Vol. 46 No. SIG 14(PRO 27) IPSJ Transactions on Programming Oct. 2005

Regular Paper

A Rewrite System with Incomplete Regular Expression Type for Transformation of XML Documents

Taro Suzuki[†] and Satoshi Okui[††]

In this paper we propose a new framework for transformations of XML documents based on an extension of regular expression type, called *incomplete* regular expression type. Incomplete regular expression type is a restricted second-order extension of regular expression (RE) type: An incomplete RE type can be applied to *arbitrary* RE types as its arguments. It is accomplished by introducing holes to types. We give a version of second-order rewrite systems, designed founded on our type system. Matching between a subterm with the left-hand side of rewrite rules yields a substitution that bind second-order terms to the variables of incomplete type. This feature is practically useful when we want to reuse "the upper parts" of XML documents in the transformation. We demonstrate that the introduction of incomplete regular expression type allows programmers much flexibility. In this paper, we mainly concentrate on the type theoretic issues and give some properties concerning for incomplete types. We also show the type preservation in rewriting.

1. Introduction

Statically typed XML transformation languages such as XDuce[8] has brought benefits to XML world; it makes us possible to statically analyze XML documents and their transformation procedures with type checking. XDuce provides a powerful pattern matching mechanism called regular expression pattern matching, based on regular expression type. However, it can extract only the substructures from a given XML document because the nature of matching in XDuce is in *first-order*.

Higher-order matching allows us to extract not only the substructures but also the functions. Higher-order matching is broadly used in program transformation[7),10)]. One of our aim is to gain the expressiveness of statically typed XML languages by extending them with the second-order matching. The second-order matching is useful when we want to extract "upper-parts" of subterms of a given term and reuse them. We will show an example in the next section.

Our another aim is to give a framework for the transformation of XML documents with much flexibility. For this purpose we adopt term rewriting systems (TRSs, for short). A TRS is a model of computation with much flexibility. For instance, the reduction strategy is not fixed. Many reduction strategies are proposed in the literature (See Refs. 1), 3) and 13) for further details). We can select one of them and combine it to the concerned TRSs in order to perform the intended computation. On the other hand, in statically typed XML transformation languages the situation is different; for instance, XDuce is based on XML and hence it employs the innermost reduction strategy.

It is worth noting that recently *higher-order* term rewriting systems are intensively investigated[17)], including some systems with higher-order matching[12),16)]. We expect to take advantage of useful results obtained from these works.

We propose a framework for statically typed XML transformation with a restricted form of the *second-order* matching. In this paper we mainly concentrate on the type theoretic issues.

This paper is organized as follows. In the next section we describe our basic idea by illustrating an example. In Section 3 we show formal definitions and some properties of our framework. In Section 4 we revisit the example in Section 2 more precisely with formalities defined in Section 3. Finally we conclude the paper with some remarks and the further research.

2. Basic Idea

We explain our basic idea using a simple example. Suppose a fragment of an XML document represents a word with specified font properties such as a name, a size, a weight and a style. The type of words with font properties

† The University of Aizu
†† Chubu University

is denoted by `Word`, which is defined as follows (we adopt notation similar to XDuce in this section).

```
  type Word =
           FL[WL[SL[String[]]] Int[]]
```

where `FL`, `WL` and `SL` are abbreviations of the unions of labels as denoted below.

```
  FL = times | helvetica | courier
  WL = bold | normal | thin
  SL = italic | roman
```

For example the word `"Hello"` with font name `times`, font size 12, font weight `bold` and font shape `italic` is represented as follows.

```
   times[bold[italic["Hello"]] 12]
```

Now we would like to transform the representation of the font properties from 'nested' forms to 'flat' forms. The new type of the words with font properties is `NewWord` defined as follows.

```
  type NewWord =
          word[String[]
                font[FL[Int[]] WL[] SL[]]]
```

The word transformed from the above is shown below.

```
  word["Hello"
        font[times[12] bold[] italic[]]]
```

The implementation of this transformation in terms of first-order matching is clumsy: actually, the number of rules to implement the transformation is equal to sum of the labels for the font properties.

```
  flatten(times[x s]) =
              flat(x, times[s])
  flatten(helvetica[x s]) =
              flat(x, helvetica[s])
  flatten(courier[x s]) =
              flat(x, courier[s])
  flat(bold[x], l) =
              flat2(x, l bold[])
  flat(normal[x], l) =
              flat2(x, l normal[])
  flat(thin[x], l) =
              flat2(x, l thin[])
  flat2(roman[w], l) =
              word[w font[l roman[]]]
  flat2(italic[w],l) =
              word[w font[l italic[]]]
```

Here the italic fonts denote (first-order) variables.

On the other hand, second-order matching provides us more direct and simpler implementation. In this paper we instantiate the second-order variables with *contexts*. Contexts are terms with special constants □ called *holes*. This restriction gives us a restricted form of second-order matching, so-called *context matching*[5),6),14)]. The holes act as pairwisely distinct bound variables. Thus, a context $l[\Box\ l'[\Box]]$ corresponds with $\lambda xy.l[x\ l'[y]]$. The holes can be replaced with the other terms later on: $l[\Box\ l'[\Box]]\{t, t'\} = l[t\ l'[t']]$.

Now that the transformation from nested representation to flat one of font properties is implemented by a single rule.

```
  flatten2(fl{wl{sl{s}}}) =
       word[s font[fl{()} wl{()} sl{()}]]
```

Here fl, wl and sl are second-order variables. When the above rule is applied to `times[bold[italic["Hello"]] 12]`, they are instantiated with contexts with a single hole as follows.

```
  fl  = times[□ 12]
  wl = bold[□]
  sl  = italic[□]
```

The desired transformed XML fragment is obtained by instantiating the right-hand side of the above rule.

Note that in the instance of each side of the rule the holes are replaced by the terms with *different types*. In the above example, the hole in `italic[□]`, substituted for sl, is replaced with `"Hello"` in the instance of the left-hand side, whereas with the empty sequence `()` in the instance of the right-hand side. In order to replace a single hole by terms with different types, we introduce holes also in types; the second-order variable sl has type `SL[□]`. Likewise, the other second-order variables fl and wl are typed with `FL[□ Int[]]` and `WL[□]`, respectively. As well as holes in terms, the holes in types can be replaced with the other types later on. So the argument of the left-hand side is typed with

```
  FL[□ Int[]]{WL[□]{SL[□]{String[]}}}
    = FL[WL[SL[String[]]] Int[]]
    = Word.
```

The right-hand side is typed with

```
  word[String[]
        font[FL[□ Int[]]{()}
              WL[□]{()}
              SL[□]{()}]]
    = word[String[]
            font[FL[Int[]] WL[] SL[]]]
    = NewWord,
```

thereby the given rule transforms fragments typed with `Word` into ones with `NewWord`, which is the desired transformation. In Section 4 we will revisit the example in more formal way after we state the formal definition of our framework in the next section.

3. Formalities

This section presents a formal definitions of our framework: languages, the subtype relation, typing rules, substitutions, and rewrite systems. we also show some properties of our framework.

3.1 Languages

As in XDuce our languages are parameterized on a language of labels, which defines as follows.

Definition 1. *A language of labels is a triple $\langle \mathcal{L}, \mathcal{T}_\mathcal{L}, [\![\,]\!] \rangle$ with the following properties.*

- *$\mathcal{L}$ is a non-empty set of labels,*
- *$\mathcal{T}_\mathcal{L}$ is a non-empty set of label types,*
- *a denotation function $[\![\,]\!]$ is a mapping from $\mathcal{T}_\mathcal{L}$ to $\mathcal{P}(\mathcal{L}) - \{\emptyset\}$,*
- *for every label l there exists a unique label type L such that $[\![L]\!] = \{l\}$,*
- *for every $L, L' \in \mathcal{T}_\mathcal{L}$ the subsumption $[\![L]\!] \subseteq [\![L']\!]$ is decidable.*

In this paper we don't fix the language of labels. An example is the following.

- a set of labels consists of all the integers, all the UNICODE strings and XML tags,
- a set of label type consists of `Int`, `String`, all labels and unions of XML tags such as `h1` | `h2` | `h3`,
- $[\![$`Int`$]\!]$ is a set of all the integers, $[\![$`String`$]\!]$ is a set of all the UNICODE strings, $[\![L]\!] = \{L\}$ if L is a label, $[\![L_1|L_2]\!] = [\![L_1]\!] \cup [\![L_2]\!]$ if L_1 and L_2 are XML tags.

Hereafter we use l and L to stand for a label and a label type, respectively.

A set of types are built from $\mathcal{T}_\mathcal{L}$ and a set $\mathcal{T}_\mathcal{N}$ of *type names*. We assume that these two sets are disjoint. The syntax of types is defined by the following grammar.

T	$::=$	types
	T_C	complete type
	$\Box$	hole
	$L\,[T]$	tree
	$T\,T$	concatenation
T_C	$::=$	complete type
	$()$	empty
	N	type name
	$L\,[T_C]$	tree
	$T_C\,T_C$	concatenation
	$T_C \mid T_C$	union
	${T_C}^*$	repetition

The concatenation type is associative, i.e., $(T_1\,T_2)\,T_3$ is equal to $T_1\,(T_2\,T_3)$, and the empty sequence type is identity with respect to concatenation, i.e., $T\,()$ and $()\,T$ are equal to T. The union type is associative and commutative. A type defined by T_C is called *complete*, which contains no hole. We call a type T *incomplete* if it is not complete. Note that the incomplete types are classified into the following two forms: $T_1\,\Box\,T_2$ with T_1 complete, or $T_1\,L[T_2]\,T_3$ with T_1 complete and T_2 incomplete. We will often use this fact in some proofs shown in Section 3.2. Let T be a type with n holes and $T_1, \ldots, T_n$ complete types. The notation $T\{T_1, \ldots, T_n\}$ represents a complete type obtained by simultaneously replacing the holes in T from left to right with $T_1, \ldots, T_n$.

Let $\mathcal{V}$ and $\mathcal{F}$ be sets of *variables* and *function symbols*, respectively. We assume $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{L}$ are mutually disjoint. The syntax of terms is defined from $\mathcal{V}$, $\mathcal{L}$ and $\mathcal{F}$ as follows.

t	$::=$	terms
	$\Box$	hole
	$()$	empty sequence
	$x\{t, \ldots, t\}$	application
	$l\,[t]$	tree
	$t\,t$	concatenation
	$f(t, \ldots, t)$	function term

Here $l \in \mathcal{L}$, $f \in \mathcal{F}$ and $x \in \mathcal{V}$. We write $x\{\}$ and $f()$ as x and f, respectively. We assume that concatenation is associative, i.e., $((t_1\,t_2)\,t_3) = (t_1\,(t_2\,t_3))$, and the empty sequence $()$ is identity with respect to concatenation, i.e., $t\,() = ()\,t = t$. Note that $t\{t_1, \ldots, t_n\}$ is a syntactically correct term only when $t \in \mathcal{V}$. A term is called *complete* if it contains no hole; Otherwise, it is called *incomplete*. A term is called *ground* if it contains no variable. A set of variables occurring in a term t is denoted by $Var(t)$.

Hereafter, we use N for type names, x for variables, f for function symbols, s and t for terms, and T for types, with subscripts or primes if necessary. We denote $t_m, \ldots, t_n$ $(m \leq n)$ by $\overline{t}_{m,n}$. If $m > n$ then $\overline{t}_{m,n}$ denotes the empty sequence. If $m = 1$ it can be denoted by $\overline{t}_n$.

We define the *size* of a type T, denoted by $|T|$, as follows.

- $|()| = 0$,
- $|\Box| = |N| = 1$,
- $|L[T]| = |T| + 1$,
- $|T_1\,T_2| = |T_1| + |T_2|$,
- $|(T_1 \mid T_2)| = |T_1| + |T_2| + 1$,
- $|T^*| = |T| + 1$.

The size of a term t, denoted by $|t|$, is defined in the same manner as follows.

- $|()| = 0$,
- $|\square| = 1$,
- $|l[t]| = |t| + 1$,
- $|t_1\, t_2| = |t_1| + |t_2|$,
- $|x\{\bar{t}_n\}| = |f(\bar{t}_n)| = 1 + \sum_{i=1}^{n} |t_i|$.

3.2 Subtype Relation

As in XDuce, our framework provides a subtype relation <: over types. In order to define subtype relation, we define a denotation function $[\![\cdot]\!]$ mapping a type to sets of terms.

Definition 2. *The denotation function $[\![\,]\!]$ mapping a type to a set of terms is defined by the least solution of the following set of equations.*

$$\begin{aligned}
[\![\square]\!] &= \{\square\}\\
[\![()]\!] &= \{()\}\\
[\![N]\!] &= [\![\mathcal{N}(N)]\!]\\
[\![L\,[T]]\!] &= \{l\,[t] \mid l \in [\![L]\!], t \in [\![T]\!]\}\\
[\![T_1\, T_2]\!] &= \{t_1\, t_2 \mid t_1 \in [\![T_1]\!], t_2 \in [\![T_2]\!]\}\\
[\![T_1|T_2]\!] &= [\![T_1]\!] \cup [\![T_2]\!]\\
[\![T^*]\!] &= \{t_1 \ldots t_n \mid n \geq 0, \forall i.t_i \in [\![T]\!]\}
\end{aligned}$$

We call $[\![T]\!]$ the denotation of a type T. Subtype relation <: is semantically defined as $T_1 <: T_2 \Leftrightarrow [\![T_1]\!] \subseteq [\![T_2]\!]$.

In the above definition $\mathcal{N}$ is a counterpart of E in 8), which stands for a mapping from the set $\mathcal{T}_{\mathcal{N}}$ of type names to a set of *complete* types. The mapping $\mathcal{N}$ is called *type name definition.* We impose on $\mathcal{N}$ the same restriction as E in XDuce in order to avoid defining types corresponding to context free languages. Intuitively all the 'loops' from a type name N to itself should pass through the body of a tree type $L[T]$ except for the case where N appears only in tail positions. See Ref. 9) for the formal definition of the restriction.

The existence of $[\![\,]\!]$ is guaranteed from the usual argument on the fixed-point. Let $[\![T]\!]^{(0)} = \emptyset$ for each type T, and $[\![\,]\!]^{(i+1)}$ be defined from the set of the equations in Definition 2 by replacing the occurrences of $[\![\,]\!]$ in the left-hand sides by $[\![\,]\!]^{(i+1)}$'s and the ones in the right-hand sides by $[\![\,]\!]^{(i)}$'s. An order $\sqsubseteq$ on a set of functions mapping a type to a set of terms is defined such that $\phi \sqsubseteq \phi'$ if and only if $\phi(T) \subseteq \phi'(T)$ for each type T. Then it follows from the Fixed-Point Theorem that for each type $[\![T]\!] = \bigcup_i [\![T]\!]^{(i)}$.

Note that the denotation of a type is a set of ground terms without function symbols. Later (in Proposition 2) we will state the relationship between the denotation function and the set of typing rules defined in the next section.

We investigate some properties of the subtype relation. It is important to note that the number of holes in types with the subtype relation are the same. In order to show this property, we first prove several lemmas.

Lemma 1. *For any complete type T, every term in $[\![T]\!]$ is complete.*

PROOF. We only have to show that for every i and every type T the set $[\![T]\!]^{(i)}$ contains only complete terms. The result is obvious when i is zero. We show that every term in $[\![T]\!]^{i+1}$ is complete for any complete type T by assuming the result holds for i. If T is a type name N then $[\![N]\!]^{(i+1)} = [\![\mathcal{N}(N)]\!]^{(i)}$. Since $\mathcal{N}(N)$ is complete, it follows from the induction hypothesis yields that the terms in $[\![\mathcal{N}(N)]\!]^{(i)}$ are complete and hence the result holds. If T is of the form $L[T']$ then $[\![L[T']]\!]^{(i+1)} = \{l[t] \mid l \in [\![L]\!], t \in [\![T']\!]^{(i)}\}$. By the induction hypothesis $[\![T']\!]^i$ consists of complete types, and so does $[\![L[T']]\!]^{i+1}$. The other cases are proved in the same way. ∎

Hereafter, we denote the number of the holes in a term t and a type T by $\#\square(t)$ and $\#\square(T)$, respectively.

Lemma 2. *For any type T and any term t in $[\![T]\!]$, the equation $\#\square(t) = \#\square(T)$ holds.*

PROOF. We show it by induction on the size of types. As for the base cases, it is obvious that the result holds for the hole $\square$. In the remaining base cases, () and the type names, the result follows from Lemma 1.

We consider the induction step. When the type is either of the form $L[T]$ or T_1T_2, the result immediately follows from the definition of $[\![\,]\!]$ and the induction hypothesis. Suppose the type is of the form $T_1|T_2$. By definition T_1 and T_2 are complete types, i.e., they have no holes. Induction hypothesis yields that for each $t \in [\![T_1]\!] \cup [\![T_2]\!]$ the term t has no hole and hence the result holds. The case where the type is of the form T^* is proved in the same manner. ∎

Now we prove that the number of the holes in two types are invariant whenever they are in the subtype relation.

Proposition 1. *Let T_1 and T_2 be types with $T_1 <: T_2$. Then $\#\square(T_1) = \#\square(T_2)$.*

PROOF. Suppose $t \in [\![T_1]\!]$. Then Lemma 2 yields $\#\square(T_1) = \#\square(t)$. From the definition of the subtype relation t belongs to $[\![T_2]\!]$ and hence $\#\square(T_2) = \#\square(t)$. Therefore we obtain $\#\square(T_1) = \#\square(T_2)$. ∎

In XDuce the type subsumption property is decidable, i.e., there exists an algorithm to know whether given two types S and T satisfies $S <: T$ [9]. The following statements guarantee us that our subtype relation enjoys this property.

Lemma 3. *Let S and T be types such that S is of the form $S_1\, L[S_2]\, S_2$ with S_1 complete and S_2 incomplete. The relation $S <: T$ holds if and only if there exist a complete type T_1, an incomplete type T_2, a type T_3 and a label type L such that $T = T_1\, L'[T_2]\, T_3$, $[\![L]\!] \subseteq [\![L']\!]$ and $S_i <: T_i$ for $i = 1, 2, 3$.*

Proof. The if direction is obvious. We prove the only-if direction. Since $S <: T$ and S is incomplete, T is incomplete from Proposition 1. Thus T is written as $T_1\, T_4\, T_3$ such that T_1 is complete and that T_4 is either $\Box$ or $L'[T_2]$ with T_2 incomplete. Let $s_i \in [\![S_i]\!]$ for $i = 1, 2, 3$ and $l \in [\![L]\!]$. Since $S = S_1\, L[S_2]\, S_3 <: T_1\, T_4\, T_3$, we have $s_1\, l[s_2]\, s_3 \in [\![T_1\, T_4\, T_3]\!]$. Hence there exist terms $t_i \in [\![T_i]\!]$ for $i = 1, 3, 4$ such that $s_1\, l[s_2]\, s_3 = t_1\, t_4\, t_3$. Suppose $T = T_1\, \Box\, T_3$. Then $t_4 = \Box$. Because S_1 and T_1 are complete and S_2 incomplete, Lemma 2 yields $\#\Box(s_1) = \#\Box(t_1) = 0$ and $\#\Box(s_2) \neq 0$. We can infer that this contradicts $s_1\, l[s_2]\, s_3 = t_1\, \Box\, t_3$. Hence T_4 should be of the form $L'[T_2]$ with T_2 incomplete. Then t_4 is written as $l'[t_2]$ with $t_2 \in [\![T_2]\!]$ and $l' \in [\![L']\!]$. Since T_2 is incomplete, Lemma 2 yields $\#\Box(s_2) \neq 0$. Together with $\#\Box(s_1) = \#\Box(t_1) = 0$ and $\#\Box(s_2) \neq 0$, it follows that $s_i = t_i$ for $i = 1, 2, 3$ and $l = l'$. Therefore $S_i <: T_i$ for every $1 \leq i \leq 3$ and $[\![L]\!] \subseteq [\![L']\!]$. ∎

Lemma 4. *Let S and T be types such that S is of the form $S_1\, \Box\, S_2$ with S_1 complete. The relation $S <: T$ holds if and only if there exist a complete type T_1 and a type T_2 such that $T = T_1\, \Box\, T_2$ and $S_i <: T_i$ for $i = 1, 2$.*

Proof. The proof is similar to the previous lemma. ∎

The decidability of type subsumption property is an immediate consequence of the above lemmas.

Theorem 1. *The type subsumption property is decidable.*

Proof. We show that for given types S and T the property $S <: T$ is decidable. We use induction on $|S|$. If S and T are complete then we employ the algorithm described in 9) to decide whether the property holds. If S is incomplete then S is of the form $S_1\, S_4\, S_3$ such that S_1 is complete and that S_4 is either $\Box$ or $L[S_2]$ with S_2 incomplete. Let $S_4 = L[S_2]$. From Lemma 3 the property $S <: T$ holds if and only if (1) T is of the form $T_1\, L'[T_2]\, T_3$ with T_1 complete and T_2 incomplete; (2) $[\![L]\!] \subseteq [\![L']\!]$; and (3) $S_i <: T_i$ for every $1 \leq i \leq 3$. It is easy to check whether the property (1) holds. The property (2) is decidable by Definition 1. The last property is also decidable owing to the induction hypothesis. The case $S_4 = \Box$ is similarly proved by Lemma 4. ∎

Before leaving this section, we show two properties concerning the replacement of the holes in types.

Lemma 5. *Let T be an incomplete type with n holes ($n \geq 0$) and $T_1, \ldots, T_n$, $T'_1, \ldots, T'_n$ complete types with $T_i <: T'_i$ for $1 \leq i \leq n$. Then $T\{T_1, \ldots, T_n\} <: T\{T'_1, \ldots, T'_n\}$.*

Proof. We use induction on n. The case for $n = 0$ is trivial. Suppose $n > 0$. Let S be a type obtained by replacing the leftmost hole in T with T'_1. Because S includes $n - 1$ holes, the induction hypothesis yields $S\{\overline{T}_{2,n}\} <: S\{\overline{T'}_{2,n}\} = T\{\overline{T'}_n\}$. Let U be an incomplete type obtained as the result of simultaneous replacements of every hole in T except the leftmost one from left to right with $T_2, \ldots, T_n$, respectively. Then $T\{\overline{T}_n\} = U\{T_1\}$ and $S\{\overline{T}_{2,n}\} = U\{T'_1\}$. Hence it suffices to show $U\{T_1\} <: U\{T'_1\}$. We use induction on $|U|$. If U is a hole then the result immediately holds. Suppose U is a tree type $L[U']$. The (second) induction hypothesis yields $U'\{T_1\} <: U'\{T'_1\}$ and hence $L[U']\{T_1\} <: L[U']\{T'_1\}$. The case for concatenation is shown in the same way. We conclude the proof because incomplete types fall into the above three cases. ∎

Lemma 6. *Let S and S' be types with $\#\Box(S) = \#\Box(S') = n \geq 0$ and $S <: S'$. If $T_1, \ldots, T_n$ are complete types then $S\{\overline{T}_n\} <: S'\{\overline{T}_n\}$.*

Proof. We use induction on $|S|$. If S is complete then the result is obvious. Let S be incomplete. We distinguish the following two cases.

(1) If $S = S_1\, \Box\, S_2$ with S_1 complete then Lemma 4 implies that S' should be of the form $S'_1\, \Box\, S'_2$ such that S'_1 is complete and $S_i <: S'_i$ for $i = 1, 2$. The induction hypothesis yields $S_2\{\overline{T}_{2,n}\} <: S'_2\{\overline{T}_{2,n}\}$. Hence we obtain

$$\begin{aligned} T\{\overline{T}_n\} &= S_1\, T_1\, S_2\{\overline{T}_{2,n}\} \\ &<: S'_1\, T_1\, S'_2\{\overline{T}_{2,n}\} = T'\{\overline{T}_n\}. \end{aligned}$$

(2) If $S = S_1\, L[S_2]\, S_3$ with S_1 complete and S_2 incomplete then Lemma 3 implies that T' should be of the form $S'_1\, L'[S'_2]\, S'_3$ such that S'_1 is complete, S'_2 incomplete, $[\![L]\!] \subseteq [\![L']\!]$ and $S_i <: S'_i$ for every $1 \leq i \leq 3$. Let $\#\Box(S_2) = k (\leq n)$. Proposition 1 yields $\#\Box(S'_2) = k$. The

induction hypothesis yields $S_2\{\overline{T}_k\} <: S'_2\{\overline{T}_k\}$ and $S_3\{\overline{T}_{k+1,n}\} <: S'_3\{\overline{T}_{k+1,n}\}$. Hence we obtain $T\{\overline{T}_n\} = S_1\ L[S_2\{\overline{T}_k\}\ S_3\{\overline{T}_{k+1,n}\} <: S'_1\ L'[S'_2\{\overline{T}_k\}]\ S'_3\{\overline{T}_{k+1,n}\}$. ∎

3.3 Typing Rules

As in the other typed systems in Church-style, we are concerned with only the *well-typed* terms. In this section we identify well-typed terms.

In the language of labels, labels are associated to label types with $\Sigma_{\mathcal{L}}$. Likewise, we associate function symbols with types. A *function signature*, denoted by $\Sigma_{\mathcal{F}}$, is a mapping from $\mathcal{F}$ to a set of finite and non-empty sequences of *complete* types. We write a sequence of $n+1$ complete types $T_1, \ldots, T_{n+1}$ as $T_1 \times \cdots \times T_n \to T_{n+1}$. A type is assigned to each variable. The set of variables with type T is denoted by $\mathcal{V}^T$. We assume that for each type T the set $\mathcal{V}^T$ is countably infinite.

Definition 3. *A judgment $\vdash t : T$ is a relation between a term t and a type T. A term t is well-typed if there exits a type T such that a judgment $\vdash t : T$ is deducible using the typing rules shown in* **Fig. 1**. *In this case we say the term t has type T.*

Note that in the rule (apply) in Fig. 1 T has n holes and $T', T_1, \ldots, T_n$ are complete from the definition of $T\{T_1, \ldots, T_n\}$.

The following proposition states that the set of the ground terms without function symbols that have type T coincides with the denotation of T.

Proposition 2. *Let t be a ground term without function symbols. $\vdash t : T$ if and only if $t \in \llbracket T \rrbracket$.*

Proof.
($\Rightarrow$)We use induction on a typing derivation of $\vdash t : T$. Since t is a ground term without function symbols, the typing rules used in the last step of the deduction are other than (variable), (apply) and (f-apply). Since most cases are trivial, we only show the case (tree). In this case $\vdash l[t] : L[T]$ is deduced from $l \in \llbracket L \rrbracket$ and $\vdash t : T$. The induction hypothesis yields $t \in \llbracket T \rrbracket$. Hence, by the definition of $\llbracket\ \rrbracket$, the desired result $l[t] \in \llbracket L[T] \rrbracket$ is obtained.
($\Leftarrow$)We use induction on $|T|$. The cases $T = ()$ and $T = \Box$ are trivial. Consider the case $L[T]$. The elements of $\llbracket L[T] \rrbracket$ are of the form $l[t]$ with $l \in \llbracket L \rrbracket$ and $t \in \llbracket T \rrbracket$. The induction hypothesis yields $\vdash t : T$. Then from (tree) we obtain $\vdash l[t] : L[T]$. Consider the case $T_1\, T_2$ with $T_i \neq ()$

(empty)
$$\frac{}{\vdash () : ()}$$

(variable)
$$\frac{x \in \mathcal{V}^T}{\vdash x : T}$$

(box)
$$\frac{}{\vdash \Box : \Box}$$

(tree)
$$\frac{l \in \llbracket L \rrbracket \quad \vdash t : T}{\vdash l\,[t] : L\,[T]}$$

(concatenation)
$$\frac{\vdash t_1 : T_1 \quad \vdash t_2 : T_2}{\vdash t_1\ t_2 : T_1\ T_2}$$

(apply)
$$\frac{x \in \mathcal{V}^T \quad \vdash t_1 : T_1\ \cdots\ \vdash t_n : T_n}{\vdash x\{t_1, \ldots, t_n\} : T'}$$
if $n > 0$ and $T' = T\{T_1, \ldots, T_n\}$

(f-apply)
$$\frac{\begin{array}{c}\Sigma_{\mathcal{F}}(f) = T_1 \times \cdots T_n \to T \\ \vdash t_1 : T_1\ \cdots\ \vdash t_n : T_n\end{array}}{\vdash f(t_1, \ldots, t_n) : T}$$

(subtype)
$$\frac{\vdash t : T_1 \quad T_1 <: T_2}{\vdash t : T_2}$$

Fig. 1 Typing rules.

for $i = 1, 2$. By definition the elements of $\llbracket T_1 T_2 \rrbracket$ are of the form $t_1\ t_2$ with $t_i \in \llbracket T_i \rrbracket$ for $i = 1, 2$. The induction hypothesis yields $\vdash t_1 : T_1$ and $\vdash t_2 : T_2$, which produces $\vdash t_1\ t_2 : T_1\ T_2$ from the (concatenation) rule. Consider the case $T_1 \mid T_2$. By definition $t \in \llbracket T_1 \mid T_2 \rrbracket$ means either $t \in \llbracket T_1 \rrbracket$ or $t \in \llbracket T_2 \rrbracket$. Without loss of generality, we assume $t \in \llbracket T_1 \rrbracket$. The induction hypothesis yields $\vdash t : T_1$. By definition $T_1 <: T_1 \mid T_2$. Hence the (subtype) rule yields $\vdash t : T_1 \mid T_2$. Finally consider the case T^*. Every element of $\llbracket T^* \rrbracket$ is written as $t_1\ \cdots\ t_n$ with $n \geq 0$ and $t_i \in \llbracket T \rrbracket$ for every $1 \leq i \leq n$. The induction hypothesis yields $\vdash t_i : T$ for every $1 \leq i \leq n$. The applications of the (concatenation) rule n times yield $\vdash t_1\ \cdots\ t_n : T'$ where T' is the concatenation of n T's. By definition $T' <: T^*$. Therefore $\vdash t_1\ \cdots\ t_n : T^*$ is deduced from the (subtype) rule. ∎

Since our typing system employs the subtype relation, a term has generally many types. However, it would be convenient if we can assign terms to the unique types. Thus we introduce a function τ that maps a term to its unique type.

Definition 4. *The partial function τ maps a term to a type as follows.*

$$\begin{array}{rcll} \tau(x) & = & T & \text{if } x \in \mathcal{V}^T \\ \tau(l[t]) & = & L[\tau(t)] & \text{if } \Sigma_{\mathcal{L}}(l) = L \\ \tau(()) & = & () & \\ \tau(\Box) & = & \Box & \\ \tau(t_1\, t_2) & = & \tau(t_1)\, \tau(t_2) & \\ \tau(x\{\overline{t}_n\}) & = & T' & \\ & & \multicolumn{2}{l}{\text{if } n > 0,\ x \in \mathcal{V}^T} \\ & & \multicolumn{2}{l}{\text{and } T' = T\{\tau(t_1), \ldots, \tau(t_n)\}} \\ \tau(f(\overline{t}_n)) & = & T & \\ & & \multicolumn{2}{l}{\text{if } \Sigma_{\mathcal{F}}(f) = T_1 \times \cdots \times T_n \rightarrow T} \\ & & \multicolumn{2}{l}{\text{and } \tau(t_i) <: T_i \text{ for } 1 \leq i \leq n} \end{array}$$

Here $\Sigma_{\mathcal{L}}$ is a mapping from $\mathcal{L}$ to $\mathcal{T}_{\mathcal{L}}$ such that $\Sigma_{\mathcal{L}}(l) = L$ if $[\![L]\!] = \{l\}$. We call $\Sigma_{\mathcal{L}}$ a *label signature.*

We show the function τ satisfies desired properties: given a well-typed term t, $\tau(t)$ constructs the least type among types of t with respect to $<:$. Furthermore, the well-typedness of a term t identifies with the definedness of $\tau(t)$.

Lemma 7. *For each term t, if $\tau(t)$ is defined then $\vdash t : \tau(t)$.*

Proof. By a trivial induction proof on the size of terms with the definitions of $\tau(\,)$ and the typing rules. We only show the case where the term is of the form $x\{\overline{t}_n\}$ with $n > 0$. Since $\tau(x\{\overline{t}_n\})$ is defined, $\tau(x\{\overline{t}_n\}) = T\{\tau(t_1), \ldots, \tau(t_n)\}$ with $x \in \mathcal{V}^T$. From induction hypothesis we obtain $\vdash t_i : \tau(t_i)$ for $1 \leq i \leq n$. Therefore $\vdash x\{\overline{t}_n\} : \tau(x\{\overline{t}_n\})$ from the rule (apply). ∎

The following theorem states that τ satisfies the desired properties mentioned above.

Theorem 2. *For each term t, t is well-typed if and only if $\tau(t)$ is defined. Moreover, for any type T if $\vdash t : T$ then $\tau(t) <: T$ holds.*

Proof. The if direction of the first half is immediate from Lemma 7. The remaining part of the result is a immediate consequence of the following claim: if $\vdash t : T$ for a type T then $\tau(t)$ is defined and $\tau(t) <: T$. The proof of the claim is done by induction on a typing deduction of $\vdash t : T$. We proceed by cases on the typing rule used in the last step of the given deduction. The first three cases in Fig. 1 are obvious. If (tree) is used then the deduced judgment is of the form $\vdash l[t] : L[T]$ where $l \in [\![L]\!]$ and $\vdash t : T$. By the induction hypothesis $\tau(t)$ is defined and $\tau(t) <: T$. By definition $\tau(l[t]) = L'[\tau(t)]$ where $[\![L']\!] = \{l\}$. Hence $\tau(l[t]) = L'[\tau(t)] <: L[T]$ from the definition of $<:$. The case for (concatenation) is shown in the same way. Suppose the rule (apply) is finally used in the deduction. Then the judgment is of the form $\vdash x\{\overline{t}_n\} : T\{\overline{T}_n\}$ $(n > 0)$ which is deduced from $\vdash t_i : T_i$ for $1 \leq i \leq n$ with $x \in \mathcal{V}^T$. From the induction hypothesis $\tau(t_i)$ is defined and $\tau(t_i) <: T_i$ for $1 \leq i \leq n$. For each $i \in \{1, \ldots, n\}$, T_i is complete and hence so is $\tau(t_i)$ from Proposition 1. Then $\tau(x\{\overline{t}_n\}) = T\{\tau(t_1), \ldots, \tau(t_n)\}$ and from Lemma 5 we obtain $\tau(x\{\overline{t}_n\}) <: T\{\overline{T}_n\}$. When the final rule is (f-apply) then the judgment is of the form $\vdash f(\overline{t}_n) : T$ deduced from $\vdash t_i : T_i$ for $1 \leq i \leq n$ and $\Sigma_{\mathcal{F}}(f) = T_1 \times \cdots \times T_n \rightarrow T$. By the induction hypothesis $\tau(t_i)$ are defined and $\tau(t_i) <: T_i$ for $1 \leq i \leq n$. Thus $\tau(f(\overline{t}_n)) = T <: T$. Finally we consider the case for (subtype). In this case $\vdash t : T_1$ and $T_1 <: T_2$ yields $\vdash t : T_2$. By the induction hypothesis $\tau(t)$ is defined and $\tau(t) <: T_1$, from which $\tau(t) <: T_2$ immediately follows. ∎

The following statement is an immediate consequence of the above theorem.

Corollary 1. *Well-typedness of a term is decidable.* ∎

3.4 Substitution

A *substitution* σ is a mapping from $\mathcal{V}$ to a set of terms[☆] satisfying the following conditions: (1) there are finitely many variables x such that $\sigma(x) \neq x$; (2) $\tau(\sigma(x)) <: \tau(x)$ for every $x \in \mathcal{V}$. Given a substitution σ, we denote the set of variables $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ by *dom*(σ). We sometimes write a substitution σ as the set $\{x \mapsto \sigma(x) \mid x \in \mathit{dom}(x)\}$.

Theorem 2 guarantees us that the above definition yields the expected substitutions. Suppose we have label types $\mathtt{L}_1, \mathtt{L}_2$, labels $\mathtt{l}, \mathtt{l}'$ and a variable $x \in \mathcal{V}^{\mathtt{L}_1[]}$ such that $[\![\mathtt{L}_1]\!] = \{\mathtt{l}\}$ and $[\![\mathtt{L}_2]\!] = \{\mathtt{l}, \mathtt{l}'\}$. Since $\tau(x) = \mathtt{L}_1\mathtt{[]}$ and $[\![\mathtt{L}_1]\!] = \{\mathtt{l}\}$, we expect that $\mathtt{l[]}$ can be substituted for x, i.e., there exists the substitution $\sigma = \{x \mapsto \mathtt{l[]}\}$. However, if $\tau(\mathtt{l[]})$ were equal to $\mathtt{L}_2\mathtt{[]}$, the term $\mathtt{l[]}$ could not be substituted for x because $\mathtt{L}_2\mathtt{[]} \not<: \mathtt{L}_1\mathtt{[]} = \tau(x)$ violates the second condition $\tau(\sigma(x)) <: \tau(x)$. But, as a matter of fact, $\tau(\mathtt{l[]}) = \mathtt{L}_1\mathtt{[]}$ by definition. Hence we can substitute $\mathtt{l[]}$ for x as expected.

Substitutions are extended to mappings over terms.

Definition 5. *For a well-typed term t and a substitution σ, a term $t\sigma$ is recursively defined*

[☆] Note that $\sigma(x)$ may contain variables. Though it is not necessary in usual XML transformation, it may be useful in investigating properties of transformation in general way.

as follows.

$$\begin{aligned}
()\sigma &= () \\
\Box\sigma &= \Box \\
x\{\overline{t}_n\}\sigma &= \sigma(x)\{\overline{t}_n\sigma\} \\
(l\,[t])\sigma &= l\,[t\sigma] \\
(t_1\,t_2)\sigma &= t_1\sigma\,t_2\sigma \\
f(\overline{t}_n)\sigma &= f(\overline{t}_n\sigma)
\end{aligned}$$

where $\overline{t}_n\sigma$ *stands for* $t_1\sigma, \ldots, t_n\sigma$.

The notation $\sigma(x)\{\overline{t}_n\sigma\}$ in the above definition represents a term defined as follows. Let t, $t_1, \ldots, t_n$ be terms with $\#\Box(\tau(t)) = n \geq 0$ and $t_1, \ldots, t_n$ are complete. Then $t\{\overline{t}_n\}$ is defined as follows.

$$\begin{aligned}
t\{\,\} &= t \\
(\Box\,s)\{\overline{t}_n\} &= t_1\,s\{\overline{t}_{2,n}\} \\
(x\{\overline{s}_m\}\,s)\{\overline{t}_n\} &= x\{\overline{s}_m\}\,s\{\overline{t}_n\} \text{ if } m > 0 \\
(x\,s)\{\overline{t}_n\} &= x\{\overline{t}_k\}\,s\{\overline{t}_{k+1,n}\} \\
&\qquad \text{where } \#\Box(\tau(x)) = k \\
(l\,[s_1]\,s_2)\{\overline{t}_n\} &= l\,[s_1\{\overline{t}_k\}]\,s_2\{\overline{t}_{k+1,n}\} \\
&\qquad \text{where } \#\Box(\tau(s_1)) = k \\
(f(\overline{s}_m)\,s)\{\overline{t}_n\} &= f(\overline{s}_m)\,s\{\overline{t}_n\},
\end{aligned}$$

The well-definedness of the above definition is an easy consequence of Lemma 2 and the definition of τ.

Composition of substitutions σ and ρ, denoted by $\sigma\rho$, is the composition of mappings σ and ρ, which is defined by $t(\sigma\rho) = (t\sigma)\rho$. Renaming is a substitution σ such that σ is an isomorphic mapping over $\mathcal{V}$.

As mentioned in the previous section, we only deal with well-typed terms. Thus, it would be good if the application of substitutions to well-typed terms always yield well-typed terms. We show this property holds with the help of the following lemma which states $\tau(t\{t_1, \ldots, t_n\})$ is defined for well-typed terms t, $t_1, \ldots, t_n$.

Lemma 8. *Let t, $t_1, \ldots, t_n$ ($n \geq 0$) be well-typed terms such that $\#\Box(\tau(t)) = n$ and each $\tau(t_i)$ is complete for $1 \leq i \leq n$. Then $\tau(t\{\overline{t}_n\}) = \tau(t)\{\tau(t_1), \ldots, \tau(t_n)\}$.*

Proof. Induction on $|t|$. The case for the empty sequence is obvious. We distinguish the following five cases.

(1) If $t = \Box\,s$ then the result follows from the following equations.

$$\begin{aligned}
&\tau(\Box\,s)\{\tau(t_1), \ldots, \tau(t_n)\} \\
&= (\tau(\Box)\,\tau(s))\{\tau(t_1), \ldots, \tau(t_n)\} \\
&= (\Box\,\tau(s))\{\tau(t_1), \ldots, \tau(t_n)\} \\
&= \tau(t_1)\,\tau(s)\{\tau(t_2), \ldots, \tau(t_n)\} \\
&= \tau(t_1)\,\tau(s\{\overline{t}_{2,n}\}) \\
&= \tau(t_1\,s\{\overline{t}_{2,n}\}) \\
&= \tau(\Box\,s\{\overline{t}_n\})
\end{aligned}$$

The fourth equation is obtained from the induction hypothesis.

(2) Consider the case $t = x\{\overline{s}_m\}\,s$. Because $\tau(x\{\overline{s}_m\})$ is complete by definition, $\tau(t)\{\tau(t_1), \ldots, \tau(t_n)\}$ equals to the sequence type $\tau(x\{\overline{s}_m\})\,\tau(s)\{\tau(t_1), \ldots, \tau(t_n)\}$. By the induction hypothesis its second half is equal to $\tau(s\{\overline{t}_n\})$. It follows the desired result.

(3) If $t = x\,s$ then $\tau(t)\{\tau(t_1), \ldots, \tau(t_n)\}$ equals to the concatenation of $\tau(x)\{\tau(t_1), \ldots, \tau(t_k)\}$ and $\tau(s)\{\tau(t_{k+1}), \ldots, \tau(t_n)\}$ where $k = \#\Box(\tau(x))$. By definition the former equals to $\tau(x\{\overline{t}_k\})$. From the induction hypothesis we obtain $\tau(s)\{\tau(t_{k+1}), \ldots, \tau(t_n)\} = \tau(s\{\overline{t}_{k+1,n}\})$. Combining them yields the desired result.

(4) Consider the case $t = l[s_1]\,s_2$. By definition $\tau(t)\{\tau(t_1), \ldots, \tau(t_n)\}$ is represented as the concatenation of $L[\tau(s_1)\{\tau(t_1), \ldots, \tau(t_k)\}]$ and $\tau(s_2)\{\tau(t_{k+1}), \ldots, \tau(t_n)\}$ where $\Sigma_{\mathcal{L}}(l) = L$ and $k = \#\Box(\tau(s_1))$. By the induction hypothesis they are equal to $L[\tau(s_1\{\overline{t}_k\}]$ and $\tau(s_2\{\overline{t_{k+1,n}}\})$, respectively. The result follows from their concatenation.

(5) If $t = f(\overline{s}_m)\,s$ then the given type equals to $\tau(f(\overline{s}_m))\,\tau(s)\{\tau(t_1), \ldots, \tau(t_n)\}$. From the induction hypothesis it is equal to $\tau(f(\overline{s}_m))\,\tau(s\{\overline{t}_n\})$, thereby we obtain the desired result. ∎

The following proposition states that the set of well-typed terms are closed under the application of substitutions. Moreover, it states that the type is preserved by the application of substitutions, i.e., for any type T the term $t\sigma$ has type T if $\vdash t : T$.

Proposition 3. *For any substitution σ and any term t, $\tau(t\sigma)$ is defined and the relation $\tau(t\sigma) <: \tau(t)$ holds.*

Proof. Induction on $|t|$. The cases for empty sequences or holes are trivial. If t is a variable x, by definition $\tau(x\sigma)$ is defined and $\tau(x\sigma) <: \tau(x)$. If t is a concatenation of non-empty sequences t_1 and t_2 then $\tau(t\sigma) = \tau(t_1\sigma)\,\tau(t_2\sigma)$. By the induction hypothesis $\tau(t_i\sigma)$ is defined and $\tau(t_i\sigma) <: \tau(t_i)$ for $i = 1, 2$, from which the result follows. The case for trees $l[t_1]$ is proved in the same way. Let $t = x\{\overline{t}_n\}$ with $n > 0$. By definition $\tau(t_i)$ is complete for $1 \leq i \leq n$. By the induction hypothesis $\tau(t_i\sigma)$ is defined and $\tau(t_i\sigma) <: \tau(t_i)$ for $1 \leq i \leq n$. Thus, from Proposition 1 $\tau(t_i\sigma)$ is complete for $1 \leq i \leq n$ and $\#\Box(\tau(\sigma(x))) = \#\Box(\tau(x)) = n$. Since $\tau(t\sigma) = \tau(\sigma(x)\{\overline{t}_n\sigma\})$, from Lemma 8 $\tau(t\sigma)$ is defined and $\tau(t\sigma) = \tau(\sigma(x))\{\tau(t_1\sigma), \ldots, \tau(t_n\sigma)\}$. Lemmas 5, 6 and the induction hypothesis yield

$$\begin{aligned}
&\tau(\sigma(x))\{\tau(t_1\sigma),\ldots,\tau(t_n\sigma)\}\\
<:\ &\tau(\sigma(x))\{\tau(t_1),\ldots,\tau(t_n)\}\\
<:\ &\tau(x)\{\tau(t_1),\ldots,\tau(t_n)\}\\
=\ &\tau(x\{t_1,\ldots,t_n\}).
\end{aligned}$$

Finally we consider the case for function terms $f(\bar{t}_n)$. By definition $\tau(t\sigma) = \tau(f(\bar{t}_n\sigma))$. From the induction hypothesis $\tau(t_i\sigma)$ is defined and $\tau(t_i\sigma) <: \tau(t_i)$ for $1 \le i \le n$. Since $f(\bar{t}_n)$ is well-typed $\tau(t_i) <: T_i$ for $1 \le i \le n$, where $\Sigma_{\mathcal{F}}(f) = T_1 \times \cdots \times T_n \to T$. Hence we obtain $\tau(t_i\sigma) <: \tau(T_i)$ for $1 \le i \le n$, which assures us that $\tau(t\sigma)$ is defined and that $\tau(t\sigma) = \tau(t) = T$. This yields the desired result. ∎

3.5 Rewrite System

Now we are ready to define rewrite systems for our language. A rewrite rule is a pair of terms l and r, denoted by $l \to r$, satisfying the following conditions: (1) $Var(r) \subseteq Var(l)$; (2) l is of the form $f(t_1,\ldots,t_n)$ with function symbol f and terms $t_1,\ldots,t_n$ containing no function symbols; and (3) $\tau(r) <: \tau(l)$.

A rewrite system is a quadruple $\mathcal{R} = \langle\mathcal{L}, \Sigma_{\mathcal{F}}, \mathcal{N}, R\rangle$, where $\mathcal{L}$ is a language of labels, $\Sigma_{\mathcal{F}}$ is a function signature, $\mathcal{N}$ a type name definition and R a set of rewrite rules.

A *new variant* of a rewrite rule $l \to r$ is $l\sigma \to r\sigma$ where σ is a renaming such that $\sigma(x)$ is a fresh variable for every $x \in dom(\sigma)$.

The rewrite relation $\to_{\mathcal{R}}$ over terms induced from a rewrite system $\mathcal{R}$ is defined as follows: $s \to_{\mathcal{R}} t$ if there exist a new variant $l \to r$ of a rewrite rule in $\mathcal{R}$, a substitution σ, and a well-typed term s' with $\#\square(s') = \#\square(\tau(s')) = 1$ such that $s = s'\{l\sigma\}$ and $t = s'\{r\sigma\}$. It follows from the above definition that if $s \to_{\mathcal{R}} t$ then $\tau(s)$ and $\tau(t)$ are complete.

The following theorem states type preservation holds for our rewrite systems, i.e., if a term s has type T and $s \to_{\mathcal{R}} t$ then t has type T.

Theorem 3. *If $s \to_{\mathcal{R}} t$ then $\tau(t) <: \tau(s)$.*

Proof. By definition there exists a term s' with a single hole and a substitution σ that satisfy $s = s'\{l\sigma\}$ and $t = s'\{r\sigma\}$. We first show $\tau(r\sigma) <: \tau(l\sigma)$. By definition $\tau(r) <: \tau(l)$. Proposition 3 yields $\tau(r\sigma) <: \tau(r)$. From the final part of the proof of Proposition 3, we know that $\tau(l\sigma) = \tau(l)$ because l is of the form $f(\bar{t}_n)$. Combining them, we obtain $\tau(r\sigma) <: \tau(l\sigma)$. The desired result immediately follows from Lemma 5. ∎

4. Example

We conclude the example illustrated in Section 2. The rewrite rule corresponding to the function `flatten2` is the following:

$$\begin{aligned}
&\texttt{flatten2}(fl\{wl\{sl\{s\}\}\}) \to\\
&\quad \texttt{word[}s\ \texttt{font[}fl\{\texttt{()}\}\ wl\{\texttt{()}\}\ sl\{\texttt{()}\}\texttt{]]}
\end{aligned}$$

where

$$\begin{aligned}
&\Sigma_{\mathcal{F}}(\texttt{flatten2}) = \texttt{Word}\to\texttt{NewWord}\\
&fl \in \mathcal{V}^{\texttt{FL[}\square\ \texttt{Int[]]}}\\
&wl \in \mathcal{V}^{\texttt{WL[}\square\texttt{]}}\\
&sl \in \mathcal{V}^{\texttt{SL[}\square\texttt{]}}\\
&s \in \mathcal{V}^{\texttt{String[]}},
\end{aligned}$$

`FL`, `WL`, `SL` are label types with denotation

$$\begin{aligned}
&[\![\texttt{FL}]\!] = \{\texttt{helvetica}, \texttt{times}, \texttt{courier}\}\\
&[\![\texttt{WL}]\!] = \{\texttt{bold}, \texttt{normal}, \texttt{thin}\}\\
&[\![\texttt{SL}]\!] = \{\texttt{roman}, \texttt{italic}\},
\end{aligned}$$

and `Word`, `NewWord` are type names such that

$$\begin{aligned}
&\mathcal{N}(\texttt{Word}) = \texttt{FL[WL[SL[Sting[]]] Int[]]}\\
&\mathcal{N}(\texttt{NewWord}) =\\
&\qquad \texttt{word[String[]}\\
&\qquad\qquad \texttt{font[FL[Int[]] WL[] SL[]]]}
\end{aligned}$$

We obtain

$$\begin{aligned}
&\tau(fl\{wl\{sl\{s\}\}\})\\
=\ &\texttt{FL[}\tau(wl\{sl\{s\}\})\ \texttt{Int[]]}\\
=\ &\texttt{FL[WL[}\tau(sl\{s\})\texttt{] Int[]]}\\
=\ &\texttt{FL[WL[SL[}\tau(s)\texttt{]] Int[]]}\\
=\ &\texttt{FL[WL[SL[String[]]] Int[]]}\\
<:\ &\texttt{Word}.
\end{aligned}$$

Thus, by definitions of τ and $\Sigma_{\mathcal{F}}$, we have $\tau(\texttt{flatten2}(fl\{wl\{sl\{s\}\}\})) = \texttt{NewWord}$. Similarly, $\tau(\texttt{word[}s\ \texttt{font[}fl\{\texttt{()}\}\ wl\{\texttt{()}\}\ sl\{\texttt{()}\}\texttt{]]})$ is obtained as follows.

$$\begin{aligned}
&\tau(\texttt{word[}s\ \texttt{font[}fl\{\texttt{()}\}\ wl\{\texttt{()}\}\ sl\{\texttt{()}\}\texttt{]]})\\
=\ &\texttt{word[String[]}\\
&\quad \texttt{font[}\tau(fl\{\texttt{()}\})\ \tau(wl\{\texttt{()}\})\ \tau(sl\{\texttt{()}\})\texttt{]}\\
=\ &\texttt{word[String[]}\\
&\qquad \texttt{font[FL[Int[]] WL[] SL[]]]}\\
<:\ &\texttt{NewWord},
\end{aligned}$$

Therefore the type of the right-hand side is a subtype of the left-hand side. The substituion

$$\begin{aligned}
\sigma = \{\ &fl \mapsto \texttt{FL[}\square\ \texttt{Int]},\ wl \mapsto \texttt{WL[}\square\texttt{]},\\
&sl \mapsto \texttt{SL[}\square\texttt{]},\qquad s \mapsto \texttt{"Hello"}\ \}
\end{aligned}$$

is obtained by matching the term `times[bold[ italic["Hello"[]]] 12]` with the left-hand side of the rule. The rewrite step starting from the above term is described as follows.

$$
\begin{aligned}
&\texttt{flatten2(}\\
&\quad\texttt{times[bold[italic["Hello"[]]] 12])}\\
&= \texttt{flatten2(}fl\{wl\{sl\{s\}\}\}\texttt{)}\sigma\\
&\to \texttt{word[}s\ \texttt{font[}fl\{\texttt{()}\}\ wl\{\texttt{()}\}\ sl\{\texttt{()}\}\texttt{]]}\sigma\\
&= \texttt{word["Hello"}\\
&\qquad\texttt{font[times[12] bold[] italic[]]]}
\end{aligned}
$$

The obtained term is the desired one.

5. Concluding Remarks

Our framework proposed in this paper provides a restricted form of second-order matching, which enables us more direct and simpler representation of the transformation rules as demonstrated in Section 2, while static type checking is still available as in XDuce. Introduction of holes in both terms and types is crucial in our work. Especially, the novel feature of our work is incomplete types, i.e., the introduction of holes in regular expression types. There are some works introducing holes in terms to provide flexible framework for XML. JWIG, a JAVA extension for high-level web service construction, supports contexts with holes[4]. The purpose of holes is different from ours: they introduce holes to provides templates, given in JAVA programs beforehand, to create XML documents. On the other hand, in our work the holes are only generated in the process of second-order matching. Very recently Kutsia and Marin proposed an approach very similar to ours[11]. They provide *context sequence matching* for querying XML data. Although their matching is more flexible than ours, it is untyped and thus static type checking is not possible.

The XML processing language CDuce[2] supports a more restricted form of second-order matching than ours. In CDuce XML tags can be substituted for variables. Thus, the function `flatten2` is written with the *same* number of equations as ours.

$$
\begin{aligned}
&\texttt{fun flatten2(}\\
&\qquad\langle\texttt{(}fl\texttt{)\&FL}\rangle\langle\texttt{(}wl\texttt{)\&WL}\rangle\langle\texttt{(}sl\texttt{)\&SL}\rangle\\
&\qquad\quad\texttt{[}s\texttt{::String }pt\texttt{::Int]:Word)}\\
&\quad\texttt{: NewWord}\\
&\quad= \texttt{word[}s\ \texttt{font[}\langle fl\rangle\texttt{[}pt\texttt{]}\ \langle wl\rangle\texttt{[]}\ \langle sl\rangle\texttt{[]]]}
\end{aligned}
$$

The notation $\texttt{p}_1\texttt{\&p}_2$ denotes that both $\texttt{p}_1$ and $\texttt{p}_2$ matches the same object. Hence $\langle\texttt{(}fl\texttt{)\&FL}\rangle$ means the variable fl has type `FL`, which corresponds to $\texttt{FL[}\Box\texttt{]}$ in our notation where the single hole occurs at just below the top-level label type `FL`. In contrast to that, our approach allows many occurrences of holes at any position. Therefore, when a rewrite rule in our framework is transformed to an equation in CDuce, the latter needs more variables than the former.

Though we allow the occurrence of hols in any position as mentioned above, the position of holes in a type is fixed due to the restriction induced from our definition of types. This suppresses the nondeterministic computation of the solutions, caused by the ordinary second-order matching, and hence implementation of the matching algorithm makes easier (note that the nondeterminism caused from the sequences in XML terms still remains). On the other hand, we should prepare extra rewrite rules to traverse given XML documents to find the subjects of transformation (though the transformation of subjects themselves is simply presented compared to the transformation with first-order matching, as shown in Section 2). As pointed out in Refs. 14) and 11), such extra rules are unnecessary in *untyped* context matching. A way to overcome this problem is to allow the presence of holes in union types and the definitions of type names.

Consider the following label type and type name definitions.

$$
\begin{aligned}
&[\![\texttt{Qf}]\!] = \{\ \texttt{helvetica}, \texttt{times}, \texttt{courier},\\
&\qquad\qquad \texttt{thin}, \texttt{bold}, \texttt{italic}\ \}\\
&\mathcal{N}(\texttt{HTML}) = \texttt{head[String[]] body[QSent]}\\
&\mathcal{N}(\texttt{QSent}) =\\
&\qquad \texttt{(String[]}\mid\texttt{Qf[QSent]) QSent}
\end{aligned}
$$

Suppose that when there are multiple occurrences of `bold`'s on a path in a XML term with type `HTML`, we want to leave only outermost `bold` on the path and remove other occurrences of them. In order to implement that, we introduce the following type name definitions.

$$
\begin{aligned}
&\mathcal{N}(\texttt{QSentBox}) = \texttt{QSent? }\Box\texttt{ QSent?}\\
&\qquad\mid \texttt{QSent? Qf[QSentBox] QSent?}\\
&\mathcal{N}(\texttt{HasBold}) = \texttt{bold[QSentBox]}\\
&\qquad\mid \texttt{QSent? Qf[HasBold] QSent?}\\
&\mathcal{N}(\texttt{NoBold}) =\\
&\qquad \texttt{(String[]}\mid\texttt{(Qf}\backslash\texttt{bold)[NoBold])}\\
&\qquad \texttt{NoBold}
\end{aligned}
$$

where `T?` is an abbreviation of $\texttt{()}\mid\texttt{T}$ and $\texttt{Qf}\backslash\texttt{bold}$ denotes a label type whose denotation contains any labels in $[\![\texttt{Qf}]\!]$ except `bold`. Obviously, every term in the sets $[\![\texttt{QSentBox}]\!]$ and $[\![\texttt{HasBold}]\!]$ has a single hole. Let $c \in \mathcal{V}^{\texttt{head[String[]] body[HasBold]}}$ and $s \in \mathcal{V}^{\texttt{NoBold}}$. Then the application of the function `uniq` defined by the following rewrite rules

$\texttt{uniq}(c\{\texttt{bold[}s\texttt{]}\}) \rightarrow c\{s\}$

$\texttt{uniq}(x) \rightarrow x$

to a term with HTML type *at its root* yields the desired transformation.

As we checked in this example, it is important that the relaxation in the occurrence of holes in type does not destroy the preservation of the number of holes with respect to the subtype relation. Finding sufficient conditions to guarantee this is our future work. Furthermore, we need to modify our framework to allow types like HasBold above since some of the notions and the proofs presented in this paper rely on the restriction of the occurrence of the holes in types.

Note that the above rewrite rules cannot be presented by CDuce function definitions because the variable c matches the arbitrary large incomplete term which includes variable number of tags.

In this paper we mainly concentrated on the type theoretic issues on incomplete types. As a further research, we should investigate the properties for our rewrite systems. We have built a pattern matching algorithm[15]. At present we only have an informal proof of completeness of the algorithm; it is necessary to give its formal proof. Moreover, the algorithm works with a restriction either on the types of the function terms to the tree types or on the reduction strategies to innermost one. We need to extend the algorithm to a general one.

Acknowledgments We are very grateful to the anonymous referee who provided us helpful comments to improve this paper. This work is supported by Japan Society for the Promotion of Science, basic research (C) No.15500014 (Suzuki), basic research (C) No.16500014 (Okui).

References

1) Baader, F. and Nipkow, T.: *Term rewriting and all that*, Cambridge University Press (1999).
2) Benzaken, V., Castagna, G. and Frisch, A.: CDuce: An XML-Centric General-Purpose Language, *Proc. ACM Int. Conf. on Functional Programming* (2003).
3) Bezem, M., et al.: *Term rewriting systems*, Cambridge University Press (2003).
4) Christensen, A. S., Moller, A. and Schwartzbach, M.I.: Extending Java for High-level web service construction, *ACM Trans. on Programming Languages and Systems*, Vol.25, No.6, pp.814–875 (2003).
5) Common, H.: Completion of rewrite systems with membership constraints Part I: Deduction rules, *J. Symbolic Computation*, Vol.25, No.4, pp.397–419 (1998).
6) Common, H.: Completion of rewrite systems with membership constraints Part II: Constraint Solving, *J. Symbolic Computation*, Vol.25, No.4, pp.421–453 (1998).
7) de Moor, O. and Sittampalam, G.: Higher-order matching for program transformation, *Theoretical Computer Science*, Vol.269, pp.135–162 (2001).
8) Hosoya, H. and Pierce, B.C.: XDuce: A typed XML processing language (preliminary report), *Int. Workshop on the Web and Databases* (*WebDB*), (May 2000). Reprinted in *The Web and Databases, Selected Papers*, LNCS Vol.1997 (2001).
9) Hosoya, H., Vouillon, J. and Pierce, B.C.: Regular Expression Types for XML, *ACM Trans. on Programming Languages and Systems* (2004).
10) Huet, G. and Lang, B.: Proving and applying program transformations expressed with second-order patterns, *Acta Informatica*, Vol.11, pp.31–55 (1978).
11) Kutsia, T. and Marin, M.: Can Context Sequence Matching be used for XML Querying? *Proc. 19th Int. Workshop on Unification* (*UNIF '05*), pp.77–92 (Apr. 2005).
12) Mayr, R. and Nipkow, T.: Higher-order rewrite systems and their confluence, *Theoretical Computer Science*, Vol.192, No.1, pp.3–29 (1998).
13) Ohlebusch, E.: *Advanced topics in term rewriting*, Springer Verlag (2002).
14) Schmidt-Schaus, M. and Stuber, J.: On the complexity of linear and stratified context matching problems, *Theory of Computing Systems*, Vol.37, pp.717–740 (2004).
15) Suzuki, T. and Okui, S.: Transformation of XML documents with incomplete regular expression type (Draft), *Third Workshop on Programmable Structured Documents* (*PSD*), pp.120–127 (Jan. 2005)
16) van Oostrom, V.: *Confluence for abstract and higher-order rewriting*, PhD Thesis, Vrije Universiteit, Amsterdam (1994).
17) Yamada, T.: Confluence and termination of simply typed term rewriting systems, *12th International Conference on Rewriting Techniques and Applications* (*RTA'01*), *LNCS 2051*, pp.338–352 (2001).

(Received February 21, 2005)
(Accepted July 12, 2005)

Taro Suzuki was born in 1964. He received his D.Sci. degree from the University of Tokyo in 1998. He has been engaged in research in higher-order rewriting, unification and functional-logic programming. He is a member of the IPSJ, ACM and JSSST.

Satoshi Okui was born in 1967. He received his D.Eng. degree from University of Tsukuba in 1995. He has been engaged in research in rewriting, unification, and functional-logic programming. He is a member of the IPSJ.