibly infinite) types built upon the signature $(\mathcal{K} \cup \mathcal{W}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, a)$. A subtyping constraint is a pair of *finite* types t_1 and t_2 in $\mathcal{T}_{\mathcal{W}}$ and is noted $\tau_1 \leq \tau_2$. For a system C of subtyping constraints, we note $V(C)$ the set of variables occurring in C.

Definition 4.2.4 *A substitution* $\rho : \mathcal{W} \to \mathcal{T}$ satisfies *the constraint* $\tau_1 \leq \tau_2$, *noted* $\rho \models \tau_1 \leq \tau_2$, *if* $\rho(\tau_1) \leq \rho(\tau_2)$. *The subtyping constraint* $\tau_1 \leq \tau_2$ *is* satisfiable *if there exist a substitution* ρ *such that* $\rho \models \tau_1 \leq \tau_2$.

Theorem 4.2.2 *[CF03] The satisfiability problem for subtyping constraints in quasi-lattices with a finite number of extrema each with an empty arity is NP-complete.*

4.3 Typed CLP Programs

CLP programs are built over a denumerable set $\mathcal{V}$ of *variables*, a finite set $\mathcal{F}$ of *function* symbols, given with their arity (constants are functions of arity 0), and a finite set $\mathcal{P}$ of *program predicate* and *constraint predicate* symbols given with their arity, containing the equality constraint $=$. A query Q is a finite sequence of constraints and atoms. A program clause is an expression noted $A \leftarrow Q$ where A is an atom formed with a program predicate and Q a query. The set variables occurring in a syntactic object O is noted $V(O)$.

A *type scheme* is an expression of the form $\forall\overline{\alpha}\tau_1, \ldots, \tau_n \rightarrow \tau$, where $\overline{\alpha}$ is the set of parameters in types $\tau_1, ..., \tau_n, \tau$. We assume that each function symbol $f \in \mathcal{F}$, has a *declared type scheme* of the form $\forall\overline{\alpha}\tau_1, \ldots, \tau_n \rightarrow \tau$, where n is the arity of f, and τ is a flat type. Similarly, we assume that each predicate symbol $p \in \mathcal{P}$ has a declared type scheme of the form $\forall\overline{\alpha}\tau_1, \ldots, \tau_n \rightarrow pred$ where n is the arity of p. The declared type of the equality constraint symbol is $\forall u\ u, u \rightarrow pred$. For notational convenience, the quantifiers in type schemes and the resulting type $pred$ of predicates will be omitted in type declarations, the declared type schemes will be indicated by writing $f_{\tau_1 \ldots \tau_n \rightarrow \tau}$ and $p_{\tau_1 \ldots \tau_n}$, assuming a fresh renaming of the parameters in $\tau_1, \ldots, \tau_n, \tau$ for each occurrence of f or p.

Throughout this paper, we assume that $\mathcal{F}$, and $\mathcal{P}$ are fixed by means of declarations in a *typed program*, where the syntactical details are insignificant for our results.

A *variable typing* is a mapping from a finite subset of $\mathcal{V}$ to $\mathcal{T}_{\mathcal{W}}$, written as $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. The type system defines well-typed terms, atoms and clauses relatively to a variable typing U. The typing rules are given in Table 1. The rules basically consist of the rules of Mycroft and O'Keefe plus a subtyping rule. Note that for the sake of simplicity constraints are not distinguished from other atoms in this system.

An object, say a term t, is *well-typed* if there exist some variable typing U and some type τ such that $U \vdash t : \tau$. Otherwise the term is *ill-typed* (and likewise for atoms, etc.). A program is well-typed if all its clauses are well-typed.

The distinction between rules *Head* and *Atom* expresses the usual *definitional genericity* principle [LR91] which states that the type of a defining occurrence of a predicate (i.e. at the left of "$\leftarrow$" in a clause) must be equivalent up-to renaming to the assigned type of the predicate. The rule *Head* used for deriving the type of the head of the clause is thus not allowed to use substitutions other than variable renamings in the declared type of the predicate. For example, the predicate $member$ can be typed polymorphically, i.e. $member : \alpha \times list(\alpha) \rightarrow pred$, if its definition does not contain special facts like $member(1, [1])$, that would force its type to be $member : int \times list(int) \rightarrow pred$, for satisfying the definitional-genericity condition.

The following proposition shows that if an expression other than a clause or a head is well-typed in a variable typing U, it remains well-typed in any instance $U\rho$.

Proposition 4.3.1 *For any variable typing U, any type judgement R other than a Head or a Clause, and any type substitution ρ, if $U \vdash R$ then $U\rho \vdash R\rho$.*

4.4 Subject Reduction w.r.t. CSLD Resolution

Subject reduction is the property that evaluation rules transform a well-typed expression into another well-typed expression. The evaluation rule for constraint logic programming is CSLD-resolution. To recall this evaluation rule, it is convenient to distinguish in a query Q, the constraint part c (where the sequence denotes the conjunction) from the other sequence of atoms $\mathcal{A}$. We use the notation $Q = c|\mathcal{A}$ to make this distinction. Given a constraint domain $\mathcal{X}$

(Sub)	$\frac{U \vdash t:\tau \quad \tau \leq \tau'}{U \vdash t:\tau'}$	
(Var)	$\{x:\tau,\ldots\} \vdash x:\tau$	
(Func)	$\frac{U \vdash t_1:\tau_1\rho \ \ldots \ U \vdash t_n:\tau_n\rho}{U \vdash f_{\tau_1\ldots\tau_n\to\tau}(t_1,\ldots,t_n):\tau\rho}$	ρ is a type substitution
(Atom)	$\frac{U \vdash t_1:\tau_1\rho \ \ldots \ U \vdash t_n:\tau_n\rho}{U \vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n) Atom}$	ρ is a type substitution
(Head)	$\frac{U \vdash t_1:\tau_1\rho \ \ldots \ U \vdash t_n:\tau_n\rho}{U \vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n) Head}$	ρ is a renaming substitution
(Query)	$\frac{U \vdash A_1 \ Atom \ \ldots \ U \vdash A_n \ Atom}{U \vdash A_1,\ldots,A_n \ Query}$	
(Clause)	$\frac{U \vdash Q \ Query \quad U \vdash A \ Head}{U \vdash A \leftarrow Q \ Clause}$	

Table 1: The type system for CLP.

which fixes the interpretation of constraints, a query $c'|\mathcal{B}$ is a *CSLD-resolvent* of a query $c|\mathcal{A}$ and a (renamed apart) program clause $p(t_1,...,t_n) \leftarrow d|\mathcal{A}_p$, if
$\mathcal{A} = A_1,\ldots,A_{k-1},p(t'_1,\ldots,t'_n),A_{k+1},\ldots,A_m$,
$\mathcal{B} = A_1,\ldots,A_{k-1},\mathcal{A}_p,A_{k+1},\ldots,A_m$,
and the constraint $c' = (c \wedge d \wedge t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n)$ is $\mathcal{X}$-satisfiable.

Theorem 4.4.1 (Subject Reduction for CSLD resolution) *[FC01] Let P be a well-typed CLP($\mathcal{X}$) program, and Q be a well-typed query, i.e. $U \vdash Q$ Query for some variable typing U. If Q' is a CSLD-resolvent of Q, then there exists a variable typing U' such that $U' \vdash Q'$ Query.*

It is worth noting that the previous result would not hold without the definitional genericity condition (expressed in rule *Head*). For example with two constants $a:\tau_a$ and $b:\tau_b$, and one predicate $p:\alpha \to pred$ defined by the non definitional generic clause $p(a)$, we have that the query $p(b)$ is well typed, but $b = a$ is a resolvent that is ill-typed if τ_a and τ_b have no upper bound.

4.5 Subject Reduction w.r.t. Substitutions

The CSLD reductions, noted $\longrightarrow_{CSLD}$, are in fact an abstraction of the operational reductions that may perform also substitution steps, noted $\longrightarrow_\sigma$, instead of keeping equality constraints. As in the CLP scheme constraints are handled modulo logical equivalence [JL87], it is clear that the diagram of both reductions commutes :

$$
\begin{array}{ccccccc}
Q_1 & \longrightarrow_{CSLD} & Q_2 & \longrightarrow_{CSLD} & \cdots & \longrightarrow_{CSLD} & Q_n \\
\downarrow_\sigma & & \downarrow_\sigma & & & & \downarrow_\sigma \\
 & \longrightarrow_{CSLD} & & & \cdots & & \\
 & & \downarrow_\sigma & & & & \downarrow_\sigma \\
 & & & \longrightarrow_{CSLD} & & & \\
 & & & & \cdots & & \cdots \\
 & & & & & & \downarrow_\sigma \\
 & & & & & & Q
\end{array}
$$

However the previous subject reduction result expresses the consistency of types w.r.t. horizontal reduction steps only, that is w.r.t. the abstract execution model which accumulates constraints, but may not hold for more concrete operations of constraint solving and substitutions. For example, with the subtype relations $int \leq term$, $pred \leq term$, the type declarations $=: \alpha \times \alpha \rightarrow pred$, $p : int \rightarrow pred$, and the program $p(X)$, the query $Y = true, p(Y)$ is well typed with $Y : int$, and succeeds with $Y = true$, although the query obtained by substitution, $p(true)$, is ill-typed.

In order to establish subject reduction for substitution steps, and be consistent with the semantical equivalence of programs, one may consider a typed execution model with type constraints on variables checked at runtime, as in [FC01]. In the example, the type constraint $Y : int$ with the constraint $Y = true$ is unsatisfiable, the query can thus be rejected at compile-time by checking the satisfiability of its typed constraints.

Another way to establish subject reduction for substitution steps, is to consider logic programs with a fixed data flow, given by modes, as in [SFD00]. The remaining part of this section recalls the main results of [SFD00].

First, we recall the notion of principal variable typing. Intuitively, a variable typing U is principal w.r.t. a term t and a type τ if it associates to each variable X of t the least instantiated and greatest (using $\leq$) possible type such that $U \vdash t : \tau$. More precisely:

Definition 4.5.1 *A variable typing U is principal for t and τ if $U \vdash t : \tau$ and for each variable typing U' such that $U' \vdash t : \tau$, there exists a type substitution ρ such that for all variable $X \in V(t)$, $U'(X) \leq U(X)\rho$.*

We will generalise concepts previously defined for terms to term *vectors*. In particular, we consider principal variable typings for a term vector $\bar{t}$ and a type vector $\bar{\tau}$. Conceptually, one could think of introducing special functors into the typed language so that any vector can be represented as an ordinary term.

Now, we define modes, which are a common concept used for verification [Apt97]. For a predicate p/n, a *mode* is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. Positions with I are called *input positions*, and positions with O are called *output positions* of p. We assume that a fixed mode is associated with each predicate in a program. To simplify the notation, an atom written as $p(\bar{s}, \bar{t})$ means: $\bar{s}$ is the vector of terms filling the input positions, and $\bar{t}$ is the vector of terms filling the output positions.

Definition 4.5.2 *Consider a derivation step where $p(\bar{s}, \bar{t})$ is the selected atom and $p(\bar{w}, \bar{v})$ is the renamed apart clause head. The equation $p(\bar{s}, \bar{t}) = p(\bar{w}, \bar{v})$ is* solvable by moded unification *if there exist substitutions θ_1, θ_2 such that $\bar{w}\theta_1 = \bar{s}$ and $V(\bar{t}\theta_1) \cap V(\bar{v}\theta_1) = \emptyset$ and $\bar{t}\theta_1\theta_2 = \bar{v}\theta_1$.*

A derivation where all unifications are solvable by moded unification is a moded *derivation.*

Moded unification is a special case of *double matching*. How moded derivations are ensured is not our problem here, and we refer to [AE93]. Note that the requirement of moded derivations is stronger than *input-consuming derivations* [Sma99] where it is only required that the MGU does not bind $\bar{s}$.

Definition 4.5.3 *A query $Q = p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ is* nicely moded *if $\bar{t}_1, \ldots, \bar{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$*

$$V(\bar{s}_i) \cap \bigcup_{j=i}^{n} V(\bar{t}_j) = \emptyset. \tag{1}$$

The clause $C = p(\bar{t}_0, \bar{s}_{n+1}) \leftarrow Q$ is nicely moded *if Q is nicely moded and*

$$V(\bar{t}_0) \cap \bigcup_{j=1}^{n} V(\bar{t}_j) = \emptyset. \tag{2}$$

A program is nicely moded *if all of its clauses are nicely moded.*

An atom $p(\bar{s}, \bar{t})$ is input-linear *if $\bar{s}$ is linear,* output-linear *if $\bar{t}$ is linear.*

Definition 4.5.4 *Let*

$$C = p_{\bar{\tau}_0, \bar{\sigma}_{n+1}}(\bar{t}_0, \bar{s}_{n+1}) \leftarrow p^1_{\bar{\sigma}_1, \bar{\tau}_1}(\bar{s}_1, \bar{t}_1), \ldots, p^n_{\bar{\sigma}_n, \bar{\tau}_n}(\bar{s}_n, \bar{t}_n)$$

be a clause. If C is nicely moded, $\bar{t}_0$ is input-linear, and there exists a variable typing U such that $U \vdash C$ Clause, and for each $i \in \{0, \ldots, n\}$, U is principal for $\bar{t}_i$ and $\bar{\tau}'_i$, where $\bar{\tau}'_i$ is the instance of $\bar{\tau}_i$ used for deriving $U \vdash C$ Clause, then we say that C is nicely typed.

A query Q is nicely typed *if the clause* $\mathsf{Go} \leftarrow Q$ *is nicely typed. A program is* nicely typed *if all of its clauses are nicely typed.*

Theorem 4.5.1 (Subject reduction) *[SFD00] Let C and Q be a nicely typed clause and query. If Q' is a resolvent of C and Q where the unification of the selected atom and the clause head is solvable by moded unification, then Q' is nicely typed.*

4.6 Typed Xcerpt Programs

In this subsection, we study the adaptation of the type system we presented for constraint logic programs to reasoning and query languages for the (semantic) web. As an example of such languages, we consider the language Xcerpt [BS02a]. A subset of Xcerpt is described in subsection 2.6.1. In addition to this subset, we allow grouping construct *all* and *some* [BS02a] to appear in construct terms. *all* t stands for the vector of data terms corresponding to all possible answers of the query part, whereas *some* n t stands for a vector of data terms corresponding to n non-deterministically chosen answers of the query part.

Example 4.6.1 *Let us consider the data term corresponding to a CD collection from example 2.6.2:*

$$\begin{array}{ll} \textit{catalogue}[& \textit{cd}[\,\textit{title}[\,\textit{"Empire Burlesque"}]\,\textit{artist}[\,\textit{"Bob Dylan"}]\,\textit{year}[\,\textit{"1985"}]\,] \\ & \textit{cd}[\,\textit{title}[\,\textit{"Hide your heart"}]\,\textit{artist}[\,\textit{"Bonnie Tyler"}]\,\textit{year}[\,\textit{"1988"}]\,] \\ & \textit{cd}[\,\textit{title}[\,\textit{"Stop"}]\,\textit{artist}[\,\textit{"Sam Brown"}]\,\textit{year}[\,\textit{"1988"}]\,]\,] \end{array}$$

The following rule extracts all the titles in the collection:

$$\begin{array}{l} \textit{result}[\,\textit{all title}[\,\textit{TITLE}]\,] \;\leftarrow \\ \qquad \textit{catalogue}\{\{\,\textit{cd}\{\{\,\textit{title}[\,\textit{TITLE}]\,\}\}\,\}\} \end{array}$$

Thus, the result returned by the rule is

$$\textit{result}[\,\textit{title}[\,\textit{"Empire Burlesque"}]\,\textit{title}[\,\textit{"Hide your heart"}]\,\textit{title}[\,\textit{"Stop"}]\,]$$

The following rule extracts any two titles in the collection:

$$\begin{array}{l} \textit{result}[\,\textit{some 2 title}[\,\textit{TITLE}]\,] \;\leftarrow \\ \qquad \textit{catalogue}\{\{\,\textit{cd}\{\{\,\textit{title}[\,\textit{TITLE}]\,\}\}\,\}\} \end{array}$$

The results returned by the rule are:

$$\begin{array}{l} \textit{result}[\,\textit{title}[\,\textit{"Empire Burlesque"}]\,\textit{title}[\,\textit{"Hide your heart"}]\,] \\ \textit{result}[\,\textit{title}[\,\textit{"Empire Burlesque"}]\,\textit{title}[\,\textit{"Stop"}]\,] \\ \textit{result}[\,\textit{title}[\,\textit{"Hide your heart"}]\,\textit{title}[\,\textit{"Stop"}]\,] \end{array}$$

The major difference between these languages and constraint logic languages is that they deal with semi-structured data. In particular, function symbols come with their arity. For example $f/1$ is different from $f/2$. On the contrary, tags in XML (or labels in Xcerpt terms) don't have a fixed arity. For example, the label f is the same in the data terms $f[a]$ and $f[a,b]$. Since the type scheme of a function symbol depends on its arity, we need an other notion of type scheme: instead of using a vector of type to specify the type of the arguments of a label, we use a regular expression of types, as it is done in *e.g.* DTDs [Ext]. We consider the following regular expression operators: * (repetition), $\mid$ (alternation) and ? (optionality).

Definition 4.6.1 *A* type regular expression*, noted Λ is a regular expression over the alphabet of types with variables $\mathcal{T}_{\mathcal{W}}$. The set of type vectors matching a type regular expression Λ is noted $L(\Lambda)$.*

Now we define some useful relations over regular expressions and vectors:

Definition 4.6.2 *The order $\sqsubseteq$ over type regular expressions is the inclusion order of type regular expressions: $\Lambda_1 \sqsubseteq \Lambda_2$ if $L(\Lambda_1) \subseteq L(\Lambda_2)$.*

For example, $(\sigma\tau)^* \sqsubseteq (\sigma \mid \tau)^*$, while $(\sigma \mid \tau)^* \not\sqsubseteq \sigma^*\tau^*$, since $\tau\sigma \in L((\sigma \mid \tau)^*)$ and $\tau\sigma \notin L(\sigma^*\tau^*)$.

Definition 4.6.3 *The order $\leq$ is extended over type regular expressions in the following way: $\Lambda_1 \leq \Lambda_2$ if for each type vector $\bar{\tau_1} \in L(\Lambda_1)$ there exists a type vector $\bar{\tau_2} \in L(\Lambda_2)$ such that $\bar{\tau_1} \leq \bar{\tau_2}$ and for each $\bar{\tau_2} \in L(\Lambda_2)$, there exists $\bar{\tau_1} \in L(\Lambda_1)$ such that $\bar{\tau_1} \leq \bar{\tau_2}$.*

For example, if $\sigma \leq \sigma'$ and $\tau \leq \tau'$, then $(\sigma\tau)^* \leq (\sigma'\tau')^*$.

Definition 4.6.4 *Let $(\Lambda_1, \ldots, \Lambda_m)$ and $(\Lambda'_1, \ldots, \Lambda'_n)$ be two vectors of type regular expressions. $(\Lambda_1, \ldots, \Lambda_m)$ is a subvector of $(\Lambda'_1, \ldots \Lambda'_n)$ if there exists $k_1, \ldots, k_m$ such that $1 \leq k_1 < k_2 < \ldots < k_m \leq n$ and for each $i \in \{1, \ldots, m\}$, $\Lambda_i = \Lambda'_{k_i}$.*

For example, $(\sigma, (\sigma \mid \tau))$ is a subvector of $(\tau, \sigma, (\tau\sigma)^*, (\sigma \mid \tau))$, while (σ, τ) is not.

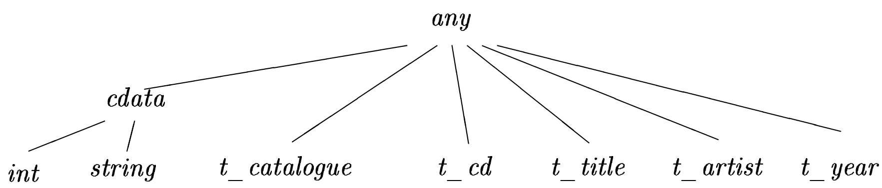
The application of a type substitution to a type regular expression Λ is defined as the application of the substitution on all the types occurring in Λ:

Definition 4.6.5 *The application of a type substitution ρ to a regular type expression Λ is inductively defined as follows: $(\tau)\rho = \tau\rho$, $(\Lambda_1\Lambda_2)\rho = \Lambda_1\rho\Lambda_2\rho$, $(\Lambda_1 \mid \Lambda_2)\rho = \Lambda_1\rho \mid \Lambda_2\rho$, $\tau?\rho = \tau\rho?$ and $(\Lambda^*)\rho = \Lambda\rho^*$.*

A *regular type scheme* is an expression of the form $\forall\bar{\alpha}\Lambda \to \tau$, where $\bar{\alpha}$ is the set of parameters occurring in Λ or τ. We assume that each label has a *declared regular type scheme* $\forall\bar{\alpha}\Lambda \to \tau$. For notational convenience, the quantifiers in type schemes will be omitted and a label l together with its declared regular type scheme will be noted $l_{\Lambda\to\tau}$.

We assume the existence of a type *any* that is greater than all other types.

Example 4.6.2 *Let us consider the following type structure:*



Here, we give type declarations for the labels used in example 4.6.1:

$catalogue : t\_cd^* \to t\_catalogue$

$cd : t\_title\ t\_artist\ t\_year \to t\_cd$

$title : string \to t\_title$

$artist : string \to t\_artist$

$year : int \to t\_year$

$result : t\_title^* \to any$

A variable typing U is a mapping from a finite set of variables to non empty finite sets of types, written $\{X_1 : \tau_1^1 \mid \ldots \mid \tau_1^{m_1}, \ldots, X_n : \tau_n^1 \mid \ldots \mid \tau_n^{m_n}\}$. When it is appropriate, we will identify the set of types associated to a given variable by a variable typing U to the type regular expression denoting this set. As opposed to variable typings for CLP, a variable may have different types depending on the regular type scheme of the label of the term they occur in. For example, if we consider the query term $cd\{\{X\}\}$, the variable X may match terms of type t_title, t_artist or t_year. Moreover, it cannot have type any, which is the lowest upper bound of these three types.

The system defines well-typed data, query and construct terms as well as well-typed rules. Typing rules are given in table 2. Judgements are of the from $U \vdash t : \Lambda$, read as "In the variable typing U, t is a vector of terms whose type matches Λ". The need for dealing with vectors of terms instead of single terms resides in handling the grouping construct *all* and *some*, which are used to build sets of terms in the head of Xcerpt rules.

Lemma 4.6.1 *If t is a query term, and there exists a variable typing U such that $U \vdash t : \Lambda$ for some type regular expression Λ then Λ is of the form $\tau_1 \mid \ldots \mid \tau_n$.*
If t is a construct term then Λ is either of the form $(\tau_1 \mid \ldots \mid \tau_n)^m$ or $((\tau_1 \mid \ldots \mid \tau_n)^m)^*$.

Proof. By induction on the derivation of query term and construct terms, and by remarking that if $\Lambda' \leq \Lambda$, then Λ' is equivalent to a regular expression of the same form that Λ. $\square$

The rule (As) simply expresses that is if a variable must be matched by terms corresponding to a specific shape, given by a query term, then the variable and this query term must have the same type. In particular, using lemma 4.6.1, one can remark that Λ indeed corresponds to a non empty finite set of types.

As judgements associates type regular expressions to terms in a given variable typing, rules (*Compl Ord*), (*Compl Unord*), (*Incompl Ord*), (*Incompl Unord*), (*Head Ord*) and (*Head Unord*) use the order $\sqsubseteq$ over type regular expressions instead of simply checking that a vector of types matches the label's type regular expression given for its arguments. A permutation of the type regular expressions of the arguments is used in rules (*Compl Unord*), (*Incompl Unord*) and (*Head Unord*) to signify that the arguments of the term are not ordered. A more precise rule would have been to check whether the language recognized by $\Lambda_1 \ldots \Lambda_n$ is included in the language of type vectors that are permutations of type vectors matching Λ. However, this last language is not regular. For example, the language of words that are permutations of words matching $(ab)^*$ is the set of words containing the same number of a's and b's, which is not regular. Similarly to the rule (*Head*), the rules (*Head Ord*) and (*Head Unord*) use a renaming instead of a substitution to express the principle of definitional genericity [LR91].

The rule (*Desc*) simply expresses that the query term t is well-typed. In particular, the possibility to give any type to the expression *desc* t express that the type of t is not related to the type of the term t' it appears in, as t may occur at an arbitrary depth in t'.

The rule (*All*) (resp. (*Some*)) expresses that the grouping construct *all* (resp. *some* n) stand for vectors of terms of an arbitrary size (resp. of size n).

4.7 Type checking

In this subsection we discuss some issues related to the type checking of Xcerpt programs.

The type system presented in table 2 is non-deterministic, since the rule (*Sub*) can be used anywhere in a typing derivation. A deterministic syntax-directed system can be obtained by replacing the rule (*Sub*) by variants of other rules with subtype relations in their premises, similarly to case of the type system for CLP [FC01]. For example, the rule (*Compl Ord'*) below is the variant (*Compl Ord*):

$$(Compl\ Ord') \quad \frac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}[t_1 \ldots t_n] : \tau\rho} \quad \begin{array}{r}\rho \text{ is a type substitution}\\ (\Lambda_1 \ldots \Lambda_n) \leq (\Lambda'_1 \ldots \Lambda'_n)\\ \text{and } (\Lambda'_1 \ldots \Lambda'_n) \sqsubseteq \Lambda\rho\end{array}$$

(*Sub*)	$\dfrac{U \vdash t : \Lambda \quad \Lambda \leq \Lambda'}{U \vdash t : \Lambda'}$	
(*Var*)	$\dfrac{X : \tau_1 \mid \ldots \mid \tau_n \in U}{U \vdash X : \tau_1 \mid \ldots \mid \tau_n}$	
(*As*)	$\dfrac{U \vdash t : \Lambda \quad X : \Lambda \in U}{U \vdash X \rightsquigarrow t : \Lambda}$	
(*Compl Ord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}[t_1 \ldots t_n] : \tau\rho}$	ρ is a type substitution and $(\Lambda_1 \ldots \Lambda_n) \sqsubseteq \Lambda\rho$
(*Compl Unord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}\{t_1 \ldots t_n\} : \tau\rho}$	ρ is a type substitution $(\Lambda'_1, \ldots, \Lambda'_n)$ is a permutation of $(\Lambda_1, \ldots, \Lambda_n)$ and $(\Lambda'_1 \ldots \Lambda'_n) \sqsubseteq \Lambda\rho$
(*Incompl Ord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}[[t_1 \ldots t_n]] : \tau\rho}$	ρ is a type substitution $(\Lambda_1, \ldots, \Lambda_n)$ is a subvector of $(\Lambda'_1, \ldots, \Lambda'_m)$ and $(\Lambda'_1 \ldots \Lambda'_n) \sqsubseteq \Lambda\rho$
(*Incompl Unord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}\{\{t_1 \ldots t_n\}\} : \tau\rho}$	ρ is a type substitution $(\Lambda_1, \ldots, \Lambda_n)$ is a subvector of a permutation of $(\Lambda'_1, \ldots, \Lambda'_m)$ and $(\Lambda'_1 \ldots \Lambda'_n) \sqsubseteq \Lambda\rho$
(*Desc*)	$\dfrac{U \vdash t : \Lambda}{U \vdash \mathit{desc}\ t : \tau}$	for any type τ
(*All*)	$\dfrac{U \vdash t : \Lambda}{U \vdash \mathit{all}\ t : \Lambda^*}$	
(*Some*)	$\dfrac{U \vdash t : \Lambda}{U \vdash \mathit{some}\ n\ t : \Lambda^n}$	
(*Head Ord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}[t_1 \ldots t_n]\ \mathit{Head}}$	ρ is a renaming type substitution and $(\Lambda_1 \ldots \Lambda_n) \sqsubseteq \Lambda\rho$
(*Head Unord*)	$\dfrac{U \vdash t_1 : \Lambda_1 \ \ldots \ U \vdash t_n : \Lambda_n}{U \vdash l_{\Lambda \to \tau}\{t_1 \ldots t_n\}\ \mathit{Head}}$	ρ is a renaming type substitution $(\Lambda'_1, \ldots, \Lambda'_n)$ is a permutation of $(\Lambda_1, \ldots, \Lambda_n)$ and $(\Lambda'_1 \ldots \Lambda'_n) \sqsubseteq \Lambda\rho$
(*Rule*)	$\dfrac{U \vdash t_c\ \mathit{Head} \quad U \vdash t_q : \mathit{any}}{U \vdash t_c \leftarrow t_q\ \mathit{Rule}}$	

Table 2: The type system for Xcerpt.

Type checking thus requires to solve the following problem: given two type regular expression Λ_1 and Λ_2, is there a type substitution ρ and a type regular expression Λ' such that $\Lambda_1 \leq \Lambda' \sqsubseteq \Lambda_2\rho$? This problem combines subtyping, parametric polymorphism and regular expression matching. The problem of subtyping in parametrized regular tree languages has been studied in [HFC05], in the context of functional languages for processing semi-structured data.

In the general case, the inclusion problem for regular expressions is PSPACE-complete [SM73, Koz77]. However, restrictions on the form of the regular expressions reduces the complexity of the problem [MNS04]. In particular checking the inclusion of 1-unambiguous regular expressions [BKW98] is in P-TIME. Thus, it seems interesting to consider restrictions on allowed regular type schemes in order to reduce the complexity of the inclusion checking algorithms, as it is the case in *e.g.* DTDs [Ext] where regular expressions must be 1-unambiguous.

However, rules (*Incompl Ord*) and (*Incompl Unord*) require the type of the arguments to be a subvector of a vector matching the type regular expression of the label. Given a regular expression Λ, it possible to compute a regular expression Λ' whose language is exactly the language of words that are subwords (in the sense of subvectors) of words in the language of Λ. Given the DFA d corresponding to Λ, a finite automaton corresponding to Λ' is obtained by adding, for each states s and s' in d and each letter a such that $s \rightarrow^* \xrightarrow{a} s'$, the transition $s \xrightarrow{a} s'$. Unfortunately, the resulting expression Λ' may be more complex and not fulfil restrictions that may have been imposed on Λ. Moreover, rules (*Compl Unord*) and (*Incompl Unord*) also require to consider permutations of type regular expressions, which thus results in a supplementary combinatorial problem.

4.8 Conclusion

We presented a prescriptive type system for constraint logic languages and showed how it can be adapted into a type system for typing rule languages used in web applications. This resulted in a prescriptive type system for the rule language Xcerpt [BS02a] for transforming and querying XML documents. Finally, we discussed some issues related to type checking.

For future work, we intend to evaluate this prescriptive type system on REWERSE applications. In particular we wish to analyse the role of types for preventing the misuse of semantic web program components and the help provided to the user for writing complex queries involving different kinds of ontologies or reasoning.

We plan also to study the theoretical properties of the type system w.r.t. the different possible execution models of Xcerpt [BS02c]. We intend to develop practical algorithms for type checking. In particular, we need an algorithm to test whether, given two type regular expression Λ_1 and Λ_2, there exists a type substitution ρ and a type regular expression Λ' such that $\Lambda_1 \leq \Lambda' \sqsubseteq \Lambda_2\rho$, which combines subtyping, parametric polymorphism and regular expression matching. In order to reduce the possible combinatorial explosions in these algorithms, we may consider restrictions on the regular type schemes of labels.

5 Final remarks

We are in contact with other working groups of the project, learning what are their needs related to type systems. We expect the approaches presented here to be adjusted and developed accordingly. As a first step, examples of applying them to representative examples obtained from the other groups are to be developed. This should show which kind of errors the typing approaches are able to discover, and how types can be employed in structuring and composition of rule applications.

References

[AE93] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, pages 1–19. Springer-Verlag, 1993.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Bar84] H. Barendregt. *Lambda Calculus: its Syntax and Semantics.* North Holland, 1984.

[Bar92] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[BBSW03] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data. In *Proceedings of the 29th Intl. Conference on Very Large Databases (VLDB03) – Demonstrations Track*, Berlin, Germany, 2003.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proc. of POPL*, pages 250–261. The ACM press, 2003.

[BDM04] F. Bry, W. Drabent, and J. Maluszynski. On subtyping of tree-structured data: A polynomial approach. In Ohlbach and Schaffert [OS04], pages 1–18.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.

[BS02a] F. Bry and S. Schaffert. A gentle introduction into Xcerpt, a rule-based query and transformation language for XML. Technical Report PMS-FB-2002-11, Computer Science Institute, Munich, Germany, 2002. Invited article at International Workshop on Rule Markup Languages for Business Rules on the Semantic Web.

[BS02b] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. of the International Conference on Logic Programming*, LNCS. Springer-Verlag, 2002.

[BS02c] F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Proc. 2nd Int. Workshop "Web and Databases"*, LNCS 2593, Erfurt, Germany, October 2002. Springer-Verlag.

[BTW01] H. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for semantic web rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*, Stanford, 2001. `http://www.dfki.uni-kl.de/ruleml`.

[CCD+04] H. Cirstea, E. Coquery, W. Drabent, F. Fages, C. Kirchner, J. Maluszynski, and B. Wack. Types for Web rule languages: a preliminary study. Deliverable I3-D2, REWERSE, 2004.

[CDG+99] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata/, 1999.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[CF03] E. Coquery and F. Fages. Subtyping constraints in quasi-lattices. In P. Pandya and J. Radhakrishnan, editors, *Proceedings of the 23rd conference on foundations of software technology and theoretical computer science, FSTTCS'2003*, LNCS, Mumbai, India, December 2003. Springer-Verlag.

[CHS72] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland, Amsterdam, 1972.

[Cir00] H. Cirstea. *Rewriting Calculus: Foundations and Applications.* PhD thesis, Université Henri Poincaré - Nancy I, 2000.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[CKL02] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Proc. of WRLA*, volume 71 of *ENTCS*, 2002.

[CKLW02] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. The Rho Cube : Some Results, Some Problems. In *Proc. of HOR*, 2002. Also as LORIA Research Report A02-R-470.

[CLW04] H. Cirstea, L. Liquori, and B. Wack. Rho-calculus with Fixpoint: First-order System. In *Proc. of TYPES*. Springer-Verlag, 2004.

[CM01] J. Clark and M. Murata (editors). RELAX NG specification, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[CM02] M. Clavel and J. Meseguer. Reflection in Conditional Rewriting Logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[Coq] E. Coquery. TCLP. `http//contraintes.inria.fr/~coquery/tclp/`.

[Cur34] H. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A.*, volume 20, pages 584–590, 1934.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Proc. of POPL*, pages 207–212. The ACM Press, 1982.

[DM01] W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs — a declarative approach. In P. Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Proceedings*, volume 2237 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[DMP02] W. Drabent, J. Maluszynski, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–610, 2002.

[DPR98] Dovier, Policriti, and Rossi. A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. *Fundamenta Informatica*, 36, 1998.

[Ext] Extensible markup language (XML) 1.0 (second edition), W3C recommendation 6 October 2000. `http://www.w3.org/TR/REC-xml`.

[FBS+04] T. Furche, F. Bry, S. Schaffert, R. Orsini, I. Horrocks, M. Krauss, and O. Bolzer. Survey over Existing Query and Transformation Languages. deliverable I4-D1, Institute for Informatics, Ludwig-Maximilians-Universität München, 2004.

[FC01] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, November 2001.

[Fe01] D. C. Fallside (ed.). XML Schema part 0: Primer. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, 2001.

[FN96] K. Futatsugi and A. Nakagawa. An Overview of Cafe Project. In *Proc. of CafeOBJ Workshop*, 1996.

[Gir86] J. Girard. The System F of Variable Types, Fifteen Years Later. *Theoretical Computer Science*, 45:159–192, 1986.

[Gog04] J. Goguen. The OBJ Family Home Page, 2004.
`http://www.cs.ucsd.edu/users/goguen/sys/obj.html`.

[GR88] P. Giannini and S. Ronchi della Rocca. Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 61–70, 1988.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[HU79] J. E. Hopcroft and J. D. Ullmann. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[HVP00] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, 2000. Full version under submission to TOPLAS.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[KK99] C. Kirchner and H. Kirchner. Rewriting, Solving, Proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *FOCS 1977*, pages 254–266. IEEE, 1977.

[Lei83] D. Leivant. Polymorphic Type Inference. In *Proc. of POPL*, pages 88–98. The ACM Press, 1983.

[LR91] T. Lakshman and U. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[LW05] L. Liquori and B. Wack. The polymorphic rewriting-calculus: Type checking vs. type inference. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Application*, volume 117 of *entcs*, pages 89 – 111. Elsevier B. V., 2005.

[Mic04] Microsoft. The C# Home Page, 2004. `http://msdn.microsoft.com/vcsharp/`.

[MLMK03] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. Submitted, 2003.

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In J. Fiala, V. Koubek, and J. Kratochvil, editors, *29th International Symposium on Mathematical Foundations of Computer Science*. Springer, 2004.

[MOM02] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[OS04] H. J. Ohlbach and S. Schaffert, editors. *Principles and Practice of Semantic Web Reasoning, Second International Workshop, PPSWR 2004, St. Malo, France, September 6-10, 2004, Proceedings*, volume 3208 of *Lecture Notes in Computer Science*. Springer, 2004.

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Pot] F. Pottier. Course DEA: Typage et programmation. `http://pauillac.inria.fr/~fpottier/mpri/dea-typage.ps.gz`.

[Pot00] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *journal of the ACM*, 12:23–41, 1965.

[SB04] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

[Sch04] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, LMU München: Fakultät für Mathematik, Informatik und Statistik, 2004.

[SFD00] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In *Proceedings of FSTTCS '2000*, number 1974 in LNCS. Springer-Verlag, 2000.

[SM73] L. Stockmeyer and A. Meyer. World problems requiring exponential time: preliminary report. In *STOC*, 1973.

[Sma99] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. D. Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.

[Ste02] M.-O. Stehr. *Programming, Specification and Interactive Theorem Proving – Towards a Unified Language based on Equational Logic, Rewriting Logic and Type Theory*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2002.

[Sun04] Sun. Java Technology, 2004. `http://java.sun.com/`.

[The03a] The Cristal Team. The Objective Caml Home Page, 2003. `http://www.ocaml.org/`.

[The03b] The GNU Prolog Team. The GNU Prolog Home Page, 2003. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.

[The04a] The Asf+Sdf Team. The Asf+Sdf Meta-Environment Home Page, 2004. `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment`.

[The04b] The Haskell Team. The Haskell Home Page, 2004. `http://www.haskell.org/`.

[The04c] The Maude Team. The Maude Home Page, 2004. `http://maude.cs.uiuc.edu/`.

[The04d] The Protheo Team. The Elan Home Page, 2004. `http://elan.loria.fr`.

[The04e] The Scheme Team. The Scheme Language, 2004. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[vDHK96] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996.

[vO90] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[WD03] A. Wilk and W. Drabent. On types for XML query language Xcerpt. In *International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003, Proceedings*, number 2901 in LNCS, pages 128–145. Springer Verlag, 2003.

[Wel99] J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.