Type-based Optimization for Regular Patterns

Michael Y. Levin and Benjamin C. Pierce

University of Pennsylvania

High Performance XML Workshop, 2004

What is it all about?

XDUCE [Hosoya,Pierce,Vouillon]: an experimental language with "native XML support" in the form of regular types and regular pattern matching

XTATIC project: a full-blown OO language (C♯) integrated with regular types and regular pattern matching; compiled into pure C♯

This talk: introduction to regular types and patterns; **type-based optimization technique** for pattern compilation; and its relation to compilation of XPATH

Background

Regular Types and Patterns

Key observations:

- **XML schema languages** are based on regular tree grammars (a simple extension of familiar regular expressions on strings); usually schemas are used for dynamic validation of documents
- **Regular types** are like schemas; they are used statically by typechecking to ensure a program's assumptions about XML fragments it operates on
- **Regular patterns** are like regular types; they are used dynamically to match XML fragments of certain shape and extract their subparts

XML Values

- Syntax

$$\begin{array}{lcl} \texttt{v} & ::= & \texttt{()} \\ & & \texttt{el}_1 \ldots \texttt{el}_k \\ \\ \texttt{el} & ::= & \texttt{l[v]} \end{array}$$

- Representing XML

$$\texttt{<a><c></c>} \sim \texttt{a[b[],c[]]}$$

Regular Types

`T`	::=	`()`	empty sequence
		`l[T]`	element labeled `l`
		$\texttt{T}_1\texttt{,T}_2$	concatenation
		$\texttt{T}_1\texttt{|T}_2$	alternation (union)
		`T*`	repetition
		`Any`	any sequence of trees
		`X`	type definition
`d`	::=	`def X = T`	type definition

E.g.

```
def Caption = caption[]
def Cell = td[]
def Row = tr[Cell+]
def Table = table[()|Caption, Row+]
```

Using Regular Types

Consider the following function skeleton:

```
T2 transform (T1 x) {
  ...
}
```

Typechecking ensures that `transform` is always called with arguments conforming to $\mathtt{T}_1$ and that it always returns values conforming to $\mathtt{T}_2$.

Regular Patterns

`p`	::=	`()`	empty sequence
		`l[p]`	element labeled `l`
		$\texttt{p}_1\texttt{,p}_2$	concatenation
		$\texttt{p}_1\texttt{|p}_2$	alternation (union)
		`p*`	repetition
		`Any`	any sequence of trees
		`X`	pattern name
		`p x`	binding
`d`	::=	`def X = p`	pattern definition

E.g.

```
def Caption = caption[]
def Cell = td[]
def Row = tr[Cell+]
table[Caption x, Row, Any y]
```

Source Language (XDUCE)

XDUCE contains:

- Conventional functional language elements: top-level function definitions; function calls; value constructors
- Pattern matching construct

```
match x with
  | p1 → t1
  ...
  | pk → tk
```

Target Language

- Has the same constructs as the source language
- But features only basic patterns

$$
\begin{array}{lll}
\texttt{t} & ::= & \ldots \\
 & & \texttt{case x of} \\
 & & \quad \texttt{|}\ \texttt{p}_1 \rightarrow \texttt{t}_1 \\
 & & \quad \ldots \\
 & & \quad \texttt{|}\ \texttt{p}_k \rightarrow \texttt{t}_k \\
 & & \quad \texttt{else t} \\
 & & \\
\texttt{p} & ::= & () \\
 & & \texttt{l[x],y}
\end{array}
$$

Type-based Pattern Compilation

Goals of Translation

The **goal** is to convert complicated regular patterns into a collection of nested `case` expressions each examining the tag of one element in the input

The **challenge** is to minimize the number of tests performed during pattern matching

Type-based optimization is a crucial technique that helps produce efficient and compact target language code

Motivating Example

```
Any f(Any x) =
  match x with
    | Any, a[] → 1
    | Any → 2
```

Motivating Example

```
Any f(Any x) =
  match x with
    | Any, a[] → 1
    | Any → 2
```

```
Any f(Any x) =
  case x of
    | () → 2
    | a[x],y →
      case x of
        | () →
          case y of
            | () → 1
            else f(y)
        else f(y)
    | ~[_],y → f(y)
```

Motivating Example

```
T = a[],(a[]|b[])
```

```
Any f(T x) =
  match x with
    | Any, a[] → 1
    | Any → 2
```

Motivating Example

```
T = a[],(a[]|b[])
```

```
Any f(T x) =
  match x with
    | Any, a[] → 1
    | Any → 2
```

```
Any f(T x) =
  case x of
    | ~[_],y →
      case y of
        | a[_],_ → 1
        else 2
```

Idea 1: Intersecting with Input Type

$$T = a[], (a[] | b[])$$

```
match x with                          match x with
  | Any,a[] → 1        ⟶               | a[],a[] → 1
  | Any → 2                            | a[],(a[]|b[]) → 2
```

Idea 2: Exhaustiveness Optimizations

```
match x with
  | p1 → 1          ⟶   1
  | p2 → 1
```

```
match x with                          match x with
  | a[p],p1 → 1     ⟶     | a[_],p1 → 1
  | a[p],p2 → 2           | a[_],p2 → 2
```

Idea 3: Maximal Branching Factor Heuristic

```
match x with
  | a[b[]],a[Any] → 1
  | a[Any] → 2
```

```
fun f(T x) : Any =
  case x of
    | a[z],y →
      case y of
        | a[_],_ → 1
        | () → 2
```

```
fun f(T x) : Any =
  case x of
    | a[z],y →
      case z of
        | b[_],_ →
          case y of
            | a[_],_ → 1
            | () → 2
        else 2
```

Optimizing XPATH

Idea

Apply type-based optimization to XPATH by:

- Translating an XPATH query into a regular pattern
- Performing type-based optimization
- Translating the result back into XPATH

XPATH vs. Regular Patterns

- Some XPATH queries involving backward axes cannot be directly converted into regular patterns
- XPATH patterns return multiple answers; regular patterns return `true` or `false` and possibly bind a collection of variables

XPath Example

```
def auctions = auctions[auction?]
def auction = auction[item, annot?]
def annot = annot[author, hap]

//auction[annot[hap]/author="John"]
```

Translation of the Example's Query

```
//auction[annot[hap]/author="John"]
```

$\longrightarrow$

```
def p = auction[Any,
                annot[Any, hap_auth, Any],
                Any]
      | Any, ~[Any, p, Any], Any

def hap_auth = hap[Any], Any, author["John"]
             | author["John"], Any, hap[Any]
```

Optimized Regular Pattern

```
def p' = ~[~[~[Any],              // item
           ~[~["John"],           // author
             Any],                // hap
           Any],                  // ()
         Any],                    // ()
       Any                        // ()
```

$\longrightarrow$

```
*/*[*[2]/*[1]="John"]
```

Questions?

http://www.cis.upenn.edu/~bcpierce/xtatic