CDuce: an XML-Centric Language

Presenter: Walter Chang

Overview

- What is CDuce?
- CDuce and XML
- A CDuce Tutorial
- CDuce Language Features
- CDuce Type System
- Other Concerns
- Discussion

What is CDuce?

- Strongly typed
- Functional (ML-Like)
- Direct syntactic support for XML
- More like embedding ML into XML than embedding XML into ML...

CDuce: What is it good for?

- Small adapters between XML applications
- Larger XML-oriented applications
- Web Applications
- Web Services

So sayeth the authors at http://www.cduce.org

Is there anything else?

What does CDuce's XML look like?

```
<?xml version="1.0"?>
<parentbook>
  <person gender="F">
    <name>Clara</name>
    <children>
      <person gender="M">
        <name>Pål André</name>
        <children/>
      </person>
    </children>
    <email>clara@lri.fr</email>
    <tel>314-1592654</tel>
  </person>
  <person gender="M">
    <name> Bob </name>
...snipped for space
  </person>
</parentbook>
```

```
let parents : ParentBook =
<parentbook>[
  <person gender="F">[
    <name>"Clara"
    <children>[
      <person gender="M">[
        <name>['Pål ' 'André']
        <children>[]
      ]
    ]
    <email>['clara@lri.fr']
    <tel>"314-1592654"
  ]
  <person gender="M">[
    <name>"Bob"
...snipped for space
  ]
]
```

Gosh, that's just like XML!

The XML...

```
<tag>some string</tag>

<tag>
  child1
  child2 ...
</tag>

<tag property=”value”>...
```

...becomes CDuce

```
<tag>”some string”

<tag>[
  child1
  child2 ...
]

<tag property=”value”>...
```

Question: If the conversion is so trivial, why not just use XML syntax?

What was that `parents : ParentBook` thing on the last slide? It isn't in the XML!

We Have Types

```
(* a ParentBook contains zero or more Persons *)
type ParentBook = <parentbook>[Person*]

(* a Person has a gender, which is either “M” or “F”,
   and contains a name, children, and possibly
   multiple phone numbers or email addresses *)
type Person = <person gender = ”M” | ”F”>[
  Name Children (Tel | Email)*]

(* a Name contains some data *)
type Name = <name>[PCDATA]

(* Children contains zero or more Persons *)
type Children = <children>[Person*]

(* a phone is one or more digits, an optional
   hyphen, and one or more digits *)
type Phone = <phone kind=?”home”|”work”>
  ['0'--'9'+ '-'? '0'--'9'+]
```

Your First Function

```
let names (ParentBook -> [Name*])
  <parentbook>x -> (map x with <person>[n _*] -> n)
```

- `names` takes a `ParentBook` and returns zero or more `Name`s
- `<parentbook>x` matches every element contained by the `<parentbook>`
- `map x with ...` performs an action on each element in the parents book
- The `n` in `[n _*]` matches the first element in the person (which is the name)
- The `_*` in `[n _*]` matches all other elements, and discards them

Your Second Function

```
let names (ParentBook -> [Name*])
  <parentbook>x ->
   (transform x with
     <person>[n <children>[Person Person]_*] -> n)
```

- `transform` will filter out anything that does not match its pattern
- `n` is bound to the first element (name)
- The pattern requires that `<children>` be present with exactly two persons
- This will return all the names of people who have exactly two children
- Regular Expression patterns work like you think they do

Function Overloading

```
let add ( (Int,Int)->Int ; (String,String)->String )
    | (x & Int, y & Int) -> x + y
    | (x & String, y & String) -> x @ y
```

- `add` is a function of type `(Int*Int)->Int` or `(String*String)->String`
- The body of `add` has an arm for each possible type of `add`
- `add` will add the arguments (if they are of type `Int`), or concatenate the arguments (if they are of type `String`)

This is actually pretty powerful...

A Complex Example

```
type Person   = FPerson | MPerson
type FPerson  = <person gender = "F">[ Name Children ]
type MPerson  = <person gender = "M">[ Name Children ]
type Children = <children>[ Person* ]
type Name     = <name>[ PCDATA ]

type Man       = <man name=String>[ Sons Daughters ]
type Woman     = <woman name=String>[ Sons Daughters ]
type Sons      = <sons>[ Man* ]
type Daughters = <daughters>[ Woman* ]

let fun split (MPerson -> Man ; FPerson -> Woman)
  <person gender=g>[ <name>n <children>[(mc::MPerson | fc::FPerson)*] ] ->
  (* the above pattern collects all the MPerson in mc,
     and all the FPerson in fc *)
     let tag = match g with "F" -> `woman | "M" -> `man in
     let s = map mc with x -> split x in
     let d = map fc with x -> split x in
     <(tag) name=n>[ <sons>s  <daughters>d ] ;;
```

A Closer Look

```
let fun split (MPerson -> Man ; FPerson -> Woman)
  <person gender=g>[ <name>n <children>[(mc::MPerson | fc::FPerson)*] ] ->
  (* the above pattern collects all the MPerson in mc,
     and all the FPerson in fc *)
     let tag = match g with "F" -> `woman | "M" -> `man in
     let s = map mc with x -> split x in
     let d = map fc with x -> split x in
     <(tag) name=n>[ <sons>s  <daughters>d ] ;;
```

- All the `MPerson`s accumulate in `mc`, and all the `FPerson`s accumulate in `fc`
- `tag` takes on the (symbolic) values `` `woman `` or `` `man `` depending on whether it saw “`F`” or “`M`”
- We map `mc` and `fc` over the split of the children
- We build either a `<man>` or a `<woman>`, with `<sons>` and `<daughters>` as appropriate
- Observe that we can compute on tags!

Higher-Order Functions

```
type f = String -> Bool
let loop (re : regexp, k : f) : f = fun (s : String) : Bool =
match re with
| <chr> p -> (match s with (c,s) -> (c = p) && (k s) | _ -> `false)
| <seq> (r1,r2) -> loop (r1, (loop (r2,k))) s
| <alt> (r1,r2) -> loop (r1,k) s || loop (r2,k) s
| <star> r -> loop (r,(loop (re,k))) s || k s

let accept (re : regexp) : f =
  loop (re, fun (String -> Bool) [] -> `true | _ -> `false)
```

- `loop` takes in a function of type `f` (String -> Bool)
- `k` can be called as any other function, and passed into other functions
- Anonymous non-recursive functions are declared with the same syntax, but without a function name (see `accept`)

Walking and Changing XML

```
type HTMLContents = <b>[HTMLContents*] |
  <p>[HTMLContents*] | <em>[HTMLContents*] | ...

let em2it (HTMLContents -> HTMLContents)
      <em>foo -> <it>foo
    | x -> x

let walk_postorder (f: HTMLContents -> HTMLContents,
                    h: HTMLContents) : HTMLContents =
  f(match h with
  | <(x)>y -> <(x)>
                (map y with z -> walk_postorder(f, z))
  | x -> x)
in
walk (em2it, my_html_contents)
```

This sort of general mechanism can fake replacement-in-place of subtrees a la XSLT...

Miscellaneous Language Features

- The usual arithmetic and boolean operators
- XML Namespace support (not discussed in paper)
- Tuples
- Sequences (you've seen them: tags have sequences of elements...)
- Records (which are used in XML attributes)
- Reference type and imperative assignment (not discussed in paper)

This is a general-purpose language, not just a query language.
Are we missing anything?

Type System Overview

- CDuce is designed around the types
- Pattern Matching seen as dynamic dispatch on types with extraction (claimed to be more powerful than dynamic dispatch in OO languages)
- Type correctness of CDuce transformations can be checked statically
- Exact type inference: the typing algorithm can find exactly the set of capturable values
- A compiler is mentioned

CDuce and DTD checking

```
<!ELEMENT person (name, children)>
<!ELEMENT children (person+)>
<!ELEMENT name (#PCDATA)>
```

Observe that no actual document of this DTD can exist: expansion would result in an infinite tree.

We can declare this in a straightforward manner:

```
type Person = <person>[ Name Children ]
type Children = <children>[ Person+ ]
type Name = <name>[ PCDATA ]
```

What do you think will happen?

CDuce and DTD checking, continued

Actual result from CDuce online demo:

```
Warning at chars 57-76:
type Children = <children>[ Person+ ]
This definition yields an empty type for Children
Warning at chars 14-39:
type Person = <person>[ Name Children ]
This definition yields an empty type for Person

Ok.
```

The paper refers you to their paper on Semantic Subtyping for a more theroetical discussion of the “magic” behind their type system

Magic, eh?

- CDuce's type system is theoretically built around the set-theoretic interpretation of types as sets of values
- Sound and complete (with respect to set inclusion)
- More powerful than most static type systems, but at a price
- Typing CDuce programs is theoretically complex: “the subtyping relation itself is already exponential...”

...but is that so bad?

Implementation Details

- Type checker: mixed top-down and bottom-up; propagates constraints (with efficient local solver for monotonic boolean constraints)
- Type-driven compilation (details forthcoming in another paper)
- Pattern matching uses “a new kind of tree automata”
- Other minor optimizations (lazy concatenation, etc)
- Good performance (typically better than XSLT)
- Not very sensitive to hand-optimization (due to type-driven compilation)

Conclusion

- CDuce is a full-featured language
- CDuce allows for very natural expression of XML and XML transformations
- CDuce has a very rich and powerful type system
- CDuce is statically checked
- CDuce has never been used for large programs

Discussion

- What features should an XML-centric language have?
- How important is static checking and performance?
- Is this the right approach? Do XML-centric languages have a place, or is extending a general-purpose language preferable?