Regular Paper

Pattern Matching of Incompletely RE-Typed Expressions via Transformation

Satoshi Okui† and Taro Suzuki††

We offer a pattern-matching algorithm based on *incomplete* regular expression (IRE, for short) types. IRE types extend the regular expression types introduced by Hosoya and Pierce in the programming language XDuce. Pattern-matching for IRE-typed expressions provides a capability to uniformly access "context" parts of XML document trees within the framework of pattern-matching theory; we do not rely on external facilities (or notations) such as XPath. In order to describe our pattern-matching algorithm, we adopt a rule-based approach; that is, we present our algorithm as a set of a few simple transformation rules. These rules simulate a transition in non-deterministic top-down tree automata (though we do not deal with tree automata explicitly), while also accumulating bindings for pattern variables. Our pattern-matching algorithm is sound and complete: it enumerates all correct but no incorrect solutions. We give rigorous proofs of these properties. A small but non-trivial example illustrates the expressiveness of our framework.

1. Introduction

Pattern matching is essential for many modern programming languages, especially for functional languages such as ML and Haskell. These languages allow rule-based programming, which makes programs shorter and clearer, and thereby reduce the labor of programming considerably.

XDuce, introduced by Hosoya and Pierce [17], is a programming language for XML document processing equipped with powerful pattern-matching facilities based on *regular expression types* [15),18)]. The development of XDuce and its successors [4),13)] demonstrates that pattern-matching is also very helpful in programming languages for XML document processing.

However, regular expression pattern-matching provides no means to access the "context" of a matched part of a document tree; for this, we need a kind of *second-order* matching called *context matching* [32] (a special case of context unification [10),31)]). To see this, consider a marked-up text such as the following:

```
"This"
<it>"is"</it>
<bf>
  "a"
  <ul><bf>"rich"</bf>"text"</ul>
</bf>
"."
```

This represents the following rich text:

This *is* **a <u>rich text</u>**.

A XDuce-like notation gives it concisely as follows:

```
"This"
it["is"]
bf[
  "a"
  ul[bf["rich"]"text"]]
]
"."
```

The type `RT` of rich texts is then given by the following (extended) regular tree grammar:

`RT` → `(Str[] | (bf|it|ul)[RT])*`

where `Str[]` is the type of plain texts.

The above example contains a redundant `<bf>` tag. This most likely occurs as a result of mechanical processing. In the framework of regular expression pattern-matching, we could write a function for eliminating all inner `<bf>` tags as follows:

```
uniq :: RT->RT
uniq(x) = uniq2(x, false)
uniq2 :: RT×Bool->RT
uniq2([], occ) = []
uniq2(bf[x] y, true)
    = uniq2(x y, true)
uniq2(bf[x] y, false)
    = bf[uniq2(x, true)] uniq2(y, false)
uniq2(it[x] y, occ)
    = it[uniq2(x, occ)] uniq2(y, occ)
uniq2(ul[x] y, occ)
    = ul[uniq2(x, occ)] uniq2(y, occ)
uniq2(s y, occ)
```

† Chubu University
†† The University of Aizu

```
    = s uniq2(y, occ)
```

Here, `x` and `y` are variables of type `RT`, while `occ` and `s` are variables of types `Bool` and `Str[]`, respectively. We have two constants, `true` and `false`, of type `Bool`. The function `uniq2` has two arguments. At the first argument, we find expressions with variables representing patterns of rich texts. For example, the pattern expression `bf[x] y` represents rich texts where the first document (sub)tree has a `bf` tag at the root. On the other hand, the second argument remembers whether we have ever encountered a `<bf>`. Label variables such as in CDuce [4] make the definition of `uniq2` slightly simpler by replacing the last two equations with the following:

```
uniq2(q[x] y, occ)
    = q[uniq2(x, occ)]uniq2(y, occ),
```

where `q` is a variable ranging over {`<it>`, `<ul>`}.

Context matching offers programmers a more convenient way. Suppose we can define the type `RC` of contexts for rich texts as follows:

`RC` → `RT` □ `RT | RT(bf|it|ul)[RC]RT`,

where a distinct symbol □ (hole) indicates that any expression of type `RC` contains exactly one hole, thereby representing a unary context. Let `c` be a variable of type `RC` and t an expression of type `RT` containing a subexpression s, i.e., $t = C[s]$ for a unary context C. A pattern `c{`s`}`☆ then matches against t, resulting in a binding $\{\texttt{c} \mapsto C\}$. The context C therefore becomes available in a function definition. The function `uniq` is now rewritten as follows:

```
uniq :: RT->RT
uniq(c1{bf[c2{bf[x]}]})
        = uniq(c1{bf[c2{x}]})
uniq(x) = x
```

Here, `c1` and `c2` are variables of type `RC`, and `x` is a variable of type `RT`.

The above `RC` is an example of *incomplete* regular expression types (IRE-types, for short). The authors have developed a basis of IRE types in their previous papers [35),36)], but as yet have given no pattern-matching algorithm for IRE-typed expressions supported by a rigorous completeness proof.

This paper presents a pattern-matching algorithm for IRE-typed expressions and demonstrates its soundness and completeness. Our aim is to provide a basic algorithm in a general setting. Issues such as matching strategies [15)], ambiguity [16)], and optimizations [23)] are not discussed.

The remainder of this paper is organized as follows: Section 2 is devoted to an explanation of IRE types. This is almost a summary of Refs. 35) and 36). Section 3 concerns IRE-pattern matching. We begin with a few basic transformation rules, then extend and refine them step by step, finally obtaining our desired algorithm. This algorithm is sound and complete. The proofs are given in Section 4. Finally, Section 5 presents our concluding remarks.

2. IRE-Typed Terms

2.1 Terms

Let $\mathcal{V}$ and $\mathcal{L}$ be disjoint sets of *variables* and *labels*, respectively. The syntax of *preterms* is defined as in **Fig. 1** ☆☆: Here, x and l range over $\mathcal{V}$ and $\mathcal{L}$, respectively. For an application $s\{t_1, \ldots, t_n\}$, s is called its *head*, and $t_1, \ldots, t_n$ its *arguments*. We often write $s\{t_i, \ldots, t_j\}$ as $s\{\bar{t}_{i,j}\}$, and $s\{\bar{t}_{1,n}\}$ as $s\{\bar{t}_n\}$. Intuitively, holes serve as placeholders that may be replaced with any terms by an application. We make no distinction between $(t)\{\}$ and t. An application whose head is a variable is called a *variable application*. A preterm that is neither an empty sequence nor a concatenation is called *individual*. We treat a preterm as a sequence of individual preterms; that is, the concatenation is associative, and () is the identity. We omit parentheses unless ambiguity occurs. The set of variables occurring in a preterm t is denoted by $var(t)$. A preterm t is called *ground* if $var(t) = \emptyset$. A preterm not containing non-variable applications is called a *term*. Note that ground terms contain no applications, representing XML document trees. Let $\#\Box(t)$ stand for the number of holes in a term t. A term t is sometimes called an n-ary *context* if $\#\Box(t) = n$.

t	::=	$()$	Empty
	|	$\Box$	Hole
	|	x	Variable
	|	$(t)\{t, \ldots, t\}$	Application
	|	$l\,[t]$	Tree
	|	$t\,t$	Concatenation

Fig. 1 Syntax of preterms.

☆ Since `[]` has been used to represent document trees, we instead use {} for contexts.

☆☆ The definition of (pre)terms in Refs. 35) and 36) includes function symbols with fixed arity. Function symbols are not treated in this paper, and are therefore omitted.

T	::=	$()$	Empty
	|	$\Box$	Hole
	|	N	Type name
	|	$L\,[T]$	Tree
	|	$T\,T$	Concatenation
	|	$(T \mid T)$	Union
	|	$(T)^*$	Repetition

Fig. 2 Syntax of types.

2.2 Types

Let $\mathcal{T}_\mathcal{L}$ and $\mathcal{T}_\mathcal{N}$ be disjoint sets of *label types* and *type names*, respectively. The syntax of *type expressions* (or just *types*) is defined as in **Fig. 2**: Here, L and N range over $\mathcal{T}_\mathcal{L}$ and $\mathcal{T}_\mathcal{N}$, respectively. The syntax of label types is left unspecified. Boolean operators may be adopted as in the example in Section 1. We assume that the concatenation is associative and that the empty type $()$ is the identity. As normal regular expressions, the repetition and the concatenation have higher precedence than the concatenation and the union, respectively. Parentheses are omitted unless ambiguity occurs. Without type names, we can not take advantage of the full expressiveness of regular tree languages. We need a *type name definition* (written $\mathcal{N}$), that is, a map from a finite set of type names to type expressions. We present a type name definition as a finite set of equations $\{N = \mathcal{N}(N)\}$.

Our treatment of regular tree grammars (unranked tree or hedge grammars [25),27),30),37)], to be precise) follows the style in Ref. 18). In that style, we impose a syntactic restriction called *well-formedness* in order to obtain regularity. Well-formedness excludes, for example, the type definition

$\mathcal{N}(\texttt{T}) = \texttt{a[] T b[] | ()}$,

which produces a context-free (word) language $\{\texttt{a}^n\texttt{b}^n \mid n \geq 0\}$. Other formulations [16),25),27)] forbid the appearance of any type names at the top level, and the repetition $(*)$ is instead built-in. The reason we adopt the style in Ref. 18) is that the repetition is definable in a type name definition. This makes type expansions (introduced later) simpler. In this paper, we consider the type expression $T*$ as an abbreviation for a type name T' defined as $\mathcal{N}(T') = T\,T' \mid ()$.

2.3 Subtyping and Well-Formed Types

In order to define subtyping and the notion of incomplete types, we need to consider the denotation of type expressions.

We define the type denotation function $[\![\cdot]\!]$ as the least function from types to sets of ground terms satisfying the following equations:

$$
\begin{array}{lcl}
[\![\Box]\!] & = & \{\Box\} \\
[\![()]\!] & = & \{()\} \\
[\![N]\!] & = & [\![\mathcal{N}(N)]\!] \\
[\![L\,[T]]\!] & = & \{l\,[t] \mid l \in [\![L]\!], t \in [\![T]\!]\} \\
[\![T_1\,T_2]\!] & = & \{t_1\,t_2 \mid t_1 \in [\![T_1]\!], t_2 \in [\![T_2]\!]\} \\
[\![(T_1|T_2)]\!] & = & [\![T_1]\!] \cup [\![T_2]\!]
\end{array}
$$

Here, $[\![L]\!]$ in the above definition is called the denotation of a label type L, an non-empty subset of $\mathcal{L}$ such that

- $[\![L]\!] \subseteq [\![L']\!]$ is decidable for any label type L, L', and
- for any label l, there exists a unique label type L such that $[\![L]\!] = \{l\}$.

The first condition is required in order to ensure that subtype relations are decidable. The second is to ensure the existence of minimal types (explained later).

Type denotations may be empty. For example, the type name `T` defined by

$\mathcal{N}(\texttt{T}) = \texttt{a[T]}$

denotes the empty set. This sometimes gives rise to difficulty. From now on, we assume that all types denote a non-empty set. As in normal regular languages, we can decide whether a type denotes the empty. See Lemma 3 (p1) in Ref. 36).

The subtype relation $<:$ on types is defined as $T_1 <: T_2 \Leftrightarrow [\![T_1]\!] \subseteq [\![T_2]\!]$.

Proposition 1 [35)] The subtype relation $<:$ is decidable. □

We further restrict the well-formed types mentioned in Section 2.2 to the types whose denotations consist of ground terms with the same number of holes. For example, the types `RT` and `RC` in Section 1 are well-formed. For any type T we can decide whether T is well-formed. See Theorem 1 in Ref. 36).

For a well-formed type T, we define $\#\Box(T) = \#\Box(t)$ for some $t \in [\![T]\!]$. For example, $\#\Box(\texttt{RT}) = 0$ and $\#\Box(\texttt{RC}) = 1$. Note that the subtyping preserves the number of holes: we have $\#\Box(T_1) = \#\Box(T_2)$ for any well-formed types T_1 and T_2 such that $T_1 <: T_2$. We say a well-formed type T is *complete* if $\#\Box(T) = 0$; otherwise, it is *incomplete*. For any well-formed type T, $\#\Box(T)$ is effectively given. See Lemma 3 (p2) in Ref. 36).

From now on, we assume that all types are well-formed.

2.4 Typing Rules

A term t is *well-typed* if a *judgement* $\vdash t : T$ (read "t has type T") is deducible for some (well-formed) type T using the *typing rules*

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

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

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

(Tree)
$$\frac{l \in [\![L]\!] \quad \vdash t : T}{\vdash l\,[t] : L\,[T]}$$

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

(Application)
$$\frac{\vdash t : T \quad \vdash t_1 : T_1 \cdots \vdash t_n : T_n}{\vdash t\{t_1, \ldots, t_n\} : T\{T_1, \ldots, T_n\}}$$
if $n = \#\Box(T) > 0$

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

Fig. 3 Typing rules.

shown in **Fig. 3**. Here, we assume that a type is assigned to each variable. The set of variables with type T is denoted by $\mathcal{V}^T$. We also assume that for each type T the set $\mathcal{V}^T$ is infinite.

In the typing rule (application), $T\{T_1, \ldots, T_n\}$ means the type expression obtained from T by replacing n holes with *complete* types $T_1, \ldots, T_n$ from left to right. See Ref. 36) for the precise definition of $T\{T_1, \ldots, T_n\}$. Note that T has exactly n holes and $T\{T_1, \ldots, T_n\}, T_1, \ldots, T_n$ are all complete. This means that our framework inhibits "partial" application.

Not all terms of an incomplete type contain holes: e.g., a variable $x \in V^T$ has no hole even if T is incomplete. It is, however, true that any context has an incomplete type.

Since our type system admits $\Box$ in type expressions, and we assume that any label belongs to at least one label type, one might expect that any preterm is well-typed. This is indeed true for ground terms, but is not generally true. The only case in which typing fails occurs in the rule (application) when $\#\Box(T)$ differs from the number of arguments.

Our type system is sound and complete in the following sense:

Proposition 2 [35] For any ground term t and any type T, $\vdash t : T$ if and only if $t \in [\![T]\!]$. □

Terms may have several different types. This is mainly due to the presence of (subtyping). It is often convenient to consider the minimum types for any well-typed terms.

Definition 1 We define a partial function τ that maps a term to a type as follows:

$$\begin{array}{rcl} \tau(()) & = & () \\ \tau(\Box) & = & \Box \\ \tau(x) & = & T \\ & & \text{if } x \in \mathcal{V}^T \\ \tau(l[t]) & = & L[\tau(t)] \\ & & \text{if } [\![L]\!] = \{l\} \\ \tau(x\{\bar{t}_n\}) & = & \tau(x)\{\tau(\bar{t}_n)\} \\ & & \text{if } n = \#\Box(\tau(x)) > 0 \\ \tau(t_1\,t_2) & = & \tau(t_1)\,\tau(t_2) \end{array}$$

When L is a singleton $\{l\}$, we identify l as L. We say a type T is *basic* if $T = \tau(t)$ for some ground term t. The following proposition states that $\tau(t)$ is the minimum type of t:

Proposition 3 [35] For each term t, t is well-typed if and only if $\tau(t)$ is defined. Moreover, $\tau(t) <: T$ holds for any type T such that $\vdash t : T$. □

Obviously, we have the following:

Corollary 1 If t is well-typed, we have $\tau(t) <: T \;\Leftrightarrow\; \vdash t : T$. □

From now on, we only consider well-typed preterms.

2.5 Substitution and Normalization

A *substitution* σ is a mapping from variables to terms satisfying the following conditions: (1) the *domain* $dom(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite and (2) $\tau(\sigma(x)) <: \tau(x)$ for every $x \in dom(\sigma)$. The first condition is a common requirement for substitutions in the literature [2),6),29)]. Later, we refer to the second condition as the *type consistency* of σ. We usually present a substitution σ as a finite set $\{x \mapsto \sigma(x) \mid x \in dom(\sigma)\}$. We define a substitution $\sigma\restriction_V$ as $\{x \mapsto \sigma(x) \mid x \in V\}$. A substitution σ is extended to a mapping over preterms as follows:

$$\begin{array}{rcl} ()\sigma & = & () \\ \Box\sigma & = & \Box \\ (l\,[t])\sigma & = & l\,[t\sigma] \\ (t\{\bar{t}_n\})\sigma & = & t\sigma\{\bar{t}_n\sigma\} \\ (t_1\,t_2)\sigma & = & t_1\sigma\,t_2\sigma \end{array}$$

where $t, t_1, \ldots, t_n$ range over preterms. The composition $\sigma\rho$ of two substitutions σ and ρ is simply the function composition; i.e., $t(\sigma\rho) = (t\sigma)\rho$ for any preterm t. We write $\vdash \theta_1 = \theta_2\ [V]$ if $x\theta_1 = x\theta_2$ for any $x \in V$. We write $\vdash \theta_1 \leq \theta_2\ [V]$ if $\vdash \sigma\theta_1 = \theta_2\ [V]$ for some substitution σ. A substitution θ is called *ground* if $x\theta$ is a ground term for any $x \in dom(\theta)$.

Note that the set of terms is not closed under substitutions: a substitution may give rise to a preterm by instantiating the head of a variable application with a term. However, reducing non-variable applications after performing a substitution always produces a term. To see this, we first give a formal definition of application by the following equations on preterms:

$$\begin{array}{rcl} \Box\{t\} & = & t \\ l[s]\{\bar{t}_n\} & = & l[s\{\bar{t}_n\}] \\ (s_1 s_2)\{\bar{t}_n\} & = & (s_1)\{\bar{t}_k\}(s_2)\{\bar{t}_{k+1,n}\} \end{array}$$

where $n > 0$, $0 \leq k = \#\Box(\tau(s_1)) \leq n$, and $s, s_1, s_2, t, t_1, \ldots, t_n$ range over terms.

We write $\vdash s = t$ if preterms s and t are equal with respect to the above equations. We say a term s is a *canonical form* of a preterm t if $\vdash s = t$. The following proposition easily follows from the above definition.

Proposition 4 (normalization) Any (well-typed) preterm has a unique canonical form. □

Hereafter, we assume that any preterm is implicitly normalized to its unique canonical form. This helps us to present our pattern matching algorithm in a simpler manner.

3. Pattern Matching

In this section, we focus on pattern matching of IRE-typed terms.

3.1 IRE Matching Problems

Given two well-typed terms s, t with s ground, an IRE pattern-matching problem asks us to find a substitution θ such that $\vdash s = t\theta$. In other words, IRE pattern matching is regarded as a restricted form of equation solving. Formally, an *IRE* (*pattern*) *matching problem* is defined as a possibly empty sequence of equations $s_1 \approx t_1; \ldots; s_n \approx t_n$ where $s_1, \ldots, s_n$ are ground terms of complete types and $t_1, \ldots, t_n$ are terms of complete types. Since holes are regarded as placeholders, it is reasonable to assume that both sides of equations contain no holes. Hence, the left-hand sides have complete types. Accordingly, the right-hand sides must also be complete. Note, however, that the right-hand sides may contain variables of incomplete types. For an IRE matching problem $E = s_1 \approx t_1; \ldots; s_n \approx t_n$ $(n \geq 0)$ and a substitution θ, we write $\vdash E\theta$ if $\vdash s_1 = t_1\theta \wedge \ldots \wedge \vdash s_n = t_n\theta$ holds. We then call $\theta\restriction_{var(E)}$ a *solution* of E, where $var(E)$ stands for the set of all variables in E. We refer to the set of all solutions of E as $\mathcal{M}(E)$.

Since the normalization to canonical forms eliminates no variables, we obtain the following:

Proposition 5 (ground solutions) Let E be an IRE pattern-matching problem. Any solutions of E are ground substitutions. □

A set S of solutions is called *complete*[3] if for any solution θ of E there exists $\theta' \in S$ such that $\theta' \leq \theta\ [var(E)]$. $\mathcal{M}(E)$ itself is, in a trivial sense, a complete set of solutions. A set S of solutions is called *minimal*[3] if its elements are pairwise incomparable; i.e., for any solutions $\theta_1, \theta_2 \in S$, either $\vdash \theta_1 \leq \theta_2\ [var(E)]$ or $\vdash \theta_2 \leq \theta_1\ [var(E)]$ implies $\vdash \theta_1 = \theta_2\ [var(E)]$. Since ground substitutions are pairwise incomparable, we immediately obtain the following:

Corollary 2 Let E be an IRE pattern-matching problem. $\mathcal{M}(E)$ is the only minimal complete set of solutions. □

3.2 Basic Transformation Rules

There is more than one way to describe pattern-matching algorithms: pseudo-programming languages, a variety of automata, and so forth. Here we take a rule-based approach [3),14),33)] familiar in unification theory. This approach gives a set of transformation rules, telling us how pattern-matching problems are transformed, step by step, into "simpler" (formally defined later) ones.

We begin with the case in which we have no variables of incomplete types. The transformation rules for that case are shown in **Fig. 4**. Any rules are of the form

$$\frac{E}{E'} \quad \text{if } P$$

meaning that a sequence of equations of the form E is transformed into E' provided that the side condition P is satisfied. Some rules have no side conditions.

We write $E \Rightarrow_{[\alpha],\sigma} E'$ if E is transformed to E' using a transformation rule $[\alpha]$, yielding a substitution σ. For rules [e] and [d], σ is regarded as the empty substitution, and is therefore dropped.

Successive applications of the transformation

[e] Removal of empty sequence

$$\frac{() \approx (); E}{E}$$

[d] Decomposition

$$\frac{l[s_1]s_2 \approx l[t_1]t_2; E}{s_1 \approx t_1; s_2 \approx t_2; E}$$

[v] Variable elimination

$$\frac{s_1 s_2 \approx x\, t; E}{(s_2 \approx t; E)\sigma}$$

if $\tau(s_1) <: \tau(x)$ where $\sigma = \{x \mapsto s_1\}$

Fig. 4 Transformation rules for IRE pattern matching (no variables of incomplete types).

[i] Imitation

$$\frac{l[s_1]s_2 \approx x\, t; E}{(s_1 \approx y; s_2 \approx z\, t; E)\sigma}$$

if $l[T_1]\, T_2 <: \tau(x)$ for some T_1, T_2, where $y \in V^{T_1}$ and $z \in V^{T_2}$ are fresh and $\sigma = \{x \mapsto l[y]z\}$

[p] Projection

$$\frac{s \approx x\, t; E}{(s \approx t; E)\sigma}$$

if $() <: \tau(x)$ where $\sigma = \{x \mapsto ()\}$

Fig. 5 Two new rules replacing [v].

rules yield a *derivation*

$E_0 \Rightarrow_{[\alpha_1],\sigma_1} E_1 \cdots E_{n-1} \Rightarrow_{[\alpha_n],\sigma_n} E_n$,

which is also written as $E_0 \Rightarrow^*_{\sigma_1 \cdots \sigma_n} E_n$. A derivation *succeeds* if E_n is the empty sequence ε of equations. A derivation *fails* if E_n is not empty and no rule is applicable to E_n.

The above rules are very similar to a rule-based presentation of the usual unification of first-order terms [14),24)]. A major difference is that in [v] the variable x can be bound to a prefix s_1 of the left-hand side $s_1 s_2$ (not only to the whole left-hand side). Thus, there are many possibilities for s_1. The side condition $\tau(s_1) <: \tau(x)$ (equivalently, $\vdash s_1 : \tau(x)$) must be invoked for every possible s_1 repeatedly.

These observations prompt us to replace [v] with the new rules ☆ shown in **Fig. 5**. While variable elimination [v] tries to make a whole binding to x at once, imitation [i] only guesses the root label to be bound to x, leaving the remaining trees (s_1 and s_2) up to subsequent steps.

Since [i] only makes a binding for the root label, say l, of the leftmost individual of the left-hand side, we only have to check whether $[\![\tau(x)]\!]$ contains a term whose leftmost individual has the root label l. This task (precisely explained later as type expansion) is much easier than checking $s_1 \in [\![\tau(x)]\!]$ for all prefix s_1, since we only deal with the leftmost individual rather than all prefixes, and only visit the root label rather than whole trees.

The other new rule, projection [p], serves for terminating the iteration of [i]. Its side condition only requires checking of whether $\tau(x)$ is nullable (a type T is *nullable* if $() \in [\![T]\!]$).

Note that [i] and [p] are not exclusive (both may be applicable to the same problem).

Example 1 Consider an IRE matching problem `a[]a[]` $\approx$ `x y` where τ(`x`) = `a[]` and τ(`y`) = `a[]*`. The type of x ensures the following unique solution:

$\{\mathtt{x} \mapsto \mathtt{a[]}, \mathtt{y} \mapsto \mathtt{a[]}\}$

The following derivation with respect to $\{[e], [d], [v]\}$ gives this solution:

$$\begin{array}{ll} & \mathtt{a[]a[]} \approx \mathtt{x\ y} \\ \Rightarrow_{[v],\{x \mapsto a[]\}} & \mathtt{a[]} \approx \mathtt{y} \\ \Rightarrow_{[v],\{y \mapsto a[]\}} & \mathtt{()} \approx \mathtt{()} \\ \Rightarrow_{[e]} & \varepsilon. \end{array}$$

On the other hand, using [i] and [p] instead of [v], we obtain the following derivation:

$$\begin{array}{ll} & \mathtt{a[]a[]} \approx \mathtt{x\ y} \\ \Rightarrow_{[i],\{x \mapsto a[x1]x2\}} & \mathtt{()} \approx \mathtt{x1}; \mathtt{a[]} \approx \mathtt{x2\ y} \\ \Rightarrow_{[p],\{x1 \mapsto ()\}} & \mathtt{()} \approx \mathtt{()}; \mathtt{a[]} \approx \mathtt{x2\ y} \\ \Rightarrow_{[e]} & \mathtt{a[]} \approx \mathtt{x2\ y} \\ \Rightarrow_{[p],\{x2 \mapsto ()\}} & \mathtt{a[]} \approx \mathtt{y} \\ \Rightarrow_{[i],\{y \mapsto a[y1]y2\}} & \mathtt{()} \approx \mathtt{y1}; \mathtt{()} \approx \mathtt{y2} \\ \Rightarrow_{[p],\{y1 \mapsto ()\}} & \mathtt{()} \approx \mathtt{()}; \mathtt{()} \approx \mathtt{y2} \\ \Rightarrow_{[e]} & \mathtt{()} \approx \mathtt{y2} \\ \Rightarrow_{[p],\{y2 \mapsto ()\}} & \mathtt{()} \approx \mathtt{()} \\ \Rightarrow_{[e]} & \varepsilon. \end{array}$$

where the fresh variables introduced intermediately are of types τ(`x1`) = τ(`x2`) = τ(`y1`) = `()` and τ(`y2`) = `a[]*`. □

To make the above derivation with respect to $\{[e], [d], [i], [p]\}$, we visit each label `a` of `a[]a[]` only once to confirm the side conditions of the rules used in the above derivation. On the other hand, if we use [v] we should look at each label

☆ The names of [i] and [p] are coined from the rules for higher-order unification [19),33)]. Sequence unification is closely related [22)] to order-sorted higher-order unification.

[h] Hole introduction

$$\frac{s_1\, s_2 \approx x\{\bar{u}_n\}t;\ E}{(s_1 \approx u_1;\ s_2 \approx y\{\bar{u}_{2,n}\}t;\ E)\sigma}$$

if $n > 0$, $\tau(s_1) <: \tau(u_1)$
and $\square T <: \tau(x)$ for some T,
where $y \in V^T$ is fresh and $\sigma = \{x \mapsto \square y\}$

[td] Decomposition via type information

$$\frac{l[s_1]\, s_2 \approx x\{\bar{u}_n\}t;\ E}{(s_1 \approx y\{\bar{u}_k\};\ s_2 \approx z\{\bar{u}_{k+1,n}\}t;\ E)\sigma}$$

if $n > 0$ and $l[T_1]T_2 <: \tau(x)$ for some T_1,T_2
where $k = \#\square(T_1)$,
$y \in V^{T_1}$, $z \in V^{T_2}$ are fresh,
and $\sigma = \{x \mapsto l[y]z\}$

Fig. 6 Additional transformation rules for variables of incomplete types.

twice because we have to check the following:

$$\begin{array}{lcl} \texttt{a[]} & <: & \tau(\texttt{x}), \\ \texttt{a[]a[]} & <: & \tau(\texttt{x}), \\ \texttt{a[]} & <: & \tau(\texttt{y}). \end{array}$$

The difference becomes more significant for more complex cases; for example, consider $a[]a[]a[] \approx xy$, where $\tau(x) = \texttt{a[]}$ and $\tau(y) = \texttt{a[]*}$.

We now consider the case in which we have variables of incomplete types. To deal with such variables, we need the additional rules shown in **Fig. 6**. One way to understand these two rules is to think of holes as variables (and variables of incomplete types as contexts). Suppose $\tau(x)$ indicates that x can be bound for a term whose leftmost individual term is a hole. The definition of an application tells us that this hole can be replaced with the first argument u_1. Thus, the hole introduction [h] tries to match u_1 against a prefix s_1 of the left-hand side, while introducing a hole into x as a placeholder. Next, suppose otherwise. The decomposition via type information [td] then guesses the root label in the same way as [d] or [i]. The arguments are distributed according to the type information.

The rule [h] has the same problem as [v]. Therefore, we replace [h] with the slightly modified rule (the partial binding via type information [tb]) in **Fig. 7**.

Example 2 Consider an IRE matching problem $\texttt{a[a[]]} \approx \texttt{x\{y\{()\}\}}$ where $\tau(\texttt{x}) = \tau(\texttt{y}) = \texttt{T}$ such that $\mathcal{N}(\texttt{T}) = \texttt{a[T]}\,|\,\square$. This problem has the following three solutions:

[tb] Partial binding via type information

$$\frac{s \approx x\{\bar{u}_n\}t;\ E}{(s \approx u_1 y\{\bar{u}_{2,n}\}t;\ E)\sigma}$$

if $n > 0$ and $\square T <: \tau(x)$ for some T,
where $y \in V^T$ are fresh and $\sigma = \{x \mapsto \square y\}$

Fig. 7 A new rule replacing [h].

$$\begin{array}{ll} \{\texttt{x} \mapsto \texttt{a[a[}\square\texttt{]]},\ & \texttt{y} \mapsto \texttt{()}\} \\ \{\texttt{x} \mapsto \texttt{a[}\square\texttt{]},\ & \texttt{y} \mapsto \texttt{a[]}\} \\ \{\texttt{x} \mapsto \square,\ & \texttt{y} \mapsto \texttt{a[a[]]}\} \end{array}$$

The second is obtained by the following derivation:

$$\begin{array}{ll} & \texttt{a[a[]]} \approx \texttt{x\{y\{()\}\}} \\ \Rightarrow_{[\mathrm{td}],\{\texttt{x}\mapsto\texttt{a[x1]x2}\}} & \texttt{a[]} \approx \texttt{x1\{y\{()\}\}}; \\ & \texttt{()} \approx \texttt{x2} \\ \Rightarrow_{[\mathrm{tb}],\{\texttt{x1}\mapsto\square\texttt{x11}\}} & \texttt{a[]} \approx \texttt{y\{()\}x11};\ \texttt{()} \approx \texttt{x2} \\ \Rightarrow_{[\mathrm{td}],\{\texttt{y}\mapsto\texttt{a[y1]y2}\}} & \texttt{()} \approx \texttt{y1\{()\}};\ \texttt{()} \approx \texttt{y2 x11}; \\ & \texttt{()} \approx \texttt{x2} \\ \Rightarrow_{[\mathrm{tb}],\{\texttt{y1}\mapsto\square\texttt{y3}\}} & \texttt{()} \approx \texttt{y3};\ \texttt{()} \approx \texttt{y2 x11}; \\ & \texttt{()} \approx \texttt{x2} \\ \Rightarrow^* & \varepsilon \end{array}$$

where the fresh variables introduced intermediately are of types $\tau(\texttt{x1}) = \tau(\texttt{y1}) = \texttt{T}$ and $\tau(\texttt{x2}) = \tau(\texttt{y2}) = \tau(\texttt{y3}) = \tau(\texttt{x11}) = \texttt{()}$. The last derivation $\Rightarrow^*$ consists of [p] and [e], instantiating `y2`, `y3`, `x11`, and `x2` to `()`. The other solutions are also obtained by some other derivations. □

If we use [h] instead of [td], we further need to check the following containments in order to confirm the side condition of [h]:

$$\begin{array}{lcl} \texttt{()} & <: & \tau(\texttt{y\{()\}}), \\ \texttt{a[]} & <: & \tau(\texttt{y\{()\}}). \end{array}$$

It would be possible to combine [i] and [td] into one rule, because [i] is derived from [td] if we allow $n = 0$ in [td]. However, we leave [i] as a distinct rule in order to emphasize the differences between variables of complete and incomplete types.

Hereafter, we refer to the set of six rules
$\{[\mathrm{e}], [\mathrm{d}], [\mathrm{i}], [\mathrm{p}], [\mathrm{tb}], [\mathrm{td}]\}$
as TRANS.

3.3 Finite Search Trees

Given IRE matching problem, TRANS transforms it into other problems. Consecutive transformations thus give rise to a search tree where each node is labeled with an IRE matching problem. We now consider an important question: given a set of equations, is the search tree finite? The answer is unfortunately "No."

This is because the rules [i], [td], and [tb] lead to infinite branching by introducing fresh variables of infinitely many different types. For example, consider a variable `x` with $\tau(\mathtt{x}) = \mathtt{T}$, where `T` is defined as $\mathcal{N}(\mathtt{T}) = \mathtt{a[T]T \mid ()}$. We then have infinitely many T_1, T_2 that fulfill a side condition $\mathtt{a}[T_1]T_2 <: \tau(\mathtt{x})$:

$$T_1, T_2 = (), \mathtt{a}[], \mathtt{a}[\mathtt{a}[]], \ldots$$

To obtain finite search trees, we need some stronger side conditions.

One way to achieve finite branching is to use non-deterministic top-down tree automata [11]. For this, we replace a type name definition with a regular tree grammar of the forms

$$N \to (), \quad N \to \Box N, \quad N \to L[N]N$$

where N ranges over type names rather than arbitrary type expressions. The side condition of, for example, [i] then takes the form

"if $\tau(x) \to L[N_1]N_2$ exists and $l \in [\![L]\!]$",

and the fresh variables y and z are now of type N_1 and N_2 respectively.

A translation from a type name definition to a tree automaton is not easy to describe formally (and is not necessary for our purpose of obtaining finite branching). We therefore enumerate only one step of the translation, each time applying transformation rules. For this, we repeatedly expand type names with their definitions until a tree or a hole appears as the leftmost individual term.

To avoid an infinite loop in the expansion, we restrict ourselves to type definitions without *left recursion* [1] (either directly or indirectly). Because detecting left recursion is a standard technique in compiler construction, we do not elaborate it here (see Ref. 1)).

We now introduce the type expansion. **Fig. 8** defines the type expansion in the way of structural operational semantics (i.e., natural semantics). We read $T \Uparrow S$ as "A type T is expanded to any types in S."

Since we consider type definitions without left recursion, a derivation tree always exists and S is uniquely determined for any type T. We refer to the set S in $T \Uparrow S$ as the set of *expansions* (write $T!$) of the type T. By abuse of notation, we write $T_1 <: T_2!$ if $T_1 <: T$ for some $T \in T_2!$.

Lemma 1 Let T_i $(i = 1, 2, 3)$ be types, and let T be a basic type. If $T <: (T_1 \mid T_2)\, T_3$, we have $T <: T_1\, T_3$ or $T <: T_2\, T_3$.

Proof: Suppose $T <: (T_1 \mid T_2)\, T_3$. Since T is basic, we can apply Corollary 1 to obtain $\vdash t : (T_1 \mid T_2)\, T_3$, where t is a ground term such that $T = \tau(t)$. Proposition 2 gives $t \in [\![(T_1 \mid T_2)\, T_3]\!]$. It follows from the definition of $[\![\ ]\!]$ that there exist ground terms t_1 and t_2 such that $t = t_1\, t_2$, $t_1 \in [\![T_1]\!] \cup [\![T_2]\!]$ and $t_2 \in [\![T_3]\!]$. Therefore we obtain $t \in [\![T_1\, T_3]\!]$ or $t \in [\![T_2\, T_3]\!]$ and hence $T <: T_1\, T_3$ or $T <: T_2\, T_3$ holds. □

$$() \Uparrow \{()\}$$

$$\Box T \Uparrow \{\Box T\}$$

$$L[T_1]T \Uparrow \{L[T_1]T\}$$

$$\frac{T_1 T \Uparrow S_1 \quad T_1 T \Uparrow S_2}{(T_1 \mid T_2)T \Uparrow S_1 \cup S_2}$$

$$\frac{\mathcal{N}(N)T \Uparrow S}{NT \Uparrow S}$$

Fig. 8 Type expansion.

Proposition 6 (finite type expansion)

(1) For any type T, the set $T!$ is finite.

(2) For any basic type T_1 and any type T_2, we have

$$T_1 <: T_2 \ \Rightarrow\ T_1 <: T_2!$$

Proof: (1) is obvious from the definition of the type expansion. We show (2) by induction on the height of derivation trees obtained from Fig. 8. Suppose $T_1 <: T_2$. We prove the essential case $T_2 = (T_{2,1} \mid T_{2,2})\, T_{2,3}$ (the other cases are obvious). From Lemma 1, we have (i) $T_1 <: T_{2,1}\, T_{2,3}$ or (ii) $T_1 <: T_{2,2}\, T_{2,3}$. Suppose the case (i). From the induction hypothesis, we have $T_1 <: T_{2,1}\, T_{2,3}!$. Hence, $T_1 <: (T_{2,1} \mid T_{2,2})\, T_{2,3}!$ by the fourth rule in Fig. 8. Case (ii) is similar. □

The assumption that T_1 is basic is necessary (consider the case $T_1 = T_2 = \mathtt{a[]|b[]}$).

Example 3 Consider the types `RT` and `RC` in Section 1. Without the repetition $(*)$, they are rewritten as follows:

```
N(RT) =
    (Str[] | (bf|it|ul)[RT])RT | ()
N(RC) =
    RT□RT | RT(bf|it|ul)[RC]RT
```

`RT!` consists of the following three elements:

```
Str[]RT,
(bf|it|ul)[RT]RT,
()
```

On the other hand, `RC!` consists of the following six elements:

```
Str[]RT□RT,
(bf|it|ul)[RT]RT□RT,
```

□RT,
Str[]RT(bf|it|ul)[RC] RT,
(bf|it|ul)[RT]RT(bf|it|ul)[RC] RT,
(bf|it|ul)[RC] RT
□

The idea of type expansion appears extensively in the literature. For regular word languages, it is known as *Brzozowski derivatives*[5),7),8)] (see also Ref. 9) and Ref. 34)). Type expansion is also related to *recursive-decent parsing*[1)], a well-known parsing technique in compiler construction, although we do not use a read-ahead and we deal with trees rather than words. A similar technique appears in Ref. 18).

We next show that any derivation is of finite length; it reaches the empty sequence or an IRE matching problem for which transformation rules are no longer applicable. To see this, we introduce a well-founded ordering on IRE matching problems. For this, we use a *multiset ordering*[6),12)]. Given a set S, a *multiset* on S is a mapping M from S to the set of non-negative integers such that $\{s \in S \mid M(s) > 0\}$ is finite. We present a multiset in the same way as a usual set $\{s_1, s_2, \ldots, s_n\}$ except that we allow duplicated elements. We denote the set of all multisets on S as $S^{\#}$. For $M, M' \in S^{\#}$, we define the membership $s \in M$ as $M(s) > 0$ and the union $M \cup M'$ as a mapping such that $(M \cup M')(s) = M(s) + M(s')$. Let $<$ be a strict partial order on a set S. We define the multiset ordering $<_{\#}$ on $S^{\#}$ induced by $<$ as the smallest transitive relation such that for all $s \in S$ and $M, M' \in S^{\#}$, if $x < s$ for all $x \in M'$ then $M \cup M' <_{\#} M \cup \{s\}$. Note that $<_{\#}$ is well-founded if $<$ is.

Definition 2 Let $E(= e_1; \ldots; e_n)$ be an IRE matching problem, and let $\{x_1, \ldots, x_m\} = var(E)$.

(1) $\#s(E)$ stands for a multiset: $\{k_1, \ldots, k_n\}$, where k_i is the number of symbols[☆] in the left hand side of e_i for $1 \le i \le n$.
(2) $\#r(E)$ stands for a multiset: $\{\#\Box(\tau(x_1)), \ldots, \#\Box(\tau(x_m))\}$.
(3) Upon IRE matching problems, we define an ordering $\lhd$ as follows: $E \lhd E'$ iff $\#s(E) <_{\#} \#s(E') \vee (\#s(E) = \#s(E') \wedge \#r(E) <_{\#} \#r(E'))$ where $<_{\#}$ is the multiset ordering induced by the ordering $<$ on non-negative integers. □

Note that $\#\Box(\tau(x))$ amounts to the rank of a variable application whose head is x since both sides of equations are well-typed.

[☆] The number of symbols in () is 0.

[e] Removal of empty sequence

$$\frac{() \approx (); E}{E}$$

[d] Decomposition

$$\frac{l[s_1]s_2 \approx l[t_1]t_2; E}{s_1 \approx t_1; s_2 \approx t_2; E}$$

[p] Projection

$$\frac{s \approx x\, t; E}{(s \approx t; E)\sigma}$$

if $() \in \tau(x)!$ where $\sigma = \{x \mapsto ()\}$

[i] Imitation

$$\frac{l[s_1]s_2 \approx x\, t; E}{(s_1 \approx y; s_2 \approx z\, t; E)\sigma}$$

if $L[T_1]\, T_2 \in \tau(x)!$ for some T_1, T_2 and $l \in L$, where $y \in V^{T_1}$ and $z \in V^{T_2}$ are fresh and $\sigma = \{x \mapsto l[y]z\}$

[tb] Partial binding via type information

$$\frac{s \approx x\{\bar{u}_n\}t; E}{(s \approx u_1 y\{\bar{u}_{2,n}\}t; E)\sigma}$$

if $n > 0$ and $\Box T \in \tau(x)!$ for some T, where $y \in V^T$ are fresh and $\sigma = \{x \mapsto \Box y\}$

[td] Decomposition via type information

$$\frac{l[s_1]\, s_2 \approx x\{\bar{u}_n\}t; E}{(s_1 \approx y\{\bar{u}_k\}; s_2 \approx z\{\bar{u}_{k+1,n}\}t; E)\sigma}$$

if $n > 0$ and $l[T_1]T_2 \in \tau(x)!$ for some T_1,T_2 and $l \in L$, where $k = \#\Box(T_1)$, $y \in V^{T_1}$, $z \in V^{T_2}$ are fresh, and $\sigma = \{x \mapsto l[y]z\}$

Fig. 9 Transformation rules for pattern matching (finitely branching).

Proposition 7 (finite derivations) For any pattern-matching problem E, the length of any derivations via TRANS issued from E is finite. □

Propositions 6–7 give the first main result:

Theorem 1 (finite search trees) Any search trees of TRANS are finite. □

The final version of TRANS capable of finite branching is given in **Fig. 9**.

4. Properties

4.1 Soundness

All solutions given by the transformation rules in TRANS are correct.

Lemma 2 For any derivation step $E \Rightarrow_{[\alpha],\sigma} E'$ with $[\alpha] \in \mathsf{TRANS}$ and a substitution θ such that $\vdash E'\theta$, we have $\vdash E\sigma\theta$.

Proof: The proof is trivial for [e], [d], and easy for [i], [p]. We give proof of the cases of [tb] and [td].

Consider the case $[\alpha] = [\mathrm{tb}]$. Let E be of the form $s \approx x\{\bar{u}_n\}t; E_1$. The assumption $\vdash E'\theta$ implies (1) $\vdash s \approx (u_1 y\{\bar{u}_{2,n}\}t)\sigma\theta$ and (2) $\vdash E_1\sigma\theta$. We can deduce

$$\begin{aligned}(x\{\bar{u}_n\}t)\sigma\theta &= (x\sigma\theta)\{\bar{u}_n\sigma\theta\}(t\sigma\theta)\\ &= (\Box y\theta)\{\bar{u}_n\sigma\theta\}(t\sigma\theta)\\ &\qquad \text{by } x\sigma = \Box y\\ &= (u_1\sigma\theta y\theta)\{\bar{u}_{2,n}\sigma\theta\}(t\sigma\theta)\\ &\qquad \text{by def. applications}\\ &= ((u_1 y)\sigma\theta)\{\bar{u}_{2,n}\sigma\theta\}(t\sigma\theta)\\ &\qquad \text{by } y\sigma = y\\ &= s\\ &\qquad \text{by (1).}\end{aligned}$$

From this and (2) we obtain $\vdash E\sigma\theta$.

Consider the case $[\alpha] = [\mathrm{td}]$. Let E be of the form $l[s_1]\, s_2 \approx x\{\bar{u}_n\}t; E_1$. The assumption $\vdash E'\theta$ implies (1) $\vdash s_1 \approx (y\{\bar{u}_k\})\sigma\theta$, (2) $\vdash s_2 \approx (z\{\bar{u}_{k+1,n}\}t)\sigma\theta$ and (3) $\vdash E_1\sigma\theta$ where $1 \le k \le n$. We can deduce

$$\begin{aligned}(x\{\bar{u}_n\}t)\sigma\theta &= (x\sigma\theta)\{\bar{u}_n\sigma\theta\}(t\sigma\theta)\\ &= ((l[y]z)\theta)\{\bar{u}_n\sigma\theta\}(t\sigma\theta)\\ &\qquad \text{by } x\sigma = l[y]z\\ &= l[y\theta\{\bar{u}_k\sigma\theta\}]z\theta\{\bar{u}_{k+1,n}\sigma\theta\}(t\sigma\theta)\\ &\qquad \text{by def. applications}\\ &= l[y\{\bar{u}_k\}\sigma\theta](z\{\bar{u}_{k+1,n}\}t)\sigma\theta\\ &\qquad \text{by } y\sigma = y,\ z\sigma = z\\ &= l[s_1]s_2\\ &\qquad \text{by (1) and (2).}\end{aligned}$$

From this and (3) we obtain $\vdash E\sigma\theta$. □

Theorem 2 (soundness) For any derivation $E \Rightarrow^*_\theta \varepsilon$ we have $\vdash E\theta$.

Proof: The proof is given by induction on the length of derivations. If the length of $E \Rightarrow^*_\theta \varepsilon$ is 0 then E is empty. So, the result obviously holds. Otherwise, the given derivation is written as $E \Rightarrow_{[\alpha],\sigma} E' \Rightarrow^*_{\theta'} \varepsilon$ with a substitution θ' such that $\sigma\theta' = \theta$. By the induction hypothesis, we have $\vdash E'\theta'$. Using Lemma 2 we obtain the conclusion $\vdash E\theta$. □

4.2 Completeness

Next, we state that all correct solutions are found by TRANS. The first lemma is standard and easy to prove.

Lemma 3 For any substitutions $\sigma, \sigma_1, \sigma_2$ and a set V of variables, $\sigma_1 = \sigma_2\ [(V \backslash dom(\sigma)) \cup vcod(\sigma)]$ implies $\sigma\sigma_1 = \sigma\sigma_2\ [V]$. □

Here, $vcod(\sigma)$ stands for $\cup_{x \in dom(\sigma)} var(\sigma(x))$. The next lemma is used to confirm the side conditions.

Lemma 4 Let x be a variable, and let θ be a substitution.

(1) If $x\theta = ()$, we have $() \in \tau(x)$!

(2) If $x\theta = l[s_1]s_2$ for some label l and ground terms s_1 and s_2, we have $L[T_1]T_2 \in \tau(x)$! such that both $l \in L$ and $\tau(s_i) <: T_i (i = 1, 2)$ hold.

(3) If $x\theta = \Box s$ for some ground term s, we have $\Box T \in \tau(x)$! such that $\tau(s) <: T$ holds.

Proof: We give a proof only of (2) and omit the other easier cases. Suppose $x\theta = l[s_1]s_2$. Let T be $\tau(l[s_1]s_2) (= l[\tau(s_1)]\tau(s_2))$. The type consistency of θ ensures that $T <: \tau(x)$. Since T is basic, we have $T <: \tau(x)!$ from Proposition 6 (2). It is not the case that $T <: ()$ or $T <: \Box T_1$ holds for some type T_1. Hence, the only possibility is to have some type expression $L[T_1]T_2 \in \tau(x)!$ such that $l \in L$, $\tau(s_1) <: T_1$ and $\tau(s_2) <: T_2$. □

The following lemma is a key for proving completeness.

Lemma 5 Suppose a non-empty sequence E of equations and a substitution θ such that $\vdash E\theta$. There exists a derivation rule $[\alpha]$ in TRANS, substitutions σ, θ_1 and an IRE matching problem E_1 satisfying the following properties:

- $E \Rightarrow_{[\alpha],\sigma} E_1$;
- $\vdash E_1\theta_1$;
- $E_1 \lhd E$;
- $\sigma\theta_1 = \theta\ [var(E)]$.

Proof: Let $E = s \approx t; E'$. We first distinguish two cases.

[1] Suppose $t = ()$. We have $s = ()$. Put $\sigma = \varepsilon$, $E_1 = E'$, and $\theta_1 = \theta$. It immediately follows that $E \Rightarrow_{[e]} E_1$, $\vdash E_1\theta_1$ and $\#s(E_1) <_{mult} \#s(E)$.

[2] Suppose $t \neq ()$. We distinguish three cases according to the leftmost individual of t.

[2-1] Suppose $t = l[t_1]t_2$ for some l, t_1, and t_2. s is of the form $l[s_1]s_2$; otherwise, E has no solutions. Put $\sigma = \varepsilon$, $E_1 = s_1 \approx t_1; s_2 \approx t_2; E'$

and $\theta_1 = \theta$. We can observe $E \Rightarrow_{[d]} E_1, \vdash E_1\theta_1$ and $\#s(E_1) <_{mult} \#s(E)$.

[2-2] Suppose $t = x\,t_2$ for some x and t_2. For some ground terms s_1, s_2 we have $s = s_1 s_2, \vdash s_1 \approx x\theta$ and $\vdash s_2 \approx t_2\theta$. We further distinguish two cases according to the leftmost individual term of a ground term $x\theta$.

[2-2-1] Suppose $x\theta = ()$. We obtain $s_1 = ()$ and $s_2 = s$. Lemma 4 (1) gives $() \in \tau(x)!$ so that [p] is applicable. Put $\sigma = \{x \mapsto ()\}$, $E_1 = (s \approx t_2; E')\sigma$ and $\theta_1 = \theta\backslash\{x \mapsto ()\}$. We can observe $E \Rightarrow_{[p]} E_1, \vdash E_1\theta_1, \#s(E_1) = \#s(E)$ and $\#r(E_1) <_{mult} \#r(E)$, and $\sigma\theta_1 = \theta\ [var(E)]$.

[2-2-2] Suppose $x\theta \neq ()$. Since neither $s_1 = ()$ nor the leftmost individual of s_1 is $\Box$ (because s has a complete type), we write s_1 as $l[s_{1,1}]s_{1,2}$. Lemma 4 (2) gives a type $L[T_1]T_2 \in \tau(x)!$ for some L, T_1, and T_2 such that $l \in L$ and $\tau(s_i) <: T_i (i = 1, 2)$. Hence, [i] is applicable. Put $\sigma = \{x \mapsto l[y]z\}$ with fresh variables $y \in V^{T_1}$ and $z \in V^{T_2}$, $E_1 = s_{1,1} \approx y; s_{1,2}s_2 \approx zt_2; E'$ and $\theta_1 = (\theta\backslash\{x \mapsto s_1\}) \cup \{y \mapsto s_{1,1}, z \mapsto s_{1,2}\}$. We can observe $E \Rightarrow_{[i]} E_1, \vdash E_1\theta_1, \#s(E_1) <_{mult} \#s(E)$, and $\sigma\theta_1 = \theta\ [var(E)]$.

[3] Suppose $t = x\{t_{1,1}, \ldots, t_{1,n}\}t_2$ for some x, $t_{1,1}, \ldots, t_{1,n}$ $(n > 0)$ and t_2. For some ground terms s_1, s_2 we have $s = s_1 s_2, \vdash s_1 \approx x\theta\{t_{1,1}\theta, \ldots, t_{1,n}\theta\}$ and $\vdash s_2 \approx t_2\theta$. We distinguish two cases according to the leftmost individual of a ground term $x\theta$.

[3-1] Suppose the leftmost individual of $x\theta$ is $\Box$; i.e., $x\theta = \Box c$ for some ground term c (c may be a context). Lemma 4 (3) gives $\Box T \in \tau(x)!$ for some T such that $\tau(c) <: T$. Hence, [tb] is applicable. Put $\sigma = \{x \mapsto \Box y\}$ for a fresh $y \in V^T$, $E_1 = (s \approx t_{1,1}y\{t_{1,2}, \ldots, t_{1,n}\}\,t_2;\ E')\sigma$, and $\theta_1 = (\theta\backslash\{x \mapsto x\theta\}) \cup \{y \mapsto c\}$. From the definition of an application, $\vdash E_1\theta_1$ holds. We can observe $E \Rightarrow_{[tb]} E_1$, $\#s(E_1) = \#s(E)$ and $\#r(E_1) <_{mult} \#r(E)$, and $\sigma\theta_1 = \theta\ [var(E)]$.

[3-2] Suppose the leftmost individual of $x\theta$ is not $\Box$. Since t is well-typed and $n > 0$, we have $\#\Box(\tau(x)) > 0$. Since $x\theta$ is ground, the type consistency of θ implies $\#\Box(x\theta) > 0$. That is, $x\theta$ is non-empty. Write $x\theta$ as $l[c_1]c_2$ for some ground terms c_1 and c_2 (those may be contexts). Then s_1 must be of the form $l[s_{1,1}]s_{1,2}$. We have

$\vdash s_{1,1} = c_1\{t_{1,1}\theta, \ldots, t_{1,k}\theta\}$,
$\vdash s_{1,2} = c_2\{t_{1,k+1}\theta, \ldots, t_{1,n}\theta\}$,
$\vdash s_2 = t_2\theta$.

Lemma 4 (3) gives $L[T_1]T_2 \in \tau(x)!$ for some L, T_1, and T_2 such that $l \in L$ and $\tau(c_i) <: T_i (i = 1, 2)$. Hence, [tb] is applicable. Define $\sigma = \{x \mapsto l[y]z\}$,

$$E_1 = (s_{1,1} \approx y\{t_{1,1}, \ldots, t_{1,k}\};\ s_{1,2}s_2 \approx z\{t_{1,k}, \ldots, t_{1,n}\}t_2; E')\sigma,$$

and $\theta_1 = (\theta\backslash\{x \mapsto x\theta\}) \cup \{y \mapsto c_1, z \mapsto c_2\}$. From the definition of application $\vdash E_1\theta_1$ holds. We can observe $E \Rightarrow_{[td]} E_1$, $\#s(E_1) <_{mult} \#s(E)$, and $\sigma\theta_1 = \theta\ [var(E)]$.

Finally, we can easily check that the σ and θ_1 constructed above are all type-consistent. Hence, the proof is complete. □

Theorem 3 (completeness) For any IRE matching problem E and a substitution θ such that $\vdash E\theta$, there exists a derivation $E \Rightarrow^*_{\theta'} \varepsilon$ such that $\theta' = \theta\ [var(E)]$.

Proof: The proof is given by Noetherian induction on the ordering $\lhd$. If E is empty, the result immediately holds. Otherwise, Lemma 5 gives a derivation $E \Rightarrow_{[\alpha],\sigma} E'$ and a substitution θ_1 such that $[\alpha] \in \mathsf{TRANS}, \vdash E'\theta_1, E' \lhd E$ and $\sigma\theta_1 = \theta\ [var(E)]$. By the induction hypothesis, we obtain a derivation $E' \Rightarrow^*_{\theta'_1} \varepsilon$ such that $\theta'_1 = \theta_1\ [var(E')]$. Put $\theta' = \sigma\theta'_1$. Using Lemma 3, we obtain $\theta' = \sigma\theta_1 = \theta\ [var(E)]$. □

5. Concluding Remarks

We have presented a pattern-matching algorithm for incompletely RE-typed expressions. We also have described the algorithm as a set of a few simple transformation rules, showing its soundness and completeness. The reminder of this section discusses related work and future directions of our own work.

There is a substantial literature concerning regular tree languages in XML processing (see Refs. 27), 28)). Because we are interested in programming languages rather than schemata or query languages, we are strongly influenced by XDuce [17]. Our incomplete RE-types are considered as an extension of regular expression types [18] of XDuce with facilities for accessing contexts. This extension could affect the programming style considerably, as we have seen in the example in Section 1.

Label variables in CDuce [4] are considered as a special case of our contexts. Our contexts are encoded using label variables and (first-order) variables if the depths of hole positions in type expressions are bounded (the size of the pattern increases considerably). Since our type system allows holes to appear at any depth, this encoding is unavailable in general.

Most comparable to ours is a very recent study by Kutsia and Marin [21]. Their *context*

sequence regular matching provides a capability for both sequential and context matching, and comes with facilities of regular expression constraints. See Ref. 20) for a comprehensive study of sequential unification or matching. Their pattern matching algorithm is also given by transformation rules.

Their treatment of regular expressions , however, is rather different from ours. While we extensively use type name definitions based on regular tree grammars, they rely on neither regular tree grammars nor tree automata. Instead, they allow variables in regular expressions. Those variables are instantiated during pattern matching. Thus, their treatment is close to *hedge regular expressions* [26] (an extension of regular expressions for tree languages [11]). A benefit of our framework is the fact that it is supported by a powerful type system capable of static type checking, which consistently deals with multiple holes in contexts, whereas Kutsia and Marin consider only unary contexts.

One reason we adopted the rule-based approach familiar in unification theory is that we are aiming to extend our algorithm with facilities for unification (i.e., bidirectional pattern matching). This extension will provide a simple way of unifying several XML documents. This will be useful in designing, for example, the template engines in web application servers. Kutsia examines in Ref. 22) a few syntactic restrictions on obtaining decidability. The key for this direction will be to find syntactic restrictions appropriate for XML document processing.

We have presented our IRE pattern-matching algorithm as generally as possible; it admits *ambiguous* [16] patterns, and enumerates all solutions for them. Applying this algorithm to existing programming languages requires appropriate restrictions ensuring unique solutions. One way to do this is to introduce a matching strategy; e.g., the *first match* strategy. TRANS will provide a suitable setting for discussion of matching strategies, because various strategies (e.g., first, longest, shortest, etc.) are simply treated by controlling the selection of transformation rules; for example, the first match strategy is easily realized by imposing a "left-to-right" precedence on elements of type expansions. An important research direction is to examine appropriate strategies.

Another important research direction is to develop program transformation techniques. For functions like `uniq` in Section 1, a naive implementation will perform a large amount of redundant pattern matching repeatedly. We expect that our type system will help us to develop such techniques.

Acknowledgments The authors are grateful to the anonymous referee for helpful comments. This work is supported by Japan Society for the Promotion of Science, basic research (C) No.16500014 (Okui) and (C) No.15500014 (Suzuki).

References

1) Aho, A., Sethi, R. and Ullman, J.D.: *Compilers, Principles, Techniques, and Tools*, Addison Wesley (1988).
2) Baader, F. and Nipkow, T.: *Term Rewriting and All That*, Cambridge University Press (1999).
3) Baader, F. and Snyder, W.: Unification Theory, *Handbook of Automated Reasoning*, Vol.I, Elsevier Science, chapter 8, pp.445–532 (2001).
4) Benzaken, V., Castagna, G. and Frisch, A.: CDuce: An XML-Centric General-Purpose Language, *ACM International Conference on Functional Programming* (2003). Available from http://www.cduce.org/
5) Berry, G. and Sethi, R.: From regular expressions to deterministic automata, *Theoretical Computer Science*, Vol.48, pp.117–126 (1987).
6) Bezem, M., et al.: *Term Rewriting Systems*, Cambridge University Press (2003).
7) Bruggemann-Klein, A.: Regular expressions into finite automata, *Latin American Theoretical Informatics* (*LATIN'92*) (1992).
8) Brzozowski, J.A.: Derivatives of regular expressions, *J. ACM*, Vol.11, No.4, pp.481–494 (1964).
9) Carme, J., Gilleron, R., Lemay, A., Terlutte, A. and Tommasi, M.: Residual finite tree automata, Technical report, GRAPPA 2003 (2003).
10) Comon, H.: Completion of rewrite systems with membership constraints, Part I: Deduction rules, *Journal of Symbolic Computation*, Vol.25, pp.397–419 (1998).
11) Comon, H., et al.: Tree Automata Techniques and Applications (2002). Available from http://www.grappa.univ-lille3.fr/tata/
12) Dershowitz, N. and Manna, Z.: Proving termination with multiset orderings, *Comm. ACM*, Vol.22, No.8, pp.465–476 (1979).
13) Gapeyev, V. and Pierce, B.C.: Regular object types, *ECOOP 2003 — Object-Oriented Programming, 17th European Conference* (2003).

14) Hölldobler, S.: *Foundations of Equational Logic Programming*, Springer Verlag (1989).
15) Hosoya, H. and Pierce, B.C.: Regular expression pattern matching for XML, *Journal of Functional Programming*, Vol.13, No.6, pp.961–1004 (2002). Available from http://xduce.sourceforge.net/
16) Hosoya, H.: Regular expression pattern matching: A simpler design, Technical Report 1397, RIMS, Kyoto University (2003).
17) Hosoya, H. and Pierce, B.C.: XDuce: A typed XML processing language, *ACM Transactions on Internet Technology*, Vol.3, No.2, pp.117–148 (2003). Available from http://xduce.sourceforge.net/
18) Hosoya, H., Vouillon, J. and Pierce, B.C.: Regular expression types for XML, *ACM Transactions on Programming Languages and Systems*, Vol.3, No.2, pp.46–90 (2004). Available from http://xduce.sourceforge.net/
19) Huet, G.P.: A unification algorithm for typed λ-calculus, *Theoretical Computer Science*, Vol.1, pp.27–57 (1975).
20) Kutsia, T.: Solving and proving in equational theories with sequence variables and flexible arity symbols, Ph.D. Thesis, Johannes Kepler University (2002).
21) Kutsia, T. and Marin, M.: Can context sequence matching be used for XML querying?, *19th Internal Workshop on Unification* (*UNIF '05*), pp.77–92 (2005).
22) Kutsia, T.: Solving equations involving sequence variables and sequence functions, *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004*, pp.157–170 (2004).
23) Levin, M. and Pierce, B.J.: Typed-based optimization for regular patterns, *First International Workshop on High Performance XML Processing* (2004).
24) Martelli, A. and Montanari, U.: An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems*, Vol.4, pp.258–282 (1982).
25) Murata, M.: Hedge Automata: A formal model for XML schemata (2000). Available online: http://www.xml.gr.jp/relax/hedge_nice.html
26) Murata, M.: Extended path expressions of XML, *Proc. 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp.126–137 (2001).
27) Murata, M., Lee, D. and Mani, M.: Taxonomy of XML schema languages using formal language theory, *Extreme Markup Languages*, Montreal, Canada (2001).
28) Neven, F.: Automata theory for XML researchers, *ACM SIGMOD Records*, Vol.31, No.3, pp.39–46 (2002).
29) Ohlebusch, E.: *Advanced Topics in Term Rewriting*, Springer Verlag (2002).
30) Pair, C. and Quere, A.: Définition et etude des bilangages réguliers, *Information and Control*, Vol.13, No.6, pp.565–593 (1968).
31) Schmidt-Schauß, M.: A decision algorithm for stratified context unification, *Journal of Logic and Computation*, Vol.12, No.6, pp.929–953 (2002).
32) Schmidt-Schauß, M.: On the complexity of linear and stratified context matching problems, *Theory of Computing Systems*, Vol.37, pp.717–740 (2004).
33) Snyder, W.: *A Proof Theory for General Unification*, Birkhäuser (1991).
34) Sperberg-McQueen, C.M.: Applications of Brzozowski derivatives to XML schema processing, *Extreme Markup Languages 2005* (2005).
35) Suzuki, T. and Okui, S.: A rewrite system with incomplete regular expression type for transformations of XML documents, *IPSJ Transactions on Programming*, Vol.46, No.SIG14 (PRO 27), pp.43–54 (2005).
36) Suzuki, T. and Okui, S.: A statically typed second-order rewrite system for XML transformation, *Proc. 8th International Conference on Humans and Computers* (*HC-2005*), pp.326–331 (2005).
37) Takahashi, M.: Generalizations of regular sets and their application to a study of context-free languages, *Information and Control*, Vol.27, pp.1–36 (1975).

(Received September 22, 2005)
(Accepted December 26, 2005)

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.

Taro Suzuki was born in 1964. He received his D.Sci. degree from Tokyo University 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.