Advances in Programming Languages

APL17: XML processing with CDuce

David Aspinall
(see final slide for the credits and pointers to sources)

School of Informatics
The University of Edinburgh

Friday 26th November 2010
Semester 1 Week 10

Topic: Bidirectional Programming and Text Processing

This block of lectures covers some language techniques and tools for manipulating structured data and text.

- Motivations, simple bidirectional transformations
- Boomerang and complex transformations
- XML processing with CDuce

This lecture introduces some language advances in text processing languages.

Outline

Outline

Evolution of XML processing languages

There is now a huge variety of special purpose XML processing languages, as well as language extensions and bindings to efficient libraries.

We might characterise the evolution like this:

- Stage 0: general purpose text manipulation; basic doc types
  - AWK, sed, Perl, ...
  - DTDs, validation as syntax checking
- Stage 1: abstraction via a parser and language bindings.
  - SAX, DOM, ...
- Stage 3: untyped XML-specific languages; better doc types
  - XSLT, XPath
  - XML Schema, RELAX NG, validation as type checking
- Stage 4: XML *document* types inside languages
  - Schema translators: HaXML, ...
  - Dedicated special-purpose languages: XDuce, XQuery
  - Embedded/general purpose: Xstatic, Cω, CDuce.

The CDuce Language

- Features:
  - General-purpose functional programming basis.
  - Oriented to XML processing. Embeds XML documents
  - Efficient. Also has OCaml integration **OCamlDuce**.
- Intended use:
  - Small "adapters" between different XML applications
  - Larger applications that use XML
  - Web applications and services
- Status:
  - Quality research prototype, though project wound down now.
  - Public release, maintained and packaged for Linux distributions.
  - My recommendation: try http://cduce.org/cgi-bin/cduce first.

Type-centric Design

Types are pervasive in CDuce:

- Static validation
  - E.g.: does the transformation produce valid XHTML ?
- Type-driven programming semantics
  - At the basis of the definition of patterns
  - Dynamic dispatch
  - Overloaded functions
- Type-driven compilation
  - Optimizations made possible by static types
  - Avoids unnecessary and redundant tests at runtime
  - Allows a more declarative style

Outline

XML syntax

```
<staffdb>
  <staffmember>
    <name>David Aspinall</name>
    <email>da@inf.ed.ac.uk</email>
    <office>IF 4.04A</office>
  </staffmember>
  <staffmember>
    <name>Ian Stark</name>
    <email>Ian.Stark@ed.ac.uk</email>
    <office>IF 5.04</office>
  </staffmember>
  <staffmember>
    <name>Philip Wadler</name>
    <email>wadler@inf.ed.ac.uk</email>
    <office>IF 5.31</office>
  </staffmember>
</staffdb>
```

CDuce syntax

```
let staffdb =
<staffdb>[
  <staffmember>[
    <name>"David Aspinall"
    <email>"da@inf.ed.ac.uk"
    <office>"IF 4.04A"]
  <staffmember>[
    <name>"Ian Stark"
    <email>"Ian.Stark@ed.ac.uk"
    <office>"IF 5.04"]
  <staffmember>[
    <name>"Philip Wadler"
    <email>"wadler@inf.ed.ac.uk"
    <office>"IF 5.31"]
]
```

CDuce Types

We can define a CDuce type a bit like a DTD or XML Schema:

```
type StaffDB = <staffdb>[StaffMember*]
type StaffMember = <staffmember>[Name Email Office]
type Name = <name>[ PCDATA ]
type Echar = 'a'--'z' | 'A'--'Z' | '0'--'9' | '_' | '.'
type Email = <email>[ Echar+ '@' Echar+ ]
type Office = <office>[ PCDATA ]
```

Using these types we can *validate* the document given before, simply by ascribing its type in the declaration:

```
let staffdb : StaffDB =
<staffdb>[
  <staffmember>[
...
```

CDuce Processing

```
 let staffdb : StaffDB =
<staffdb>[
  <staffmember>[
    <name>"David Aspinall"
    <email>"da@inf.ed.ac.uk"
    <office>"IF 4.04A"]
  ...
]

let staffers : [String*] =
  match staffdb with <staffdb>mems ->
      (map mems with (<_>[<_>n _ _])->n)
```

```
val staffers : [ String* ] =
[ "David Aspinall" "Ian Stark" "Philip Wadler" ]
```

Outline

Type-safe XML Processing

XML has evolved into a text-based general purpose data representation language, used for storing and transmitting everything from small web pages to enormous databases.

Roughly, two kinds of tasks:

transforming changing XML from one format to another, inc. non-XML

querying searching and gathering information from an XML document

Both activities require having prescribed document formats, which may be partly or wholly specified by some form of typing for documents.

Regular Expression Types

Regular expression types were pioneered in XDuce, an ancestor of CDuce.

We have already seen these in Boomerang.

The idea is to introduce *subtypes* of the type of strings, defined by regular expressions. The values of a regular expression R type are exactly the set of strings matching R.

$$\mathrm{R} \quad ::= \quad \emptyset \quad | \quad s \quad | \quad \mathrm{R}|\mathrm{R} \quad | \quad \mathrm{R}*$$

CDuce takes this idea and runs with it, starting with basic set-theoretic type constructors and recursion. Types are treated as flexibly as possible and type inference as precisely as possible.

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- CDuce has a rich type structure built with simple combinators
- Many types, included those for XML, are encoded.
- Types stand for sets of *values* (i.e., fully-evaluated expressions).
- A sophisticated type inference algorithm works with rich equivalences and many subtyping relations derived from the set interpretation.

CDuce Types

t	$::=$	Int | Char | Atom |	*type constants*
		Any | Empty	*everything/nothing*
	|	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	|	(t_1, t_2) | $(t_1 \to t_2)$	*products and functions*
	|	$t_1 \& t_2$ | $t_1 \mid t_2$ | $t_1 \setminus t_2$	*set combinations*
	|	v	*singletons*
	|	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	|	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Int is arbitrary precision, Char set of Unicode
- Can write integer or character ranges as $i--j$.
- Atoms are symbolic constants (like symbols in lisp)
- For example, `'nil`

CDuce Types

t	$::=$	Int $\mid$ Char $\mid$ Atom $\mid$	*type constants*
	$\mid$	Any $\mid$ Empty	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Any is the universal type, any value belongs
- Empty is the empty type, no value belongs
- These are used to define richer types or constraints for patterns

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Record values are written $\{a_1 = v_1; \ldots; a_1 = v_n\}$
- Records are used to define attribute lists

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- By default record types are *open* (match records with more fields)
- Closed records are allowed too: $\{|a_1 = t_1; \ldots; a_1 = t_n|\}$.

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Pairs are written (v_1, v_2).
- Longer tuples and sequences are encoded, Lisp-style.
- For example, $[v_1\ v_2\ v_3]$ means $(v_1, (v_2, (v_3, \text{'nil})))$.

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	$T \text{ where } T_1 = t_1 \text{ and } \cdots \text{ and } T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Function types are used as *interfaces* for function declarations.
- A simple function declaration has the form:

```
let foo (t->s) x -> e
```

CDuce Types

t	$::=$	Int $\mid$ Char $\mid$ Atom $\mid$	*type constants*
	$\mid$	Any $\mid$ Empty	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \backslash t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- The general function declaration has the form:

  let foo (t_1->s_1;...;t_n->s_n) | p_1->e_1 | ... p_m->e_m

  where $p_1 \ldots p_m$ are *patterns*.

CDuce Types

t	$::=$	Int $\mid$ Char $\mid$ Atom $\mid$	*type constants*
	$\mid$	Any $\mid$ Empty	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Boolean connectives: intersection $t_1 \& t_2$, union $t_1 \mid t_2$ and difference $t_1 \setminus t_2$
- These have the expected set-theoretic semantics.
- Useful for overloading, pattern matching, precise typing

CDuce Types

t	$::=$	Int $\mid$ Char $\mid$ Atom $\mid$	*type constants*
	$\mid$	Any $\mid$ Empty	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- A value used v in place of a type stands for the single-element type whose unique element is v.

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- Sequences $[t*]$ are defined with recursive types, e.g.:

$$[\mathsf{Char}*] \equiv (T \text{ where } T = (\mathsf{Char}, T) \mid \mathsf{nil})$$

- Strings are encoded as $[\mathsf{Char}*]$, like in Haskell.
- This interpretation matches XML parsers well.

CDuce Types

t	$::=$	Int $\mid$ Char $\mid$ Atom $\mid$	*type constants*
	$\mid$	Any $\mid$ Empty	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- XML fragments have a tag, attribute list and child elements
- This is actually a shorthand, again...

CDuce Types

t	$::=$	$\mathsf{Int} \mid \mathsf{Char} \mid \mathsf{Atom} \mid$	*type constants*
	$\mid$	$\mathsf{Any} \mid \mathsf{Empty}$	*everything/nothing*
	$\mid$	$\{a_1 = t_1; \ldots; a_n = t_n\}$	*records*
	$\mid$	$(t_1, t_2) \mid (t_1 \to t_2)$	*products and functions*
	$\mid$	$t_1 \& t_2 \mid t_1 \mid t_2 \mid t_1 \setminus t_2$	*set combinations*
	$\mid$	v	*singletons*
	$\mid$	T where $T_1 = t_1$ and $\cdots$ and $T_n = t_n$	*recursive types*
	$\mid$	$\langle t_1\ t_2 \rangle t_3$	*XML: tags, attrs, elts*

- For example: **type** Book = <book>[Title (Author+|Editor+) Price?] is encoded as

$$\begin{array}{lcl} \mathsf{Book} & = & (\text{'book}, (\mathsf{Title}, X \mid Y)) \\ X & = & (\mathsf{Author}, X \mid (\mathsf{Price}, \text{'nil}) \mid \text{'nil}) \\ Y & = & (\mathsf{Editor}, Y \mid (\mathsf{Price}, \text{'nil}) \mid \text{'nil}) \end{array}$$

From types to patterns

Conventional idea: patterns are values with capture variables, wildcards, constants.

New idea: Patterns = Types + Capture Variables

```
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
    | ('nil, n) -> n
    | ((_,t), n) -> length(t, n+1)
```

- Same syntax for types as for values (s, t) not $s \times t$
- Values stand for singleton types (e.g., nil)
- Wildcard: _ synonym of Any

Why?

From types to patterns

Conventional idea: patterns are values with capture variables, wildcards, constants.

New idea: Patterns = Types + Capture Variables

```
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
    | ('nil, n) -> n
    | ((_,t), n) -> length(t, n+1)
```

- Same syntax for types as for values (s, t) not $s \times t$
- Values stand for singleton types (e.g., nil)
- Wildcard: _ synonym of Any

Why? Natural simplification: fewer concepts. Execution model based on pattern matching and grammars defined by type language.

Rich patterns for XML structure

Suppose an XML type:

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

Then we can pattern match against *sequences*:

```
match bibs with
  <bib>[(x::<book year="1990">[ * Publisher\"ACM"] | )*] -> x
```

This binds x to the sequence of books published in 1990 from publishers other than ACM.

Advanced constructs: map and transforms

CDuce has built-in map, transform (map+flatten) and xtransform (tree recursion) operations.

```
let bold (x:[Xhtml]):[Xhtml]=
  xtransform x with <a (y)>t -> [<a (y)>[<b> t]]
```

This emboldens all hyper-links in an XHTML document.

The user could write these as higher-order functions in the language, but the built-ins have more accurate typings than user-defined versions could. For example, by understanding sequences, result types like C^*D^* are possible from argument types A^*B^* and map operations $A \rightarrow C$ and $B - D$.

Advanced constructs: querying

SQL-like queries using a pattern-based query sub-language.

Contents of `bstore1.example.com/bib.xml`:

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison–Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison–Wesley</publisher>
        <price>65.95</price>
    </book>
    ...
```

Advanced constructs: querying

SQL-like queries using a pattern-based query sub-language.

Contents of `http://bstore2.example.com/reviews.xml`:

```
<reviews>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review> A very good discussion of semi–structured database
                 systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
                A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
   ...
```

Advanced constructs: querying

SQL-like queries using a pattern-based query sub-language.

In XQuery:

```
<books–with–prices>
  {
    for $b in doc("http://bstore1.example.com/bib.xml")//book,
        $a in doc("http://bstore2.example.com/reviews.xml")//entry
    where $b/title = $a/title
    return
        <book–with–prices>
            { $b/title }
            <price–bstore2>{ $a/price/text() }</price–bstore2>
            <price–bstore1>{ $b/price/text() }</price–bstore1>
        </book–with–prices>
  }
</books–with–prices>
```

Advanced constructs: querying

SQL-like queries using a pattern-based query sub-language.

In CDuce:

```
<books−with−prices>
select <book−with−price>[t1
                         <price−bstore2>p2
                         <price−bstore1>p1 ]
from <bib>[b::Book∗] in [bstore1],
     <book>[t1 & Title _∗ <price>p1] in b
     <reviews>[e::Entry∗] in [bstore2],
     <entry>[t2 & Title <price>p2; _] in e
where t1=t2
```

See XQuery's XML Query Use Case examples, Q5

Outline

Online Demo

Go here: http://cduce.org/cgi-bin/cduce

Try these too: http://cduce.org/demo.html

Outline

Summary

XML processing with CDuce

- A general purpose language designed for XML processing
- Functional, with a very rich type/subtyping structure
- Idea: Patterns = Types + Capture Variables
- Patterns used to drive evaluation, further language constructs

Homework

- Visit http://www.cduce.org and try the tutorial, then the sample problems.

References

See http://www.cduce.org/papers.html for a list of sources.

Some slides were based on Giuseppe Castagna's invited talk *CDuce, an XML Processing Programming Language from Theory to Practice* at at SBLP 2007: The 11th Brazilian Symposium on Programming Languages Symposium on Programming Languages.