OCaml + XDuce

Alain Frisch
INRIA Rocquencourt
Alain.Frisch@inria.fr

Abstract

This paper presents the core type system and type inference algorithm of OCamlDuce, a merger between OCaml and XDuce. The challenge was to combine two type checkers of very different natures while preserving the best properties of both (principality and automatic type reconstruction on one side; very precise types and implicit subtyping on the other side). Type inference can be described by two successive passes: the first one is an ML-like unification-based algorithm which also extracts data flow constraints about XML values; the second one is an XDuce-like algorithm which computes XML types in a direct way. An optional preprocessing pass, called strengthening, can be added to allow more implicit use of XML subtyping. This pass is also very similar to an ML type checker.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Languages

Keywords OCaml, XDuce, CDuce, XML, type inference, regular expression types

1. Introduction

This paper presents the core type system of OCamlDuce, a merger between OCaml [L+01] and XDuce [Hos00, HP00, HP03, HVP00]. OCamlDuce source code, documentation and sample applications are available at `http://www.cduce.org/ocaml`.

OCaml is a widely-used general-purpose multi-paradigm programming language with automatic type reconstruction based on unification techniques. XDuce is a domain specific and type-safe functional language adapted to writing transformations of XML documents. It comes with a very precise and flexible type system based on regular expression types and a natural notion of subtyping. The basic type-checking primitives for XDuce constructions are rather involved, but the structure of the type checker is simple: types are computed in a bottom-up way along the abstract syntax tree; the input and output types of functions are explicitly provided by the programmer. The high-level objective of the OCamlDuce project is to enrich OCaml with XDuce features in order to provide a robust development platform for applications that need to deal with XML but which are not necessarily focused on XML.

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

The challenge was to combine two type checkers of very different natures while preserving as much as possible the best properties of both (principality and automatic type reconstruction on one side; very precise types and implicit subtyping on the other side).

Our main guideline was to design a type system which can be implemented by reusing existing implementations of OCaml and CDuce [BCF03, Fri04]. (CDuce can be seen as a dialect of XDuce with first-class and overloaded functions – for the merger with OCaml, we don't consider these extra features). Because of the complexity of OCaml's type system, it was out of question to reimplement it. The typing algorithm we describe in this paper has been successfully implemented simply by combining a slightly modified OCaml type checker with the CDuce type checker, and by adding some glue code. As a result, OCamlDuce is a strict extension of OCaml: programs which don't use the new features will be treated exactly the same by OCaml and OCamlDuce. It is thus possible to compile any existing OCaml library with OCamlDuce. Also, we believe our modifications to the OCaml compiler are small enough to make it easy to maintain OCamlDuce in sync with future evolutions of OCaml. Our experience so far confirms that: OCamlDuce was initially developed over OCaml 3.08.3, and then adapted without any problem for each release until the current 3.09.2.

Another guideline in the design of OCamlDuce was that XDuce programs should be easily translatable to OCamlDuce in a mechanical way. In XDuce, all the functions are defined at the toplevel and comes with an explicit signature. We can obtain an OCamlDuce program by some minor syntactical modifications (the new constructions in the language are delimited to avoid grammatical overloading of notations). Explicit function signatures are simply translated to type annotations.

The design goals pushed us into the direction of simplicity. We choose to segregate XDuce values from regular ML values. Of course, a constructed ML value can contain nested XDuce values, but from the point of view of ML, XDuce values are black boxes, and similarly for types. Also, we decided not to have parametric polymorphism on XDuce types. A type variable can of course be instantiated to an XDuce type (or to a type which contains a nested XDuce type), but it is not possible to force a generalized variable to be instantiated only to XDuce types or to use a type variable within an XDuce type. The technical presentation introduces a notion of foreign type variables, but they are nothing more than a technical device for inferring ground XDuce types.

Overview In Section 2, we give some intuitions about the behavior of OCamlDuce's type-checker.

The formalization of the type system will be developed by abstracting away from details about XDuce. In Section 3, we introduce an abstract notion of *extension* (foreign types and foreign operators) and show of XDuce can be seen as an extension. In Section 4, we present the type-system and type inference algorithm for a calculus made of ML [Mil78, Dam85] plus an arbitrary extension. The basic idea is to rely on standard techniques for ML type inference. Indeed, we start from a type system which is an instance of

ML where foreign types are considered as atomic types and foreign operators are explicitly annotated with their input and output types. Then we present an algorithm to *infer* these annotations. This algorithm is described as two successive passes: the first one is a slightly modified version of an ML type-checker, and the second one is a simple forward computation on foreign types.

In Section 5, we present a preprocessing pass, called strengthening, whose purpose is to make more programs accepted by the type system by allowing implicit use of subtyping.

In Section 6, we present other details of the concrete integration in OCaml. In Section 7, we compare our approach to related works.

2. An example

In this section, we illustrate the behavior of OCamlDuce's typechecker on the following code snippet:

```
let f x = match x with
  {{ [ (y::<a>_ | _)* ] }} -> {{ y @ y }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ] }};
      {{ [ <a>[<c>[]] ] }} ]
```

The example is intended to illustrate the use of the OCaml type checker to perform a data-flow analysis of XML values, and also how OCaml features (here, higher-order functions and data-structures) interact with XDuce features.

Double curly-braces `{{...}}` are used in OCamlDuce only to avoid ambiguities in the grammar; they carry no typing information. For instance, the symbol `@` used for list concatenation in OCaml is re-used for denote XML sequence concatenation. Similarly, the square brackets `[...]` are used both to denote OCaml list literals (whose elements are separated by semi-colons) and XML sequences literals when used within double curly braces (their elements are separated by whitespace). XML element literals are written in the form `<tag>content`.

The first line of the program above declares a function `f` which consists of an XML pattern matching on its argument, with a single branch. The XML pattern `p = [ (y::<a>_ | _)* ]` extracts from an XML sequence all the elements with a tag `<a>` and put them (in order) in the capture variable `y`. The function is then used twice, including once indirectly through a call to the function `List.map` (from the OCaml standard library) of type $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$. For the purpose of explaining type-checking, we will rewrite the body of the function `f` as:

```
let f x =
  let y' = match[y;p](x) in
  {{ y' @ y' }}
```

The `y` and `p` parameters of the `match` operator represent the capture variable under consideration and the pattern itself.

In OCamlDuce, XML values (elements, sequences, ...) and regular OCaml values are kept apart. An XML value can of course appear as part of an OCaml value (e.g. the XML elements which are put into an OCaml list), but an OCaml value cannot appear within an XML value. The same applies to types: an XML type can appear as part of a complex OCaml type expression, but the converse is impossible. XML operators can be applied to XML values and return new XML values. In the example, we can see three kind of XML operators: XML literals (no argument), XML concatenation (two arguments), and XML pattern matching (one argument).

The basic idea of the OCamlDuce type system is to infer XML types for the inputs and outputs of XML operators. This is done by introducing internally a new kind of type variables, called XML type variables. Before proper type-checking starts, each XML operator used in the program is annotated with fresh XML type variables (in subscript position for the inputs, and in superscript position for the outputs):

```
let f x =
  let y' = match[y;p]^{ι2}_{ι1}(x) in
  {{ y' @^{ι5}_{ι3,ι4} y' }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ]^{ι6} }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ]^{ι7} }};
      {{ [ <a>[<c>[]] ]^{ι8} }} ]
```

The regular OCaml type-checker is then applied. It gives to each XML operator an arrow type following the annotations and then proceeds as usual (generalizes types of `let`-bound identifiers, instantiates ML type-schemes when an identifier is used, and performs unifications to make type compatible).

For instance, the concatenation operator in our example is given the type $\iota_3 \to \iota_4 \to \iota_5$, and the type-checker performs the following unifications: $\iota_2 = \iota_3 = \iota_4$ (the type for `y'`), $\iota_1 = \iota_6 = \iota_7 = \iota_8$ (the type for the argument of `f`). It also produces the following types for the top-level identifiers:

```
val f  : ι1 → ι5
val z1 : ι5
val z2 : ι5 list
```

Of course, we must still instantiate the XML type variables with ground XML types. Each occurrence of an XML operator in the program gives one constraint on the instantiation. Indeed, we can interpret each n-ary operator as as n-ary function from XML types to XML types. If we choose ι_1 and ι_2 as representatives for their classes of equivalence modulo unification, the program is:

```
let f x =
  let y' = match[y;p]^{ι2}_{ι1}(x) in
  {{ y' @^{ι5}_{ι2,ι2} y' }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ]^{ι1} }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ]^{ι1} }};
      {{ [ <a>[<c>[]] ]^{ι1} }} ]
```

from which we read the following constraints:

$$\begin{array}{lcl} \iota_2 & \geq & \texttt{match[y;p]}(\iota_1) \\ \iota_5 & \geq & \iota_2 @ \iota_2 \\ \iota_1 & \geq & \texttt{[<a>[] <b>[] <a>[<b>[]]]} \\ \iota_1 & \geq & \texttt{[<a>[<a>[]]]} \\ \iota_1 & \geq & \texttt{[<a>[<c>[]]]} \end{array}$$

In this system, we consider `match[y;p]` as a function from XML types to XML types, given by XDuce's type inference algorithm for pattern matching. Similarly, the operator `@` is now interpreted as a function from pair of types to types.

The set of constraints generates dependencies between variables. We say that a variable on a left-hand side of a constraint depends on variables of the right-hand side. In our example, the graph of dependencies between variables is acyclic. In this case, we can topologically order the variables and find the least possible ground XML type for each of them: we assign to a variable the

union of all its lower bounds. In the example, we will thus compute the following instantiation:

$$\begin{array}{lcl} \iota_1 & = & \texttt{[R1]} \\ \iota_2 & = & \texttt{match[y;p]([R1])} = \texttt{[R2]} \\ \iota_5 & = & \iota_2 @ \iota_2 = \texttt{[R2 R2]} \end{array}$$

where `R1` is the regular expression

```
(<a>[]<b>[]<a>[<b>[]])|<a>[<a>[]|<c>[]]]
```

and `R2` is the regular expression

```
(<a>[]<a>[<b>[]])|<a>[<a>[]|<c>[]]
```

Type-checking is over: we have found an instantiation for XML type variables which satisfies all the constraints. In essence, the type-checker has collected all the XML types that can flow to the input of the function, and then type-checked the body of the function with the union of all these types. In general, the OCaml type-checker is used to infer the data flow of XML values in the programs. The way to solve the resulting set of constraints by forward computation corresponds roughly to the structure of the XDuce type-checker.

Implicit subtyping Let's see what happens if we add an explicit type constraint for `z1`:

```
let z1 : {{[ <a>_* ]}} =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
```

(The type `[ <a>_* ]` denotes the set of all sequences made of XML elements with tag `<a>`.) The algorithm described above will infer a much less precise type for `z2` as well, which is unfortunate. The reason is that the OCaml type-checker unifies ι_5 with `[ <a>_* ]`. Basically, the unification-based type system forgets about the direction of the data flow. There is some dose of implicit subtyping in the algorithm, but only for the result of XML operators (because of the way we interpret them as subtyping - not equality - constraints).

In order to address this lack of implicit subtyping, we use a preprocessing pass whose purpose is to detect which sub-expressions are of kind XML and to introduce around them a special unary XML operator `id` which behaves semantically as the identity, but allows subtyping. This preprocessing pass would rewrite the definition for `z1` as:

```
let z1 : {{[ <a>_* ]}} =
  id^ι10_ι9(f {{[<a>[] <b>[] <a>[<b>[]]]^ι1}})
```

The variable ι_9 will then be unified with ι_5 and ι_{10} with `[<a>_*]`. The additional constraint corresponding to the `id` operator is thus simply:

$$\texttt{[<a>\_*]} \geq \iota_5$$

which is satisfied by the same instantiation for ι_5 as in the original example. As a consequence, the type for `z2` is not changed.

The preprocessing pass is quite simple. It consists of another run of the OCaml type-checker, where all the XML types are considered equal. This allows to identify which sub-expressions are of kind XML. Section 5 describes this pass formally.

Breaking cycles The key condition which allowed us to compute an instantiation for XML type variables in the example was the acyclicity of the constraints. As we will see in Section 4, the acyclicity condition corresponds to the structure of the XDuce's type checker, which does not try to infer argument and result types for recursive functions. Of course, this acyclicity property does not always hold. For instance, let us extend the original example with the following definition:

```
let z3 = f z1
```

Without the preprocessing pass mentioned above, this line would force the OCaml type-checker to unify ι_1 and ι_5. The preprocessing pass actually replaces this definition by:

```
let z3 = f id^ι12_ι11(z1)
```

The type-checker then unifies ι_{11} with ι_5 and ι_{12} with ι_1; the resulting constraint for `id` is thus:

$$\iota_1 \geq \iota_5$$

which corresponds to the fact that the output of `f` can flow back to its input. We observe that the set of constraints has now a cycle between variables ι_1, ι_5 and ι_2.

Our type-system cannot deal with such a situation. It would issue an error explaining that the inferred data flow on XML values has a cycle. The programmer is then required to break explicitly this cycle by providing more type annotations. For instance, the programmer could use the same annotation as above on `z1`:

```
let z1 : {{[ <a>_* ]}} =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
```

or maybe he will prefer to annotate the input or output type of `f`.

3. Abstract extension of ML

The previous section explained the behavior of OCamlDuce's type checker on a example. It should be clear from this example that the type system is largely independent of the actual definitions of values, types, patterns and operators from XDuce and could be applied to other extensions of OCaml as well. In this section, we will thus introduce an abstract notion of extension and show how XDuce fits into this notion. This more abstract presentation should help the reader to understand the structure of the type checker, without having to care about the details of XDuce's type system.

Definition 1. *An* **extension** X *is defined by:*

- *a set of ground foreign types* $\mathcal{T}$*;*
- *a subtyping relation* $\leq$ *on* $\mathcal{T}$*, which is a partial order with a binary least-upper bound operator* $\sqcup$*;*
- *a set of foreign operators* $\mathcal{O}$*;*
- *for each operator* $\texttt{o} \in \mathcal{O}$*: an arity* $n \geq 0$ *and an abstract semantics* $\hat{\texttt{o}} : \mathcal{T}^n \to \mathcal{T}$ *which is monotone with respect to* $\leq$ *on each of its argument.*

We use the meta-variable τ to range over ground foreign types. The foreign operators are used to model both foreign value constructors and operations on these foreign values. Since we are not going to formalize the dynamic semantics, we don't need to distinguish between these two kinds of operators.

The monotonicity requirement on the abstract semantics ensures the completeness of our resolution strategy for constraints (combining the lower bounds for each variable with the $\sqcup$ operator).

We don't formalize in this paper the operational semantics of operators. Instead, we assume informally that it is given and compatible with the abstract semantics.

XDuce as an extension Now we show how XDuce features can be seen as an extension. We consider here a simple version of XDuce, with the following kind of expressions: element constructor $\texttt{a}[\texttt{e}]$ (seen as a unary operator), empty sequence `()`, concatenation $\texttt{e}_1, \texttt{e}_2$, and pattern matching $\texttt{match e with p} \to \texttt{e} \mid \ldots \mid \texttt{p} \to \texttt{e}$. OCamlDuce is actually built on CDuce, which considers for instance XML element constructors as ternary operators (the tag and a specification for XML attributes are also considered as arguments).

The meta-variable $\mathtt{p}$ ranges over XDuce patterns. We don't need to recall here what they are. We just need to know that for any pattern $\mathtt{p}$ we can define an accepted type $\wr\mathtt{p}\wr$, a finite set of capture variables $\mathtt{Var}(\mathtt{p})$, and for any type τ and any variable $\mathtt{x}$ in $\mathtt{Var}(\mathtt{p})$, a type $\mathtt{match}[\mathtt{x};\mathtt{p}](\tau)$ (which represents the set of values possibly bound to $\mathtt{x}$ when the inspected value is in τ and the pattern succeeds).

Here is the formal definition of an extension X for XDuce. The set of ground foreign types $\mathcal{T}$ is the set of regular XML tree languages, that is XDuce types quotiented by the equivalence induced by the subtyping relation (types with the same set-theoretic interpretation are considered equal). The subtyping relation $\leq$ directly comes from XDuce. The least-upper bound operator $\sqcup$ corresponds to XDuce's union type constructor (usually written |). We use the following families of foreign operators:

- a unary operator for each XML label $\mathtt{a}$, a unary operator;
- a binary operator corresponding to the concatenation;
- a constant operator corresponding to the empty sequence;
- for any pattern $\mathtt{p}$ and variable $\mathtt{x}$ in $\mathtt{Var}(\mathtt{p})$, a unary operator written $\mathtt{match}[\mathtt{x};\mathtt{p}]$ (its semantics is to return the value bound to $\mathtt{x}$ when matching its argument against the pattern $\mathtt{p}$).

The abstract semantics for all these operators follows directly from XDuce's theory.

Element constructor, concatenation and the empty sequence expressions can directly be seen as foreign operators. This is not the case for a pattern matching $\mathtt{match\ e\ with\ p_1} \to \mathtt{e_1} \mid \ldots \mid \mathtt{p}_n \to \mathtt{e}_n$. We are going to present an encoding of pattern-matching in terms of operators and normal ML expressions. This encoding is rather heavy; in practice, the implementation deals with pattern matching directly.

First, we define the translation $\overline{\mathtt{p} \to \mathtt{e}}$ of a single branch where $\mathtt{Var}(\mathtt{p}) = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ as the expression:

$$\begin{array}{l}\lambda\mathtt{x}.\\ \mathtt{let\ x}_1 = \mathtt{match}[\mathtt{x}_1;\mathtt{p}]\mathtt{x\ in}\\ \ldots\\ \mathtt{let\ x}_n = \mathtt{match}[\mathtt{x}_n;\mathtt{p}]\mathtt{x\ in}\\ \mathtt{e}\end{array}$$

Then, the translation of $\mathtt{match\ e\ with\ p_1} \to \mathtt{e_1} \mid \ldots \mid \mathtt{p}_n \to \mathtt{e}_n$ is defined as:

$$\begin{array}{l}\mathtt{let\ x = e\ in}\\ \mathtt{dispatch}[\tau_1, \ldots, \tau_n]\ \mathtt{x\ x}\ (\overline{\mathtt{p}'_1 \to \mathtt{e}_1})\ \ldots\ (\overline{\mathtt{p}'_n \to \mathtt{e}_n})\end{array}$$

where $\tau_i = \wr\mathtt{p}_i\wr$ and $\mathtt{p}'_i = \mathtt{p}_i \backslash (\tau_1 \sqcup \ldots \sqcup \tau_{i-1})$ (the restriction of $\mathtt{p}_i$ to values which do not match any pattern from an preceding branch). The $\backslash$ operator denotes set-theoretic type difference (in XDuce, it is a meta-operation; in CDuce, it is part of the type algebra). We have used in this translation a new built-in ML constant $\mathtt{dispatch}[\tau_1, \ldots, \tau_n]$ of type scheme: $\forall\alpha.(\tau_1 \sqcup \ldots \sqcup \tau_n) \to \alpha \to (\alpha \to \beta) \to \ldots \to (\alpha \to \beta) \to \beta$, which we assume to be present in the initial typing environment. Its intuitive semantics is to drop the first argument (it is used only to force the type-checker to verify that $\mathtt{x}$ has type $\tau_1 \sqcup \ldots \sqcup \tau_n$, which corresponds to the XDuce's pattern matching exhaustivity condition), and to call the k^{th} functional argument ($1 \leq k \leq n$) on the second argument when k is the smallest integer such that this argument has type τ_k.

In principle, the technique described in this paper could be used to integrate many of the existing extensions to the original XDuce design (such as attribute-element constraints [HM03] or XML filters [Hos04]) without any additional theoretical complexity. In its current form, however, OCamlDuce integrates all the CDuce extensions except overloaded functions: XML attributes as extensible records, sequence and tree pattern-based iterators, strings as sequences of characters (hence string regular expression types and patterns), etc.

4. Type system

In this section, we present a type system and a type inference algorithm for a fixed extension X. This section and the following one do not depend on a particular extension X. Our language will be the kernel of an ML-like type system, enriched with types and operators from the extension X.

Types and expressions The syntax of types and expressions is given in Figure 1. We use a vector notation to represent tuples. E.g. $\vec{\mathtt{t}}$ stands for an n-tuple $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$.

We assume a set of ML type constructors, ranged over by the meta-variable $\mathtt{P}$. Each ML type constructor comes with a fixed arity and we assume all the types to be well-formed with respect to these arities. The arrow $\to$ is considered as a distinguished binary type constructor for which we use an infix and right-associative syntax.

We assume two infinite families of type variables and foreign type variables, respectively ranged over by the meta-variables α and ι. Let us emphasize that a foreign type variable cannot appear within a ground foreign type τ. In an expression $\exists\alpha.\mathtt{e}$, the type variable α is bound in $\mathtt{e}$. Expressions are considered modulo α-conversion of bound type variables. The construction $\exists\alpha.\mathtt{e}$ thus serves to introduce a fresh type variable α to be used in a type annotation somewhere in $\mathtt{e}$.

Foreign operators are annotated with the type of their arguments (in subscript position) and of their result (in superscript); the number of type arguments is assumed to be coherent with the arity of the foreign operator. However, in practice, the source language does not include the annotations: they are automatically filled with fresh foreign type variables by the compiler (we also use this convention in this paper for some examples). Putting the annotations in the syntax is just a way of simplifying the presentation. The main technical contribution of the paper is an algorithm to infer ground foreign types for the foreign type variables.

The ML(X) fragment We call ML(X) the fragment of our calculus where all the foreign types are restricted to be ground. Figure 2 defines a typing judgment $\Gamma \vdash \mathtt{e} : \mathtt{t}$ for ML(X). It is exactly an instance of the ML type system [Mil78, Dam85] if we see ground foreign types as atomic ML types and ground-annotated foreign type operators $\mathtt{o}^{\vec{\tau}}_{\tau}$ as built-in ML constants or constructors (we also introduce explicit type annotation and type variable introduction). We recall classical notions of type scheme, typing environment and generalization. A **type scheme** is a pair of a finite set $\bar{\alpha}$ of type variables and of a type $\mathtt{t}$, written $\forall\bar{\alpha}.\mathtt{t}$. The variables in $\bar{\alpha}$ are considered bound in this scheme. We write $\sigma \ll \mathtt{t}$ if the type $\mathtt{t}$ is an instance of the type scheme σ. A **typing environment** is a finite mapping from program variables to type schemes. The **generalization** of a type $\mathtt{t}$ with respect to a typing environment Γ, written $\mathtt{Gen}_\Gamma(\mathtt{t})$ is the type scheme $\forall\bar{\alpha}.\mathtt{t}$ where $\bar{\alpha}$ is the set of variables which are free in $\mathtt{t}$, but not in Γ.

Type-soundness of the ML(X) fragment We *assume* that a *sound* operational semantics is given for the ML(X) calculus. This amounts to defining δ-reduction rules for the $\mathtt{o}^{\vec{\tau}}_{\tau}$ operators which are coherent with the abstract semantics for the foreign operators. Well-typed expressions in ML(X) (in an empty typing environment, or an environment which contains built-in ML operators) cannot go wrong. We also assume that the operational semantics for an $\mathtt{o}^{\vec{\tau}}_{\tau}$ operator depends only on $\mathtt{o}$, not on the annotations $\vec{\tau}$, τ. This allows us to lift the semantics of ML(X) to the full calculus of Figure 1.

Typing problems A **substitution** ϕ is an idempotent function from types to types that maps type variables to types, foreign type

$\varepsilon ::=$	Foreign types:	$\mathtt{e} ::=$	Expressions:
τ	ground foreign type	$\mathtt{x}$	program variable
ι	foreign type variable	$\lambda\mathtt{x}.\mathtt{e}$	abstraction
		$\mathtt{e}\ \mathtt{e}$	application
$\mathtt{t} ::=$	Types:	$\mathtt{let\ x = e\ in\ e}$	local definition
$\mathtt{P}\ \vec{\mathtt{t}}$	constructed	$(\mathtt{e} : \mathtt{t})$	annotation
α	type variable	$\exists\alpha.\mathtt{e}$	existential variable
ε	foreign type	$\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$	foreign operator

Figure 1. Types and expressions

$$\frac{\Gamma(\mathtt{x}) \ll \mathtt{t}}{\Gamma \vdash \mathtt{x} : \mathtt{t}} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1 \to \mathtt{t}_2 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{t}_1}{\Gamma \vdash \mathtt{e}_1\mathtt{e}_2 : \mathtt{t}_2} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1 \qquad \Gamma, \mathtt{x} : \mathtt{Gen}_\Gamma(\mathtt{t}_1) \vdash \mathtt{e}_2 : \mathtt{t}_2}{\Gamma \vdash \mathtt{let\ x = e}_1\ \mathtt{in\ e}_2 : \mathtt{t}_2}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash (\mathtt{e} : \mathtt{t}) : \mathtt{t}} \qquad \frac{\Gamma \vdash \mathtt{e}[\mathtt{t}_0/\alpha] : \mathtt{t}}{\Gamma \vdash \exists\alpha.\mathtt{e} : \mathtt{t}} \qquad \frac{\hat{\mathtt{o}}(\vec{\tau}) \leq \tau}{\Gamma \vdash \mathtt{o}^{\tau}_{\vec{\tau}} : \tau_1 \to \ldots \to \tau_n \to \tau}$$

Figure 2. Type system for the ML(X) fragment

variables to foreign types, ground foreign types to themselves, and that commutes with ML type constructors. We use a post-fix notation to denote a capture-avoiding application of this substitution to typing environments, expressions, types or constraints.

A substitution ϕ_1 is **more general** than a substitution ϕ_2 if $\phi_2 = \phi_2 \circ \phi_1$[1].(Or equivalently, because substitutions are idempotent: there exists a substitution ϕ such that $\phi_2 = \phi \circ \phi_1$.)

A **typing problem** is a tuple $(\Gamma, \mathtt{e}, \mathtt{t})$. (Usually, $\mathtt{t}$ is a fresh type variable.) A **solution** to this problem is a substitution ϕ such that $\Gamma\phi \vdash \mathtt{e}\phi : \mathtt{t}\phi$ is a valid judgment in ML(X). We will now rephrase this definition in terms of a typing judgment on the full calculus. This judgment $\Gamma \vdash_X \mathtt{e} : \mathtt{t}$ is defined by the same rules as in Figure 2, except for foreign operators, for which we take:

$$\overline{\Gamma \vdash_X \mathtt{o}^{\varepsilon}_{\vec{\varepsilon}} : \varepsilon_1 \to \ldots \to \varepsilon_n \to \varepsilon}$$

Typing environment and type schemes that are used in the judgment $\vdash_X$ are allowed to contain foreign type variables. We say that ϕ is a **pre-solution** to the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ if the assertion $\Gamma\phi \vdash_X \mathtt{e}\phi : \mathtt{t}\phi$ holds. Of course, the new rule for foreign operators forgets the constraints that relates the input and output types of foreign operators. In order to ensure type soundness, we must also enforce these constraints.

Formally, we define a **constraint** $\mathtt{C}$ as a finite set of annotated foreign operators $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$. We write $\Vdash \mathtt{C}$ if all the elements of $\mathtt{C}$ are of the form $\mathtt{o}^{\tau}_{\vec{\tau}}$ with $\hat{\mathtt{o}}(\vec{\tau}) \leq \tau$. For an expression $\mathtt{e}$, we collect in a constraint $\mathtt{C}(\mathtt{e})$ all the instances of foreign operators $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$ that appear in $\mathtt{e}$. Note that for any substitution ϕ, we have $\mathtt{C}(\mathtt{e})\phi = \mathtt{C}(\mathtt{e}\phi)$.

We are ready to rephrase the notion of solution.

Lemma 1. *A substitution ϕ is a solution to the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ if and only if the following three assertions hold:*

- *$\Gamma\phi$, $\mathtt{e}\phi$ and $\mathtt{t}\phi$ do not contain foreign type variables;*
- *ϕ is a pre-solution to the typing problem;*
- $\Vdash \mathtt{C}(\mathtt{e}\phi)$.

Type soundness Type soundness for our calculus is a trivial consequence of the type soundness assumption for the ML(X) fragment. Indeed, we can see a solution ϕ to a typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ as an *elaboration* into a well-typed program in this fragment.

[1] As usual, the symbol $\circ$ denotes the composition of functions. The composition of two substitutions is not in general a substitution.

Type inference Let us consider a fixed typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$. We want to find solutions to this problem. Thanks to Lemma 1, we will split this task into two different steps:

- find a most-general pre-solution ϕ_0;
- instantiate the remaining foreign type variables so as to satisfy the resulting constraint.

It is almost straightforward to adapt unification-based existing algorithms for ML type inference (and their implementations) to compute a most general pre-solution if there exists a pre-solution, or to report a type error otherwise. Indeed, the typing judgment $\vdash_X$ is very close to a normal ML type system. In particular, it satisfies a substitution lemma: if $\Gamma \vdash_X \mathtt{e} : \mathtt{t}$, then $\Gamma\phi \vdash_X \mathtt{e}\phi : \mathtt{t}\phi$ for any substitution ϕ.

Of course, if the typing problem has no pre-solution, it has no solution as well. For the remaining of the discussion, we assume given a most general pre-solution ϕ_0. Let us write V for the set of foreign type variables that appear in $(\Gamma\phi_0, \mathtt{e}\phi_0, \mathtt{t}\phi_0)$ and $\mathtt{C}_0$ for the constraint $\mathtt{C}(\mathtt{e}\phi_0)$.

A solution to the typing problem is in particular a pre-solution. As a consequence, a substitution ϕ is a solution if and only if $\phi = \phi \circ \phi_0$ and if it maps foreign type variables in V to ground foreign types in such a way that $\Vdash \mathtt{C}_0\phi$. The "minimal" modification we need to bring to ϕ_0 to get a solution is to instantiate variables in V so as to validate $\mathtt{C}_0$. Formally, we define a **solvent** as a function $\rho : V \to \mathcal{T}$ such that $\Vdash \mathtt{C}_0\rho$. To any solvent ρ, we can associate a solution ϕ defined by $\mathtt{t}\phi = \mathtt{t}\phi_0\rho$ and any solution is less general than the solution obtained this way from some solvent. In particular, a solution exists if and only if a solvent exists. So we are now looking for a solvent.

We won't give a *complete* algorithm to check for the existence of a solvent. This would lead to difficult constraint solving problems which might be undecidable (this of course depends on the extension X). Even if they are decidable for a given extension, they might be intractable in practice and so we prefer to stick to our design guideline that type inference shouldn't be significantly more complicated than both ML type inference and XDuce-like type inference. XDuce computes in a bottom-up way, for each subexpression, a type which over approximates all the possible outcomes of this sub-expression. The basic operations and their typing discipline corresponds respectively to our foreign operators and their static semantics. XDuce's type system uses subsumption only

when necessary (e.g. to join the types of the branches of a pattern matching, or when calling a function). So we can say that XDuce tries to compute a minimal type for each sub-expression, by applying basic type-checking primitives. We will do the same, and to make it work, we need some acyclicity property, which corresponds to the bottom-up structure of XDuce's type checker.

Definition 2. *Let $\mathtt{C}$ be a constraint. We write $\iota_1 \overset{\mathtt{C}}{\rightsquigarrow} \iota_2$ if $\mathtt{C}$ contains an element $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$ such that $\iota_2 = \varepsilon$ and ι_1 appears in $\vec{\varepsilon}$. We say that $\mathtt{C}$ is* **acyclic** *if the directed graph defined by this relation is acyclic.*

Our type inference algorithm only deals with the case of an acyclic constraint $\mathtt{C}_0$ (this condition does not depend on the particular choice of the most general pre-solution). If the condition is not met, we issue an error message. It is not a type error with respect to the type system, but a situation where the algorithm is incomplete.

Remark. *The acyclicity criterion is of course syntactical (it does not depend on the semantics of constraints but on their syntax), but it is not defined in terms of a specific inference algorithm. Instead, it is defined in terms of the most-general pre-solution of an ML-like type system. In particular, it does not depend on implementation details such as the order in which sub-expression are type-checked.*

Below we furthermore assume that $\mathtt{C}_0$ is acyclic. We define the function $\rho_0 : V \to \mathcal{T}$ by the following equation:

$$\forall \iota \in V.\ \iota\rho_0 = \bigsqcup \{\hat{\mathtt{o}}(\vec{\varepsilon}\rho_0) \mid \mathtt{o}^{\iota}_{\vec{\varepsilon}} \in \mathtt{C}_0\}$$

The acyclicity condition ensures that this definition is well-founded and yields a unique function ρ. Furthermore, this function is a solvent if and only if the typing problem has a solution. To check this property, only constraints whose right-hand side is a ground foreign type need to be considered:

$$(1)\quad \forall \mathtt{o}^{\tau}_{\vec{\varepsilon}} \in \mathtt{C}_0.\ \hat{\mathtt{o}}(\vec{\varepsilon}\rho_0) \leq \tau$$

Also, any other solvent ρ is such that:

$$\forall \iota \in V.\ \iota\rho_0 \leq \iota\rho$$

In other words, under the acyclicity condition, we can check in a very simple way whether a given typing problem has a solution, and if this is the case, we can compute the smallest solvent (for the point-wise extension of the subtyping relation). This computation only involves one call to the abstract semantics for each application of a foreign operator in the expression to be type-checked.

Remark. *In some cases, it is possible to find manifest type errors even when the constraint is not acyclic. In practice, the computation of ρ_0, the verification of (1), and the check for acyclicity can be done in parallel, e.g. with a deep-first graph traversal algorithm. It can detect some violation of (1) before a cycle. In this case, we know that the typing problem has no solution, and thus a proper type error can be issued.*

Manually working around the incompleteness When the algorithm described above infers a cyclic constraint, it cannot detect whether the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ has a solution or not. However, we have the following property. If a solution ϕ exists, then we can always produce an expression $\mathtt{e}'$ by adding annotations to $\mathtt{e}$ such that the algorithm succeeds for the typing problem $(\Gamma, \mathtt{e}', \mathtt{t})$ and that ϕ is equivalent (for the equivalence induced by the more-general ordering) to the solution ϕ_0 computed by the algorithm.

In other words, even if the algorithm is not complete (because of the acyclicity condition) and makes a choice between most-general solutions (the smallest one for the subtyping relation), for any solution to a typing problem, the programmer can always add annotations so that the algorithm infers this very solution (or an equivalent one).

Partial operators The foreign operators were assumed to be total. This means they should apply to any foreign value. We can simulate partial operators by adding a new top element $\top$ to the set of ground foreign types $\mathcal{T}$, and by requiring the abstract semantics of operators to be such that whenever an argument is $\top$, the result is also $\top$. Since the typing algorithm infers the smallest solvent for foreign type variables, we can simply look at it and check that no foreign type variable is mapped to $\top$.

5. Strengthening

As we mentioned above, we can see the type system for the calculus as an elaboration into its ML(X) fragment, which immediately gives type soundness.

In this section, we consider another elaboration from the calculus into itself. Namely, this elaboration is intended to be used as a preprocessing pass (rewriting expressions into expressions) in order to make the type system accept more programs. We call this elaboration procedure strengthening.

The issue addressed by strengthening is a lack of implicit subsumption in our calculus. We already hinted at this issue in Section 2. We will now give more examples.

Subsumption missing in action We consider the typing problem $(\Gamma_1, \mathtt{e}_1, \beta)$ where $\Gamma_1 = \{\mathtt{x} : \tau_1, \mathtt{y} : \tau_2, \mathtt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\mathtt{e}_1 = \mathtt{f\ x\ y}$. It admits a solution if and only if $\tau_1 = \tau_2$. In a system with implicit subtyping, we might expect to give type $\tau = \tau_1 \sqcup \tau_2$ to both $\mathtt{x}$ and $\mathtt{y}$, so that the application succeeds and the result type is τ.

Similarly, the expression $(\lambda\mathtt{x}.\mathtt{x} : \tau_1 \to \tau_2)$ is not well-typed even if $\tau_1 \leq \tau_2$ (unless $\tau_1 = \tau_2$).

A naive solution Let us see how to implement the amount of implicit subtyping we need to make these examples type-check. The following rule could be a reasonable candidate as an addition to the type system (we write $\vdash_{\leq}$ for the new typing judgment):

$$\frac{\Gamma \vdash_{\leq} \mathtt{e} : \tau \qquad \tau \leq \tau'}{\Gamma \vdash_{\leq} \mathtt{e} : \tau'}$$

A concrete way to see this rule is that any subexpression $\mathtt{e}'$ can be magically transformed to the application $\mathtt{id}^{\iota 2}_{\iota 1}\ \mathtt{e}'$, where $\mathtt{id}$ is a distinguished foreign operator such that $\hat{\mathtt{id}}(\tau) = \tau$ and ι_1, ι_2 are fresh foreign type variables.

The type system extended with this rule would accept the examples given above to illustrate the lack of implicit subsumption. However, this rule as it stands would add a lot of complexity to the type inference algorithm. As a matter of fact, the type system would not admit most-general pre-solutions anymore. We can see this on a very simple example with the typing problem $(\{\mathtt{x} : \tau\}, \mathtt{x}, \alpha)$. We could argue that a more liberal definition of being more-general should allow some dose of subtyping. So let us consider the more complex example $\Gamma_3 = \{\mathtt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\mathtt{e}_3 = \lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\lambda\mathtt{g}.\mathtt{g}\ (\mathtt{f\ x\ y})\ (\mathtt{f\ x\ z})$. In ML, the inferred type scheme would be $\forall\alpha, \beta.\alpha \to \alpha \to \alpha \to (\alpha \to \alpha \to \beta) \to \beta$ which forces the first three arguments to have the same type. But if the arguments turn out to be of a foreign-type, another family of types for the function is possible, namely $\forall\beta.\tau_1 \to \tau_2 \to \tau_3 \to ((\tau_1 \sqcup \tau_2) \to (\tau_1 \sqcup \tau_3) \to \beta) \to \beta$, and these types cannot be obtained as instances of the ML type scheme above.

A practical solution We will now describe a practical solution. Instead of modifying the type system by adding a new subsumption rule, we will formulate the extension as a rewriting preprocessing pass. The rewriting consists in inserting applications of the identity foreign operator $\mathtt{id}$. The challenge is then to choose which subexpressions $\mathtt{e}'$ should be rewritten to $\mathtt{id\ e}'$. If we had an oracle

to tell us so, the composition of the rewriting pass and the type system of Section 4 would be equivalent to the type system $\vdash_{\leq}$. Unfortunately, we don't have such an oracle. We could try all the possible choices of sub-expressions, and this would give a complete type-checking algorithm for the type system $\vdash_{\leq}$.

We prefer to use a computationally simpler solution. We also expect it to be simpler to understand by the programmer. The idea is to use an incomplete oracle. The oracle first runs a special version of an ML type-checker on the expression to be type-checked. This type-checker identifies all the foreign types together. The effect is to find out which sub-expressions have a foreign type in a principal derivation, that is, which sub-expression have necessarily a foreign type in all the possible derivations. The preprocessing pass consists in adding an application of the identity operator above all these sub-expressions and only those.

The important point here is that the oracle may be overly conservative. Let us consider a type variable which has been generalized in the principal derivation. In a non-principal derivation, it could have been instead instantiated to a foreign type. If this derivation had been considered instead of the principal one, the preprocessing pass would have added more applications of the identity operator. Maybe this would have been necessary in order to make the resulting expression type-check. An example is given by the expression $\texttt{let h} = \texttt{e}_3 \texttt{ in } (\texttt{h} : \tau_1 \to \tau_2 \to \tau_3 \to ((\tau_1 \sqcup \tau_2) \to (\tau_1 \sqcup \tau_3) \to \texttt{t}) \to \texttt{t})$ where $\texttt{e}_3$ is from the example above. Here, the preprocessing pass succeeds but does not change the expression because no sub-expression has a foreign type in the principal type derivation. The type-scheme inferred for h is a pure ML type-schema, which makes the type-system subsequently fail on the expression.

We believe that this restriction of the $\vdash_{\leq}$ system is reasonable. It can be implemented very simply by reusing the same type-checker as in Section 4 in a different mode (where all the foreign types can be unified). The simple examples at the beginning of this section are now accepted. Indeed, the preprocessing pass transforms the expressions to $\texttt{f } (\texttt{id x}) \ (\texttt{id y})$ and $((\lambda \texttt{x.id x}) : \tau_1 \to \tau_2)$ respectively. This allows the type system $\vdash$ to use subtyping where needed.

Properties The strengthening pass cannot transform a well-typed program into an ill-typed one. Note, however, that it might break the acyclicity condition if it was already met. See below for a way to relax the acyclicity condition.

Also, if strengthening fails, the typing problem has no pre-solution (for the typing judgment $\vdash$), and thus no solution. However, it is not true that if it succeeds, a pre-solution necessarily exists (for the new program where applications of the id operators have been added). As an example, let us consider the situation where $\Gamma = \{\texttt{x} : \tau_1 \to \tau_1, \texttt{y} : \tau_2 \to \tau_2, \texttt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\texttt{e} = \texttt{f x y}$. The preprocessing succeeds, because all the foreign types are considered equal but does not touch the expression (because no sub-expression has a foreign type in a principal typing derivation). Still, the next pass of the type inference algorithm attempts to unify the types τ_1 and τ_2 and thus fails.

Relaxing the acyclicity condition Inserting applications of the id operator can break the acyclicity condition. We can actually relax this condition to deal with the id operator more carefully. Let us consider a constraint C with a cycle $\iota_1 \overset{\texttt{C}}{\rightsquigarrow} \ldots \overset{\texttt{C}}{\rightsquigarrow} \iota_1$, such that all the edges in this cycle come from elements of the form $\texttt{id}_\iota^{\iota'}$. Clearly, any solvent ρ such that $\Vdash \texttt{C}\rho$ will map all the ι_i in the cycle to the same ground foreign type. So instead of considering the most-general pre-solution and then face a cyclic constraint, we may as well unify all these ι_i first: all the solutions can still be obtained from this less-general pre-solution.

The relaxed condition is: There must be no cycle in the constraint except maybe cycles whose edges are all produced by the id operator.

To illustrate the usefulness of the relaxed condition, let us consider the expression $\texttt{e} = \texttt{fix}(\lambda\texttt{g}.\lambda\texttt{x.f c } (\texttt{g x}))$ with $\Gamma = \{\texttt{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha, \texttt{f} : \forall\alpha.\alpha \to \alpha \to \alpha, \texttt{c} : \tau\}$. The strengthening pass builds a principal typing derivation for e in a type algebra where all the foreign types are identified. Here is such a derivation, where we write $\star$ for foreign types and $\texttt{t} = \alpha \to \star$, $\Gamma' = \Gamma, \texttt{g} : \texttt{t}, \texttt{x} : \alpha$ (we collapse rules for multiple abstraction and application):

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{f} : \star \to \star \to \star} \quad \dfrac{\dfrac{}{\Gamma' \vdash \texttt{g} : \texttt{t}} \quad \dfrac{}{\Gamma' \vdash \texttt{x} : \alpha}}{\Gamma' \vdash \texttt{g x} : \star}}{\Gamma' \vdash \texttt{f c } (\texttt{g x}) : \star} \quad \dfrac{}{\Gamma \vdash \texttt{fix} : (\texttt{t} \to \texttt{t}) \to \texttt{t}} \quad \dfrac{}{\Gamma \vdash \lambda\texttt{g}.\lambda\texttt{x.f c } (\texttt{g x})) : \texttt{t} \to \texttt{t}}}{\Gamma \vdash \texttt{e} : \alpha \to \star}$$

On this principal derivation, we observe three sub-expressions of a foreign type. Accordingly, strengthening introduces three instances of the id operator and thus rewrites the expression to:

$$\texttt{e}' = \texttt{fix}(\lambda\texttt{g}.\lambda\texttt{x.id}_{\iota_1}^{\iota_2} \ (\texttt{f } (\texttt{id}_{\iota_3}^{\iota_4} \ \texttt{c}) \ (\texttt{id}_{\iota_5}^{\iota_6} \ (\texttt{g x}))))$$

The type-checker which is then applied performs some unifications: $\iota_1 = \iota_4 = \iota_6$, $\iota_2 = \iota_5$, $\iota_3 = \tau$. We can for instance assume that the computed most-general pre-solution maps ι_4 and ι_6 to ι_1 and ι_5 to ι_2. The first and third instances of the id operator in $\texttt{e}'$ thus generate the dependencies $\iota_1 \overset{\texttt{C}_0}{\rightsquigarrow} \iota_2$ and $\iota_2 \overset{\texttt{C}_0}{\rightsquigarrow} \iota_1$. Strictly speaking, the constraint is cyclic, but we can break the cycle simply by unifying ι_1 and ι_2. The smallest solvent is then given by $\iota_1\rho = \tau$. We would have obtained the same solution if we had applied the type-checker directly on e without the strengthening pass. In this example, strengthening is useless and the relaxed acyclicity condition is just a way to break a cycle introduced by strengthening. We can easily imagine more complex examples where strengthening is really necessary but introduces cycles that can be broken by the relaxed condition.

6. Integration in OCaml

We have described a type system for basic ML expressions. Of course, OCaml is much more than an ML kernel. We found no problem to extend it to deal with the whole OCaml type system, including recursive types, modules, classes, and other fancy features. The two ML-like typing passes (the one used during strengthening and the one using for the real type-checking) are done on whole compilation units (in the toplevel, they are done on each phrase). The information from the compilation unit interface (the `.cmi` file) is integrated before checking the acyclicity condition. Indeed, this information acts as additional type annotations on the values exported by the compilation unit and can thus contribute to obtaining this condition. Also, in addition to type annotations on expressions, OCaml provides several ways to introduce explicit type informations (and thus obtain the acyclicity condition): datatype definitions (explicit types for constructor and exception arguments, record fields), module signatures, type annotations on ML pattern variables.

Because of its global flow analysis flavor, OCamlDuce's type system is much less modular than OCaml's. In particular, all the call sites of a function which expects XML values as arguments can contribute to the result type of the function. In practice, to alleviate this lack of modularity and to get better error messages,

the programmer should probably give more type annotations than what it strictly required.

OCaml subtyping OCaml comes with a structural subtyping relation (generated by object types and polymorphic variants subtyping and extended structurally by considering the variance of ML type constructors). The use of this subtyping relation in programs is explicit. The syntax is $(\texttt{e} : \texttt{t}_1 :> \texttt{t}_2)$ (sometimes, the type $\texttt{t}_1$ can be inferred) and it simply checks that $\texttt{t}_1$ is a subtype of $\texttt{t}_2$. Of course, the OCaml subtyping relation has been extended in OCamlDuce to take XDuce subtyping into account. For instance, if τ_1 is a XDuce subtype of τ_2 and `e` has type τ_1`list`, then it is possible to coerce it to type τ_2 `list`: $(\texttt{e} :> \tau_2\ \texttt{list})$.

Crossing the boundary In our system, XDuce values are opaque from the point of view of ML and XDuce types cannot be identified with other ML type constructors. Sometimes, we need to convert values between the two worlds. For instance, we have a foreign type `String` which is different from OCaml `string`. This foreign type conceptually represents immutable sequences of arbitrary Unicode characters, whereas the OCaml type should be thought as representing mutable buffers of bytes. As a consequence, we don't even try to collapse these two types into a single one. Instead, OCamlDuce comes with a runtime library which exports conversion functions such as `Utf8.make: string -> String`, `Utf8.get: String -> string` and `Latin1.make: string -> Latin1`, `Latin1.get: Latin1 -> string`. The type `Latin1` is a subtype of `String`: it represents all the strings which are only made of latin-1 characters (latin-1 is a subset of the Unicode character set). The function `Utf8.make` checks at runtime that the OCaml string is a valid representation of a Unicode string encoded in utf-8.

Similarly, we often need to translate between XDuce's sequences and OCaml's lists. For any XDuce type τ, we can easily write two functions of types $[\tau*] \to \tau$ `list` and τ `list` $\to [\tau*]$ (the star between square brackets denotes Kleene-star). Similarly, we can imagine a natural XDuce counterpart of an OCaml product type $\tau_1 \times \tau_2$, namely $[\tau_1\ \tau_2]$, and coercion functions. However, writing this kind of coercions by hand is tedious. OCamlDuce comes with built-in support to generate them automatically. This automatic system relies on a structural translation of *some* OCaml types into XDuce types: lists and arrays are translated to Kleene-star types, tuples are translated to finite-length XDuce sequences, variant types are translated to union types, etc. Some OCaml types such as polymorphic or functional types cannot be translated. OCamlDuce comes with two magic unary operators `to_ml`, `from_ml` (both written `{: ...:}` in the concrete syntax). The first one takes an XDuce value and applies a structural coercion to it in order to obtain an OCaml value; this coercion is thus driven by the output type of the operator. The type-checker requires this type to be fully known (polymorphism is not allowed). Similarly, the operator `from_ml` takes an OCaml value and apply a structural coercion in order to obtain an XDuce value. Since the type of its input drives its behavior, the type-checker requires this type to be fully known.

This system can be used to obtain coercions from complex OCaml types (e.g. obtained from big mutually recursive definitions of concrete types) to XDuce types, whose values can be seen as XML documents. This gives parsing from XML and pretty-printing to XML for free.

7. Related work

The CDuce language itself comes with a typed interface with OCaml. The interface was designed to: (i) let the CDuce programmers use existing OCaml libraries; (ii) develop hybrid projects where some modules are implemented in OCaml and other in CDuce. The interface is actually quite simple: each monomorphic OCaml type `t` is mapped in a structural way to a CDuce type $\hat{\texttt{t}}$. A value defined in an OCaml module can be used from CDuce (the compiler introduces a natural translation $\texttt{t} \to \hat{\texttt{t}}$). Similarly, it is possible to provide an ML interface for a CDuce module: the CDuce compiler checks that the values exported by the module are compatible with the ML-to-CDuce translation of these types and produces stub code to apply a natural translation $\hat{\texttt{t}} \to \texttt{t}$ to these values. This CDuce/OCaml interface is used by many CDuce users and served as a basis to the `to_ml` and `from_ml` operators described in Section 6.

Sulzmann and Zhuo Ming Lu [SL05] pursue the same objective of combining XDuce and ML. However, their contribution is orthogonal to ours. Indeed, they propose a compilation scheme from XDuce into ML such that the ML representation of XDuce values is driven by their static XDuce type (implicit use of subtyping are translated to explicit coercions). Their type system supports in addition used-defined coercions from XDuce types to ML types. However, they do not describe a type inference algorithm for their abstract specification of a type system and do not study the interaction between XDuce type-checking and ML type inference (XDuce code can call ML functions but their type must be fully known). These last points are precisely the issues tackled by our contribution. For instance, our system makes it possible to avoid some type annotation on non-recursive XDuce functions. Another difference is that in our approach, the XDuce/CDuce type checker and back-end (compilation of pattern matching) can be re-used without any modification whereas their approach requires a complete reengineering of the XDuce part (because subtyping and pattern matching relations must be enriched to produce ML code) and it is not clear how some XDuce features such as the `Any` type can be supported in a scenario of modular compilation. We believe our approach is more robust with respect to extensions of XDuce and that the XDuce-to-ML translation can be seen as an alternative implementation technique for XDuce which allows some interaction between XDuce and ML (the same kind of interaction as what can be achieved with the CDuce/OCaml interface described above).

The Xtatic project [GP03] is another example of the integration of XDuce types into a general purpose language, namely C#. Since both C#'s and XDuce's type checkers operate with bottom-up propagation (explicit types for functions/methods, no type inference), the structure of Xtatic type-checker is quite simple. The real theoretical contribution is in the definition of a subtyping relation which combines C# named subtyping (inheritance) and XDuce set-theoretic subtyping. Since the resulting type algebra does not have least-upper bounds, the nice locally-complete type inference algorithm for XDuce patterns [HP02] cannot be transferred to Xtatic. In Xtatic, XDuce types and C# types are stratified, but the two algebras are mutually recursive: XDuce types can appear in class definitions and C# classes can be used as basic items in XDuce regular expression types. This does not really introduce any difficulty because C# types are not structural. The equivalent in OCamlDuce would be to allow OCaml abstract types as part of XDuce types, which would not be difficult, except for scoping reasons (abstract types are scoped by the module system).

In the last ten years, a lot of research effort has been put into developing type inference techniques for extensions of ML with subtyping and other kinds of constraints. For instance, the HM(X) framework [OSW99] could serve as a basis to express the type system presented here. The main modification to bring to HM(X) would be to make foreign-type variables global. Another way to express it is to disallow constraints in type-schemes (which is what we do in the current presentation). We have chosen to present our system in a setting closer to ML so as to make our message more

explicit: our system can be easily implemented on top of existing ML implementations.

8. Conclusion and future work

We have presented a simple way to integrate XDuce into OCaml. The modification to the ML type-system is small enough so as to make it possible to easily extend existing ML type-checkers.

Realistic-sized examples of code have been written in OCamlDuce, such as an application that parses XML Schema documents into an internal OCaml form and produces an XHTML summary of its content. Compared to a pure OCaml solution, this OCamlDuce application was easier to write and to get right: XDuce's type system ensures that all possible cases in XML Schema are treated by pattern-matching and that no invalid XHTML output can be produced). We refer the reader to OCamlDuce's website for the source code of this application.

The main limitation of our approach is that it doesn't allow parametric polymorphism on XDuce types. Adding polymorphism to XDuce is an active research area. In a previous work with Hosoya and Castagna [HFC05], we presented a solution where polymorphic functions must be explicitly instantiated. Integrating this kind of polymorphism into the same mechanism as ML polymorphism is challenging and left for future work. The theory recently developed by Vouillon [Vou06] could be a relevant starting point for such a task.

Another direction for improvement is to further relax the acyclicity conditions, that is, to accept more programs without requiring extra type annotations. Once the set of constraints representing XML data flow and operations have been extracted by the ML type-checker, we could use techniques which are more involved than simple forward computation over types. The static analysis algorithm used in Xact [KMS04] could serve as a starting point in this direction.

Acknowledgments

The author would like to thank Didier Rémy and François Pottier for fruitful discussion about the design and formalization of type systems.

References

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Pr ogramming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[Dam85] Luis Manuel Martins Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, Scotland, April 1985.

[Fri04] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmatio n fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[GP03] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darms tadt, Germany*, 2003.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *POPL*, 2005.

[HM03] Haruo Hosoya and Makoto Murata. Boolean operations and inclusion test for attribute-element constraint s. In *Eighth International Conference on Implementation and Application of Automata*, 2003.

[Hos00] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, December 2000.

[Hos04] Haruo Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, 2004.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Data bases (WebDB2000)*, 2000.

[HP02] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(4), 2002.

[HP03] Haruo Hosoya and Benjamin C. Pierce. A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.

[KMS04] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[L+01] Xavier Leroy et al. The Objective Caml system release 3.08; Documentation and user's manual, 2001.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[SL05] Martin Sulzmann and Kenny Zhuo Ming Lu. A type-safe embedding of XDuce into ML. In *The 2005 ACM SIGPLAN Workshop on ML*, 2005.

[Vou06] Jérôme Vouillon. Polymorphic regular tree types and patterns. In *POPL*, 2006.