Binding-Ambiguity for Regular Expression Patterns

Haruo Hosoya

The University of Tokyo

Regular expression pattern matching

[Hosoya and Pierce, POPL01]

- a main feature of the XDuce language for XML processing
- ML-style pattern matching based on regular expressions
  - a “natural” design since XML data are typed by regular expressions
  - But the design still leaves many possibilities in how to handle ambiguity of patterns...

Regular expression pattern matching (detail)

- A pattern is a regular expression with variable annotations
- Matching yields a single binding (like in ML)

E.g.,

```
match e with    (* input value *)
  (name as n), email*, (tel? as t)
     ->  ...    (* do something with n and t *)
| ...
```

Ambiguity

In general, regular expressions can match in multiple possible ways $\Longrightarrow$ bindings may not be unique

E.g., `(email* as x), email*`

What should be the semantics for such patterns?

Our previous approach

uses built-in “first-match” rules:

> Earlier patterns have higher priority (roughly)

quite “standard” (cf. Perl, Emacs) and sometimes convenient, but other time behaves in an unintuitive way...

E.g., How does it behave? (`AnyOne` matches any label)

```
AnyOne*, (email as x), AnyOne*
```

E.g., How does it behave? (`AnyOne` matches any label)

```
AnyOne*, (email as x), AnyOne*
```

Answer: `x` is bound to the last `email`

Why?

- A repetition is actually defined by recursion: `AnyOne*` $\equiv$ `X` where

  ```
  X  =  AnyOne,X  |  ()
  ```

- The first clause `AnyOne,X` for the first pattern `AnyOne*` has the highest priority

Accurately remembering the priority rules can be a heavy burden!

Our new approach

requires that a pattern yields a unique binding, e.g.,

- `AnyOne*, (email as x), AnyOne*`

  is rejected

- We have to rewrite it to, e.g.,

  `(^email)*, (email as x), AnyOne*`

This “disambiguation” requires a bit of work, but the behavior is clearer afterward

Algorithmic issue

How can we check if a pattern always yields a unique binding?

A conventional “matching unambiguity” [Brüggemann-Klein, 93]

- can ensure unique bindings
- but is too strong — some important patterns yielding unique bindings are rejected (examples coming later)

We’ve developed two exact algorithms for binding-ambiguity based on

- a “physical” uniqueness
- a “structural” uniqueness

Outline for the rest

- Regular expression patterns and binding ambiguity
- Algorithms for checking binding ambiguity
  - Physical binding-ambiguity
  - Structural binding-ambiguity
- Related and future work

Regular Expression Patterns and Binding Ambiguity

Values

are sequences of labeled values and base values (thus representing any fragments of XML documents)

E.g.,

```
addrbook[
  person[
    name["haruo"],email["hosoya"]],
  person[
    name["pierce"],tel["987"]]]
```

Patterns

are regular expressions over labeled patterns or base types

- where variables can be added on subexpressions

E.g.,

- `person[(name[String] as n), email[String]*] as p`
- `email[String as x] | tel[String] as x`

A type is a pattern containing no variable

Linearity restriction

imposed for avoiding missing or duplicated bindings, e.g.,

- × `email[String as x] | tel[String as y]`
- × `name[String as x], email[String as x]`
- × `email[String as x]*`

Note: Linearity restriction can be a bit relaxed [CDuce, PLAN-X02]

Conventional “matching” unambiguity

[Brüggemann-Klein93], etc.

For any input value,

- there is a unique mapping from the labels in the value to the labels in the pattern

(Variables are irrelevant)

E.g.,

× `email[String]*, (email[String]* as x)`

Well-known definition. However...

There are some patterns that are matching-ambiguous, but always yield a unique binding, e.g.,

- `(email[String] as x) | (email[String] as x)`
- More important: In XDuce, we can
  - import a type `T` from a (possibly ambiguous) external schema
    - cf. RELAX NG schemas allow ambiguity
  - and use it as a part of pattern
    - e.g., `(l[T] as x)`
  - But we have no control on schemas written by other people!

Binding unambiguity (physical)

For any input value where each node is given a unique ID,

- if the pattern yields two bindings B and C, then each bound value $B(x)$ is equal to $C(x)$ respecting the nodes' IDs

✓ `(a[String] as x) | (a[String] as x)`

✓ `a[T] as x`   (where `T` is matching-ambiguous)

× `((a[] as x), a[]) | (a[], (a[] as x))`

- In either case, x is bound to `a[]` but with a different ID

Binding unambiguity (structural)

For any input value,

- if the pattern yields two bindings B and C, then each bound value $B(x)$ is structurally equal to $C(x)$

Strictly more permissive than physical binding-unambiguity

E.g.,

✓ `((a[] as x), a[]) | (a[], (a[] as x))`

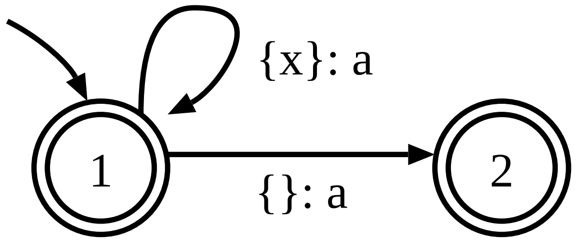
- The structural one has a simpler specification
- The physical one has a simpler algorithm (next section)
- In practice, the physical one is probably enough

Algorithm for Binding-Ambiguity

Automata with variables

Each transition is annotated with a set of variables

E.g., `(a* as x), a?` translates to



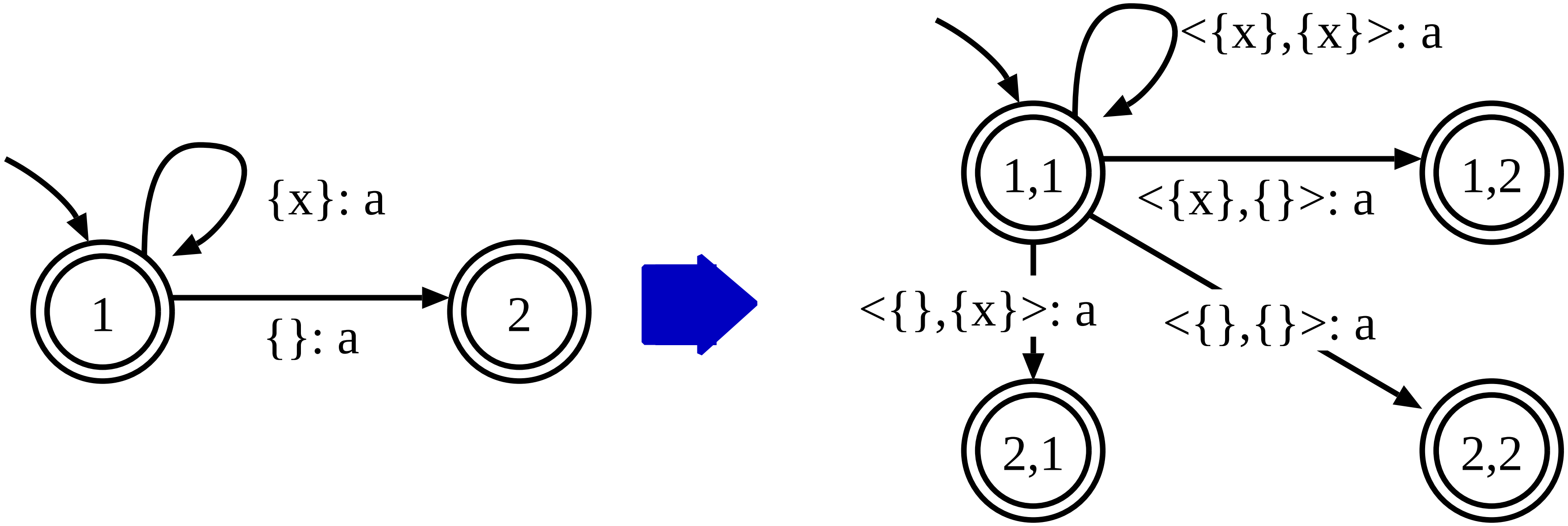
(We use string automata just for explanation)

During matching,

- we maintain a “sequence accumulator” associated to each variable
- each time we follow a transition with `x`, we append the label to the accumulator

Checking physical binding-ambiguity

1. Take the self-product of the given automaton where each transition has a pair of variables sets



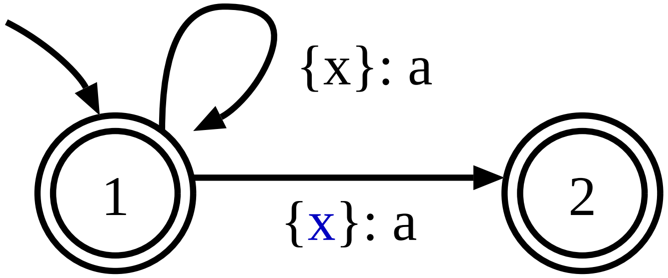
2. Eliminate useless states

3. Answer “physically binding-ambiguous” iff some transition has a pair of different variable sets

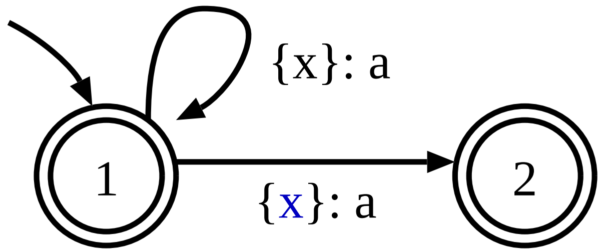
Another example (physically binding-unambiguous)

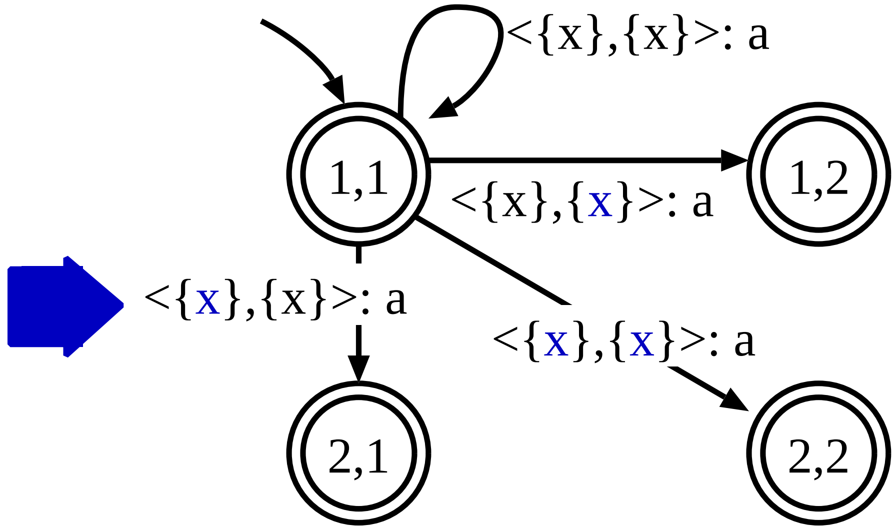
```
(a*, a?) as x
```

which translates to



Take the self-product:





Every transition has a pair of the same variable sets $\Longrightarrow$ physically binding-unambiguous

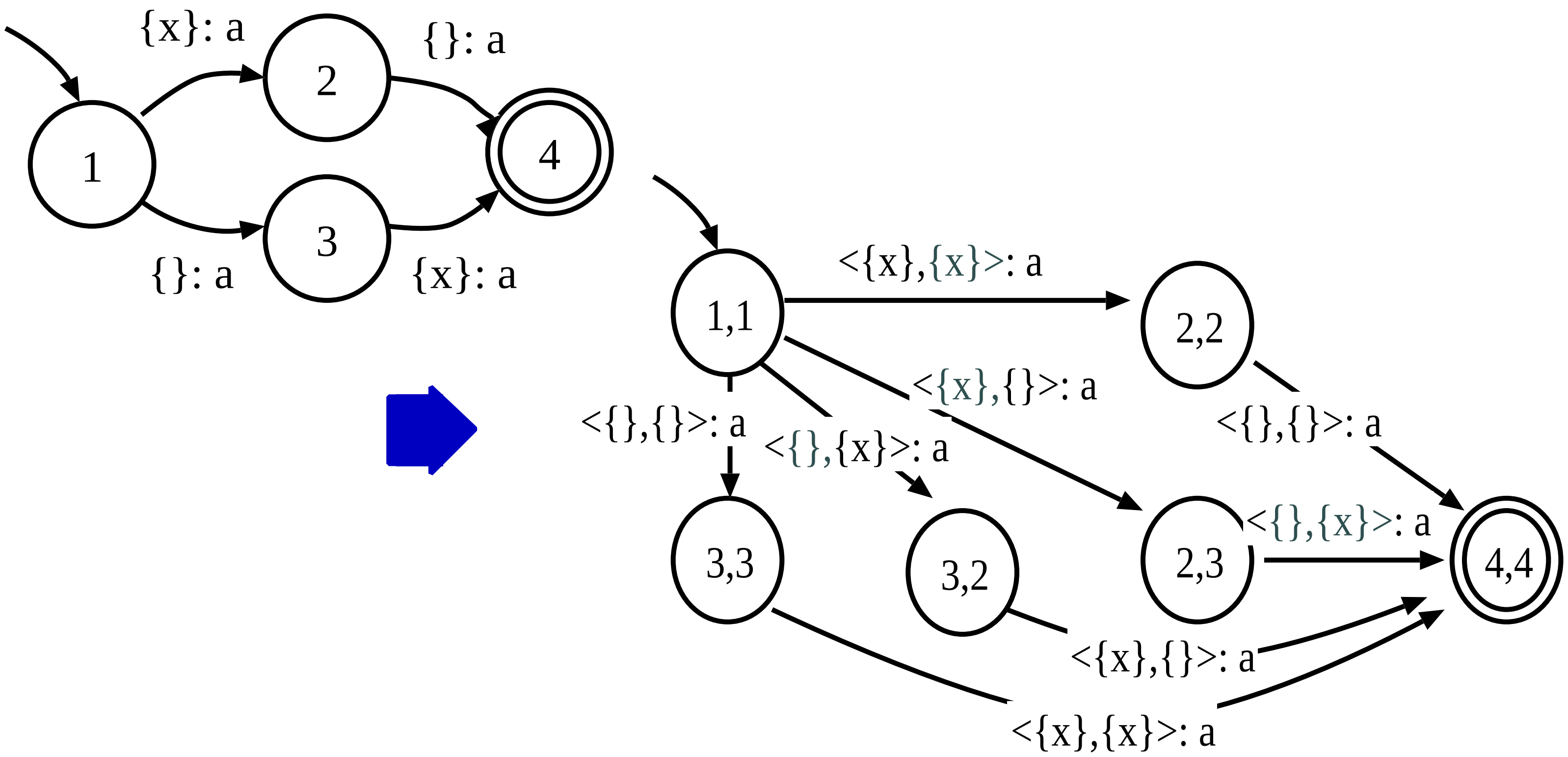
Remarks

- $O(n^2)$ complexity (since the algorithm is based on product construction)
- easily extensible to tree automata

Checking structural binding-ambiguity

(The case where there is at most one variable)

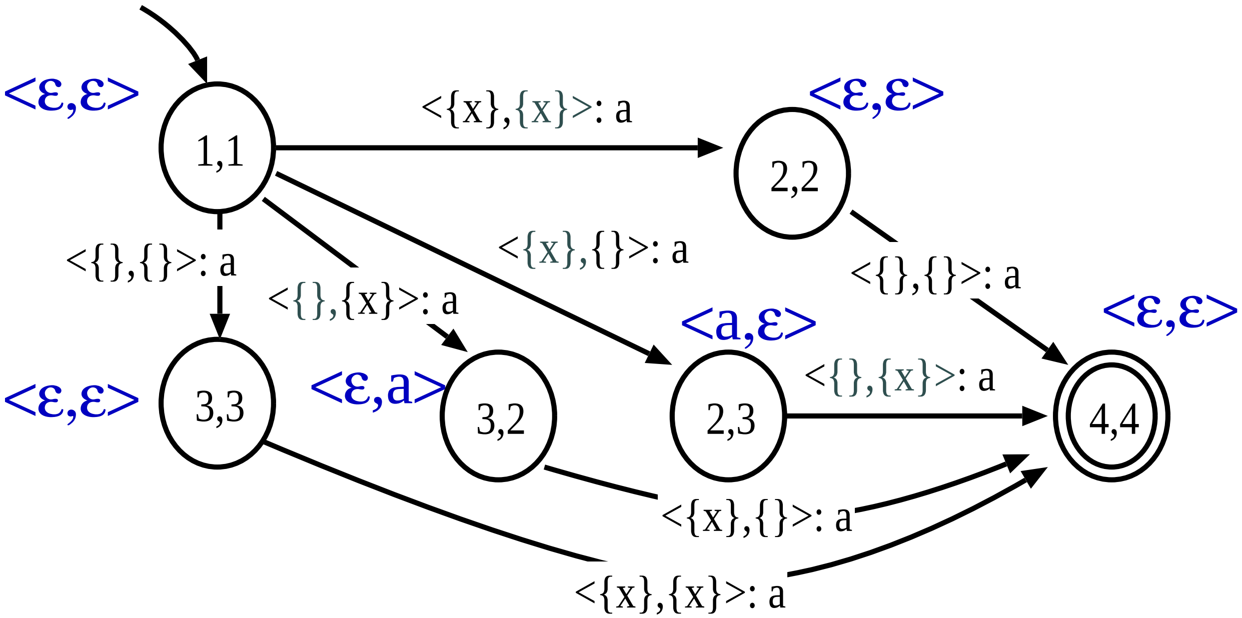
1. Take the self-product



2. Eliminate useless states

3. Associate each state q with a pair $\langle v, w\rangle$ of values such that

> any pair of values that can be captured up to q has the form $\langle uv, uw\rangle$ for some u



(There is a linear time algorithm to find such an association)

4. Answer “structurally binding-unambiguous” iff we can find such an association and it maps each final state to $\langle \epsilon, \epsilon\rangle$

Related work

- First-match regular expression patterns
  - Old XDuce [Hosoya and Pierce, POPL01]
  - CDuce [Frisch, Castagna, and Benzaken, PLAN-X02]
  - XPerl (for strings) [Tabuchi and Sumii, TIP02]
- [Vansummeren03]
  - points out that "longest-match by first-match plus recursion" (`T*` $\equiv$ `X` where `X = T,X | ()`) is not longest-match
  - proposes a further more complex "true" longest-match

- Matching ambiguity for string regular expressions [Book, Even, Gribach, and Ott, 71] [Brüggemann-Klein, 93] etc.
  - weakly unambiguous: unique mapping from value labels to expression labels
  - strongly unambiguous: unique decomposition by regular expressions
  - E.g.,
    - `a*,a*` — strongly and weakly ambiguous
    - `a**` — strongly but not weakly ambiguous
- 1-unambiguity [Brüggemann-Klein and Wood]
  - deterministic by 1-character lookahead
  - E.g., `a*,a` — 1-ambiguous

- Matching ambiguity for tree regular expressions
  - weak ambiguity [Kawaguchi, 01]
  - strong ambiguity [Hosoya, TR03]

Future work

- Extension for attribute-element constraints (RELAX-NG-style attribute types)
- Implementation in the XDuce system