Towards Practical Typechecking for Macro Tree Transducers

Alain Frisch[1] and Haruo Hosoya[2]

[1] INRIA Rocquencourt `alain@frisch.fr`
[2] The University of Tokyo `hahosoya@is.s.u-tokyo.ac.jp`

Abstract. Macro tree transducers (mtt) are an important model that both covers many useful XML transformations and allows decidable exact typechecking. This paper reports our first step toward an implementation of mtt typechecker that has a practical efficiency. Our approach is to represent an input type obtained from a backward inference as an alternating tree automaton, in a style similar to Tozawa's XSLT0 typechecking. In this approach, typechecking reduces to checking emptiness of an alternating tree automaton. We propose several optimizations (Cartesian factorization, state partitioning) on the backward inference process in order to produce much smaller alternating tree automata than the naive algorithm, and we present our efficient algorithm for checking emptiness of alternating tree automata, where we exploit the explicit representation of alternation for local optimizations. Our preliminary experiments confirm that our algorithm has a practical performance that can typecheck simple transformations with respect to the full XHTML in a reasonable time.

1 Introduction

Static typechecking for XML transformations is an important problem that expectedly has a significant impact on real-world XML developments. To this end, several research groups have made efforts in building typed XML programming languages [8, 3] with much influence from the tradition of typed functional languages [2, 10]. While this line of work has successfully treated general, Turing-complete languages, its approximative nature has resulted in an even trivial transformation like the identity function to fail to typecheck unless a large amount of code duplicates and type annotations are introduced [7]. Such situation has led us to pay attention to completely different approaches that have no such deficiency, among which *exact typechecking* has emergingly become promising. The exact typechecking approach has extensively been investigated for years [12, 20, 16, 23, 26, 24, 11, 15, 1, 13, 18, 14], in which *macro tree transducers* (mtt) have been one of the most important computation models since they allow decidable exact typechecking [5], yet cover many useful XML transformations [5, 11, 4, 19]. Unfortunately, these studies are mainly theoretical and their practicality has never been clear except for some small cases [23, 26].

This paper reports our first step toward a *practical* implementation of typechecker for mtts. As a basic part, we follow an already-established scheme called *backward inference*, which computes the preimage of the output type for the subject transformation and then checks it against the given input type. (This is because, as is well known, the more obvious, forward inference does not work since the image of the input type is not always a regular tree language and can even go beyond context-free tree languages.[3]) However, our proposal is, on top of this scheme, to use a representation of the preimage by an *alternating tree automaton* [21], extending the idea used in Tozawa's typechecking algorithm for XSLT0 [23]. In this approach, typechecking reduces to checking emptiness of an alternating tree automaton.

Whereas normal tree automata use only disjunctions in the transition relation, alternating tree automata can use both disjunctions and conjunctions. This extra freedom permits a more compact representation (they can be exponentially more succinct than normal tree automata) and make them a good intermediate language to study optimizations. Having explicit representation of transitions as Boolean formulas (with disjunctions and conjunctions) allows us to derive optimized versions of the rules for backward inference, such as Cartesian decomposition and state partitioning (Section 4), from which we obtain a typechecking algorithm that scales to large types. Also, in our emptiness algorithm for alternating tree automata, we exploit various simple facts on Boolean formulas (e.g., a formula $\phi_1 \wedge \phi_2$ denotes an empty set if ϕ_1 does so) to perform efficient shortcuts—these exploited facts are not immediately available in normal tree automata (our emptiness algorithm is omitted from this abstract for lack of space; see our technical report [**?**]).

For preliminary experiments on our implementation, we have written several sizes of transformations and verified against the full XHTML type automatically generated from its DTD. (In reality, transformations are often small, but types that they work on are quite big in many cases; excellent statistical evidences are provided in [17].) The results show that, for this scale of transformations, our implementation has successfully completed typechecking in a reasonable time (about 1 second or less on a stock PC) even with XHTML, which is considered to be quite large. We have also compared the performance of our implementation with Tozawa and Hagiya's [26] and confirmed that ours has comparable speed for their small examples that are used in their own experiments.

Related work Numerous techniques for exact typechecking for XML transformations have been proposed. Many of these take their target languages from the tree transducer family. Those include techniques for macro tree transducers [12, 4], for macro forest transducers [20], for k-pebble tree transducers [16,

[3] Special thanks to Sebastian Maneth providing a simple proof for this: a macro (or even a top-down) tree transducer can produce the tree language over $\{a, b, c\}$ that consists of trees $a(t, t')$ where t' is identical to t except that every symbol b in t is replaced with c in t', but this language is not in context-free tree languages according to [**?**].

4], for subsets of XSLT [23, 26], for high-level tree transducers [24], and a tree transformation language TL [11]. Other techniques treat XML query languages in the select-construct style [15, 1, 13] or even simpler transformations [18, 14]. Most of the above mentioned work provides only theoretical results; the only exceptions are [23, 26], where some experimental results are shown though we have examined much bigger examples (in particular in the size of types).

Several algorithms in pragmatic approaches have been proposed to address high complexity problems related to XML typechecking. A top-down algorithm for inclusion test on tree automata has been developed and used in XDuce typechecker [9]; an improved version is proposed in [22]. A similar idea has been exploited in the work on ℂDuce on the emptiness check for alternating tree automata [6]; the emptiness check algorithm in our present work is strongly influenced by this. Tozawa and Hagiya have developed BDD-based algorithms for inclusion test on tree automata [25] and for satisfiability test on a certain logic related to XML typechecking [26].

Lastly, another relevant piece of work is on static typechecking for XSLT programs by Møller, Olesen, and Schwartzbach [17]. They employ a context-sensitive flow analysis and have experimentally proved its high precision by using a number of style sheets taken from real applications. However, their technique is, in a sense, based on a forward inference and, in theory, cannot be exact (even if we exclude obscure features such as complex conditionals and external function calls). Whether or not the lack of exactness can be problematic in practice is yet to be seen. (A remark worthwhile here is that their analysis is precise enough to typecheck a trivial identity function with respect to a given type.)

Overview This paper is organized as follows. In Section 2, we recall the classical definitions of macro tree transducers (mtt), bottom-up tree automata (bta), and alternating tree automata (ata). In Section 3, we present a basic construction of our backward type inference that produces an ata from an mtt and a deterministic bta. In Section 4, we revisit this construction from a practical point of view and describe important optimizations and implementation techniques. In Section 5, we report the results of our experiments with our implementation of the typechecker for several XML transformations. In Section 6, we conclude this paper with future research directions.

Our accompanied technical report [**?**] describes, in addition to proofs of theorems and our emptiness check for atas, our theoretical contributions omitted from this abstract for lack space. Namely, we establish an exact relationship with two major existing algorithms for mtt typechecking, a classical algorithm based on "function enumeration" [4] and an algorithm proposed by Maneth, Perst, and Seidl [12]. In this, we show that each of these algorithms can be retrieved from ours by composing it with a known algorithm.

2 Preliminaries

2.1 Macro Tree Transducers

We assume an alphabet Σ where each *symbol* $a \in \Sigma$ is associated with its arity; often we write $a^{(n)}$ to denote a symbol a with arity n. We assume that there is a symbol ϵ with zero-arity. *Trees*, ranged over by $v, w, \ldots$, are defined as follows: $v ::= a^{(n)}(v_1, \ldots, v_n)$. We write ϵ for $\epsilon()$ and $\vec{v} = (v_1, \ldots, v_n)$ to represent a tuple of trees. Assume a set of *variables*, ranged over by $x, y, \ldots$. A *macro tree transducer* (mtt) $\mathcal{T}$ is a tuple (P, P_0, Π) where P is a finite set of *procedures*, $P_0 \subseteq P$ is a set of *initial procedures*, and Π is a set of *(transformation) rules* each of the form $p^{(k)}(a^{(n)}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to e$ where each y_i is called *(accumulating) parameter* and e is a (n,k)-expression, defined below. We will abbreviate the tuples $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_k)$ to $\vec{x}$ and $\vec{y}$. Note that each procedure is associated with its arity, i.e., the number of parameters; we write $p^{(k)}$ to denote a procedure p with arity k. An (n,k)-*expression* e is defined by the following grammar

$$e ::= a^{(m)}(e_1, \ldots, e_m) \mid p^{(l)}(x_h, e_1, \ldots, e_l) \mid y_j$$

where only y_j with $1 \leq j \leq k$ and x_h with $1 \leq h \leq n$ can appear as variables. We assume that each initial procedure has arity zero.

We describe the call-by-value semantics of an mtt (P, P_0, Π) by a denotation function $[\![\cdot]\!]$. First, the semantics of a procedure $p^{(k)}$ takes a tree $a^{(n)}(v_1, \ldots, v_n)$ and parameters $\vec{w} = (w_1, \ldots, w_k)$ and returns the set of trees resulting from the evaluation of p's body expressions.

$$[\![p^{(k)}]\!](a^{(n)}(\vec{v}), \vec{w}) = \bigcup_{(p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi} [\![e]\!](\vec{v}, \vec{w})$$

Then, the semantics of an (n,k)-expression e takes a current n-tuple $\vec{v} = (v_1, \ldots, v_n)$ of trees and a k-tuple of parameters $\vec{w} = (w_1, \ldots, w_k)$, and returns a set of trees. It is defined as follows.

$$\begin{aligned}
[\![a^{(m)}(e_1, \ldots, e_m)]\!](\vec{v}, \vec{w}) &= \{a^{(m)}(v'_1, \ldots, v'_m) \mid v'_i \in [\![e_i]\!](\vec{v}, \vec{w}),\ \text{for } i = 1, \ldots, m\} \\
[\![p^{(l)}(x_h, e_1, \ldots, e_l)]\!](\vec{v}, \vec{w}) &= \{[\![p^{(l)}]\!](v_h, (w'_1, \ldots, w'_l)) \mid w'_j \in [\![e_j]\!](\vec{v}, \vec{w}), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{for } j = 1, \ldots, l\} \\
[\![y_j]\!](\vec{v}, \vec{w}) &= \{w_j\}
\end{aligned}$$

Note that an mtt is allowed to inspect only the input tree and never a part of the output tree being constructed. Also, parameters only accumulate subtrees that will potentially become part of the output and never point to parts of the input.

The whole semantics of the mtt with respect to a given input tree v is defined by $\mathcal{T}(v) = \bigcup_{p_0 \in P_0} [\![p_0]\!](v)$. An mtt $\mathcal{T}$ is *deterministic* when $\mathcal{T}(v)$ has at most one element for any v; also, $\mathcal{T}$ is *total* when $\mathcal{T}(v)$ has at least one element for any v. We will also use the classical definition of *images* and *preimages*: $\mathcal{T}(V) = \bigcup_{v \in V} \mathcal{T}(v)$ and $\mathcal{T}^{-1}(V') = \{v \mid \exists v' \in V'.v' \in \mathcal{T}(v)\}$.

2.2 Tree Automata and Alternation

A *(bottom-up) tree automaton* (bta) $\mathcal{M}$ is a tuple (Q, Q_F, Δ) where Q is a finite set of *states*, $Q_F \subseteq Q$ is a set of *final states*, and Δ is a set of *(transition) rules* each of the form $q \leftarrow a^{(n)}(q_1, \ldots, q_n)$ where each q_i is from Q. We will write $\vec{q}$ for the tuple $(q_1, \ldots, q_n)$. Given a bta $\mathcal{M} = (Q, Q_F, \Delta)$, acceptance of a tree by a state is defined inductively as follows: $\mathcal{M}$ *accepts* a tree $a^{(n)}(\vec{v})$ by a state q when there is a rule $q \leftarrow a^{(n)}(\vec{q})$ in Δ such that each subtree v_i is accepted by the corresponding state q_i. $\mathcal{M}$ accepts a tree v when $\mathcal{M}$ accepts v by a final state $q \in Q_F$. We write $[\![q]\!]_{\mathcal{M}}$ for the set of trees that the automaton $\mathcal{M}$ accepts by the state q (we drop the subscript $\mathcal{M}$ when it is clear), and $\mathcal{L}(\mathcal{M}) = \bigcup_{q \in Q_F} [\![q]\!]$ for the set of trees accepted by the automaton $\mathcal{M}$. Also, we sometimes say that a value v *has type* q when v is accepted by the state q. A bta (Q, Q_F, Δ) is *complete and deterministic* when, for any symbol $a^{(n)}$ and n-tuple of states $\vec{q}$, there is exactly one transition rule of the form $q \leftarrow a^{(n)}(\vec{q})$ in Δ. Such a bta is called *deterministic bottom-up tree automaton* (dbta). For any value v, there is exactly one state q such that $v \in [\![q]\!]$. In other words, the collection $\{[\![q]\!] \mid q \in Q\}$ is a partition of the set of trees.

An *alternating tree automaton* (ata) $\mathcal{A}$ is a tuple $(\varXi, \varXi_0, \varPhi)$ where $\varXi$ is a finite set of *states*, $\varXi_0 \subseteq \varXi$ is a set of *initial states*, and $\varPhi$ is a function that maps each pair $(X, a^{(n)})$ of a state and an n-ary symbol to an n-formula, where *n-formulas* are defined by the following grammar.

$$\phi ::= \downarrow_i X \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \top \mid \bot$$

(with $1 \leq i \leq n$). In particular, note that a 0-ary formula evaluates naturally to a Boolean. Given an ata $\mathcal{A} = (\varXi, \varXi_0, \varPhi)$, we define acceptance of a tree by a state. $\mathcal{A}$ *accepts* a tree $a^{(n)}(\vec{v})$ by a state X when $\vec{v} \vdash \varPhi(X, a^{(n)})$ holds, where the judgment $\vec{v} \vdash \phi$ is defined inductively as follows: $\vec{v} \vdash \phi_1 \wedge \phi_2$ if $\vec{v} \vdash \phi_1$ and $\vec{v} \vdash \phi_2$; $\vec{v} \vdash \phi_1 \vee \phi_2$ if $\vec{v} \vdash \phi_1$ or $\vec{v} \vdash \phi_2$; $\vec{v} \vdash \top$; $\vec{v} \vdash \downarrow_i X$ if $\mathcal{A}$ accepts v_i by X. That is, $\vec{v} \vdash \phi$ intuitively means that ϕ holds by interpreting each $\downarrow_i X$ as "v_i has type X." We write $[\![X]\!]$ for the set of trees accepted by a state X and $[\![\phi]\!] = \{\vec{v} \mid \vec{v} \vdash \phi\}$ for the set of n-tuples accepted by an n-formula ϕ. We write $\mathcal{L}(\mathcal{A}) = \bigcup_{X_0 \in \varXi_0} [\![X_0]\!]$ for the language accepted by the ata $\mathcal{A}$. Note that a bta $\mathcal{M} = (Q, Q_F, \Delta)$ can be seen as an ata with the same set of states and final states by defining the function $\varPhi$ as $\varPhi(q, a^{(n)}) = \bigvee_{(q \leftarrow a^{(n)}(\vec{q})) \in \Delta} \bigwedge_{i=1,\ldots,n} \downarrow_i q_i$, and the definitions for the semantics of states and the language accepted by the automaton seen as a bta or an ata then coincide. We will use the notation $\simeq$ to represent semantical equivalence of pairs of states or pairs of formulas.

3 Typechecking

Given a dbta $\mathcal{M}_{\text{out}}$ ("output type"), a bta $\mathcal{M}_{\text{in}}$ ("input type"), and an mtt $\mathcal{T}$, the goal of typechecking is to verify that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}})) \subseteq \mathcal{L}(\mathcal{M}_{\text{out}})$. It is well known that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}}))$ is in general beyond regular tree languages and hence the forward inference approach (i.e., first calculate an automaton representing

$\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}}))$ and check it to be included in $\mathcal{L}(\mathcal{M}_{\text{out}}))$ does not work. Therefore an approach usually taken is the backward inference, which is based on the observation that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}})) \subseteq \mathcal{L}(\mathcal{M}_{\text{out}}) \iff \mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M})) = \emptyset$, where $\mathcal{M}$ is the complement automaton of $\mathcal{M}_{\text{out}}$. Intuitively, if the intersection $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ is not empty, then it is possible to exhibit a tree v in this intersection; since this tree satisfies that $v \in \mathcal{L}(\mathcal{M}_{\text{in}})$ and $\mathcal{T}(v) \not\subseteq \mathcal{L}(\mathcal{M}_{\text{out}})$, it means that there is a counter-example of the well-typedness of the mtt with respect to the given input and output types. Algorithmically, the approach consists of computing an automaton $\mathcal{A}$ representing $\mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ and then checking that $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(\mathcal{A}) = \emptyset$. Since the language $\mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ is regular and indeed such automata $\mathcal{A}$ can effectively be computed, the above disjointness is decidable.

The originality of our approach is to compute $\mathcal{A}$ as an alternating tree automaton. Let a dbta $\mathcal{M} = (Q, Q_F, \Delta)$ and an mtt $\mathcal{T} = (P, P_0, \Pi)$ be given. Here, note that the automaton $\mathcal{M}$, which denotes the complement of the output type $\mathcal{M}_{\text{out}}$, can be obtained from $\mathcal{M}_{\text{out}}$ in a linear time since $\mathcal{M}_{\text{out}}$ is deterministic. From $\mathcal{M}$ and $\mathcal{T}$, we build an ata $\mathcal{A} = (\Xi, \Xi_0, \Phi)$ where

$$\begin{array}{ll}
\Xi & = \{\langle p^{(k)}, q, \vec{q}\rangle \mid p^{(k)} \in P,\ q \in Q, \vec{q} \in Q^k\} \\
\Xi_0 & = \{\langle p_0, q\rangle \mid p_0 \in P_0,\ q \in Q_F\} \\
\Phi(\langle p^{(k)}, q, \vec{q}\rangle, a^{(n)}) = & \displaystyle\bigvee_{(p^{(k)}(a^{(n)}(\vec{x}),\vec{y}) \to e) \in \Pi} \operatorname{Inf}(e, q, \vec{q}).
\end{array}$$

Here, the function Inf is defined inductively as follows.

$$\begin{array}{ll}
\operatorname{Inf}(b^{(m)}(e_1, \ldots, e_m), q, \vec{q}) & = \displaystyle\bigvee_{(q \leftarrow b^{(m)}(\vec{q'})) \in \Delta} \bigwedge_{j=1,\ldots,m} \operatorname{Inf}(e_j, q'_j, \vec{q}) \\
\operatorname{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), q, \vec{q}) & = \displaystyle\bigvee_{\vec{q'} \in Q^l} \left(\downarrow_h \langle p^{(l)}, q, \vec{q'}\rangle \wedge \bigwedge_{j=1,\ldots,l} \operatorname{Inf}(e_j, q'_j, \vec{q}) \right) \\
\operatorname{Inf}(y_j, q, \vec{q}) & = \begin{cases} \top & (q = q_j) \\ \bot & (q \neq q_j) \end{cases}
\end{array}$$

Intuitively, each state $\langle p, q, \vec{q}\rangle$ represents the set of trees v such that the procedure p may transform v to some tree u of type q, assuming that the parameters y_i are bound to trees w_i each of type q_i. Formally, we can prove the following invariant

$$\forall \vec{w} \in [\![\vec{q}]\!].\ v \in [\![\langle p^{(k)}, q, \vec{q}\rangle]\!] \iff [\![p^{(k)}]\!](v, \vec{w}) \cap [\![q]\!] \neq \emptyset \qquad (1)$$

where $\vec{w} \in [\![\vec{q}]\!]$ means $w_1 \in [\![q_1]\!], \ldots, w_k \in [\![q_k]\!]$. Note that this invariant implies that whether the right-hand side holds or not does not depend on the specific choice of the values w_i from the sets $[\![q_i]\!]$. From this invariant, the initial states Ξ_0 represent the set of trees that we want. Then, the function $\operatorname{Inf}(e, q, \vec{q})$ infers an n-formula representing the set of n-tuples $\vec{v}$ such that the expression e may transform $\vec{v}$ to some tree of type q, assuming that the parameters y_i are bound to trees w_i each of type q_i. Each case can be understood as follows.

- In order for a tree u of type q to be produced from the constructor expression $b^{(m)}(e_1, \ldots, e_m)$, first, there must be a transition $q \leftarrow b^{(m)}(\vec{q'}) \in \Delta$. In addition, u's each subtree must have type q'_i and must be produced from the corresponding subexpression e_i.
- In order for a tree u of type q to be produced from the procedure call $p(x_h, e_1, \ldots, e_l)$, first, a tree w'_j of some type q'_j must be yielded from each parameter expression e_j. In addition, the h-th input tree must have type $\langle p, q, (q'_1, \ldots, q'_l)\rangle$ since the result tree u must be produced by the procedure p from the h-th tree with parameters $w'_1, \ldots, w'_l$ of types $q'_1, \ldots, q'_l$.
- In order for a tree of type q to be produced from the variable expression y_j, this variable must have type q.

Theorem 1. $\mathcal{L}(\mathcal{A}) = \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$.

Finally, it remains to check $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(A) = \emptyset$, for which we first calculate an ata $\mathcal{A}'$ representing $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(A)$ (this can easily be done since an ata can freely use intersections) and then check the emptiness of $\mathcal{A}'$. For lack of space, we give our emptiness checking algorithm in [**?**].

Note that the size of the ata $\mathcal{A}$ is polynomial in the sizes of $\mathcal{M}_{\text{out}}$ and of $\mathcal{T}$. The size of $\mathcal{A}'$ is thus polynomial in the sizes of $\mathcal{M}_{\text{in}}$, $\mathcal{M}_{\text{out}}$, and $\mathcal{T}$.

4 Optimization techniques

In this section, we describe some optimization techniques for speeding up the backward inference presented in Section 3.

A simple algorithm to compute the input type as an alternating tree automaton is to follow naively the formal construction given in Section 3. A first observation is that it is possible to build the automaton lazily, starting from the initial states, producing new states and computing $\Phi(\_)$ only on demand. This is sometimes useful since our emptiness check algorithm [**?**] works in a top-down way and will not always materialize the whole automaton.

The defining equations for the function Inf as given in Section 3 produce huge formulas. We will now describe new equations that produce much smaller formulas in practice. Before describing them, it is convenient to generalize the notation $\mathrm{Inf}(e, q, \vec{q})$ by allowing a *set of states* $\overline{q} \subseteq Q$ instead of a single state $q \in Q$ for the output type. Intuitively, we want $\mathrm{Inf}(e, \overline{q}, \vec{q})$ to be semantically equivalent to $\bigvee_{q \in \overline{q}} \mathrm{Inf}(e, q, \vec{q})$. We obtain a direct definition of $\mathrm{Inf}(e, \overline{q}, \vec{q})$ by adapting the rules for $\mathrm{Inf}(e, q, \vec{q})$:

$$\begin{aligned}
\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) &= \bigvee_{(q \leftarrow b^{(m)}(\vec{q'})) \in \Delta, q \in \overline{q}} \; \bigwedge_{j=1\ldots,m} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \\
\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q}) &= \bigvee_{\vec{q'} \in Q^l} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \vec{q'} \rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \right) \\
\mathrm{Inf}(y_j, \overline{q}, \vec{q}) &= \begin{cases} \top \; (q_j \in \overline{q}) \\ \bot \; (q_j \notin \overline{q}) \end{cases}
\end{aligned}$$

We have used the notation $\downarrow_h \langle p^{(l)}, \overline{q}, \vec{q'} \rangle$. Intuitively, this should be semantically equivalent to the union $\bigvee_{q \in \overline{q}} \downarrow_h \langle p^{(l)}, q, \vec{q'} \rangle$. Instead of using this as a definition, we prefer to change the set of states of the automaton:

$$\begin{array}{ll} \Xi & = \{\langle p^{(k)}, \overline{q}, q_1, \ldots, q_k \rangle \mid p^{(k)} \in P,\ \overline{q} \subseteq Q, \vec{q} \in Q^k\} \\ \Xi_0 & = \{\langle p_0, Q_F \rangle \mid p_0 \in P_0\} \\ \Phi(\langle p^{(k)}, \overline{q}, \vec{q} \rangle, a^{(n)}) & = \bigvee_{(p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi} \mathrm{Inf}(e, \overline{q}, \vec{q}). \end{array}$$

In theory, this new alternating tree automaton could have exponentially many more states. However, in practice, and because of the optimizations we will describe now, this actually reduces significantly the number of states that need to be computed.

The sections below will use the semantical equivalence $\bigvee_{q \in \overline{q}} \mathrm{Inf}(e, \{q\}, \vec{q}) \simeq \mathrm{Inf}(e, \overline{q}, \vec{q})$ mentioned above in order to simplify formulas.

Cartesian factorization The rule for the constructor expression $b^{(m)}(e_1, \ldots, e_m)$ can be rewritten:

$$\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) = \bigvee_{\vec{q'} \in \Delta(\overline{q}, b^{(m)})} \bigwedge_{j=1\ldots,m} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q})$$

where $\Delta(\overline{q}, b^{(m)}) = \{\vec{q'} \mid q \leftarrow b^{(m)}(\vec{q'}) \in \Delta, q \in \overline{q}\} \subseteq Q^m$. Now assume that we have a decomposition of this set $\Delta(\overline{q}, b^{(m)})$ as a union of l Cartesian products:

$$\Delta(\overline{q}, b^{(m)}) = (\overline{q}_1^1 \times \ldots \times \overline{q}_m^1) \cup \ldots \cup (\overline{q}_1^l \times \ldots \times \overline{q}_m^l)$$

where the $\overline{q}_j^i$ are sets of states. It is always possible to find such a decomposition: at worst, using only singletons for the $\overline{q}_j^i$, we will have as many terms in the union as m-tuples in $\Delta(\overline{q}, b^{(m)})$. But often, we can produce a decomposition with fewer terms in the union. Let us write $\mathrm{Cart}(\Delta(\overline{q}, b^{(m)})$ for such a decomposition (seen as a subset of $(2^Q)^m$). One can then use the following rule:

$$\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) = \bigvee_{(\overline{q}_1, \ldots, \overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q}, b^{(m)}))} \bigwedge_{j=1,\ldots,m} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q})$$

State partitioning

Intuition The rule for procedure call enumerates all the possible states for the values of parameters of the called procedure. In its current form, this rule always produces a big union with $|Q|^l$ terms. However, it may be the case that we don't need fully precise information about the value of a parameter to do the backward type inference.

Let us illustrate that with a simple example. Assume that the called procedure $p^{(1)}$ has a single parameter y_1 and that it never does anything else with y_1 than copying it (that is, any rule for p whose right-hand side mentions y_1

is of the form $p^{(1)}(a^{(n)}(x_1,\ldots,x_n),y_1) \to y_1$). Clearly, all the states $\langle p,\overline{q},q'_1\rangle$ with $q'_1 \in \overline{q}$ are equivalent, and similarly for all the states $\langle p,\overline{q},q''_1\rangle$ with $q''_1 \notin \overline{q}$. This is because whether the result of the procedure call will be or not in $\overline{q}$ only depends on the input tree (because there might be other rules whose right-hand side doesn't involve y_1 at all) and on whether the value for the parameter is itself in $\overline{q}$ or not. In particular, we don't need to know exactly in which state the accumulator is. So the rule for calling this procedure could just be:

$$
\begin{aligned}
&\mathrm{Inf}(p(x_h,e_1),\overline{q},\vec{q})\\
&\quad= \bigvee_{q'_1\in Q} \downarrow_h \langle p,\overline{q},q'_1\rangle \wedge \mathrm{Inf}(e_1,\{q'_1\},\vec{q})\\
&\quad= \left(\bigvee_{q'_1\in \overline{q}} \downarrow_h \langle p,\overline{q},q'_1\rangle \wedge \mathrm{Inf}(e_1,\{q'_1\},\vec{q})\right)\\
&\qquad \vee \left(\bigvee_{q''_1\in Q\backslash\overline{q}} \downarrow_h \langle p,\overline{q},q''_1\rangle \wedge \mathrm{Inf}(e_1,\{q''_1\},\vec{q})\right)\\
&\quad= (\downarrow_h \langle p,\overline{q},q'_1\rangle \wedge \mathrm{Inf}(e_1,\overline{q},\vec{q})) \vee (\downarrow_h \langle p,\overline{q},q''_1\rangle \wedge \mathrm{Inf}(e_1,Q\backslash\overline{q},\vec{q}))
\end{aligned}
$$

where in the last line q'_1 (resp. q''_1) is chosen arbitrarily from $\overline{q}$ (resp. $Q\backslash\overline{q}$).

A new rule More generally, in the rule for a call to a procedure $p^{(l)}$, we don't need to consider all the l-tuples $\vec{q'}$, but only a subset of them that capture all the possible situations. First, we assume that for given procedure $p^{(l)}$ and output type $\overline{q}$, one can compute for each $j=1,\ldots,l$ an equivalence relation $E\langle p^{(l)},\overline{q},j\rangle$ such that:

$$(\forall j=1,\ldots,l.\ (q'_j,q''_j)\in E\langle p^{(l)},\overline{q},j\rangle) \Rightarrow \langle p^{(l)},\overline{q},\vec{q'}\rangle \simeq \langle p^{(l)},\overline{q},\vec{q''}\rangle \quad (*)$$

Let us see again the right-hand side of the definition for $\mathrm{Inf}(p^{(l)}(x_h,e_1,\ldots,e_l),\overline{q},\vec{q})$:

$$\mathrm{Inf}(p^{(l)}(x_h,e_1,\ldots,e_l),\overline{q},\vec{q}) = \bigvee_{\vec{q'}\in Q^l}\left(\downarrow_h \langle p^{(l)},\overline{q},\vec{q'}\rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j,\{q'_j\},\vec{q})\right)$$

Let us split this union according to the equivalence class of the q'_j modulo the relations $E\langle p^{(l)},\overline{q},j\rangle$. If for each j, we choose an equivalence class $\overline{q}_j$ for the relation $E\langle p^{(l)},\overline{q},j\rangle$ (we write $\overline{q}_j \lhd E\langle p^{(l)},\overline{q},j\rangle$), then all the states $\langle p^{(l)},\overline{q},\vec{q'}\rangle$ with $\vec{q'}\in \overline{q}_1\times\ldots\times\overline{q}_l$ are equivalent to $\langle p^{(l)},\overline{q},\mathrm{C}(\overline{q}_1\times\ldots\times\overline{q}_l)\rangle$, where C is a choice function (it picks an arbitrary element from its argument). We can thus rewrite the right hand-side to:

$$\bigvee_{\overline{q}_1\lhd E\langle p^{(l)},\overline{q},1\rangle,\ldots,\overline{q}_l\lhd E\langle p^{(l)},\overline{q},l\rangle} \left(\downarrow_h \langle p^{(l)},\overline{q},\mathrm{C}(\overline{q}_1\times\ldots\times\overline{q}_l)\rangle \wedge \bigvee_{\vec{q'}\in\overline{q}_1\times\ldots\times\overline{q}_l} \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j,\{q'_j\},\vec{q})\right)$$

The union of all the formulas $\bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q})$ for $\vec{q'} \in \overline{q}_1 \times \ldots \times \overline{q}_l$ is equivalent to $\bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q})$. Consequently, we obtain the following new rule:

$$\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q}) = \bigvee_{\overline{q}_1 \lhd E\langle p^{(l)}, \overline{q}, 1\rangle, \ldots, \overline{q}_l \lhd E\langle p^{(l)}, \overline{q}, l\rangle} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \mathrm{C}(\overline{q}_1 \times \ldots \times \overline{q}_l)\rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q}) \right)$$

In the worst case, all the equivalence relations $E\langle p^{(l)}, \overline{q}, j\rangle$ are the identity, and the right-hand side is the same as for the old rule. But if we can identify larger equivalence classes, we can significantly reduce the number of terms in the union on the right-hand side.

Computing the equivalence relations Now we will give an algorithm to compute the relations $E\langle p^{(k)}, \overline{q}, j\rangle$ satisfying the condition $(*)$. We will also define equivalence relations $E[e, \overline{q}, j]$ for any (n, k)-expression e (with $j = 1, \ldots, k$), such that:

$$(\forall j = 1, \ldots, k.(q'_j, q''_j) \in E[e, \overline{q}, j]) \Rightarrow \mathrm{Inf}(e, \overline{q}, \vec{q'}) \simeq \mathrm{Inf}(e, \overline{q}, \vec{q''})$$

We can use the rules used to define the formulas $\mathrm{Inf}(e, \overline{q}, \vec{q})$ in order to obtain sufficient conditions to be satisfied so that these properties hold. We will express these conditions by a system of equations. Before giving this system, we need to introduce some notations. If E_1 and E_2 are two equivalence relations on Q, we write $E_1 \sqsubseteq E_2$ if $E_2 \subseteq E_1$ (when equivalence relations are seen as subsets of Q^2). The smallest equivalence relation for this ordering is the equivalence relation with a single equivalence class. The largest equivalence relation is the identity on Q. For two equivalence relations E_1, E_2, we can define their least upper bound $E_1 \sqcup E_2$ as the set-theoretic intersection. For an equivalence relation E and a set of states $\overline{q}$, we write $\overline{q} \lhd E$ if $\overline{q}$ is one of the equivalence class modulo E. Abusing the notation by identifying an equivalence relation with the partition it induces on Q, we will write $\{Q\}$ for the smallest relation and $\{\overline{q}, Q\backslash\overline{q}\}$ for the relation with the two equivalence classes $\overline{q}$ and its complement. The system of equations is derived from the rules used to define the function Inf:

$$\begin{array}{ll}
E[b^{(m)}(e_1, \ldots, e_m), \overline{q}, i] & \sqsupseteq \bigsqcup \{E[e_j, \overline{q}_j, i] \mid (\overline{q}_1, \ldots, \overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q}, b^{(m)})), \\
 & \qquad\qquad \text{for } j = 1, \ldots, m\} \\
E[p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, i] & \sqsupseteq \bigsqcup \{E[e_j, \overline{q}_j, i] \mid \overline{q}_j \lhd E\langle p^{(l)}, \overline{q}, j\rangle, \text{ for } j = 1, \ldots, l\} \\
E[y_j, \overline{q}, i] & \sqsupseteq \begin{cases} \{\overline{q}, Q\backslash\overline{q}\} & (i = j) \\ \{Q\} & (i \neq j) \end{cases} \\
E\langle p^{(k)}, \overline{q}, j\rangle & \sqsupseteq \bigsqcup \{E[e, \overline{q}, j] \mid p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi\}
\end{array}$$

Let us explain why these conditions imply the required properties for the equivalence relation and how they are derived from the rules defining Inf. We will use an intuitive induction argument (on expressions), even though a formal proof actually requires an induction on trees. Consider the rule for the procedure call. The new rule we have obtained above implies that in order to have $\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q'}) \simeq \mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q''})$, it is sufficient to have $\mathrm{Inf}(e_j, \overline{q}_j, \vec{q'}) \simeq \mathrm{Inf}(e_j, \overline{q}_j, \vec{q''})$ for all $j = 1, \ldots, l$ and for all $\overline{q}_j \triangleleft E\langle p^{(l)}, \overline{q}, j\rangle$, and thus, by induction, it is also sufficient to have $(q'_i, q''_i) \in E[e_j, \overline{q}_j, i]$ for all i, for all $j = 1, \ldots, l$ and for all $\overline{q}_j \triangleleft E\langle p^{(l)}, \overline{q}, j\rangle$. In other words, a sufficient condition is $(q'_i, q''_i) \in \bigcap\{E[e_j, \overline{q}_j, i] \mid \overline{q}_j \triangleleft E\langle p^{(l)}, \overline{q}, j\rangle,\ j = 1, \ldots, l\}$, from which we obtain the equation above (we recall that $\sqcup$ corresponds to set-theoretic intersection of relations). The reasoning is similar for the constructor expression. Indeed, the rule we have obtained in the previous section tells us that in order to have $\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q'}) \simeq \mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q''})$, it is sufficient to have $\mathrm{Inf}(e_j, \overline{q}_j, \vec{q'}) \simeq \mathrm{Inf}(e_j, \overline{q}_j, \vec{q''})$ for all $(\overline{q}_1, \ldots, \overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q}, b^{(m)}))$ and $j = 1, \ldots, m$.

As we explained before, it is desirable to compute equivalence relations with large equivalence classes (that is, small for the $\sqsubseteq$ ordering). Here is how we can compute a family of equivalence relations satisfying the system of equations above. First, we consider the CPO of functions mapping a triple $(e, \overline{q}, i)$ to an equivalence relation on Q and we reformulate the system of equation as finding an element x of this CPO such that $f(x) \sqsubseteq x$, where f is obtained from the right-hand sides of the equations. To compute such an element, we start from x_0 the smallest element of the CPO, and we consider the sequence defined by $x_{n+1} = x_n \sqcup f(x_n)$. Since this sequence is monotonic and the CPO is finite, the sequence reaches a constant value after a finite number of iterations. This value x satisfies $f(x) \sqsubseteq x$ as expected. We conjecture that this element is actually a smallest fixpoint for f, but we have no proof of this fact (note that the function f is not monotonic).

Sharing the computation Given the rules defining the formulas $\mathrm{Inf}(e, \overline{q}, \vec{q})$, we might end up computing the same formula several times. A very classical optimization consists in memoizing the results of such computations. This is made even more effective by hash-consing the expressions. Indeed, in practice, for a given mtt procedure, many constructors have identical expressions.

Complementing the output In the example at the beginning of the section on "state partitioning," we have displayed a formula where both $\mathrm{Inf}(e, \overline{q}, \vec{q})$ and $\mathrm{Inf}(e, Q\backslash\overline{q}, \vec{q})$ appear. One may wonder what the relation is between these two sub-formulas. Let us recall the required properties for these two formulas:

$$[\![\mathrm{Inf}(e, \overline{q}, \vec{q})]\!] = \{v \mid [\![p]\!](\vec{v}, \vec{w}) \cap [\![\overline{q}]\!] \neq \emptyset\}$$

$$[\![\mathrm{Inf}(e, Q\backslash\overline{q}, \vec{q})]\!] = \{v \mid [\![p]\!](\vec{v}, \vec{w}) \cap [\![Q\backslash\overline{q}]\!] \neq \emptyset\}$$

(for $\vec{w} \in [\![\vec{q}]\!]$). Note that $[\![Q\backslash\overline{q}]\!]$ is the complement of $[\![\overline{q}]\!]$. As a consequence, if $[\![p]\!]$ is a total deterministic function (that is, if $[\![p]\!](\vec{v}, \vec{w})$ is always a singleton), then $[\![\mathrm{Inf}(e, Q\backslash\overline{q}, \vec{q})]\!]$ is the complement of $[\![\mathrm{Inf}(e, \overline{q}, \vec{q})]\!]$. If we extend the syntax of formula in alternating tree automata with negation (whose semantics is trivial to define), we can thus introduce the following rule:

$$\mathrm{Inf}(e, \overline{q}, \vec{q}) = \neg\mathrm{Inf}(e, Q\backslash\overline{q}, \vec{q})$$

to be applied e.g. when the cardinal of $\overline{q}$ is strictly larger than half the cardinal of Q. In practice, we observed a huge impact of this optimization: the number of constructed states is divided by two in all our experiences, and the emptiness algorithm runs much more efficiently. Also, because of the memoization technique mentioned above, this optimization allows us to share more computation. That said, we don't yet have a deeper understanding of the very important impact of this optimization.

The rule above can only be applied when the expression e denotes a total and deterministic function. We use a very simple syntactic criterion to ensure that: we require all the reachable procedures $p^{(k)}$ to have exactly one rule $p^{(k)}(a^{(n)}(x_1, \ldots, x_n), y_1, \ldots, y_k) \rightarrow e$ for each symbol $a^{(n)}$.

5 Experiments

We have experimented on our typechecker with various XML transformations implemented as mtts. Although we did not try very big transformations, we did work with large input and output tree automata automatically generated from the XHTML DTD (without taking XML attributes into account). Note that because this DTD has many tags, the mtts actually have many transitions since they typically copy tags, which requires all constructors corresponding to these tags to be enumerated. They do not have too many procedures, though. The bottom-up deterministic automaton that we generated from the XHTML DTD has 35 states.

Table 1 gives the elapsed times spent in typechecking several transformations and the number of states of the inferred alternating tree automaton that have been materialized. The experiment was conducted on an Intel Pentium 4 processor 2.80Ghz, running Linux kernel 2.4.27, and the typechecking time includes the whole process (determinization of the output type, backward inference, intersection with the input type, and emptiness check). The typechecker is implemented in and compiled by Objective Caml 3.09.3.

We also indicate the number of procedures in each mtt, the maximum number of parameters, and the minimum integer b, if any, such that the mtt is syntactically b-bounded copying. Intuitively, the integer b captures the maximum number of times the mtt traverses any node of the input tree. This notion has been introduced in [12] where the existence of b is shown to imply the polynomiality of the algorithm described in that paper (see [**?**]). Here, we observe that even unbounded-copying mtts can be typechecked efficiently.

Unless otherwise stated, transformations are checked to have type `XHTML`$\rightarrow$`XHTML` (i.e., both input and output types are `XHTML`). Transformation (1) removes all the `<b>` tags, keeping their contents. Transformation (2) is a variant that drops

Transformation:	(1)	(2)	(3)	(4)	(5)	(6)	(7)
# of procedures:	2	2	3	5	4	6	6
Max # of parameters:	1	1	1	1	2	2	2
Bounded copying:	1	1	2	∞	∞	2	1
Type-checking time (ms):	1057	1042	0373	0377	0337	0409	0410
# of states in the ata:	147	147	43	74	37	49	49

Table 1. Results of the experiments

the `<div>` tags instead. The typechecker detects that the latter doesn't have type `XHTML`→`XHTML` by producing a counter-example:

```
<html><head><title/></head><body><div/></body>
```

Indeed, removing the `<div>` element may produce a `<body>` element with an empty content, which is not valid in XHTML. This kind of error is quite common in XML transformations but it is difficult to find with testing or with a simple type system. Transformation (3) copies all the `<a>` elements (and their corresponding subtrees) into a new `<div>` element and prepends the `<div>` to the `<body>` element. Transformation (4) groups together adjacent `<b>` elements, concatenating their contents. Transformation (5) extracts from an XHTML document a tree of depth 2 which represents the conceptual nesting structure of `<h1>` and `<h2>` heading elements (note that, in XHTML, the structure among headings is flat). Transformation (6) builds a tree representing a table of contents for the top two levels of itemizations, giving section and subsection numbers to them (where the numbers are constructed as Peano numerals), and prepends the resulting tree to the `<body>` element. Transformation (7) is a variant that only returns the table of contents.

We have also translated some transformations (that can be expressed as mtts) used by Tozawa and Hagiya in [26] (namely `htmlcopy`, `inventory`, `pref2app`, `pref2html`, `prefcopy`). Our implementation takes between 2ms and 6ms to typecheck these mtts, except for `inventory` for which it takes 22 ms. Tozawa and Hagiya report performance between 5ms and 1000ms on a Pentium M 1.8 Ghz for the satisfiability check (which corresponds to our emptiness check and excludes the time taken by backward inference). Although these results indicate our advantages over them to some extent, since the numbers are too small and they have not undertaken experiments as big as ours, it is hard to draw a meaningful conclusion.

6 Conclusion and Future Work

We have presented an efficient typechecking algorithm for mtts based on the idea of using alternating tree automata for representing the preimage of the given mtt obtained from the backward type inference. This representation was useful for deriving optimization techniques on the backward inference phase such as state partitioning and Cartesian factorization, and was also effective for speeding up the subsequent emptiness check phase by exploiting Boolean equivalences among formulas. Our experimental results confirmed that our techniques allow us to typecheck small sizes of transformations with respect to the full XHTML type.

The present work is only the first step toward a truly practical typechecker for mtts. In the future, we will seek for further improvements that allow typechecking larger and more complicated transformations. In particular, transformations with upward axes can be obtained by compositions of mtts as proved in [11] and a capability to typecheck such compositions of mtts in a reasonable time will be important. We have some preliminary ideas for the improvement and plan to pursue them as a next step. In the end, we hope to be able to handle (at least a reasonably large subset of) XSLT.

References

1. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, 2001.
2. A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, Aug. 1991.
3. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.
4. J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9):613–698, 2003.
5. J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):710–146, 1985.
6. A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis, Université Paris 7, 2004.
7. H. Hosoya. Regular expression filters for XML. *Journal of Functional Programming*, 16(6):711–750, 2006. Short version appeared in Proceedings of Programming Technologies for XML (PLAN-X), pp.13–27, 2004.
8. H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.
9. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2004. Short version appeared in Proceedings of the International Conference on Functional Programming (ICFP), pp.11-22, 2000.
10. X. Leroy, D. Doligez, J. Garrigue, J. Vouillon, and D. Rémy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.
11. S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 283–294, 2005.

12. S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *International Conference on Database Theory (ICDT)*, pages 254–268, 2007.
13. W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In *Proceedings of International Conference on Database Theory*, pages 64–78, 2003.
14. W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 23–34, 2004.
15. T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of Symposium on Principles of Database Systems*, pages 215–226, Philadelphia, May 1999.
16. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.
17. A. Møller, M. Ø. Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005. Draft, accepted for TOPLAS.
18. M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.
19. K. Nakano and S.-C. Mu. A pushdown machine for recursive XML processing. In *APLAS*, pages 340–356, 2006.
20. T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.
21. G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.
22. T. Suda and H. Hosoya. Non-backtracking top-down algorithm for checking tree automata containment. In *Proceedings of Conference on Implementation and Applications of Automata (CIAA)*, pages 83–92, 2005.
23. A. Tozawa. Towards static type checking for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.
24. A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming (FLOPS)*, pages 81–96, 2006.
25. A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *8th International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 213–225. Springer-Verlag, 2003.
26. A. Tozawa and M. Hagiya. Efficient decision procedure for a logic for XML. unpublished manuscipt, 2004.

Acknowledgments This work is partly supported by Japan Society for the Promotion of Science and by The Okawa Foundation for Information and Telecommunications. We are grateful to Sebastian Maneth for useful discussions.