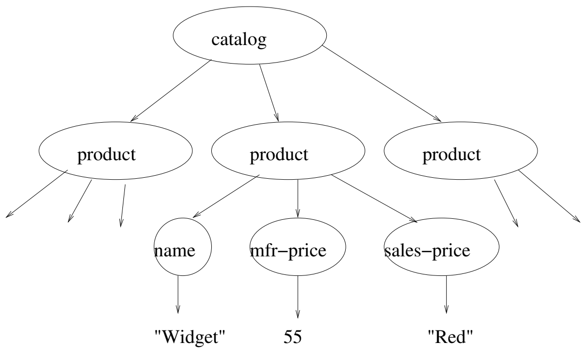
Typechecking for Semistructured Data

Dan Suciu
University of Washington

No Institute Given

1 The Problem

Semistructured data is used in data exchange applications, like B2B and EAI, and represents data in a flexible format. Every data item has a unique tag (also called *label*), and data items can be *nested*. Formally, a semistructured data instance is a tree whose nodes are labeled with tags and leaves are labeled with data values. XML [Con98] is a standard syntax for describing such trees; Fig. 1 shows a tree representing a semistructured data instance and its XML syntax. We will refer interchangeably to semistructured data instances as trees or XML trees.




```
<catalog>
  <product>
    <name> Widget </name>
    <mfr-price> 55 </mfr-price>
    <color> Red </color>
  </product>
  <product>
    <name> Gizmo </name>
    <mfr-price> 99 </mfr-price>
    <sales-price> 79 </sales-price>
  </product>
  <product>
    <color> Green </color>
    <color> Blue </color>
  </product>
</catalog>
```

Fig. 1. An XML Example and its Tree Representation

In a data exchange application some agreement is necessary on the data's structure in order to allow one program to understand the data produced by another. At minimum one should agree on the *taxonomy*, i.e. the meaning of the different tags. For example, an application must be able to understand the difference between tags like `sales-price`, `mfr-price`, and `suggested-price`. Little or no formalism is needed for such an agreement, and the common wisdom is that applications will look for the tags they need, and ignore all the other tags in the data. At a more detailed level, an agreement should also constrain the *structure*. For example a data item with a `price` tag must contain an integer value, while a data item with a `product` tag must contain nested items labeled `name`, `price`, and `description`. An agreement on the structure enables applications to navigate the data in a meaningful way. We call a collection of constraints on the structure a *type*. Several type formalism have been proposed for semistructured data [BDFS97,GW97,BM99], and several are considered for XML [Con98,BLM$^+$99,BFRW84].

A type in semistructured data, τ, is simply a set of trees, $\tau \subseteq \mathcal{U}$, where $\mathcal{U}$ denotes all (unranked) trees. Such a set is also called a tree language. While type formalisms differ in what sets of trees they can define as types, it is always easy to check if a *data instance* (i.e. a tree) is of a given type [1]. In the case of XML,

[1] In some powerful type systems [BM99] this problem is hard.

this is done at parse time, and is called *validation*. Most XML parsers today are validating parsers. A much harder problem arises when the semistructured data instance is generated automatically by a program, since then one needs to *typecheck* the program.

Typechecking Formally, given a program $P : \mathcal{V} \rightarrow \mathcal{U}$ that takes an input x from a set $\mathcal{V}$ and returns a tree $P(x)$, and given a type $\tau \subseteq \mathcal{U}$, the typechecking problem consists of checking: $\forall x \in \mathcal{V}$, $P(x) \in \tau$.

Two particular instances of typechecking are especially important in applications. The first is in *XML publishing*, where an XML view is defined over a relational database. We have a fixed relational schema, and $\mathcal{V}$ denotes all database instances of that schema; $P : \mathcal{V} \rightarrow \mathcal{U}$ is typically expressed in a declarative language, e.g. an extension of SQL in [SSB+00] or a new language in SilkRoute [FST00]. Often we also have a set of constraints over the relational database, e.g. a set of functional dependencies FD. In this case typechecking becomes: $\forall DB \in \mathcal{V}, FD \models DB \Rightarrow P(DB) \in \tau$.

The second application is in XML to XML transformations; such transformations where a program P is expressed in languages like XQuery, or XSL, and map XML trees to XML trees, $P : \mathcal{U} \rightarrow \mathcal{U}$. Often we have both an input type τ_1 and an output type τ_2, and the typechecking problem becomes: $\forall t \in \mathcal{U}, t \in \tau_1 \Rightarrow P(t) \in \tau_2$.

Type inference Related to the typechecking problem is the *type inference problem*: given the program P, infer the type $\{P(x) \mid x \in \mathcal{V}\}$. If this is a valid type, and we can infer it, then typechecking can be done by checking containment between two types: $\{P(x) \mid x \in \mathcal{V}\} \subseteq \mathcal{U}$, which is in general decidable for types in semistructured data. As we shall see however, such a strategy fails, even in the simplest cases, and we need to address typechecking with different methods.

Connection to programming languages Typechecking and type inference have been intensively studied in functional programming languages [Mit96]. There are three significant differences in semistructured data, however. First, types in semistructured data are *global* constraints on the data, while in programming languages they are *local* constraints. For example, a type for semistructured data may define a type of lists by a regular expression, say $(\mathtt{a}^*.\mathtt{b}^*.\mathtt{c}.\mathtt{b}^*) \mid (\mathtt{a}^*.\mathtt{c}.\mathtt{a}^*.\mathtt{b}^*)$, which says in which all `a` elements come before all the `b` elements, and that there be exactly one `c` element, anywhere in the list. Types in programming languages define only local properties, e.g. they cannot define the list type above, but only the approximation $(\mathtt{a} \mid \mathtt{b} \mid \mathtt{c})^*$, in which all elements of the list are either `a` or a `b` or a `c`. Second, the typechecking and type inference problems are defined differently. For example, in the *typed lambda calculus* we are given an expression in which each variable is annotated with a type, and are asked to check whether all operations in that program are of correct type. In type inference, the types of the variables have been erased, and we are asked to re-discover them. Both are problems are decidable are are formulated on the syntactic representation of the program. They differ from their counterparts in semistructured data, which require us to check a property of the semantics of a program: this is in general undecidable, if we don't impose any restrictions on the programming language. Third, the goals of typechecking and type inference differ from semistructured data to programming languages [Tan01]. In the former the goal is to validate documents; in the latter it is to ensure that there will be no runtime type error.

Clearly, the bar for typechecking is set higher in semistructured data than in programming languages, and one may wonder if the bar is too high [Wad01], making the problem undecidable and, thus, impractical. We argue that it is not, for several reasons. First, typechecking *is* decidable for certain restricted languages. There is precedence in query languages, for example one can check if a view defined by a conjunctive query satisfies certain integrity constraints (e.g. a functional dependency). Second, alternative (syntactic) approaches to typechecking in semistructured data have limitations that may be annoying in practice. As we show here, correct programs cannot be typechecked by a system relying on type inference, and have to be rejected, leaving the user wondering what to do. And, finally, the best way to find out if the bar is too high is to set it there and try ! This is precisely what has been done in [MSV00,AMN+01b,AMN+01a], leading to a characterization of the boundary of the decidable and undecidable cases.

2 Background

2.1 The Data

We consider an alphabet Σ, whose elements we call tags, or labels and assume Σ is finite. We denote with $\mathcal{U}_\Sigma$ the set of ordered, finite trees whose nodes are labeled with elements from Σ. Sometimes we allow the semistructured data to also carry some *data values* on the leaves. For this, we assume an infinite set D of data values to be given, and denote with $\mathcal{U}_\Sigma(D)$ the set of trees in which every internal node is labeled with a value in Σ, and every leaf is labeled either with some value in Σ or some value in D. A *semistructured data instance* is a tree in $\mathcal{U}_\Sigma(D)$.

To denote trees, we use a simple syntax:

$$T ::= \sigma(F) \mid \sigma' \quad F ::= T \mid T, F$$

where $\sigma \in \Sigma$ and $\sigma' \in \Sigma \cup D$.

Example 1. The following is an example of a semistructured data instance, whose tree representation is shown in Fig. 1:

```
catalog(product(name("Widget"),
                mfr-price(55),
                color("Red")),
        product(name("Gizmo"),
                mfr-price(99),
                sales-price(79)),
        product(name("Super-Widget"),
                color("Green"),
                color("Blue")))
```

2.2 The Types

One may be tempted to define a type as a regular tree language, but here we run into a technical problem. Tree automata (and regular tree languages) are defined in terms of ranked alphabets, while our Σ is unranked. Several approaches have been proposed to extend regular tree languages to unranked trees. They all turn out to be equivalent, but each may be useful in a different situation. We present them here, and begin by revising regular tree languages.

Ranked trees and tree automata A ranked alphabet, Ω, is a set partitioned into k subsets, $\Omega = \Omega_0 \cup \ldots \cup \Omega_k$. A ranked tree over Ω is a finite, ordered tree where every node with i children is labeled with a symbol from Ω_i. We denote $\mathcal{T}_\Omega$ the set of ranked Ω-trees.

A tree automaton is a device that accepts or rejects ranked trees. The definition below is adapted from [Tho90], and slightly modified.

Definition 1. *A* non-deterministic top-down tree automaton *is $A = (\Omega, Q, q_0, P)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and $P \subseteq \bigcup_{i=0,k} \Omega_i \times Q \times Q^i$.*

We write $(a, q) \rightarrow q_1 q_2 \ldots q_i$ whenever $(a, q, q_1, \ldots, q_i) \in P$.

Given $t \in \mathcal{T}_\Omega$, the automaton computes by assigning states to nodes in t. First it assigns q_0 to the root node. Whenever some node u with symbol a has been assigned a state q, and there exists a transition $(a, q) \rightarrow q_1 \ldots q_i$ in P, then the state q is removed from u, and the states $q_1, \ldots, q_i$ are assigned to its children. In particular, if u is a leaf ($i = 0$) then we only remove the state from u. The tree is accepted if we can remove all states from the tree. $L(A)$ denotes the set of trees accepted by A, and is called *regular tree language*. Notice that we can equally well describe the computation in reverse, from the leaves to the root; this defines a *bottom-up* tree automaton.

What is known about tree automata ? Bottom-up and top-down tree automata define the same set of regular tree languages, regular tree languages are closed under boolean operations (union, intersection, difference), and it is decidable whether one regular language is contained in another.

Regular Unranked Tree Languages Brüggemann-Klein et al. [BKMW98] extend tree automata to unranked trees. The problem here is that a symbol $a \in \Sigma$ has no rank, hence we need to specify potentially infinitely many transitions $(a, q) \rightarrow q_1 \ldots q_n$, for all $n \geq 0$. An unranked tree automaton is defined as before, but now P is a subset of $\Sigma \times Q \times Q^*$, and may be infinite, but is such that for every $a \in \Sigma$ and $q \in Q$, the set $\{w \mid (a, q, w) \in P\} \subseteq Q^*$ is a regular set. This allows us to give a finite description of P, as a set of triples (a, q, E), where E is a regular expression over the alphabet Q. Moreover, for each $a \in \Sigma, q \in Q$, we need exactly one pair (a, q, E) in P (we can collapse multiple pairs by taking the union of the regular expressions, and, for missing pairs, we introduce one with an empty regular expression). Computations on unranked trees are now straightforward extensions of computations on ranked trees. A *unranked regular tree language* is a set $\tau \subseteq \mathcal{U}_\Sigma$ accepted by some unranked tree automaton.

XDuce Types XDuce [HH00] is a functional programming language for XML. It defines XML types in a practical syntax, which has also been adopted by the XQuery working group [FFM+01]. We give here a slightly simplified definition, and start with an example. Assuming `Integer` and `String` to be predefined types, the following defines three new types, `CatalogType`, `ProductsType`, and `ProductType`:

```
CatalogType  = catalog(ProductsType)
ProductsType = (product(ProductType))*
ProductType  = (name(String), mfr-price(Integer),
                (sales-price(Integer))?, (color(String))*)
```

`CatalogType` defines a set of trees; each tree in this set has the root labeled `catalog`, and its children form a forest which belongs to the type `ProductsType`. This defines a set of forests, in which each forest has the root nodes labeled `product`, and the children of type `ProductType`, etc. As syntactic sugar, we allow a more concise syntax, where the type identifier `ProductType` has been substituted with its definition:

```
CatalogType = catalog(ProductsType)
ProductsType = product(name(String), mfr-price(Integer),
                       sales-price(Integer)?, color(String)*)*
```

Formally, we are given a set Q type identifiers. A *unit type* is a pair (σ, q) with $\sigma \in \Sigma$ and $q \in Q$. We denote $\sigma(q)$ such a unit type. Next, we consider regular expressions over unit types:

$$E ::= \sigma(q) \mid \epsilon \mid E, E \mid (E|E) \mid E* \mid E?$$

where $\sigma(q)$ is a unit type. A *type definition*, TD, is a set of pairs (q, E), where $q \in Q$ and E is a regular expression, such that for every $q \in Q$, there exists exactly one pair $(q, E) \in TD$. The semantics of a type definition is given by a straightforward fixpoint construction, see [HH00]. Under this semantics each type identifier in Q denotes a set of *forests*, not of trees. For our purposes, we will assume a distinguished type identifier q_0 (the "root") in Q and a pair $(q_0, \sigma(q))$ in TD, where $\sigma(q)$ is a unit type. This guarantees that the meaning of q_0 is a set of trees. In the example above, q_0 is `CatalogType`.

In our definition we have ignored the base types; they can be added easily.

There has been considerable effort at the W3C to define schemas for XML. The XML standard itself [Con98] defines DTDs, *Document Type Definitions*. Several attempts have been made to extend DTDs, and the current official recommendation is XML-Schema [BLM+99,BFRW84]. While both DTDs and XML-Schema types can be expressed as XDuce types, neither DTDs nor XML-Schema can express *all* XDuce types. In DTDs the restriction is that, if two unit types with the same tag, $\sigma(q)$ and $\sigma(q')$, occur in a type definition TD, then their types identifiers must be the same, i.e. $q = q'$. In other words there is a 1-1 correspondence from tags to type identifier, and therefore we can drop the latter and use tags instead. A DTD is then a set of pairs (σ, E), where $\sigma \in \Sigma$ and E is a regular expression over Σ. In XML-Schema the restriction is relaxed to the following: for every pair $(q_1, E_1) \in TD$, if two unit types with the same tag, $\sigma(q)$ and $\sigma(q')$ occur in E_1, their types have to be identical, i.e. $q = q'$. Here the same tag may still have different types, but only when occurring in different contexts, one in (q_1, E_1), the other in (q_2, E_2), with $q_1 \neq q_2$.

Specialized DTDs Papakonstantinou and Vianu [PV00] generalize DTD's to *specialized DTDs*. A specialized DTD consists of two alphabets, Q and Σ, a DTD over Q, and a function f from Q to Σ. The set it defines consists of homeomorphic images of the first DTD under the function f.

They are all equivalent It turns out that these three distinct type definitions are equivalent; they define the same class of unraked, regular tree languages. More interestingly, they correspond precisely to regular (ranked) tree languages, in two different ways. In a very naive sense, one is to view any ranked alphabet Ω as an unranked alphabet, and define a function, $u : \mathcal{T}_\Omega \to \mathcal{U}_\Omega$ that just forgets the ranks. Then for every regular tree language τ, $u(\tau)$ is a unranked regular trees language. More interestingly, we can encode unranked trees as ranked trees. Several such encodings are possible. We present here a particular encoding, which will be useful later, in Sec. 4.

Given the unranked alphabet Σ, associate to it the ranked alphabet Ω: $\Omega_0 = \{|\}, \Omega_2 = \Sigma \cup \{-\}$. Define the following "encoding" function on trees, $e : \mathcal{U}_\Sigma \to \mathcal{T}_\Omega$, and the associated encoding function on forests, $f : \mathcal{U}_\Sigma^* \to \mathcal{T}_\Omega$:

$$\begin{array}{llll} e(\sigma) & \stackrel{\text{def}}{=} \sigma(|,|) & e(\sigma(F)) & \stackrel{\text{def}}{=} \sigma(f(F),|) \\ f(T,F) & \stackrel{\text{def}}{=} -(e(T),f(F)) & f(T) & \stackrel{\text{def}}{=} e(T) \end{array}$$

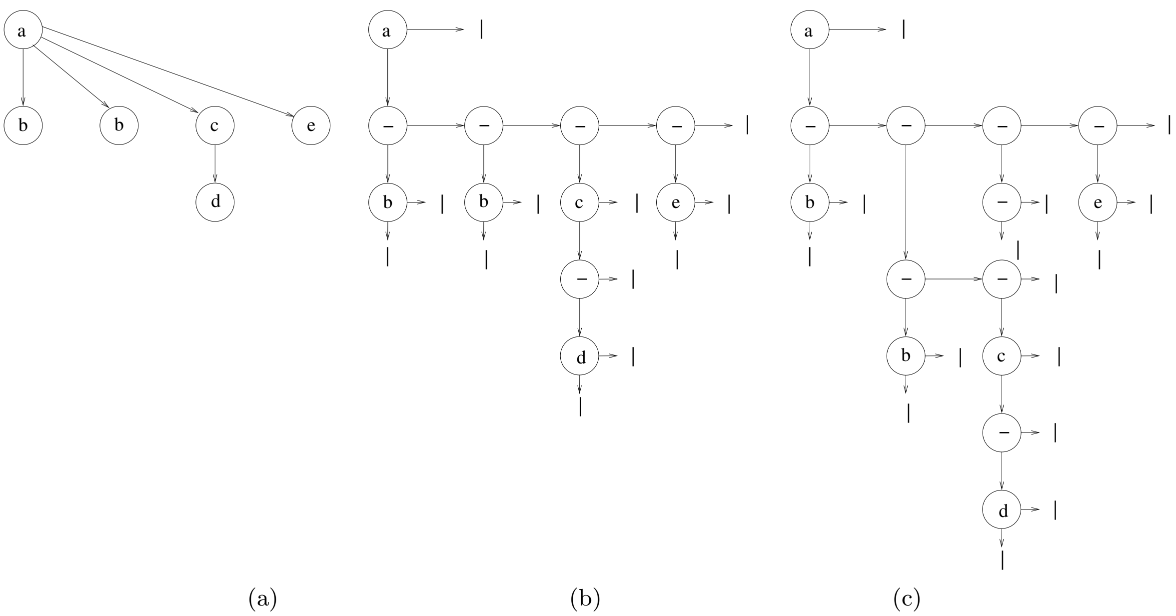


Fig. 2. Encoding of an unranked tree (a) into a binary tree (b); another binary tree (c).

Fig. 2 (a) and (b) illustrates the effect of the encoding function. In the other direction, we define a decoding function, $d : \mathcal{T}_\Omega \to \mathcal{U}_\Sigma^*$:

$$\begin{array}{lll} d(-(T_1,T_2)) & \stackrel{\text{def}}{=} d(T_1), d(T_2) & \\ d(\sigma(T_1,T_2)) & \stackrel{\text{def}}{=} \sigma(d(T_1), d(T_2)) & \sigma \in \Sigma \\ d(|) & \stackrel{\text{def}}{=} \varepsilon & \end{array}$$

Here F_1, F_2 denotes the concatenation of the forests F_1 and F_2. It is easy to check that $\forall t \in \mathcal{U}_\Sigma$, $d(e(t)) = t$, and that some trees in $\mathcal{T}_\Omega$ are not correct encodings, for example if they have too many occurrences of $-$, like the tree illustrated in Fig. 2 (c). The decoding function works on all trees in $\mathcal{T}_\Omega$; for example it maps the tree in Fig. 2 (c) to that in (a); we will need this in Sec. 4. For now, we summarize below the common knowledge on the relationship between unranked and ranked tree regular languages. The theorem below is straightforward to prove, and similar statements can be found in [BKMW98,MSV00,HH01].

Theorem 1.

1. *Let Σ be an unranked alphabet, and $\tau \subseteq \mathcal{U}_\Sigma$. The following are equivalent: (a) τ is accepted by an automaton on unranked trees, (b) τ is definable by an XDuce type, (c) τ is definable by a specialized DTD.*
2. *Given an unranked alphabet Σ, let Ω be its associated alphabet of rank 2 and let $\tau \subseteq \mathcal{U}_\Sigma$. Then the following are equivalent: (a) τ is an unranked regular tree language (b) $e(\tau)$ is a regular tree language.*
3. *Given a ranked alphabet Ω, a set $\tau \subseteq \mathcal{T}_\Omega$ is regular iff $u(\tau)$ is a regular set of unranked trees.*

It is interesting to see what happens to the ranked trees that are not valid encodings. If we take some arbitrary regular language $\tau \subseteq \mathcal{T}_\Sigma$ and decode it, $d(\tau)$ is *not* necessarily regular. To see why this happens, recall that the sequence of leaves on all trees in some regular tree language form a context free, not necessarily regular language. However, the following two properties hold:

Proposition 1. *(1) Let τ be an unranked, regular tree language in $\mathcal{U}_\Sigma$. Then $d^{-1}(\tau)$ is a regular tree language in $\mathcal{T}_\Omega$. (2) Let τ be a (ranked) regular tree language in $\mathcal{T}_\Omega$. Then $e^{-1}(\tau)$ is an unranked regular tree language.*

Containment is decidable The important property we take from the connection to regular tree languages is that, for any two types of semistructured data τ_1, τ_2, it is decidable whether $\tau_1 \subseteq \tau_2$. Indeed, this is equivalent to $e(\tau_1) \subseteq e(\tau_2)$, and containment of regular tree languages is decidable.

2.3 The Query Languages

There are two different paradigms for processing XML data.

Declarative XML-QL and XQuery are declarative languages for mapping XML to XML; systems like Xperanto [SSB+00] and SilkRoute [FST00] have declarative languages for mapping relational databases to XML. It is debatable what exactly makes a query language "declarative", but it is agreed that evaluation in a declarative language proceeds in two steps: first find all bindings of the variables, then compute some result fragment for each binding. This process results in a flat collection; to build nested collections, queries have to be nested. An important consequence for typechecking is that the nesting structure in the query defines precisely the nesting structure in the output XML document. Hence we can statically determine how deep the resulting XML document will be.

Functional languages In XSL and XDuce, an XML to XML transformation is expressed by traversing the input document with recursive functions. In XSL for example we start from the root and normally proceed downwards; at any point however we may continue with the current node's parent, or even start again at the root. Along the way we can bind "parameters" to certain nodes, and access them later during the recursion.

We describe two small language formalisms capturing the two paradigms.

TreeQL TreeQL is a language for mapping relational databases to trees. It was introduced in [AMN+01a] as an abstraction of RXL [FST00]. TreeQL uses *conjunctive queries* to construct trees. We denote with CQ the set of conjunctive queries. Given $q \in CQ$, $\mathit{headVar}(q)$ denotes its free variables; if $\bar{X} \subseteq \mathit{headVar}(q)$, then $\Pi_{\bar{X}}(q)$ denotes the projection of q on the variables $\bar{X}$: this is still a conjunctive query. For some database instance $DB = (D, R_1, \ldots, R_k)$, $q(DB)$ denotes the result of q on DB. We assume here that the databases are ordered, hence $q(DB)$ is an ordered set of tuples. Finally, we denote $q \subseteq q'$ if[2] $q(DB) \subseteq q'(DB)$ for all database instances DB, and say that q is contained in q'.

A program P in TreeQL is a tree in $\mathcal{U}_{\Sigma \times CQ}$. Hence, each node u in P is labeled both with a tag, and with a conjunctive query, denoted q_u. Formally:

Definition 2. *A TreeQL program is a tree $P \in \mathcal{U}_{\Sigma \times CQ}$ such that, whenever u is the parent node of v then (1) $\mathit{headVar}(q_u) \subseteq \mathit{headVar}(q_v)$ and (2) $\Pi_{\mathit{headVar}(q_u)}(q_v) \subseteq q_u$.*

Given an ordered database DB $P(DB) \in \mathcal{U}_\Sigma$ is defined as follows. Its nodes are pairs $(u, \bar{x})$, such that $u \in \mathit{nodes}(P)$ and $\bar{x} \in q_u(DB)$. The tag of $(u, \bar{x})$ is the same as that of u. A node $(u, \bar{x})$ is the parent of $(v, \bar{y})$ if u is the parent of v and $\bar{x} = \Pi_{\mathit{headVar}(q_u)}(\bar{y})$. The containment condition (2) above guarantees that every node has a parent (except the root). The nodes are ordered lexicographically, using the order in P and in DB.

[2] Recall that containment of conjunctive queries is decidable.

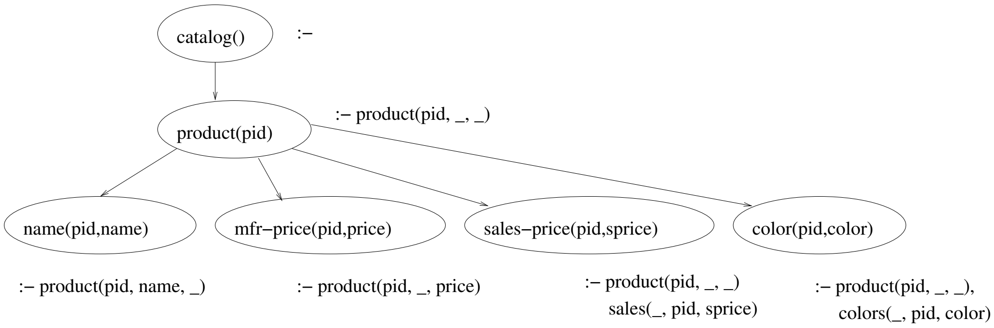


Fig. 3. TreeQL query for Example 2

Example 2. Consider the following relational database schema:

$$\texttt{product}(\underline{\texttt{pid}}, \texttt{name}, \texttt{mfrprice}), \texttt{colors}(\underline{\texttt{cid}}, \texttt{pid}, \texttt{color}), \texttt{sales}(\underline{\texttt{sid}}, \texttt{pid}, \texttt{price})$$

The following is a TreeQL program:

```
catalog() :-
  product(pid) :- product(pid, _, _)
     name(pid, name) :- product(pid, name, _)
     mfr-price(pid, price) :- product(pid, _, price)
     sales-price(pid, sprice) :- product(pid, _, _), sales(_, pid, sprice)
     color(pid, color) :- product(pid, _, _), colors(_, pid, color)
```

We wrote the conjunctive query in Datalog notation, used the XML output tag as head predicate, and indentation to define the tree structure. Fig. 3 illustrates the same program graphically. On some database instance, the program computes an XML tree as follows. First, it computes the query for `catalog()`, returning a single value (the empty tuple): this corresponds to the root node. Next it computes the query for `product(pid)`. For each tuple in the answer it constructs a node labeled `product`. Next, it computes `name(pid, name)`, and for each answer it constructs a node labeled `name`: its parent is the `product` node with the same `id`, etc.

RecQL We introduce here RecQL, a functional language mapping an input tree to an output tree. RecQL is designed to express precisely the tree transformations expressible in k-pebble tree transducers [MSV00]. Fig. 4 defines RecQL. A program consists of an expression, E, in which has several mutually recursive functions. Functions take a single node as input parameter, and can construct result forests, test predicates, or call other functions recursively. The forest constructors are: the tree denoted by a variable x; singletons $\sigma(F)$, where $\sigma \in \Sigma$, and $\mathsf{tag}\ (x)(F)$, where $\mathsf{tag}\ (x)$ denotes the tag associated to the node x; forest concatenation F_1, F_2; and the empty forest (). Recursive calls can be applied either to a node ($f(x)$), or to one of its neighbors: left-most child ($\mathsf{down}\ (x)$), parent ($\mathsf{up}\ (x)$), left sibling ($\mathsf{left}\ (x)$), or right sibling ($\mathsf{right}\ (x)$). Finally, the predicate $\mathsf{tag}\ (x) = \sigma$ checks whether the label of node x is σ, $x = y$ checks whether x and y are the same node, while $\mathsf{hasDown}\ (x)$, $\mathsf{hasUp}\ (x)$, $\mathsf{hasLeft}\ (x)$, $\mathsf{hasRight}\ (x)$ test whether the node x has the corresponding neighbor. Function definitions may be nested, and usual scoping rules apply.

We impose the following restriction on function calls: the variable x must be defined in the same block as f or a surrounding block For example, considering the definition $\mathsf{fun}\ f(x) = \mathsf{let\ fun}\ g(y) = \mathsf{let\ fun}\ h(z) = F_h\ \mathsf{in}\ F_g g'(y') = F_{g'}\ \mathsf{in}\ F_f$, then F_g (the body of g) may contain the function calls $g(Up(y))$, $g'(Down(x))$, $h(y)$, and $f(Up(x))$, but not the function call $f(y)$, because y is defined in an inner block from f. This restriction is imposed in order to make the proof of the typechecking theorem (Theorem 4) possible[3].

A program is an expression with a unique free variable, t.

The fact that we construct *forests* as opposed to *trees* gives us extra expressive power, as we shall see in a moment. Since the function definitions may be nested, we can refer to variables in their outer scope variables; this also gives us extra power. Notice that programs do not necessarily terminate.

[3] We leave open the question whether this restriction is necessary.

$$
\begin{aligned}
E ::= {} & \mathsf{let\ fun}\ f_1(x) = E_1 \\
& \quad \mathsf{fun}\ f_2(x) = E_2 \\
& \quad \ldots \\
& \quad \mathsf{fun}\ f_n(x) = E_n \\
& \mathsf{in}\ \ F \\
& \mid F \\
F ::= {} & x \quad \mid \quad \sigma(F) \quad \mid \ \mathsf{tag}\ (x)(F) \quad \mid \quad F_1, F_2 \quad \mid \quad () \quad \mid \qquad\qquad\qquad\qquad (*) \\
& \mathsf{if}\ P\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2 \mid \\
& f(x) \quad \mid \quad f(\mathsf{down}\ (x)) \quad \mid \quad f(\mathsf{up}\ (x)) \quad \mid \quad f(\mathsf{left}\ (x)) \quad \mid \quad f(\mathsf{right}\ (x)) \\
P ::= {} & (\mathsf{tag}\ (x) = \sigma) \quad \mid x = y \quad \mid \quad \mathsf{hasDown}\ (x) \quad \mid \quad \mathsf{hasUp}\ (x) \quad \mid \quad \mathsf{hasLeft}\ (x) \quad \mid \quad \mathsf{hasRight}\ (x)
\end{aligned}
$$

Fig. 4. The Language RecQL

Example 3. The following program takes a tree t as input and returns a list of all leaves:

```
result(let fun f(x) = if hasDown(x) then aux(down(x)) else x
           fun aux(x) = f(x), (if hasRight(x) then aux(right(x)) else ())
       in f(t))
```

On input `a(b(c,d(e)),f(g))` the program returns `result(c,e,g)`. The function `aux` helps us applying `f` on all children of the current node. This is a common idiom, and we will use the abbreviation `apply[f]` for such a construction. That is, the function above can be rewritten as:

```
result(let fun f(x) = if hasDown(x) then apply[f](x) else x in f(t))
```

As a second example, we illustrate a program which copies the entire tree, renaming all `a`'s to `b`'s:

```
result(let fun f(x) = if tag(x) = a then b(apply[f](x)) else tag(x)(apply[f](x))
       in f(t))
```

Finally, the last program takes a list, `root(a1(..), ..., an(..))` and returns the cartesian product, `result(pair(a1,a1), pair(a1, a2), ..., pair(an,an))`:

```
result(let fun f(x) = let g(y) = pair(tag(x)(), tag(y)())
                      in apply[g](t)
       in apply[f](t))
```

Notice that this function satisfies the restriction on recursive calls: g is called recursively on on t, a variable which is visible within its scope.

Relationship to practical languages TreeQL is defined almost identical to RXL, a language for publishing XML from relational databases used in SilkRoute [FST00]. Xperanto [SSB$^+$00] uses a much richer language which is an extension of SQL. RecQL corresponds to a subset of XSL and, more closely, to the formal XSLT0 fragment described in [BMN00a,BMN00b]. Like XSL it can express recursive calls on children, parents, siblings, and can simulate recursive calls on descendants and ancestors that are allowed in XSL. It can simulate to some extent parameters in XSL, through the use of global variables. XSL however allows a new tree to be constructed and bound to a parameter: this cannot be simulated in RecQL. Another limitation in RecQL is that we cannot compare two data values, i.e. cannot express joins.

3 Type Inference and Its Limitations

Consider the program in Fig. 3. Reading it carefully, we can actually infer its output type, written in XDuce syntax (and ignoring the base types):

```
T   = catalog(T1*)
T1  = product(name, mfr-price, sales-price*, color*)
```

The result has a `catalog` tag on the root, and multiple `product` subelements. Each product has exactly one `name` and one `mfr-price`, because `pid` is a key in `product`, but may have arbitrary many `sales-price`s and `color`s.

This approach is very simple, and very powerful. For example if we are given an output type:

```
T'   = catalog(T1'*)
T1'  = product(name, description?, mfr-price*, (sales-price,sales-date?)*, color*)
```

then we are able to deduce that P typechecks w.r.t. T'. For that, we use the fact that containment of unranked regular tree languages is decidable (Sec. 2).

What makes this even more appealing is that we can use constraints on the input database to refine the inferred input type. For example, assume that in the table `sales(pid, price)` the following Functional Dependency holds: `pid` $\rightarrow$ `sid` (i.e. `pid` is also a key). Then every product has at most one sales prices, and we can refine the inferred type to:

```
T   = catalog(T1*)
T1  = product(name, mfr-price, sales-price?, color*)
```

XDuce [HH00] describes type inference rules for a more complex language, and XQuery algebra, which is currently discussed at the W3C, follows a similar approach.

Unfortunately, type inference is not complete, not even for the simplest language and the simplest restrictions on types. Consider the following example. The relational schema has a single table, `R(x,y)`. The TreeQL query is:

```
result() :-
   a(x, y) :- R(x, y)
   b(x, y) :- R(x, y)
```

The inferred output type would be:

$$\mathtt{T} = \mathtt{result}(\mathtt{a}*, \mathtt{b}*)$$

But in fact, the "real" inferred type is:

$$\mathtt{T_r} = \{\mathtt{result}(\mathtt{a^n}, \mathtt{b^n}) \mid \mathtt{n} \geq 0\}$$

since we have the same number of `a`'s and `b`'s. Obviously, this is not a regular tree language, hence we cannot "infer" it. Is perhaps `T` the best type one can infer ? Unfortunately not; consider the following type:

$$\mathtt{T1} = \mathtt{result}() \mid \mathtt{result}(\mathtt{a}, \mathtt{a}*, \mathtt{b}, \mathtt{b}*)$$

It rules out the cases when there are no `a`'s but there are `b`'s, and vice versa. For the same reason `T2` below is better than `T1`:

$$\mathtt{T2} = \mathtt{result}() \mid \mathtt{result}(\mathtt{a}, \mathtt{b}) \mid \mathtt{result}(\mathtt{a}, \mathtt{a}, \mathtt{a}*, \mathtt{b}, \mathtt{b}, \mathtt{b}*)$$

Obviously, we will not be able to obtain a "best" type.

As a result, a system using type inference to do typechecking may reject correct programs. Assume that we want to check if our program outputs XML trees of type `T1`. The answer is "yes", but the system would first infer the output type `T`, observe that $\mathtt{T} \not\subseteq \mathtt{T1}$, and answer "no".

RecQL programs have the same problem. Consider the following program `P`:

```
result(let fun f(x) = apply[g](x), apply[h](x)
           fun g(x) = a
           fun h(x) = b
       in f(t))
```

On some input tree `root(t1, ..., tn)` it returns the tree $\mathtt{result}(\mathtt{a^n}, \mathtt{b^n})$, and the same arguments apply as above.

4 Type Checking and Its Limitations

In practice, however, we need to solve the typechecking, not the type inference problem. Are there any methods that allow us to do typechecking without type inference ? The answer is yes, in some cases, but not in general. We illustrate below with both TreeQL and RecQL.

4.1 Typechecking TreeQL

We illustrate on an example. Consider the following TreeQL program `P`:

```
result() :-
    a(x,y) :- q1(x,y)
    b(x,y) :- q2(x,y)
```

where `q1` and `q2` are some conjunctive queries. Assume that we want to check whether its output belongs to the type `T2` in Sec. 3. We can proceed as follows. First compute the complement of `T2`:

```
T2' =  result(a,a*) | result(b,b*) | result(a,b,b,b*) | result(a,a,a*,b)
```

If `P` does not typecheck w.r.t. `T2`, then there exists some database instance `DB` s.t. `P(DB)` is of type `T2'`. We have to consider four cases, we illustrate here the case when `P(DB)` is in `result(a,b,b,b*)`. Hence `q1(DB)` has exactly one tuple in the answer, $\bar{x1}$, and `q2(DB)` has at least two tuples, let $\bar{x2}, \bar{x2}'$ be two such tuples. Since `q1`, `q2` are conjunctive, we can find a subset database $\mathtt{DB_0}$ s.t. $\bar{x1} \in \mathtt{q1}(\mathtt{DB_0})$ and $\bar{x2}, \bar{x2}' \in \mathtt{q2}(\mathtt{DB_0})$, and the size of $\mathtt{DB_0}$ is bound by the sizes of `q1` and `q2` only. In other words, if there exists an counterexample database, then there exists a "small" one, and we can typecheck by simply computing `P` on all "small" databases.

This generalizes, provided that we restrict the output type τ as follows:

- τ is a DTD.
- every regular expression occurring in τ is "star-free".

Theorem 2. *[MSV00] Typechecking is decidable for TreeQL over databases with functional dependencies, and for star-free output DTDs.*

Star-free regular expressions [Tho90] are much more expressive than their name suggests. We are not allowed to use the Kleene closure, however we can use the complement, denoted *compl*, and the empty set $\emptyset$. For example, if $\Sigma = \{a, b, c\}$, then *compl*($\emptyset$) denotes Σ^*, *compl*($\Sigma^*.b.\Sigma^* \mid \Sigma^*.c.\Sigma^*$) denotes a^*. Thus, regular expressions like $a^*.b^*.(c \mid a.b^*)$ and even $(a.b)^*$ are star-free. In fact, *all* examples of regular expressions used in this paper are star-free. An example which is not star-free is $(a.a)^*$.

The algorithm that results from the proof of Theorem 2, unfortunately, is not practical, since it enumerates all databases up to a certain size.

What happens if we drop the restrictions on the output type ? In that case typechecking becomes undecidable [AMN$^+$01a].

4.2 Typechecking for RecQL

RecQL seems much more powerful than TreeQL but, surprisingly, we can decide typechecking for RecQL, and without any restrictions to the output type. The catch, as we will see, is that there are no joins in RecQL.

We start with an example. Consider the program `P` at the end of Sec.3. On some input tree $\mathtt{root}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ it returns the tree $\mathtt{result}(\mathtt{a^n}, \mathtt{b^n})$. Suppose that we are given some input type τ_1 and some output type τ_2, and we want to typecheck: $\forall \mathtt{t} \in \mathcal{U}_\Sigma.\mathtt{t} \in \tau_1 \Rightarrow \mathtt{P}(\mathtt{t}) \in \tau_2$. The test $\mathtt{P}(\tau_1) \subseteq \tau_2$ would suffice, but we cannot "infer" the type $\mathtt{P}(\tau_1)$. It turns out, however, that we can always do inverse type inference, i.e. compute $\mathtt{P}^{-1}(\tau_2)$. In other words, this is always an unraked regular tree language. Then typechecking reduces to: $\tau_1 \subseteq P^{-1}(\tau_2)$. To see an example, consider τ_2 to be: `result((a,a)*,(b,b)*)`, i.e. we must have an even number of `a`'s, and an even number of `b`'s. Then $\mathtt{P}^{-1}(\tau_2)$ consists of all trees in which the root has an even number of children, $\mathtt{root}(\mathtt{t_1}, \ldots, \mathtt{t_{2n}})$, which is indeed a unranked regular tree language.

Proving typechecking for the general case relies on a theorem in [MSV00], where typechecking is done for k-pebble tree transducers. Without revising these devices here, we restate that result in terms of RecQL. Let Ω be a ranked alphabet, $\Omega = \Omega_0 \cup \Omega_2$. Trees in $\mathcal{T}_\Omega$ are complete binary trees, i.e. every node is either a leaf or has two children identified as the left and the right child. Notice that our encoding of unranked trees into ranked trees are special cases of such trees, see Fig. 2. Define RecQL$_\Omega$ to be the following adaptation of RecQL to trees in $\mathcal{T}_\Omega$. RecQL$_\Omega$ has the same definition as RecQL in Fig. 4, except for the forest constructors (the line marked $(*)$ in Fig. 4), which now becomes:

$$F ::= x \mid \sigma \mid \sigma(F, F) \mid \mathsf{tag}\ (x) \mid \mathsf{tag}\ (x)(F, F)$$

We show here the following:

Proposition 2. *RecQL$_\Omega$ expresses precisely the transformations expressed by deterministic k-pebble tree transducers.*

Proof. (Sketch) Recall that a k-pebble transducer has its set of states partitioned into k subsets, $Q = Q_1 \cup \ldots \cup Q_k$, with all states from Q_i describing transitions for pebble i. First we show how to translate a k-pebble transducer into a RecQL$_\Omega$ program. The program will have one function $f_{q_1 q_2 \ldots q_i}(x_i)$ for each i-tuple of states $q_1 \in Q_1, \ldots, q_i \in Q_i$, for $i = 1, \ldots, k$: thus there will be exponentially more functions than states. All functions of the form $f_{q_1 \ldots q_i q_{i+1}}$ are defined within the scope of the function $f_{q_1 \ldots q_i}$, and they can access all pebbles $x_1, x_2, \ldots, x_i$ (the variables of the functions $f_{q_1}, f_{q_1 q_2}, \ldots, f_{q_1 q_2 \ldots q_i}$). Each function $f_{q_1 q_2 \ldots q_i}(x_i)$ simulates the transitions of state q_i: it test the current tag ($\mathsf{iftag}\ (x_i) = \sigma\ \mathsf{then} \ldots$) and the presence/absence of the other pebbles ($\mathsf{if}\ x_i = x_j\ \mathsf{then} \ldots$), then simulates the corresponding transition in the transducer. For example we simulate an *up* move, to some state q_i', with the expression $f_{q_1 q_2 \ldots q_i'}(\mathsf{up}\ (x_i))$. To simulate a *place* move, which places the next pebble, $i+1$, on the root and continue in some state $q_{i+1} \in Q_{i+1}$ we call the function $f_{q_1 \ldots q_i q_{i+1}}(t)$, where t is the variable bound to the root of the tree. To simulate a *pick* move, in which we pick the pebble i and continue with pebble $i-1$ in some state $q_{i-1}' \in Q_{i-1}$, we call the function $f_{q_1 q_2 \ldots q_{i-1}'}(x_{i-1})$. Notice that this function satisfies the restriction since x_{i-1} is defined in the same block as $f_{q_1 q_2 \ldots q_{i-1}'}(x_{i-1})$. Finally, for an output transition, in which some symbol σ is output and the computation proceeds in states q_i' and q_i'', we write the expression $\sigma(f_{q_1 \ldots q_i'}(x_i), f_{q_1 \ldots q_i''}(x_i))$.

For the reversed direction, we are given a RecQL$_\Omega$ program and need to construct an equivalent k-pebble transducer. We do this by a number of transformations that essentially transform it into a k-pebble transducer, with each function corresponding to a state. In the first transformation we pull the conditionals up, e.g. rewrite $\sigma(\mathsf{if}\ P\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2, F)$ into $\mathsf{if}\ P\ \mathsf{then}\ \sigma(F_1, F)\ \mathsf{else}\ \sigma(F_2, F)$. In the second transformation we ensure that all expressions on the leaves of the conditionals are function calls, like $f(x)$ or $f(\mathsf{up}\ (x))$, or an output symbol followed by two calls, e.g. $\sigma(f(\mathsf{up}\ (x)), g(\mathsf{down}\ (y)))$. For that we break more complicated expressions by introducing additional functions. The third transformation is to ensure that each call to an inner function is the root as argument: for that we replace a call $f(x_i)$ with a call $f'(t)$ where f' is a new inner function, i.e. defined in the same block as f_i, which performs a depth-first search of the tree until it finds the node x_i, and there it calls f (it is allowed to call f with some other argument than the root because it is at the same level). The fourth transformation does something similarly for calls to function in outer blocks. That is, if $f_i(x_j)$ is a function call inside some block $> i$ to some function f_i define in block i and the argument x_j is a variable defined by the block $j < i$, then we do the following. We replace the expression $f_i(x_j)$ with $f_i'(x_i)$ where f_i' is a new function added to block i, which does a traversal of the tree until it places x_i on the same node as x_j, then calls $f_i(x_i)$. Finally, the last transformation is one in which we remove predicates on global variables. For example, if a function at nesting depth i, $f_i(x_i)$, contains a predicate of the form $\mathsf{if}\ x_j = \sigma \ldots$ where x_j is defined in some lower block $j < i$, then we replace f_i with two copies, f_i', f_i'', one which corresponds to the case $x_j = \sigma$ the other to the case $x_j \neq \sigma$, and let the functions that call f_i perform the test $x_j = \sigma$ and call either f_i' or f_i''. This results in an exponential growth of the number of functions. We treat similarly predicates of the form $x_i = x_j$. After all these transformations are completed the resulting program corresponds directly to a k-pebble transducer.

We can now formally restate the main result in [MSV00].

Theorem 3. *[MSV00] For any program P in RecQL$_\Omega$ and any regular tree language τ, $P^{-1}(\tau)$ is also a regular tree language. In consequence typechecking for RecQL$_\Omega$ is decidable.*

This result is for ranked trees. We now transfer this result to RecQL and unranked trees, using the encoding in Sec. 2, and its properties in Prop. 1. In the following we fix an unranked alphabet Σ, and denote Ω is associated, ranked alphabet of rank 2.

Proposition 3. *For every RecQL program P, there exists some RecQL$_\Omega$ program P_Ω s.t. $\forall t \in \mathcal{U}_\Sigma$, $d(P_\Omega(e(t))) = P(t)$.*

Proof. (Sketch) P_Ω differs from P in the way it traverses the input tree and the way it constructs the output tree.

- On the input tree, $e(t)$, P_Ω simulates P by traversing on the nodes labeled with $-$. The moves $\mathsf{down}\ (x)$, $\mathsf{up}\ (x)$, $\mathsf{left}\ (x)$, and $\mathsf{right}\ (x)$ now become $\mathsf{down}\ (\mathsf{down}\ (x))$, $\mathsf{up}\ (\mathsf{up}\ (x))$, $\mathsf{up}\ (x)$ and $\mathsf{right}\ (\mathsf{down}\ (x))$ (technically these expressions are illegal; we need to introduce a new function, for every occurrence of such a move). Whenever we need $\mathsf{tag}\ (x)$ (in a forest constructor or a conditional) we need to move one step down and get it (this, again, requires us to introduce a new function).
- On the output tree, we add extra nodes labeled $-$. We change the forest constructors, F to F_Ω, as follows:

$$
\begin{aligned}
(\sigma(F))_\Omega &\stackrel{\text{def}}{=} \sigma(-(F_\Omega, |), |) \qquad \text{same for } \mathsf{tag}\ (x)(F) \\
(F_1, F_2)_\Omega &\stackrel{\text{def}}{=} -((F_1)_\Omega, (F_2)_\Omega) \\
()_\Omega &\stackrel{\text{def}}{=} |
\end{aligned}
$$

Notice that the new program P_Ω will not return a correct encoding, because it adds too many nodes labeled $-$. That is, its result will look more like Fig. 2 (c) rather than (b). Hence, $P_\Omega(e(t))$ is in general not equal to $e(P(t))$; only the weaker statement in Prop. 3 holds. However, here we can use Prop. 1 and obtain the following result:

Theorem 4. *For any program P in RecQL and any unranked regular tree language τ, $P^{-1}(\tau)$ is also an unraked regular tree language. In consequence typechecking for RecQL is decidable.*

Proof. From Proposition 3 it follows:

$$P^{-1}(\tau) = e^{-1}(P_\Omega^{-1}(d^{-1}(\tau)))$$

By Prop. 1 $d^{-1}(\tau)$ is a regular tree language, hence, by Theorem 3 $P_\Omega^{-1}(d^{-1}(\tau))$ is a regular tree language, hence $e^{-1}(P_\Omega^{-1}(d^{-1}(\tau)))$ is an unranked regular tree language.

Despite its appeal, this positive result has little practical consequences in its current form. The complexity of the decision procedure that follows from the proof in [MSV00] is hyper-exponential. While no lower bound has been shown so far, typechecking is at least as complex as containment of regular tree languages, which is DEXPTIME complete [Sei90].

Moreover, the result in Theorem 4 is brittle: typechecking becomes undecidable if we add data value joins. More precisely, assume that we add the data values to our input trees, i.e. consider trees in $\mathcal{U}_\Sigma(D)$. Now leaf nodes in the input tree may be labeled with elements from D. Assume that we extend the definition of RecQL in Fig. 4 with a predicate that checks the equality of two data values:

$$P ::= \mathsf{tag}\ (x) = \mathsf{tag}\ (y) \tag{1}$$

where x, y are two node variables. We did not need such a predicate before, when Σ was finite and known in advance (we could have simply compared each x and y which each symbol in Σ). Denoting RecQL$^=$ the extended language, it turns out that typechecking for RecQL$^=$ is undecidable. This is easily seen by reduction from the finite satisfiability problem for First Order Logic. Namely, given a vocabulary with one binary relation $R(x, y)$, one can encode finite structures over R as trees in $\mathcal{U}_\Sigma(D)$, with a small alphabet Σ. For example, we could take $\Sigma = \{a, b\}$ and encode the finite structure $\{(3,5), (5,1), (3,2)\}$ as $a(b(3,5), b(5,1), b(3,2))$. Given a formula in first order logic, φ, one can write a program P in RecQL$^=$ which checks whether the formula is true on the (structure encoded by the) input tree. If yes, then it returns a c; otherwise it returns a d. Assume the output type to be $\tau = \{d\}$. Then P typechecks if and only if φ is not satisfiable. Notice that we need the equality operator (1) because φ may contain the equality predicate.

5 Related Work

An extensive treatment of semistructured data, with additional references can be found in [ABS99]. Types for semistructured data have been proposed in [BDFS97,GW97,BM99]; for XML, DTDs are described in the XML recommendation [Con98], while XML-Schema is now a separate recommendation [BLM+99]. Automata for unranked trees are described in [BKMW98], and specialized DTD's in [PV00]. The XDuce programming language and it's associated XML type inference problem are discussed in [HH00,HH01]; the XQuery Semantics document describes type inference for a much richer language [FFM+01], which is used to define formally the semantics of XQuery. k-Pebble tree transducers are introduced in [MSV00] and shown to have a decidable typechecking problem. A connection between k-pebble tree transducers and a fragment of XSL is discussed in [BMN00a,BMN00b]. The typechecking problem for other languages, with joins, is studied in [AMN+01b,AMN+01a].

6 Discussion

XML typechecking is clearly an important problem, because we need to validate dynamically generated XML documents. We have seen two approaches to typechecking: an incomplete method, based on heuristic type inference, and two complete methods, using specialized algorithms. None is completely satisfactory in practice. The first results in systems that reject correct programs without warnings. The second may work in some cases, but the decidable cases seem to be more the exception than the rule. Moreover, even where typechecking is decidable, the complexity is high.

Today, the front-runner in practice is the type inference method, because it is easy to extend, even if incompletely, to a variety of language constructs. What is intriguing about type inference is that, despite its theoretical incompleteness, it seems to be "complete" for practical applications. In our example in Sec. 3, the types `T1` an `T2` that caused typechecking to fail are artificial, and one can argue that such types do not occur in practice. One could make the argument that with the inferred type `T` the system can correctly typecheck the program for all "practical" output types (since the output types on which it fails, like `T1`, are "artificial"). But so far no formal statements of this kind have been proven. This raises several questions:

– Is there a notion of "practical" types for which type inference is complete ? This notion should exclude the types `T1, T2` in Sec. 3, while allow interesting types with Kleene closures (perhaps with the limitations of star-free languages), alternations, and optional elements.
– Is it possible to issue a warning when type inference fails ? We would like the typechecking algorithm to return `yes`, `no`, or `unknown` rather than return false negatives.
– As alternative techniques to type inference one might consider approximation, or randomized typechecking algorithm. Of course, type inference offers an approximation algorithm already, but it is not clear yet that is is the best in practice.

Acknowledgments The author is supported by the NSF CAREER Award 0092955, a Sloan Fellowship and a gift from Microsoft Research.

References

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and Xml*. Morgan Kaufmann, 1999.

[AMN+01a] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking xml views of relational databases. In *LICS*, 2001.

[AMN+01b] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Xml with data values: Typechecking revisited. In *PODS*, 2001.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BFRW84] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: a model for W3C XML Schema. In *Proceedings of WWW10*, Hong Kong, May 1984.

[BKMW98] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998. Available at `ftp://ftp11.informatik.tu-muenchen.de/pub/misc/caterpillars/`.

[BLM+99] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. Xml schema part 1: Structures, May 1999. `http://www.w3.org/TR/xmlschema-1/`.

[BM99] Catriel Beeri and Tova Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 296–313, 1999.

[BMN00a] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In Lloyd et al., editor, *Computational Logic – CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1137–1151. Springer, 2000.

[BMN00b] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT, 2000. To appear in Information Systems.

[Con98] World Wide Web Consortium. Extensible markup language (xml) 1.0, 1998. `http://www.w3.org/TR/REC-xml`.

[FFM+01] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 formal semantics, 2001. available from the W3C, `http://www.w3.org/TR/query-semantics`.

[FST00] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.

[HH00] Benjamin C. Pierce Haruo Hosoya. Xduce: An xml processing language (preliminary report). In *WebDB'2000*, 2000. http://www.research.att.com/conf/webdb2000/.

[HH01] Benjamin C. Pierce Haruo Hosoya. Regular expression pattern matching for xml. In *ACM SIGPLAN, SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2001.

[Mit96] John C. Mitchell. *Foundations for Programmng Languages*. MIT Press, 1996.

[MSV00] T. Milo, D. Suciu, and V. Vianu. Typechecking for xml transformers. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 11–22, Dallas, TX, 2000.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of PODS*, pages 35–46, Dallas, TX, 2000.

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput*, 19(3):424–437, 1990.

[SSB+00] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egipt, September 2000.

[Tan01] Val Tannen. Personal communication, 2001.

[Tho90] W. Thomas. Automata on infinite objects. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 4, pages 133–192. Elsevier, Amsterdam, 1990.

[Wad01] Phil Wadler. Personal communication, 2001.