Regular Expression Types and Patterns in CDuce

Advances in Programming Languages

Paul McEwan (0452900)

14/03/2008

Abstract

This report examines the CDuce language, a typed functional programming language designed for general purpose programming. Unlike other functional programming languages, CDuce incorporates native support for XML documents in the language. This report looks at the language, related work and then at the use of regular expression types and patterns. Specifically, how these particular types and patterns are used to query/manipulate the XML data, as well as allow static checking by the compiler that the XML data used is always valid.

1. Introduction

The CDuce language is a functional, typed programming language allowing the creation of general purpose programs. The key difference between CDuce and other typed, functional programming languages (such as Haskell and ML) is that it was designed to be used with XML from the start. The language has features included that allow the programer to manipulate and query XML trees directly in the code, instead of using additional tree parsers (such as Document Object Model (DOM) parsers). The language allows for XML files to be read in or created directly in the code and be exported back to a file. The XML handled by programs written in CDuce is guaranteed to be valid (both well-formed/syntactically correct and corresponding to a specific structure).

Of particular interest, the CDuce language allows the inclusion of regular expressions when defining types and patterns. The use of these regular expressions allows the programmer to not only enquire and alter the XML, it also allows the compiler to perform checks statically on the code to ensure the XML is valid. This report first examines the CDuce language at a high level (§2), then a look at some related work (including languages which inspired the creation of CDuce) (§3). The use of regular expressions in patterns and types will then be examined (§4) with an example program created using these features of the language (§5). Finally, the report will be concluded by looking back at the use of regular expressions in the types and patterns of the CDuce language (§6).

2. **CDuce**

The paper [CDuceXCGPL] presents CDuce in detail and is an ideal resource to use to understand the language. Here, points raised in the paper shall be summarised in order to present a general introduction to the CDuce language: what it is, how it came about and how it works.

As stated previously, the CDuce language is a typed functional programming language designed to allow the processing of XML data directly in the language while still being a “general purpose” language (i.e. not specific to XML processing but allowing programs to be created that include the functionality). The CDuce project was an off-shoot extension of the XDuce language, but was designed to be less “XML-centric”. To this effect, the CDuce language extends upon XDuce by addressing - what the paper referred to as - limitations in three areas:

- *Type System*
  The XDuce type system allowed the user to create types specific to dealing with XML data, which included having “regular expression types” and “type-based patterns”. The use of the

former will be discussed in (§4); the latter allows the programmer to query and manipulate the XML. However, the language only contained these XML-specific types and therefore needed to be linked into a program written in a different high-level language in order to create more general applications with the XML features.

CDuce was designed to still be geared towards the development of XML programs. However, to allow more general programs to be created while still utilising the XML types created as part of XDuce, CDuce includes additional types found in most functional languages, including:

- *"Arrow Types (T→T)"* - Function type
- *"Boolean Types (T&T, T|T, T\T)"*
- *"Records ({l(T)})"*

- *Language Design*
  CDuce was designed to include features that would be useful for both XML and general programs. These include (from [CDuceXCGPL]) having "overloaded functions" and the ability to iterate over sequences, as well as other extensions.

- *Run-Time System*
  The CDuce language was designed to allow programs created with it to execute in the most efficient manner possible. This involved the utilisation of a "new deterministic tree automata", as documented in [CDuceXCGPL].

Further information about the structure of the language can be found in the paper. The rest of this section will examine how the CDuce language represents the XML data and the methods presented to manipulate and query that data.

The XML type structure in CDuce has the following format: "*<T Y> [Z]*". The *T* specifies to the name of the tag the XML element must have; *Y* represents the attributes (zero, one or more) that the element may or must have; *Z* specifies the children/content for this particular XML element. With this type feature, it is possible to represent an XML file inside the program by creating a variable with this type. To have a specific XML structure, the programmer defines new types using the XML type structure corresponding to the desired XML structure (similar to creating a DTD[1] for the XML file[2]). The figure below better illustrate this idea[3]:

```
type Car = <car> [MOT ]
type MOT = <mot> [<passed> String | String ]
```

Here, two types are created to correspond to an XML structure. Type *Car* represents the root element of the XML document, and any variables created to represent an XML file with this structure will have this type. The *Car* type has a tag with the name "car" and no attributes. The contents of the element are contained in the square brackets (referred to as the *sequence*); here, the

[1] For more information about DTDs (Document Type Definition) please visit http://www.w3schools.com/dtd/dtd_intro.asp

[2] A CDuce program *dtd2Cduce* is available that takes a DTD and creates the corresponding CDuce types

[3] Inspired by the example in [OCCDuceT]

sequence contains just one item of type *MOT*. This means that the root element of the XML file has just one child, the format of which depends on the type *MOT* structure.

The *MOT* type has a tag with name "mot" and the sequence identifies that the child may take one of two forms: a tag "<passed>" which has a child of type *String* (when it passed); or just a *String* (to say it has not been taken). The String type is simply represented in CDuce as a list of zero or more characters, and can be used interchangeably with the *PCDATA* type (corresponding to the PCDATA item in XML).

The two figures below represent an XML document corresponding to this structure. The one on the left is written directly into CDuce, while the right illustrates the XML file representation:

```
let doc : Car =
    <car> [
      <mot> [
        <passed> "June 2005" ] ]
```

```
<car>
    <mot>
    <passed> June 2005 </passed>
</mot></car>
```

In CDuce, the variable (identified by the use of the *let* keyword) "doc" contains the XML data. Looking at the left figure, the declaration of the XML document does not include any closing tags. In CDuce, the closing tag is inferred by knowledge that everything within the square brackets is the contents of the XML element. When a variable is declared with the root element type (or indeed, any of the other types for the XML document) the compiler statically performs a pattern matching check to make sure that the structure corresponds to the given types (more on this in §4).

CDuce not only provides a means by which to represent XML data within the program, it also allows the programmer to have the XML data read in from, and written out to, an XML file. The programmer creates a variable of the root element type as illustrated in the left hand figure above. The command "load_xml" can then be used to load in an XML file and - provided it corresponds to the structure defined by the types - store it in the variable[4]. The user can also write the XML data to a valid XML file by using the "print_xml" command (which converts the XML data into a String in the standard XML format) and the "dump_to_file" command (which takes the name of the file to write to, a String containing the data to write and creates the file).

Once the XML data has been read into the program and stored into a variable of the root element type, the programmer can use it to query and manipulate the data.

To query the XML data, the programmer can create variables or functions, whose value or definition involves pattern matching using the XML document variable. The follow operations are provided to allow querying:

- *"/"*
  The "/" operator allows the programmer to traverse the XML tree structure of the document variable, allowing access to the children and grandchildren. The figure on the right illustrates a variable whose value is

```
let cString =
    [doc ]/<mot> _/<passed>
_ /_

>> val cString : String =
                 "June 2005"
```

[4] Technically, the variable the XML is loaded to is actually of type "Any". The variable has to be cast to the type of the root element using the ":?" notation, which states that it should be cast to this type if it conforms to the structure pattern.

determined by looking at the document, seeing if the root node has a “<passed>’ tag as grandchild and appending the child to the variable if it exists. The operations result is presented on the right of the “>>”. Note here that, not only does it correctly assign the value “June 2005”, the compiler is able to infer the type of the variable as this is the type of the “<passed>” child. The “_” represents the “Any” type and is used in the example to identify the children of the XML element.

- *“match ... with”*
  This is known as the “fundamental operation” in the CDuce language [CDuceSE]. The role of the operation is to actually perform pattern matching. Given an expression, the operation compares the expression with a list of patterns. When the first pattern is found that matches the expression, a corresponding expression is fired. An example of this operation is presented to the right. Here, since the sequence contains three Strings (“car”, “bus” and “bike”), the second pattern is matched as the first requires four elements. The sub-sequence of the first two elements is returned. The “::” operand is discussed later.

```
match ["car" "bus" "bike"] with
      [_ x :: (_ _ _) ] -> x
    | [x :: (_ _) _] -> x

>> [['car'] ['bus']] = ["car" "bus"]
```

- *“map ... with”*
  The “map … with” operation is a basic pattern matching operation that attempts to match the expression (converted into a sequence) with a list of patterns. A sequence is returned which attempts to map each item input sequence with the expression in the matched patterns [CDuceSE]. An example of it’s use is given to the right. In the example, the function takes in a variable of the root element type, extracts the child information and then maps it with the structure of a “<mot>” tag with either a “<passed>” tag child and string grandchild or a string child. The String child/grandchild in a list is then returned. Using the function with the XML document variable produces a list containing just the“June 2005” String.

```
let rootChildren ( Car -> [String])
  <car> x -> map x with
    <mot>[ (<passed> s) | (s) ] -> s

let rC = rootChildren doc

>> val rC : [String] = ["June 2005"]
```

- *“transform ... with”*
  The “transform … with” operation is similar to the “map … with” operation (and indeed can be created using this operator [CDuceSE]). The essential difference is the returned sequence does not have to be the same length as the input sequence.
  The example to the right illustrates the use of the transform operation to obtain the String item associated with either the “<passed>” element’s child or the “<mot>” element’s child.

```
let rootChildren ( Car -> String )
 <car> x -> transform x with
    <mot>[(<passed> s) | (s) ] -> s

let rC = rootChildren doc

>> val rC : String = "June 2005"
```

- *“&”*
  This operator is a Boolean connective that allows the binding of two patterns or types

```
match ["car" "bus" "bike"] with
      [_ x & (_) _] -> x

>> ['bus'] = "bus"
```

together[5]. The most common use for the operator is to append the result of one pattern to another. In the example on the left, the "&" operator is used to map the "x" pattern with the pattern of second element of the list ("bus"). The "x" is an empty variable and matches anything, hence why the pattern is matched against it.

- "::"
  This final operator is similar to the "&" operator. However, where the previous operator only works with single patterns, the "::" operator works on sub-sequences. As before, the most common usage in CDuce is to use the operator to bind a sub-sequence to a variable. The example on the right is a modified form of the previous example using this operator. Here, the "x" pattern is bound to the sub-sequence of the list that includes the final two elements ("bus" and "bike").

```
match ["car" "bus" "bike"] with
      [ _  x::(_ _)] -> x

>> [['bus']['bike']] = ["bus" "bike"]
```

Having looked at ways by which to query the XML data in CDuce, the final area to examine is ways in which the programmer may manipulate and alter the data in the XML file. The operators defined already may be used to produce these results, but the key is how they are used. The following example illustrates how to manipulate XML data. Imagine the following documents have been declared:

```
let doc : Car =
   <car> [
        <mot> [
          <passed> "June 2005"]]
```

```
let doc2 : Car =
   <car> [
        <mot> [
          "Not Taken"]
```

A programmer may wish to manipulate the "mot" element, changing the contents to identify the car has passed it's MOT and when (or, updating the last date when the car passed it's MOT). The following function can be created:

```
let updateMOTPass ( oDoc : Car ) (nDate : String) : Car =
   match oDoc with
        <car>[<mot> [<passed> _ ]] -> <car>[<mot>[<passed> nDate]]
      | <car>[<mot> [_ ]] -> <car>[<mot>[<passed> nDate]]
```

The function takes an XML document (variable with root element type), a string representing the new passed date and returns an item of the root element type. The function then pattern matches to decide how to update: if the car previously passed an MOT then the string child of "<passed>" is updated; else the "<mot>" element has a "<passed>" element added containing the new passed

[5] CDuce also contains the Boolean connectives "|" and "\". "|" is the disjunction connective: when used with types accepts the values that accept both types; when used with patterns accepts value of either the left or right hand pattern (if value is matched by both, the value is matched to the left pattern). "\" is known as the difference connective and allows the left hand side to be either a pattern or a type but the right hand side must be a type. [CDuceSTP]

date[6]. Since the function returns an item of the root element type, we can reassign this to the original variable to update the XML data:

```
let doc = updateMOTPass doc "March 2008"
>>doc : Car = <car>[ <mot>[ <passed>[ 'March 2008' ] ] ]
let doc2 = updateMOTPass doc "January 2007"
>>doc2 : Car = <car>[ <mot>[ <passed>[ 'January 2007' ] ] ]
```

The XML data stored in the two variables "doc" and "doc2" has therefore been altered to update the last time the cars passed their MOTs.

3. Related Work

The foremost work related to the CDuce language is on XDuce, of which CDuce was an extension (as discussed in the previous section). Consulting [CDuceXCGPL], the paper identifies two particular projects that were inspired by XDuce:

- *Xtatic*
  Xtatic [XtaticS] is very similar to CDuce in that both projects are attempts to integrate the useful features of XDuce into general purpose languages, for the creation of general purpose programs which allow native processing of XML data. Both languages attempt to do this through the use of regular expression patterns and types. Unlike CDuce, Xtatic was not designed to be a stand-alone functional programming language, but rather as an extension for C#. Therefore, Xtatic follows an object-oriented approach and can be used natively as part of any C# program.

- *XQuery*
  Designed primarily for querying XML, the type system designed for XQuery [XQueryS] was based upon the one used in XDuce (as with CDuce). The method used to make an XML query is similar in XQuery to that of CDuce and includes support for XPath [CDuceXCGPL]. Unlike CDuce, XQuery does not have regular expression patterns or structural typing [CDuceXCGPL].

More information regarding related work may be found in [CDuceXCGPL] and [CDuceSR].

4. Regular Expressions in Types and Patterns

Having looked at the CDuce language and related work, this section will examine the use of and reasons for the inclusion of regular expressions in both patterns and types. The next section presents example code containing types and patterns that utilise these regular expressions.

The regular expression operators used in the CDuce language are:

- *"?"* - zero or one items
- *"+"* - one or more items
- *"*"* - zero or more items

[6] Note that this could have been simplified to the one pattern matching expression "*<car>[<mot> [_*]] -> <car>[<mot>[<passed> nStr]]*" (since the resulting expression is the same despite the input pattern).

These regular expression operators are used in XML and retain the same meaning. These operators are referred to as “greedy” operators that attempt to match with as many items as possible [CDuceXCGPL]. Non-greedy variants are constructed by placing a “?” suffix to the operator[7].

The inclusion of regular expressions in the type system of CDuce follows a similar reasoning as that for the inclusion of regular expressions in DTDs. The type system, when used to define XML elements, allows the structure of acceptable documents to be defined in advance. This enforcement of a structure allows for consistency in XML documents which conform to this structure, and ensures that valid XML documents used in operations cause expected results to be produced. However, it may be desirable to define the structure of a valid XML document but allow some flexibility for exact numbers. For example, an XML document containing vehicle information may contain “car” elements. Stating an exact number of car elements allowed would not be appropriate, as the initial document to be written could need to contain more or less cars than the definition allowed for. Furthermore, modification to a different number would be impossible. Regular expressions allow the definition to state a particular structure but allow flexibility. For example, the vehicle definition could state there can be zero or more cars. Another example would be that an owner may have more than one vehicle, so a regular expression stating “(car | bike | lorry)*” could be used to say that the owner as zero or more of any of the given vehicles.

When the CDuce definition of the type - containing regular expressions - is compiled, the type is represented internally as a recursive pattern. When a variable of the given type is declared, the definition used (either explicitly in the CDuce code or read from a file) is pattern matched against the associated recursive pattern. If the pattern matching fails, the definition did not conform to the structural definition for an XML document of this type (i.e. it may be well-formed but is not valid). The compiler will either throw an error (if the definition was stated explicitly in the program) or assign the variable the type “Any”, if read in from the file[8].

The inclusion of regular expressions in the type definition therefore provide the same advantages as using them in a DTD: namely, the programmer can state the formal structure for a valid XML document for the program purposes which guarantees that valid documents will be used in operations but is not so rigid as to restrict the data that is contained in the document.

When constructing patterns as part of pattern matching, CDuce allows the construction of regular expression patterns, which can be used to pattern match against types that use regular expressions. This allows the programmer to write queries and manipulation functions which can use XML data that utilises the use of regular expressions when defining it’s type structure. Although the general structure of the XML file is known, the exact layout of the particular XML data may not be known. Regular expressions in patterns can therefore capture this uncertainty, in effect allowing the programmer to assert “there will be some number of these, the exact number is unknown, but it will be roughly this”. It is also possible for the programmer to bind a regular expression pattern to a

---

[7] The difference between greedy and non-greedy versions is documented in [CDuceXCGPL]. Essentially, the difference comes from the structure of the internal pattern used for the internal representation of the expression. For greedy expressions, the internal pattern places the pattern to match on the left of the “|” operator(meaning it is evaluated first), while the non-greedy version places it on the right.

[8] This highlights a slight limitation of CDuce. Since it is not possible to explicitly assign the root element type to an XML document that has been read in (see footnote 4), non-valid documents may be read in and assigned to variables. However, these will cause the program to fail if used in a function that expected the variable to be of the root element type.

variable using a "*sequence capture variable*" which is structured as "*x::(regular expression pattern)*".

Internally, these regular expression patterns are converted by the compiler into a recursive pattern. For example, if a regular expression pattern states that, for some type "Owner", there can be one or more occurrences, a pattern of the form "P = (p & Owner, Q) | (_ , P) and Q = (p & Owner, Q) | (p & nil)" would be created. The pattern binds the occurrences of "Owner" to some sequence variable "p", with the "nil" pattern used to signal the end of the sequence[9].

5. Example

The example code written here can be copied into the online CDuce interpreter [CDuceOP] and run to see the results[10]. To allow the reader to see the results of CDuce programming without the need to install the interpreter on their home machine, the code only uses functionality available on the online interpreter and does not include code saving the XML to a file or loading from a file.

The example CDuce file was inspired by the example used in [CDuceXCGPL] (§2). The example illustrates how an XML document containing vehicle information stored as part of the DVLA records can be represented and instantiated in a CDuce program, with regular expressions used in the type definitions for the XML. Finally, functions and variables are defined which contain patterns that query the XML for information. The example will be divided into steps to illustrate the approach used to create the program. The code for each step will be presented within a boxed area.

Step 1 - Create the Types

The first step involves creating the types for the XML model. The code for the type declarations is as follows:

```
(* DVLARecord contains zero or more vehicles*)
type DVLARecord = <dvla> [ Vehicle*]
(* A vehicle has an attribute stating whether car or motor bike, another
stating whether taxed or not. It has as children one or more owners and
a registration number *)
type Vehicle = <vehicle vehType=(”car”|”mbike”) taxed=("yes" | "no")>
            [ Owner+ Reg ]

(*An owner has a name and may have multiple addresses or telephone
numbers*)
type Owner = <owner> [ Name (Address | Tel)* ]
(*Registration number has one or more letters followed by either a "-"
number "-" letter or just "-" letter*)
type Reg = <reg> [ ('A'--'Z')+ ( '-' ('0'--'9') )? '-' 'A'--'Z' ]

(*Name is just text*)
type Name = <name> [PCDATA ]
(*Address has either a number and street name or just a name*)
type Address = <addr> [ ( 0--* Name | Name ) ]
(*Telephone number has some numbers, possibly separated by dashes*)
type Tel = <phone> [ 0--* ]
```

[9] Inspired by an example in [CDuceXCGPL] (§3.6)

[10] A slight problem was noticed when running the code, copied directly from the word processor, into the prototype interpreter. The open quote character (“) was not recognised correctly. Re-typing the quotes in the interpreter fixes this problem.

Regular expressions have been used throughout these type declarations. The “DVLARecord” type (type root element) states that the XML can contain zero or more occurrences of vehicles. The “Car” and “MotorBike” types contain a regular expression stating that there may be one or more owners.

Step 2 - Create an XML document using the defined types
The next step is to create a new variable that will represent the XML document conforming to the structure defined in the type definitions. The code for the XML definition for this example will be the following:

```
let doc : DVLARecord =
  <dvla>[ <vehicle vehType=”car” taxed="yes"> [
            <owner> [
              <name> ['Tim']
              <addr> [123 <name>['Edin']]
              <phone> [12345]]
            <reg> ['ABC-Z'] ]
          <vehicle vehType=”car” taxed="yes"> [
            <owner> [
              <name> [ 'Kim' ]
              <addr> [ 153 <name>['Glas'] ]
              <phone> [14656] ]
            <owner> [
              <name> [ 'Tim' ]
              <addr> [ 123 <name>['Edin'] ]
              <phone> [12345] ]
            <reg> ['ABC-3-Z'] ]
          <vehicle vehType=”mbike” taxed="no"> [
            <owner> [
              <name> [ 'Jim' ]
              <addr> [ <name>['RichPlace'] ]
              <addr> [ 123 <name>['RicherPlace'] ]
              <phone> [32156] ]
            <reg> ['ABR-6-Z'] ]
        ]
```

Step 3 - Query the XML
Now that an XML document has been created, functions will be created to query the XML to discover information regarding untaxed vehicles. The following functions are defined:

```
(* The first function returns the names of drivers with untaxed
vehicles. When extracting the owners, the "+" operator is used to
create a regular expression stating each vehicle may have more than one
owner. The compiler knows this from the type definition and will throw
an error if simply the variable 'o' is used*)
let getUntaxedNames ( DVLARecord -> [Name*])
  <dvla> x
    -> transform x with <vehicle taxed=y ..> [ o::(Owner+) _*]
      -> if y="no" then (map o with <owner>[n _*] -> n) else []

(* This second function returns the owners of untaxed vehicles numbers.
The regular expression for owner is repeated here. This time, another
regular expression is used to append all the possible phone numbers to
t (which may be zero or more)*)
let getUTPhoneNumbers ( DVLARecord -> [Tel*])
  <dvla> x
    -> transform x with <vehicle taxed=y ..> [ o::(Owner+) _]
      ->if y="no" then ( transform o with <owner>[_ (t::Tel | _)*]
          -> t)
        else []
```

When the functions are applied to the XML document defined in the first step, the following information is returned:

```
let untaxedName = getUntaxedNames doc
>> val untaxed : [ Name* ] = [ <name>[ 'Jim' ] ]

let utPhoneNum = getUTPhoneNumbers doc
>> val utPhoneNum : [ Tel* ] = [ <phone>[ 32156 ] ]
```

Jim will probably be getting a call from the DVLA very soon!

6. Conclusion

The CDuce language was developed to be a general purpose language which was tailored for the development of XML programs. Extending the XML type system developed in XDuce, CDuce allows programmers to create programs that treat XML data as types, allowing manipulation and querying to be performed in a manner familiar to a functional programmer.

The inclusion of regular expression types and patterns allows not only the programmer to create, query and manipulate complex XML structures with ease, but their inclusion also facilitates the compiler in identifying valid XML data. The types and patterns that utilise regular expressions are converted into recursive patterns by the compiler, with values that use these patterns or types pattern matched against these internal patterns.

Overall, CDuce is an ideal platform for the development of XML and XML based applications. The type and pattern systems guarantee statically valid XML. This guarantee allows XML data to be used or transferred between sources and still retain its meaning. It also guarantees that functions written to query the XML behave as expected and return only valid results.

Bibliography

The following resources were used and consulted while researching for this report:

[CDuceSite] *http://www.cduce.org/ - CDuce home page*[11]

[CDuceOP] *http://reglisse.ens.fr/cgi-bin/cduce - CDuce online prototype interpreter*

[CDuceXCGPL] *CDuce: An XML-Centric General-Purpose Language - CDuce paper*

[CDuceGREM] *Greedy regular expression matching - CDuce Paper*

[CDuceEMREP] *Error Mining for Regular Expression Patterns - CDuce paper*

[OCCDuceT] *http://www.ocaml-tutorial.org/manipulating_xml_documents_with_cduce - Manipulating XML file with CDuce*

[CDuceSTP] *http://www.cduce.org/manual_types_patterns.html - CDuce page on types and patterns*

[CDuceSE] *http://www.cduce.org/manual_expressions.html - CDuce page on expressions*

[CDuceSR] *http://www.cduce.org/#research - CDuce page (regarding the research project)*

[XtaticS] *http://www.cis.upenn.edu/~bcpierce/xtatic/ - Xtatic Home page*

[XQueryS] *http://www.w3.org/TR/xquery/ - XQuery 1.0: An XML Query Language*

[11] As an interesting note, the CDuce website itself is generated by a CDuce program generating XML in the format of XHTML