

A Type-Safe Embedding of XDuce into ML

Martin Sulzmann¹

*School of Computing
National University of Singapore
S16 Level 5, 3 Science Drive 2
Singapore 117543*

Kenny Zhuo Ming Lu²

*School of Computing
National University of Singapore
S16 Level 5, 3 Science Drive 2
Singapore 117543*

Abstract

We consider the problem of integrating XDuce into ML. This is difficult because of incompatible type and value representations. Our solution is a type-driven translation scheme from XDuce to ML based on a structured representation of XDuce values. XDuce type inference guides the insertion of appropriate coercion functions to translate regular expression pattern matching and uses of semantic subtyping. We can extend our translation scheme to include ML function calls and patterns into XDuce. Thus, we can embed XDuce into ML. Our results allow to enrich the ML language with support for dealing with semi-structured data.

Key words: Semi-structured data handling, program transformation, language integration/extension.

1 Introduction

There has been some notable interest in making use of typed programming languages for XML processing [16,13,3,11]. The advantages of such an approach are clear. In a typed setting we can provide some static guarantees about the well-formedness of XML documents and transformations. Previous work can be roughly divided into two categories.

On one hand we find special-purpose languages such as XDuce [16,13] and CDuce [3] which are specifically designed for processing XML data. Such

¹ Email:sulzmann@comp.nus.edu.sg

² Email:luzm@comp.nus.edu.sg

languages offer great expressiveness while providing strong static guarantees. However, it is difficult to promote their widespread use due to limited library support and integration with existing languages.

On the other hand, we find approaches [27,30,6] which try to integrate XML features into existing languages such as Haskell [12]. Note that the approaches [27,30] only support type-safe construction of XML values but do not provide the strong type guarantees as e.g. XDuce when processing XML values. Approach [6] comes with no type safety guarantee and is merely a limited, untyped XDuce interpreter written in Haskell.

What we would like is to enhance languages such as Haskell and ML with the main XDuce features which are semantic subtyping among regular expression types and regular expression pattern matching. The difficulty with language integration lies often in incompatible type and value representations.

Our approach is based on a structured representation of regular expression types as opposed to the flat representation found in XDuce and CDuce. We translate XDuce to ML by inserting some appropriate coercions among structured values to mimic uses of semantic subtyping. We can derive these coercions out of the proof of language containment among regular expression types obtained while performing type inference of XDuce.³ Similarly, regular expression patterns are translated to ordinary pattern matching via another set of coercions. The regular expression containment proof system to derive coercion functions is extensible. Hence, we can extend XDuce (and the translation scheme to ML) with ML function calls and ML patterns. Thus, we achieve a type-safe embedding of XDuce into ML.

In summary, our contributions are:

- We give a type and semantic preserving translation from XDuce to ML (Section 4).
- Based on our translation scheme we show how to integrate ML patterns and ML function calls into XDuce (Section 5).
- We provide evidence that our approach is practical (Section 6).

We give an overview of our translation scheme in Section 2. In Section 3, we review some background materials on how to derive coercions out of the proof of language containment among regular expressions. Related work is discussed in Section 7. We conclude in Section 8.

Full proofs of all results stated can be found in an accompanying technical report [26]. Due to space limitations, we restrict ourselves to a very simple set of examples. More realistic examples can be found here [26]. Note that throughout the paper we use Haskell-style syntax for XDuce and ML programs.

³ XDuce demands explicit type annotations. Hence, it may be more appropriate to talk about ‘type checking’ of XDuce. But, we decided to use here the more general term ‘type inference’.

2 Overview

We illustrate our translation scheme via a very simple XDuce program.

Example 2.1 Consider the following.

```
regtype A = A[String]
regtype B = B[String]
f :: (A|B)*->B*
f (x as A*, y as B, z as B*) = y
f (x as A*) = ()
f (x as B*, y as (A|B)*) = f y
```

Let us first understand how this program behaves. Function `f` takes a sequence of `As` or `Bs` as input and returns a sequence of `Bs`. This is specified via *regular expression types* [18], see `f`'s annotation. The result type tells us that the output can be any sequence of `Bs`. Note that we often adopt the convention that regular expressions in type-writer font refer to types and expressions in math-font refer to values.

In the first function clause, we find a *regular expression pattern* stating that this case applies if the input consists of zero or more `As` followed by at least one `B`. In the function body we return a value of type `B` which is a *semantic subtype* (viewed in terms of language containment) of the result type `B*`. Hence, this clause is type correct. Note that we often drop the “semantic” part and say subtype or subtyping for short.

The second clause empties all sequences consisting of zero or more `As`. Note that `()` represents the empty sequence. Again, the function body is type correct due to semantic subtyping. The third clause seems to indicate that this case applies if we find zero or more `Bs` followed by zero or more `As` or `Bs`. However, in XDuce we take into account “failure” of the previous clauses (function clauses are processed from top to bottom). Hence, we actually can give more precise types to the last pattern (`x as B*`, `y as (A|B)*`). Here, we assume here the longest-match policy. Hence, from pattern type inference [17] we can conclude that variable `x` binds to values of type `B*` and `y` only binds to values of type `(A, (A|B)*)`. Details of the actual inference process for this example can be found in the Appendix. In the body, `y` is applied to function `f` recursively. Obviously it is type-safe because type `(A, (A|B)*)` is a subtype of `(A|B)*`. Furthermore, based on type inference we can verify that patterns are exhaustive, i.e. covering all possible input values.

Let us attempt to translate the above XDuce program into a ML program. The first step in our translation consists of finding a suitable representation for regular expression types. We represent “*” (zero or more occurrences) by lists and “|” (choice) via the data type `data Or a b = L a | R a`. For each regular expression definition we introduce a data type definition. E.g.,

```
data A_U = A_U String
data B_U = B_U String
```

where the suffix U indicates that they are the underlying representations. This is essentially the representation suggested by Wallace and Runciman [30]. Thus, the translation of \mathbf{f} 's type annotation yields $\mathbf{f} :: [\mathsf{Or} \ \mathsf{A_U} \ \mathsf{B_U}] \rightarrow [\mathsf{B_U}]$.

In general, we write $\llbracket R \rrbracket$ to denote the ML representation of R where

$$\begin{aligned} \llbracket () \rrbracket &= [\mathsf{Phi}] & \llbracket R^* \rrbracket &= \llbracket [R] \rrbracket & \llbracket l \rrbracket &= \mathsf{1_U} \\ \llbracket (R_1 | R_2) \rrbracket &= (\mathsf{Or} \ \llbracket [R_1] \rrbracket \ \llbracket [R_2] \rrbracket) & \llbracket (R_1, R_2) \rrbracket &= (\llbracket [R_1] \rrbracket, \llbracket [R_2] \rrbracket) \end{aligned}$$

We assume that type Phi inhabits no values. Thus, the empty sequence $()$ has the empty list $[]$ as its ML representation. Note that we could also give $\llbracket () \rrbracket$ the unit type with no change in results. For each label type l we introduce `data 1_U = 1_U`. Clearly, we can imagine other more efficient structured representations for sequences etc. However, we leave the study of such issues for future work.

The real challenge is to translate semantic subtyping and regular expression pattern matching. Consider the first clause above. Earlier, we observed that although the expected type is B^* and we return a value y of type B the program is type-safe because B is a semantic subtype of B^* . In a naive translation we could keep y but then we are in trouble. In our structured translation of regular expression types we find that the translated return type of \mathbf{f} is $[\mathsf{B_U}]$. This type does not match $\mathsf{B_U}$, the type of y in the translation. To ensure that the translated ML program is type correct, our idea is to insert an appropriate coercion. E.g.,

```
u1 :: B_U -> [B_U]
u1 x = [x]
```

The big question is how to build these coercions systematically? Fortunately, in our own previous work [23] we show how to construct such coercions out of the proof of language containment among regular expression types. There, we define a proof system for deciding regular containment where a judgment $\vdash R_1 \leq^u R_2$ is derivable iff $L(R_1) \subseteq L(R_2)$ where $L(R)$ denotes the language described by R . As a side effect, we derive a coercion u of type $\llbracket [R_1] \rrbracket \rightarrow \llbracket [R_2] \rrbracket$. Note that these coercions have the property to inject a smaller value into a larger value. Hence, we often refer to these coercions as “up-cast” functions.

For regular expression pattern matching we apply a similar translation method. Consider the pattern type $(\mathsf{A}^*, \mathsf{B}, \mathsf{B}^*)$ of the first clause which is a (semantic) subtype of the input type $(\mathsf{A} | \mathsf{B})^*$. At run-time we will need to check whether some input value matches this pattern. In terms of our structured representation we will need to check whether a value of type $[\mathsf{Or} \ \mathsf{A_U} \ \mathsf{B_U}]$ can be turned into a value of type $([\mathsf{A_U}], (\mathsf{B_U}, [\mathsf{B_U}]))$. Again we make use of coercions but this time we need a “down-cast” rather than “up-cast” function. E.g.,

```
data Maybe a = Just a | Nothing
```

```
d1 :: [Or A_U B_U] -> Maybe ([A_U], (B_U, [B_U]))
d1 []                = Nothing
```

```

d1 ((A_U v):r) = case (d1 r) of
    Just (as,(b,bs)) -> Just (((A_U v):as),(b,bs))
    Nothing -> Nothing
d1 ((B_U v):r) = case (d1' r) of
    Just bs -> Just ((B_U v),bs)
    Nothing -> Nothing
d1' :: [Or A_U B_U] -> Maybe [B_U]
d1' ((A_U _):_) = Nothing
d1' [] = Just []
d1' ((B_U v):r) = case (d1' r) of
    Just bs -> Just ((B_U v):bs)
    Nothing -> Nothing

```

Note that matching may fail, therefore, we make use of the `Maybe` data type. Derivation of down-cast functions is similar to the up-cast case (out of the proof of language containment). Hence, a valid judgment $\vdash R_1 \leq_d^u R_2$ implies a down-cast function $d :: \llbracket R_2 \rrbracket \rightarrow \text{Maybe } \llbracket R_1 \rrbracket$ and a up-cast function $u :: \llbracket R_1 \rrbracket \rightarrow \llbracket R_2 \rrbracket$. Details of the regular expression containment proof system can be found in Section 3.

Based on XDuce style type inference we can calculate the remaining coercion functions which are as follow. For brevity, we omit their function bodies.

```

d2 :: [Or A_U B_U] -> Maybe [A_U]
d3 :: [Or A_U B_U] -> Maybe ([A_U],(B_U,([B_U],(A_U,[Or A_U B_U])))
u2 :: ([A_U],(B_U,([B_U],(A_U,[Or A_U B_U]))) ->
    ([B_U],(A_U,([Or A_U B_U])))
u3 :: (A_U,([Or A_U B_U])) -> [Or A_U B_U]

```

Then, the translation of XDuce function `f` to ML yields

```

f :: [Or A_U B_U] -> [B_U]
f v = case (d1 v) of
    Just vp1 -> let (x,(y,z)) = vp1 in (u1 y)
    Nothing -> case (d2 v) of
        Just vp2 -> []
        Nothing -> case (d3 v) of
            Just vp3 -> let (x,y) = (u2 vp3)
                in (f (u3 y))
            Nothing -> error "Non-exhaustive pattern."

```

The observant reader may wonder about the type of down-cast function `d3` used in the translation of the last clause. In the above, function `d3` checks whether values of type `[Or A_U B_U]` can be turned into values of type `([A_U],(B_U,([B_U],(A_U,[Or A_U B_U])))`. These (underlying) types correspond to the regular expression types $(A|B)^*$ and $(A^*,B^+,A,(A|B)^*)$ ⁴ However, as said earlier, pattern inference yields that the precise regular expression type of the last pattern is $(B^*,(A,(A|B)^*))$. The point is that type $(A^*,B^+,A,(A|B)^*)$ represents the set of input values which could not be matched by any of the earlier clauses intersected with the type of the last

⁴ Silently, we use B^+ as a short-hand for (B,B^*) .

pattern. Hence, we first apply `d3 v` to *check* whether the last pattern applies.⁵ Then, we use up-cast function `u2` to *distribute* the matched value to the pattern. As mentioned earlier, we assume the longest-match policy for distribution [28].

In case of the above situation, we can optimize our translation by combining the effect of `d3` and `u2`.

```
d3u2 :: [Or A_U B_U] -> Maybe ([B_U], (A_U, ([Or A_U B_U])))
...
-- replaces (case (d3 v) of ...)
case (d3u2 v) of
  Just (x,y) -> f (u2 y)
  Nothing -> error "Non-exhaustive pattern."
```

In essence, `d3u2` combines the pattern matching check and value distribution. However, this may not always be possible (see upcoming Example 4.1). Obviously, there are further optimizations possible, some of which we briefly discuss in Section 6.

Our methods developed so far allow us to integrate ML function calls in XDuce expressions. For example, consider some expression (`head xs`) where `head` is a ML function, say of type `head :: [A_U] -> A_U`, and `xs` is a XDuce variable of type `A*`. The above expression certainly makes sense. We can translate the above program text by simply extending our proof system for language containment. by providing additional facts such as $\vdash A^* \leq^u [A_U]$ (plus some appropriate proof term representing the up-cast coercion u). The general rule may look as follows.

$$\begin{array}{c}
 (*-\square) \quad \frac{\vdash t_1 \leq^{u'} t_2 \quad \begin{array}{l} u \square = \square \\ u (x : xs) = (u' x) : (u xs) \end{array}}{\vdash t_1^* \leq^u [t_2]}
 \end{array}$$

where t refers to an extended type language which may also include ML types. Issues regarding the extension of our proof system are discussed in Section 3.2

The integration of ML patterns in regular expression patterns can be achieved similarly by extending the proof system.

Example 2.2 Consider the following program fragment.

```
f :: [A*] -> ...
f ((x as A, xs as A*) : ys as [A*]) = ...
```

We find a regular expression pattern, `(x as A, xs as A*)`, within an enclosing ML list pattern. Note that we provide the ML pattern annotations only for clarity. They could be inferred here. The trick is to rephrase ML pattern inference as an instance of XDuce pattern inference by using singleton types. Thus, we generate that `((x as A, xs as A*) : ys as [A*])` has type `(CONS`

⁵ In fact, the function is exhaustive. Hence, the last pattern will always succeed.

$(A, A^*) [A^*]$). Then, we derive $\vdash (\text{CONS } (A, A^*) [A^*]) \leq_d [A^*]$ by appropriately extending our proof system (here we will only need the down-cast coercion d). Hence, our translation scheme yields the following.

```

data NIL = NIL
data CONS x xs = CONS x xs
-- nested regular pattern
d' :: [A_U] -> Maybe (A_U, [A_U])
d' (x:xs) = Just (x,xs)
d' _      = Nothing
-- composition of ML and XDuce pattern matching
d :: [[A_U]] -> Maybe (CONS (A_U, [A_U]) [[A_U]])
d (y:ys) = case (d' y) of
             Just (x,xs) -> Just (CONS (x,xs) ys)
             -           -> Nothing
f :: [[A_U]] -> ...
f v = case (d v) of
        Just (CONS (x,xs) ys) -> ...

```

Note that as an optimization we have combined the down-cast coercion (pattern check) with the up-cast coercion (pattern value distribution).

3 Regular Expression Containment Proof System

We review our own work reported in [23] where we devise a proof system for deciding regular expression language containment which additionally computes down-cast and up-cast coercions among structured ML representations of regular expressions. Note that similar proof systems can be found for recursive type equality and subtyping [5]. A major achievement in [23] is to derive coercions for regular hedges. Although, we will not make use of this case here. A new topic is the extension of proof rules which is necessary to allow for an interaction between XDuce and ML programs.

3.1 Proof Rules

The rules of our proof system are in Figure 1. We make use of ordinary judgments $\vdash R_1 \leq_d^u R_2$ and normalized judgments $\vdash_{\text{inf}} N_1 \leq_{d'}^{u'} N_2$ to derive proof terms $u : \llbracket R_1 \rrbracket \rightarrow \llbracket R_2 \rrbracket$, $d : \llbracket R_2 \rrbracket \rightarrow \text{Maybe } \llbracket R_1 \rrbracket$, $u' : \llbracket N_1 \rrbracket \rightarrow \llbracket N_2 \rrbracket$ and $d' : \llbracket N_2 \rrbracket \rightarrow \text{Maybe } \llbracket N_1 \rrbracket$. Such judgments are valid, i.e. can be derived with the proof rules, if $R_1 \leq R_2$ and $N_1 \leq N_2$. Note that we use ML style syntax to define proof terms describing coercions functions.

The proof rules are divided into two sets of ordinary and normalized rules. In rule (Norm) we switch from the ordinary to the normalized system. We normalize regular expressions based on the partial derivatives operation proposed by Antimirov [1]. Antimirov observed that every (non-empty) regular expression can be turned into a union of (l, R) s and $()$ where leading l s are distinct. E.g., $(A|B)^*$ is normalized to $(A, (A|B)^*) | (B, (A|B)^*) | ()$. For-

Ordinary Rules:

$$\begin{array}{c}
u = \lambda v1.(fn2 (u' (tn1 v1))) \\
d = \lambda v2.case d' (tn2 v2) of \\
\quad Just n1 \rightarrow Just (fn1 n1) \\
\quad Nothing \rightarrow Nothing \\
\vdash R_1 \rightsquigarrow_{tn1}^{fn1} N_1 \quad \vdash R_2 \rightsquigarrow_{tn2}^{fn2} N_2 \\
\vdash_{\text{Inf}} N_1 \leq_{d'}^{u'} N_2 \\
\hline
\vdash R_1 \leq_d^u R_2
\end{array}$$

Normalized Rules:

$$\begin{array}{c}
u = \lambda v.\square \\
d = \lambda v.(Just \square) \quad (\text{m-m}) \\
\vdash_{\text{Inf}} () \leq_d^u () \\
\hline
\vdash_{\text{Inf}} (l_1, R_1) \leq_d^u (l_2, R_2)
\end{array}$$

$$\begin{array}{c}
u = \lambda n1.(R (u' n1)) \\
d = \lambda (R n3).(d' n3) \\
\vdash_{\text{Inf}} N_1 \leq_{d'}^{u'} N_3 \\
\hline
\vdash_{\text{Inf}} N_1 \leq_d^u (N_2|N_3)
\end{array}$$

$$\begin{array}{c}
u = \lambda n1.(L (u' n1)) \\
d = \lambda (L n2).(d' n2) \\
\vdash_{\text{Inf}} N_1 \leq_{d'}^{u'} N_2 \\
\hline
\vdash_{\text{Inf}} N_1 \leq_d^u (N_2|N_3)
\end{array}$$

$$\begin{array}{c}
u = \lambda x.case x of \\
\quad (L n1) \rightarrow (u' n1) \\
\quad (R n2) \rightarrow (u'' n2) \\
d = \lambda n3.case (d' n3) of \\
\quad Just n1 \rightarrow Just (L n1) \\
\quad Nothing \rightarrow case (d'' n3) of \\
\quad \quad Just n2 \rightarrow Just (R n2) \\
\quad \quad Nothing \rightarrow Nothing \\
\vdash_{\text{Inf}} N_1 \leq_{d'}^{u'} N_3 \quad \vdash_{\text{Inf}} N_2 \leq_{d''}^{u''} N_3 \\
\hline
\vdash_{\text{Inf}} (N_1|N_2) \leq_d^u N_3
\end{array}$$

Fig. 1. Regular Expression Containment Proof System

mally, we write $\vdash R \rightsquigarrow_{tn}^{fn} N$ which implies (bijections) $fn : \llbracket R \rrbracket \rightarrow \llbracket N \rrbracket$, $tn : \llbracket N \rrbracket \rightarrow \llbracket R \rrbracket$. Details of normalization are rather tedious and can be found in [26]. It is then clear that in the normalized system we simply need to find matching leading labels, see rule (m-m). A more subtle point is that ordinary rules need to be interpreted co-inductively. That is, along the proof derivation, if a ordinary judgment has already appeared in the history, it is reduced to True immediately.

In our first example, we build an up-cast coercion for our example in the Introduction. The second examples shows the necessity for co-induction.

Example 3.1 Consider proving $(A, (A|B)*) \leq^{u3} (A|B)*$. Along the derivation we build the resulting coercion functions.

Proof Derivations	Proof Terms
$\vdash (A, (A B)*) \leq^{u3} (A B)*$	$u3 = \lambda v.(fn2 (u31 (tn1 v)))$
$\longrightarrow_{Norm} \vdash (A, (A B)*) \rightsquigarrow_{tn1} (A, (A B)*),$	$tn1 = \lambda v.v$
$\vdash (A B)* \rightsquigarrow^{fn2}$	$fn2 = \lambda v.case\ v\ of$
$(A, (A B)*) (B, (A B) ()),$	$L(a, r) \rightarrow ((L\ a) : r)$
	$R\ s \rightarrow case\ s\ of$
	$L(b, r) \rightarrow ((R\ b) : r)$
	$R\ [] \rightarrow []$
$\vdash_{\text{Inf}} (A, (A B)*) \leq^{u31}$	$u31 = \lambda v.(L (u32 v))$
$(A, (A B)*) (B, (A B) ())$	
$\longrightarrow_{\perp 1} \vdash (A, (A B)*) \leq^{u32} (A, (A B)*)$	$u32 = \lambda(A_U, v).(A_U, (u33 v))$
$\longrightarrow_{m-m} \vdash (A B)* \leq^{u33} (A B)*$	$u33 = \lambda v.v$
$\longrightarrow_{taut} True$	

Example 3.2 Consider $\vdash A* \leq (A|B)*$.

Proof Derivations	Proof Terms
$\vdash A* \leq^u (A B)*$	$u = \lambda v.(fn2 (u1 (tn1 v)))$
$\longrightarrow_{Norm} \vdash_{\text{Inf}} ((A, A*) ()) \leq^{u1}$	$u1 = \lambda(L\ v).(u2\ v)) (R\ v).(u3\ v)$
$((A, (A B)*) (B, (A B)*) ()),$	
$\vdash A* \rightsquigarrow^{tn1} ((A, A*) ()),$	$tn1 = \dots$
$\vdash (A B)* \rightsquigarrow^{fn2}$	$fn2 = \dots$
$((A, (A B)*) (B, (A B)*) ()),$	
$\longrightarrow_{\perp 1} \vdash_{\text{Inf}} (A, A*) \leq^{u2}$	$u2 = \lambda v.(L (u4 v))$
$((A, (A B)*) (B, (A B)*) ()),$	
$\vdash_{\text{Inf}} () \leq^{u3} ((A, (A B)*) (B, (A B)*) ())$	$u3 = \dots$
$\longrightarrow_{\perp 1} \vdash_{\text{Inf}} (A, A*) \leq^{u4} (A, (A B)*),$	$u4 = \lambda(A_U, r).(A_U, (u5 r))$
\dots	
$\longrightarrow_{m-m} \vdash A* \leq^{u5} (A B)*,$	$u5 = u$
\dots	
$\longrightarrow_{CoInd} True, \dots$	

Note that we omit some less-important proof terms like `tn1`, `fn2` and `u3`, and we suppose it is clear how to construct them. The derivation shows that after a few steps, the judgment $\vdash A^* \leq (A|B)^*$ appears again. Hence, an inductive interpretation of judgments would lead to non-termination. We apply co-induction. Hence, $\vdash A^* \leq (A|B)^*$ will be reduced to *True*. On the level of proof terms, we simply built the fix-point by defining `u5=u`.

The following result from [23] is an important corner stone on our way to achieve a type-safe embedding of XDuce into ML.

Theorem 3.3 *Let R_1 and R_2 be two regular expressions. Then, $R_1 \leq R_2$ iff $\vdash R_1 \leq_d^u R_2$ where $u :: \llbracket R_1 \rrbracket \rightarrow \llbracket R_2 \rrbracket$, $d :: \llbracket R_2 \rrbracket \rightarrow \text{Maybe } \llbracket R_1 \rrbracket$ are typable in ML.*

3.2 Extending the Proof System

As we saw earlier, the integration of ML function calls makes it necessary to extend the proof rules. E.g., for each label l we need to add the fact that $\vdash l \leq_{\lambda x.(Just\ x)}^{\lambda x.x} \mathbf{1}_U$. In general, we wish to extend the proof system such that for mixed types t_1 and t_2 we have that $\vdash t_1 \leq_d^u t_2$ holds iff the semantic meaning of t_1 is contained in t_2 . We assume that types t_1 and t_2 may contain a mix of regular expression and user-defined ML types and u and d refer to the up-cast and down-cast coercion functions between them. As in case of regular expressions, the semantic meaning of a mixed type t can be described by its implied language. That is, the set of values of type t where we may want to “erase” the occurrence of value constructors to allow for a canonical representation.

In Section 2, we saw the following additional rule to incorporate ML list types.

$$\begin{array}{c}
 (*-\square) \quad \vdash t_1 \leq^{u'} t_2 \quad \begin{array}{l} u \square = \square \\ u (x : xs) = (u' x) : (u xs) \end{array} \\
 \hline
 \vdash t_1^* \leq^u [t_2]
 \end{array}$$

Unfortunately, the above rule does not yield a complete extension of the proof system. For example, $(A,B)^*$ is semantically contained in $[Or\ A_U\ B_U]$. However, the above rule is not sufficient to derive $\vdash (A,B)^* \leq [Or\ A_U\ B_U]$. The subtle point here is that we may also need to extend the normalization rules. We saw earlier that $(A|B)^*$ is normalized to $(A, (A|B)^*) | (B, (A|B)^*) | ()$. Hence, we need additional rules to normalize $[Or\ A_U\ B_U]$ to $(A_U, [Or\ A_U\ B_U]) | (B_U, [Or\ A_U\ B_U]) | ()$.

Another tricky point is that the semantic denotation of mixed types may not be regular anymore. E.g., consider the following data type definition describing a context-free language.

```
data T a = L | N a (T a) a
```

Note that the containment problem between context-free languages is undecidable in general. Hence, a complete extension of the proof system to arbitrary mixed types is not tractable.

We conclude. A sound extension of the proof system to deal with mixed types is mostly straightforward (only-if direction of Theorem 3.3). Completeness requires more care and a closer examination of the particular set of mixed types.

4 Formal Translation of XDuce to ML

The language of expressions and types in XDuce is as follow.

Expressions	$e ::= x \mid () \mid l \mid (e,e) \mid (e e) \mid$ $\text{let } \begin{array}{l} f :: R \rightarrow R \\ f [p_i = e_i]_{i \in I} \end{array} \text{ in } e$
Patterns	$p ::= x \text{ as } R \mid (p, p)$
Values	$v ::= l \mid (v, \dots, v) \mid ()$
Labels	$l ::= A_1 \mid \dots$
RegExp Types	$R ::= () \mid l \mid R^* \mid (R R) \mid (R,R)$

We promote the use of \mid in BNF to eliminate the confusion with the regular expression operator $|$. As usual, patterns are linear, i.e. occur at most once. For simplicity, we omit the treatment of *regular hedges*, which extend labels l to labeled-expression $l[R]$. All our results carry over to regular hedges. We also omit **Any** type which denotes all possible values. Note that **Any** can be replaced by the Kleene-star of the union of all possible labels. E.g., if we have only two labels **A** and **B**, **Any** is just $(A|B)^*$. Hence in this paper, we will not deal with **Any** type.

The translation from XDuce to ML is driven by XDuce inference. For concreteness, we follow the inference scheme described in [28] and assume a longest-match policy. The actual translation process is described in terms of *translation* judgments of the form $\Gamma \vdash e : R \rightsquigarrow E$ where Γ is the XDuce type environment, e is a XDuce expression, E is its translation to ML and R is the regular expression type of e . We omit to give a formal definition of the syntax of ML expressions. Note that we obtain the XDuce typing judgments by simply omitting E . As expected we introduce a translation rule for each XDuce expression. The details are in Figure 2.

Rules (Var), (Fun), (Empty), (Label) and (Pair) contain no surprises. In rule (Sub), we make use of the up-cast function u derived from $R_1 \leq R_2$ to ensure well-typing of the resulting ML expression. We maintain the invariant that E has the ML type $\llbracket R_1 \rrbracket$. Hence, $(u E)$ has type $\llbracket R_2 \rrbracket$.

Rule (Let) translates regular expression pattern clauses into a series of ML style pattern matchings based on the result of pattern inference. Auxiliary judgment $R, \{p_1, \dots, p_n\} \vdash \{R_1, \dots, R_n\}, \{\Gamma_1, \dots, \Gamma_n\}$ computes the type R_i which can reach pattern p_i and the environment Γ_i which holds the binding of pattern variables. We assume that patterns are processed sequentially and

$$\begin{array}{c}
\text{(Var)} \frac{(x : R) \in \Gamma}{\Gamma \vdash x : R \rightsquigarrow x} \quad \text{(Fun)} \frac{(f : R \rightarrow R') \in \Gamma}{\Gamma \vdash f : R \rightarrow R' \rightsquigarrow f} \\
\\
\text{(Label)} \Gamma \vdash l : l \rightsquigarrow l_U \quad \text{(Pair)} \frac{\Gamma \vdash e_1 : R_1 \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : R_2 \rightsquigarrow E_2}{\Gamma \vdash (e_1, e_2) : (R_1, R_2) \rightsquigarrow (E_1, E_2)} \\
\text{(Empty)} \Gamma \vdash () : () \rightsquigarrow \square \\
\\
\text{(Sub)} \frac{\Gamma \vdash e : R_1 \rightsquigarrow E \quad \vdash R_1 \leq^u R_2}{\Gamma \vdash e : R_2 \rightsquigarrow (u E)} \quad \text{(App)} \frac{\Gamma \vdash e : R_1 \rightarrow R_2 \rightsquigarrow E \quad \Gamma \vdash e' : R_1 \rightsquigarrow E'}{\Gamma \vdash e e' : R_2 \rightsquigarrow E E'} \\
\\
\frac{\Gamma.(f : R \rightarrow R') \vdash e : R'' \rightsquigarrow E \quad I = \{1, \dots, n\} \quad R, \{p_1, \dots, p_n\} \vdash \{R_1, \dots, R_n\}, \{\Gamma_1, \dots, \Gamma_n\} \quad \vdash R_i \leq_{d_i} R \quad \Gamma_i \vdash ((p_i) \downarrow_v) : R_{\Gamma_i} \quad \vdash R_i \leq^{u_i} R_{\Gamma_i} \quad \Gamma \cup \Gamma_i.(f : R \rightarrow R') \vdash e_i : R' \rightsquigarrow E_i \quad \text{for } i \in I}{\Gamma \vdash \text{let } \begin{array}{l} f :: R \rightarrow R' \\ f [p_i = e_i]_{i \in I} \\ \text{let } f :: [R] \rightarrow [R'] \\ f x = \text{case } (d_1 x) \text{ of} \\ \quad \text{Just } vp_1 \rightarrow \text{let } ((p_1) \downarrow_v) = u_1 vp_1 \text{ in } E_1 \\ \quad \text{Nothing} \rightarrow \\ \quad \vdots \\ \quad \text{Nothing} \rightarrow \\ \quad \text{case } (d_n x) \text{ of} \\ \quad \quad \text{Just } vp_n \rightarrow \text{let } ((p_n) \downarrow_v) = u_n vp_n \text{ in } E_n \\ \quad \quad \text{Nothing} \rightarrow \text{error "non - exhaust pat"} \end{array} \text{ in } E} \\
\text{(Let)}
\end{array}$$

Fig. 2. Translating XDuce to ML

each pattern follows the longest-match policy. Details are in Figure 3, a brief description follows.

The results of `lbreak` and `rbreak` denote regular languages assuming the input is regular. Intuitively, `lbreak`(R, R_1, R_2) denotes the set of all longest possible prefixes of words in R such that the prefixes are in R_1 and the correspondent suffices are in R_2 . Function `rbreak`(R, R_1, R_2) denotes the corresponding suffices. For instance, `lbreak`(A^*, A^*, A^*)= A^* because A^* is the set of the longest prefixes that satisfy that condition and `rbreak`(A^*, A^*, A^*)= $()$ because the corresponding suffix is $()$. We also assume some standard regular

$$\begin{array}{c}
\text{(VarPX)} \frac{(R \cap R') = R''}{R, (x \text{ as } R') \vdash \{x : R''\}} \quad \text{(PairPX)} \frac{R_1 = (p_1) \downarrow_t \quad R_2 = (p_2) \downarrow_t}{R, (p_1, p_2) \vdash (\Gamma_1 \cup \Gamma_2)} \\
\frac{(\text{lbreak } R \ R_1 \ R_2), p_1 \vdash \Gamma_1}{(\text{rbreak } R \ R_1 \ R_2), p_2 \vdash \Gamma_2}
\end{array}$$

$$\text{(SeqPX)} \frac{R, p_1 \vdash \Gamma_1 \quad R_1 = (R \cap ((p_1) \downarrow_t))}{R, \{p_1, p_2, \dots, p_n\} \vdash \{R_1, R_2, \dots, R_n\}, \{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}}$$

$$((x \text{ as } R) \downarrow_t) = R \quad ((p_1, p_2) \downarrow_t) = ((p_1) \downarrow_t), ((p_2) \downarrow_t)$$

$$\begin{aligned}
\text{lbreak}(R, R_1, R_2) &= \{w_1 \in R_1 \mid \exists w_2 \in R_2 \text{ such that } (w_1, w_2) \in R \wedge \\
&\quad \neg(\exists v_1 \neq (), v_2 \text{ such that } (v_1, v_2) = w_2 \wedge (w_1, v_1) \in R_1 \wedge v_2 \in R_2)\}
\end{aligned}$$

$$\begin{aligned}
\text{rbreak}(R, R_1, R_2) &= \{w_2 \in R_2 \mid \exists w_1 \in R_1 \text{ such that } (w_1, w_2) \in R \wedge \\
&\quad \neg(\exists v_1 \neq (), v_2 \text{ such that } (v_1, v_2) = w_2 \wedge (w_1, v_1) \in R_1 \wedge v_2 \in R_2)\}
\end{aligned}$$

Fig. 3. Pattern Inference (Longest-Match)

expression operations such as \cap (intersection), $-$ (difference). The interested reader is referred to [28] for more details.

The results of pattern inference allow us to derive $R_i \leq_{d_i} R$ where the down-cast in combination with ML pattern matching allows us to check which function clause applies. A subtle point is that the structured ML representation of R_i may not match the shape of the ML representation of pattern p_i . Let us consider an example to illustrate this point. Before, we define function $(\cdot) \downarrow_v$ to translate a XDuce pattern to ML. We have that $((x \text{ as } R)) \downarrow_v = x$ and $((p_1, p_2)) \downarrow_v = (((p_1) \downarrow_v), ((p_2) \downarrow_v))$. E.g., $((x \text{ as } A), (y \text{ as } B)) \downarrow_v = (x, y)$.

Example 4.1 consider,

```
f :: ((A,B) | (B,A)) -> ((A|B), (A|B))
f (x as (A|B), y as (A|B)) = (x,y)
```

Intuitively, the pattern can only accept values of type (A, B) or (B, A) , because that is what is solely given by the input type. Note that $((A, B) | (B, A)) \cap ((A|B), (A|B)) = ((A, B) | (B, A))$. Hence, in the translation the input value is first down-casted from $\llbracket ((A, B) | (B, A)) \rrbracket$ to $\llbracket ((A, B) | (B, A)) \rrbracket$. Under the pattern inference algorithm in Figure 3, we infer environment $\Gamma_1 = \{(x : (A|B)), (y : (A|B))\}$. However, we cannot directly distribute the intermediate ML value of type $\llbracket ((A, B) | (B, A)) \rrbracket$, which is equal to $(\text{Or } (A_U, B_U) (B_U, A_U))$, to the expected ML pattern (x, y) , because they are not of the same shape.

In order to distribute the input value according to the translated ML

pattern, we need to infer another type R_{Γ_i} . This type must satisfy that $\Gamma_i \vdash ((p_i) \downarrow_v) : R_{\Gamma_i}$ ⁶ and $\vdash R_i \leq^{u_i} R_{\Gamma_i}$ hold (see premise of rule (Let)). Thus, we can distribute (via up-cast function u_i) the result of a successful pattern match to the ML representation of the regular expression pattern.

A common assumption is that patterns are exhaustive which we do not enforce explicitly. To ensure exhaustiveness we simply need to add the condition $\vdash R \leq \bigcup_{i \in I} R_i$ to the premise of rule (Let). That is, the union of the inferred types R_i of patterns p_i subsumes the input type R .

The full translation of Example 2.1 can be found in Appendix A. We conclude this section by stating some formal results about our translation scheme.

The first result shows that (as expected) resulting expressions are typable in ML. We write $\llbracket \Gamma \rrbracket$ to denote $\{(x : \llbracket R \rrbracket) \mid (x : R) \in \Gamma\}$. We assume $\Gamma \vdash^{ML} E : t$ denotes that expression E is typable in ML with type t under environment Γ .

Theorem 4.2 (Type Preservation) *Let $\Gamma \vdash e : R \rightsquigarrow E$. Then $\llbracket \Gamma \rrbracket \vdash^{ML} E : \llbracket R \rrbracket$.*

Thus, we obtain that our translation scheme is sound. We can also state that the semantics of the resulting ML program is “equivalent” to the original XDuce program by “flattening” ML values.

We assume that judgments $\Delta \vdash e \Downarrow v$ describe the big-step operational semantics for XDuce. Recall we assume the longest match policy. We write $\Delta \vdash \Gamma$ to denote that a value environment Δ satisfies Γ . A formalization can be found here [28].

We make use of a standard big-step operational semantics [31] of ML specified in terms of judgments of the form $\Delta \vdash^{ML} E \Downarrow v'$.

We establish a relation among a XDuce value environment Δ and a ML value environment Δ' . We write $\Gamma \vdash \Delta \rightsquigarrow \Delta'$ iff $\Delta' = \{(x, v') \mid (x, v) \in \Delta \wedge \Gamma \vdash v : \Gamma(x) \rightsquigarrow v'\}$.

We define the flattening function `flat` turning a structured ML value into a flat XDuce value as follow:

$$\begin{aligned} \text{flat } [] &= () & \text{flat } A_U &= A & \text{flat } (L \ x) &= \text{flat } x & \text{flat } (R \ x) &= \text{flat } x \\ \text{flat } (x, y) &= (\text{flat } x, \text{flat } y) & \text{flat } (x : xs) &= ((\text{flat } x), (\text{flat } xs)) \end{aligned}$$

For instance `flat [(L A_U), (R B_U)] = (A, B)`.

Theorem 4.3 (Semantic Preservation) *Let $\Delta \vdash \Gamma$, $\Gamma \vdash e : R \rightsquigarrow E$, $\Delta \vdash e \Downarrow v$ and $\Gamma \vdash \Delta \rightsquigarrow \Delta'$. Then $\Delta' \vdash^{ML} E \Downarrow v'$ where `flat` $v' = v$.*

Theorem 4.3 implies that our translation scheme is coherent, i.e. different translations yield the same (flattened) result.

⁶ Note that we silently treat $((p_i) \downarrow_v)$ as an expression.

Additional translation rule:

$$\text{(Var-ML)} \frac{(x : \forall \bar{a}. T) \in \Gamma}{\Gamma \vdash x : [\bar{t}/\bar{a}]T \rightsquigarrow x}$$

Examples of additional proof rules:

$$\begin{array}{c} u = \lambda x.L(u' x) \\ \text{(OrL)} \frac{\vdash t_1 \leq^{u'} t_2}{\vdash t_1 \leq^u \text{Or } t_2 t_3} \end{array} \quad \begin{array}{c} u = \lambda x.R(u' x) \\ \text{(OrR)} \frac{\vdash t_1 \leq^{u'} t_3}{\vdash t_1 \leq^u \text{Or } t_2 t_3} \end{array}$$

Fig. 4. ML Function Call Extension

Corollary 4.4 (Coherence) *Let $\Gamma \vdash e : R \rightsquigarrow E_1$ and $\Gamma \vdash e : R \rightsquigarrow E_2$. Then, E_1 and E_2 are equivalent (assuming we compare flattened values).*

A silent assumption is that the coercion functions of the regular expression containment proof system obey the specific matching policy (longest-match in our case here). Note that the semantic preservation result and thus coherence may not hold for arbitrary matching policies. E.g., the latest description of XDuce [14,15] assumes a indeterministic (matching) semantics. Further coherence problems arise once we integrate additional language features such as ML-style polymorphism (see upcoming Example 5.1).

5 Extending XDuce with ML Patterns and ML Function Calls

We extend XDuce with ML types $TC \bar{t}$ and ML patterns $K p_1 \dots p_n$ as follows.

$$\begin{array}{l} \text{Patterns} \quad p ::= (K p_1 \dots p_n) \parallel x \text{ as } t \parallel (p, p) \\ \text{Mixed Types} \quad t ::= TC \bar{t} \parallel () \parallel t^* \parallel (t|t) \parallel (t, t) \end{array}$$

As usual, we assume that constructors K of user-definable types $TC \bar{t}$ are recorded in some initial environment. We assume $K : \forall \bar{a}. t_1 \rightarrow \dots \rightarrow t_n \rightarrow TC a_1 \dots a_n \in \Gamma_{init}$ where $fv(t_1, \dots, t_n) \subseteq \bar{a}$.

Expressions are silently extended with ML function calls. We assume that the types of ML functions are recorded in some initial environment when translating (extended) XDuce. This is a realistic assumption (see the discussion in Section 6).

5.1 ML Function Calls

It is fairly easy to allow for ML function calls thanks to our extensible proof system for deriving coercions. In case of polymorphic ML functions we need to build a type instance via an additional rule. Details are in Figure 4 where we also give further samples of additional proof rules. We assume that T refers to a ML type where \bar{a} are some bound variables. We write $\overline{[t/\bar{a}]}$ to denote a substitution replacing variables a_i by types t_i . It is straightforward to show that the Type Preservation Theorem 4.2 applies to a richer language which includes ML function calls. As discussed in Section 3.2, we may reject reasonable programs because the particular proof system extension may be incomplete.

A more serious issues is that we cannot maintain coherence of our translation scheme. That is, for the extended language our translation scheme may produce two programs which behave differently.

Example 5.1 Consider

```
-- extended XDuce
g :: (B,A) -> (A|B)
g x = f x

-- ML function
f :: Or (c,A) (B,c) -> c
f (L (y,_)) = y
f (R (_,y)) = y
```

Note that variable c in f 's annotation is universally quantified. Hence, in $f\ x$ we can build the instance $\text{Or } (B,A) (B,B) \rightarrow B$ by taking c to be B . Note that $\vdash (B,A) \leq^{u_1} \text{Or } (B,A) (B,B)$ and $\vdash B \leq^{u_2} (A|B)$ are derivable. For the first derivation we will need the additional rules in Figure 4. Hence, translation of g yields

```
u1 :: (B,A) -> Or (B,A) (B,B)
u2 :: B -> Or A B
g :: (B,A) -> Or A B
g x = u2 (f (u1 x))
```

However, there is another possible translation. This time we instantiate c by A . Then, the type of f becomes $\text{Or } (A,A) (B,A) \rightarrow A$. We can verify that $\vdash (B,A) \leq^{u_3} \text{Or } (A,A) (B,A)$ and $\vdash A \leq^{u_4} (A|B)$ holds. This time function g translates to

```
u3 :: (B,A) -> Or (A,A) (B,A)
u4 :: A -> Or A B
g :: (B,A) -> Or A B
g x = u4 (f (u3 x))
```

We conclude that the first translation of g applied to a pair (B,A) yields (after flattening) B whereas the second yields A .

Additional pattern inference rules:

$$(ML-x) \quad t, x \text{ as } t \vdash^{ML} t, \{x : t\}$$

$$(ML-K) \quad \frac{K : \forall \bar{a}. t_1 \rightarrow \dots \rightarrow t_n \rightarrow TC \ a_1 \dots a_n \in \Gamma_{init} \quad [t'_1/a_1, \dots, t'_m/a_m] t_i, p_i \vdash t''_i, \Gamma_i \text{ for } i = 1, \dots, n}{TC \ t'_1 \dots t'_m, K \ p_1 \dots p_n \vdash^{ML} K _T \ t''_1 \dots t''_n, \bigcup_{i=1, \dots, n} \Gamma_i}$$

$$(ML-Seq) \quad \frac{t, p_1 \vdash^{ML} \Gamma_1 \quad t_1 = ((p_1) \downarrow_t) \quad t, \{p_2, \dots, p_n\} \vdash^{ML} \{t_2, \dots, t_n\}, \{\Gamma_2, \dots, \Gamma_n\}}{t, \{p_1, p_2, \dots, p_n\} \vdash^{ML} \{t_1, t_2, \dots, t_n\}, \{\Gamma_1, \Gamma_2, \dots, \Gamma_n\}}$$

$$(Switch) \quad \frac{t, p \vdash^{ML} t, \Gamma}{t, p \vdash t, \Gamma}$$

Additional proof rule:

$$(K) \quad \frac{K : \forall \bar{a}. t_1 \rightarrow \dots \rightarrow t_n \rightarrow TC \ a_1 \dots a_n \in \Gamma_{init} \quad t'_i \leq_{d_i}^{u_i} [\bar{t}/\bar{a}] t_i \text{ for } i = 1, \dots, n \quad u(K _T \ x_1 \dots x_n) = K \ (u_1 \ x_1) \dots (u_n \ x_n) \quad d _ = \text{Nothing} \quad d(K \ x_1 \dots x_n) = \text{case } (d_1 \ x_1) \text{ of} \quad \begin{array}{l} \text{Just } v_1 \rightarrow \\ \vdots \\ \text{case } (d_n \ x_n) \text{ of} \\ \text{Just } v_n \rightarrow K _T \ v_1 \dots v_n \\ \text{Nothing} \rightarrow \text{Nothing} \end{array} \quad \text{Nothing} \rightarrow \text{Nothing}}{K _T \ t'_1 \dots t'_n \leq_d^u TC \ \bar{t}}$$

Fig. 5. Integrating ML Patterns

In fact, this result is not surprising given that similar observations can be made when extending XDuce with polymorphism [15]. We conjecture that we maintain coherence if we require that polymorphic functions are unambiguous following the approach in [15].

- (1) $[A^*], ((x \text{ as } A, \text{ xs as } A^*) : \text{ys as } [A^*]) \vdash t, \Gamma$ (Switch)
- (2) $[A^*], ((x \text{ as } A, \text{ xs as } A^*) : \text{ys as } [A^*]) \vdash^{ML}$
 $CONS\ t_1\ t_2, \Gamma_1 \cup \Gamma_2$ (ML-K)
 where $t = CONS\ t_1\ t_2, \Gamma = \Gamma_1 \cup \Gamma_2$
- (2.1) $[A^*], \text{ys as } [A^*] \vdash t_2, \Gamma_2$ (Switch)
- (2.2) $[A^*], \text{ys as } [A^*] \vdash^{ML} t_2, \Gamma_2$ (ML-x)
 where $t_2 = [A^*], \Gamma_2 = \{\text{ys} : [A^*]\}$
- (3.1) $A^*, (x \text{ as } A, \text{ xs as } A^*) \vdash t_1, \Gamma_1$ (SeqPX)
 where $t_1 = (A, A^*)$
- (3.2) $A^*, (x \text{ as } A, \text{ xs as } A^*) \vdash \Gamma_3 \cup \Gamma_4$ (PairPX)
 where $\Gamma_1 = \Gamma_3 \cup \Gamma_4, \text{lbreak } A^* \ A \ A^* = A, \text{rbreak } A^* \ A \ A^* = A^*$
- (3.2.1) $A, x \text{ as } A \vdash \Gamma_3$ (VarPX)
 where $\Gamma_3 = \{x : A\}$
- (3.2.2) $A^*, \text{xs as } A^* \vdash \Gamma_4$ (VarPX)
 where $\Gamma_4 = \{\text{xs} : A^*\}$

Fig. 6. Pattern Inference Example

5.2 ML Pattern Matching

We integrate ML patterns as follow. For each constructor $K : \forall \bar{a}. t_1 \rightarrow \dots \rightarrow t_n \rightarrow TC\ a_1 \dots a_n \in \Gamma_{init}$ we introduce a singleton type `data K_T x1 ... xn = K_T x1 ... xn`. In Figure 5, we introduce a new (regular expression containment) proof rule which models construction and deconstruction (i.e. pattern matching) based on the singleton-types information. Note that we assume patterns are evaluated from left to right.

ML pattern obey the common rules, see (ML-x), (ML-K) and (ML-Seq) in Figure 5. For simplicity, we omit the treatment of inference of pattern variables (which is possible for ML pattern variables). An important point in our formulation is that we return the singleton-type of each pattern in rule (ML-K). We assume that $((K\ p_1 \dots p_n) \downarrow_t) = K_T\ p'_1 \dots p'_n$ where $((p_i) \downarrow_t) = p'_i$ for $i = 1, \dots, n$ and similarly $((K\ p_1 \dots p_n) \downarrow_v) = K_T\ p'_1 \dots p'_n$ where $((p_i) \downarrow_v) = p'_i$ for $i = 1, \dots, n$. Thus, we can compute the appropriate coercions (in our extended proof system). It is straightforward to verify that thus we can precisely mimic ML pattern matching.

Rule (Switch) is necessary to allow for combinations of ML and regular expression patterns. Consider pattern inference for Example 2.2. We

find input type $[A^*]$ and pattern $((x \text{ as } A, xs \text{ as } A^*):ys \text{ as } [A^*])$ and need to compute the actual type t and the pattern binding Γ such that $[A^*], (x \text{ as } A, xs \text{ as } A^*):ys \text{ as } [A^*] \vdash t, \Gamma$. The necessary calculations can be found in Figure 6 (making use of rules in Figures 3 and 5). Hence, we conclude that $t = CONS(A, A^*) [A^*]$ and $\Gamma = \{x : A, xs : A^*, ys : [A^*]\}$. Based on this information we can compute the appropriate coercions. Note that in Example 2.2 we performed a slight optimization compared to our actual translation scheme (see rule (let) in Figure 2) by combining the down-cast and up-cast coercion.

In our current formulation, we prohibit a mixing of ML patterns with regular expression types, and respectively regular patterns with ML types.

Example 5.2 Here is a very simple program which cannot be dealt with in our system.

```
f1 :: A* -> ..
f1 (x as A, xs A*) = ...
f1 [] = ...
```

There is no pattern inference rule for the case of regular expression type $()$ and ML pattern $[]$. In fact, we could further extend our system to deal with such cases. However, it is questionable whether we want to allow such programs.

6 Discussion

It should be fairly straight-forward to integrate our approach in the type inference and translation process of an existing ML implementation. Note that all XDuce functions are type annotated. Hence, we can first perform type inference and translation of the ML program text. Thus, we obtain type information about all ML functions which we might call from XDuce. This allows us then to apply our type-directed translation scheme. Note that there is no (significant) overhead in terms of type inference time. Note that we derive the necessary coercions from the same operations which are already necessary for type inference of XDuce.

We would like to stress that the XDuce-ML (application) programmer does *not* need to predict the result of XDuce inference or know anything about coercions. The worst what can happen is that the regular expression containment proof system has not been sufficiently extended. In such a situation, the system could respond “containment rule $\vdash A^* \leq T A$ undefined” where T is some user-defined data type. Clearly, it is the responsibility of the designer of these data types to provide the necessary rules. In our current implementation, the regular expression proof system is formulated as a library. Hence, any extension can be done in a modular way.

A curious reader may wonder whether our translation scheme incurs a runtime penalty compared to other native XDuce style implementations. There are many possibilities to optimize our translation scheme.

For example, we have already sketched that it is sometimes possible to

combine down-cast $\vdash R_i \leq_{d_i} R$ and up-cast $\vdash R_i \leq^{u_i} R_{\Gamma_i}$ (see rule (Let) in Figure 2) if $\vdash R_{\Gamma_i} \leq_{d'_i} R$ exists (which may not always be the case, see Example 4.1). In such a situation, we can generate the ‘more optimal’ code case $(d'_i x)$ of *Just* $((p_i) \downarrow_v) \rightarrow \dots$ instead of

$$\text{case } (d_i x) \text{ of } \textit{Just} \textit{ vp}_i \rightarrow \text{let } ((p_i) \downarrow_v) = u_i \textit{ vp}_i \text{ in } \dots$$

Another optimization is to apply coercions ‘lazily’, or use a more optimized structured representation etc.

We have done a small comparison of our translation scheme against existing XDuce style implementation trying to focus on some ‘difficult’ cases. The results are encouraging. Details can be found here [26].

7 Related Work

We review some other works incorporating XML features into general-purpose languages.

Wallace and Runciman introduce HaXML [30], a Haskell combinator library to model XML data in Haskell. We adopt their encoding scheme to translate XDuce types to ML.

Thiemann [27] introduces WASH, a combinator library to generate XML values. He makes use of the Haskell type class system to check for correctness of constructed values. However, he does not consider destruction of values.

Broberg, Farre and Venningson [6] introduce a pre-processor for Haskell to mimic some of the XDuce features. In essence, their approach is untyped and they can only allow for a limited form of regular pattern matching (over lists) but do not allow for semantic subtyping.

Frisch, Castagna and Benzaken introduce CDuce [3] which is like XDuce interpreter-based and supports semantic subtyping and regular expression pattern matching. Additionally, CDuce is capable of higher-order functions and explicit function overloading.

We recently became aware of the OCamlDuce project [8] which integrates CDuce in OCaml [25]. A significant difference compared to our approach seems that CDuce style regular expression types, patterns and values are strictly separated from their OCaml counter-parts. Therefore, the semantic subtyping relation does not extend to OCaml type constructors. For example, $\mathbf{A}^* \leq [\mathbf{A}]$ is considered invalid in the OCamlDuce type system, unless user annotations are provided. It is unclear to us whether this approach applies to arbitrary user-definable types. On the other hand, we have a systematic way to deal with such cases thanks to our extensible regular expression proof system.

Extending Java and C# with XML support is also a popular topic. Gapeyev and Pierce incorporated some of the XDuce features in C# and Java, leading the XTATIC language [11]. Kempa’s and Linnemann’s work on Xobe [19] and Kirkegarrd’s, Møller’s and Schwartzbach’s XACT [21] language follow similar goals. None of these works seems related to ours given that they use a uniform

representation whereas ours is structured. Bierman’s, Meijer’s and Schulte’s work on C_ω [4] seems closer to ours. They also give a type-preserving translation scheme for C_ω . However, they do not seem to capture the full set of semantic subtype relations.

We also would like to mention work on coercive subtyping, e.g. consider work by Mitchell [24], Kierssling and Luo [20], which are superficially related to our work. However, none of these works considers semantic subtyping or deals with down-cast coercions.

There is also some connection to work on type isomorphisms. E.g., consider [2] and the references therein. It would be interesting to re-formulate some of this work in terms of a variant of our regular expression containment proof system.

8 Conclusion

We have presented a type-safe embedding of XDuce into ML based on a type and semantic preserving translation. To the best of our knowledge our work appears to be novel. The gist of our approach is fairly simple but we believe will be of high value to incorporate XDuce features into main stream functional languages such as ML and Haskell. Our translation scheme proved to be flexible enough to deal with ML functions calls and ML patterns. We could easily cater for other matching policies but we have focused here on a variant of XDuce following the longest-match policy.

We have seen that incorporating ML function calls leads to coherence issues in combination with polymorphic ML programs. We foresee several possible solutions to the problem by either demanding explicit annotations or restricting the form of polymorphism. We plan to pursue this topic in the future following the work of Hosoya and Frisch and Castagna [15].

Another interesting question is the study of systematic extensions of our proof system which retain completeness. E.g., the containment problem between a context-free and a regular language is known to be decidable. This suggests that we can always up-cast ML into XDuce values.

In another direction, we plan to investigate optimization techniques for our scheme. There has already been some work done in this area assuming a uniform representation of regular expression values [9,10,22,7]. It will be interesting to see what techniques can be applied to our setting where values have a structured representation. We conjecture that we might obtain some optimizations for free based on existing techniques such as deforestation [29].

Acknowledgments

We are grateful to the referees for their comments.

References

- [1] V. M. Antimirov. Rewriting regular inequalities. In *Proc. of FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer-Verlag, 1995.
- [2] F. Atanassow and J. Jeuring. Inferring type isomorphisms generically. In *Proc. of MPC'04*, volume 3125 of *LNCS*, pages 32–53. Springer-Verlag, 2004.
- [3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proc. of ICFP '03*, pages 51–63. ACM Press, 2003.
- [4] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in c_ω . In *Proc of ECOOP 05*, pages 287–311. Spring-Verlag, 2005.
- [5] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.
- [6] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proc. of ICFP'04*, pages 67–78. ACM Press, 2004.
- [7] B. Emir. Compiling regular patterns to sequential machines. Technical report, École Polytechnique Fédérale de Lausanne, 2004. http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200472.pdf.
- [8] A. Frisch. OCamlDuce. <http://www.cduce.org/ocaml.html>.
- [9] Alain Frisch. Regular tree language recognition with static information. In *IFIP TCS*, pages 661–674. Kluwer, 2004.
- [10] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Runtime representations for Xtatic. In *14th International Conference on Compiler Construction*, pages 43–58, April 2005.
- [11] V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP '03*, volume 2743 of *LNCS*, pages 151–175. Springer, 2003. A preliminary version was presented at FOOL '03.
- [12] Haskell 98 language report. <http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/>.
- [13] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, December 2000.
- [14] H. Hosoya. Regular expression filter. In *PLAN-X '04 Informal Proceedings*, 2004.
- [15] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *Proc. of POPL'05*. ACM Press, 2005.
- [16] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997, pages 226–244, 2000.

- [17] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL '01*, pages 67–80. ACM Press, 2001.
- [18] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.
- [19] M. Kempa and V. Linnemann. Type checking in XOBJE. In *Proc. Datenbanksysteme fur Business, Technologie und Web, BTW '03*, LNI, pages 227–246. GI, 2003.
- [20] R. Kiersling and Z. Luo. Coercions in Hindley-Milner systems. In *Proc. of Inter. Conf. on Proofs and Types*, volume 3085 of *LNCS*, pages 259–275. Springer-Verlag, 2004.
- [21] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transaction on Software Engineering*, 30(3):181–0_6, 2004.
- [22] Michael Y. Levin. Compiling regular patterns. In *Proc of ICFP 03*, pages 65–77, 2003.
- [23] K. Z. M. Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, LNCS, pages 57–73. Springer-Verlag, 2004.
- [24] J. C. Mitchell. Coercion and type inference. In *Proc of POPL 84*, pages 175–185. ACM-Press, 1984.
- [25] Objective Caml. <http://pauillac.inria.fr/ocaml/>.
- [26] M. Sulzmann and K.Z.M. Lu. A type-safe embedding of XDUCE into ML. Technical report, The National University of Singapore, 2005. forthcoming.
- [27] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4 and 5):435–468, July 2002.
- [28] S. Vansummeren. Type inference for unique pattern matching. <http://alpha.luc.ac.be/~lucg5855/files/matchinfer.ps.gz>. To appear in ACM TOPLAS.
- [29] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proc. of ESOP'88*, pages 231–248. North-Holland Publishing Co., 1988.
- [30] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP '99*, pages 148–159. ACM Press, 1999.
- [31] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

A Full translation of Example 2.1

Recall Example 2.1 from Section 2. We first consider applying the pattern inference as stated in Figure 3. We have

$$(A|B)^*, \{p_1, p_2, p_3\} \vdash \{R_1, R_2, R_3\}, \{\Gamma_1, \Gamma_2, \Gamma_3\}$$

where $p_1=(x \text{ as } A^*, (y \text{ as } B, z \text{ as } B^*)), p_2=(x \text{ as } A^*)$ and $p_3 = (x \text{ as } B^*, y \text{ as } (A|B)^*)$ and R_i s, Γ_i s are unknown.

According to rule (SeqPX), we consider patterns p_i from left to right.

First, we compute R_1 by intersecting $(A|B)^*$ with $(A^*, (B, B^*))$ which yields $(A^*, (B, B^*))$ as the result.

Second, we infer Γ_1 . Note that pattern p_1 is pair. We apply rule (PairPX) twice. We perform the following operations to infer the type for sub-patterns.

- (i) $\text{lbreak}((A|B)^*, A^*, (B, B^*)) = A^*$,
- (ii) $\text{rbreak}((A|B)^*, A^*, (B, B^*)) = (B, B^*)$,
- (iii) $\text{lbreak}((B, B^*), B, B^*) = B$ and
- (iv) $\text{rbreak}((B, B^*), B, B^*) = B^*$

Hence we have $\Gamma_1 = \{(x:A^*), (y:B), (z:B^*)\}$.

Third, we compute R_2 . We have to take into account that patterns are processed sequentially. Hence, values matching p_1 will not match p_2 . We take the difference $(A|B)^* - (A^*, (B, B^*)) = ((A^*, (B+, (A, (A|B)^*))) | A^*)$, let's call it R'_1 . By intersecting R'_1 with A^* , we have $R_2 = A^*$.

Fourth, we infer Γ_2 . Rule (VarPX) applies, hence, $\Gamma_2 = \{(x:A^*)\}$.

At last, we compute R_3 . Again we compute what is left after pattern p_2 by calculating $R'_1 - R_2$, we have $(A^*, (B+, (A, (A|B)^*)))$, let's call it R'_2 . Now R_3 is obtained by another intersection between R'_2 and $(B^*, (A|B)^*)$, the result is $(A^*, (B+, (A, (A|B)^*)))$.

Finally, we infer Γ_3 . Applying rule (PairPX) again. We calculate.

- (i) $\text{lbreak}((A^*, (B+, (A, (A|B)^*))), B^*, (A|B)^*) = B^*$ and
- (ii) $\text{rbreak}((A^*, (B+, (A, (A|B)^*))), B^*, (A|B)^*) = (A, (A|B)^*)$.

We find $\Gamma_3 = \{(x:B^*), (y:(A, (A|B)^*))\}$.

Based on the above, we can verify the following containment relations in our proof system. We leave out the exact details of the resulting proof terms for simplicity.

$$\begin{aligned} &\vdash (A|B)^* \leq_{d1} (A^*, B, B^*), \vdash (A|B)^* \leq_{d2} A^*, \vdash (A|B)^* \leq_{d3} (A^*, B+, A, (A|B)^*), \\ &\vdash B \leq^{u1} B^*, \vdash (A^*, B+, A, (A|B)^*) \leq^{u2} (B^*, A, (A|B)^*), \vdash (A, (A|B)^*) \leq^{u3} (A|B)^*, \\ &\vdash (A^*, B, B^*) \leq^{u4} (A, B, B^*), \vdash A^* \leq^{u5} A^*, \vdash () \leq^{u6} B^* \text{ and } \vdash B^* \leq^{u7} B^* \end{aligned}$$

Hence, the unabridged translation is as follow.

```

d1 :: [Or A_U B_U] -> Maybe ([A_U], (B_U, [B_U]))
d2 :: [Or A_U B_U] -> Maybe [A_U]
d3 :: [Or A_U B_U] -> Maybe ([A_U], (B_U, ([B_U], (A_U, [Or A_U B_U]))))
u1 :: B_U -> [B_U]
u2 :: ([A_U], (B_U, ([B_U], (A_U, [Or A_U B_U])))) ->
      ([B_U], (A_U, ([Or A_U B_U])))
u3 :: (A_U, ([Or A_U B_U])) -> [Or A_U B_U]
u4 :: ([A_U], (B_U, [B_U])) -> ([A_U], (B_U, [B_U]))

```

```
u5 :: [A_U] -> [A_U]
u6 :: [Phi] -> [B_U]
u7 :: [B_U] -> [B_U]

f :: [Or A_U B_U] -> [B_U]
f v = case d1 v of
  Just vp1 -> let (x,(y,z)) = u4 vp1 in (u1 y)
  Nothing -> case d2 v of
    Just vp2 -> let x = u5 vp2 in (u6 [])
    Nothing -> case d3 v of
      Just vp3 -> let (x,y) = (u2 vp3)
                    in (u7 (f (u3 y)))
      Nothing -> error "Non-exhaustive pattern."
```