Regular Object Types

Vladimir Gapeyev   Benjamin C. Pierce
University of Pennsylvania

Abstract

Regular expression types have been proposed as a foundation for statically typed processing of XML and similar forms of tree-structured data. To date, however, regular expression types have been explored in special-purpose languages (e.g., XDuce, CDuce, and XQuery) with type systems designed around regular expression types "from the ground up." The goal of the Xtatic language is to bring regular expression types to a broad audience by offering them as a lightweight extension of a popular object-oriented language, C$^{\#}$.

We develop here the formal core of the Xtatic design—a combination of the tree-structured data model of XDuce and the classes-and-objects data model of a conventional object-oriented language. Our tool for this investigation is a tiny language called FX with features drawn from Featherweight Java (FJ) and from the core of XDuce. Points of interest include (1) a smooth interleaving of the two value spaces, in which XDuce's tree structures are grafted into of FJ's class hierarchy while objects and object types play the role of XDuce's label values and label types; (2) a "semantic" definition of the subtype relation, inherited from XDuce and extended to objects; and (3) a natural encoding of XML documents and their schemas using a simple form of singleton classes.

1 Introduction

The popularity of XML can be attributed, in part, to the existence of a number of formalisms for specifying the structure of XML documents. By supporting dynamic consistency checking, ensuring that information being exchanged (e.g., between modules in an application or nodes in a distributed system) has the expected structure, these schema languages significantly increase the robustness of complex XML-based information systems.

However, the exploitation of schema languages by current XML technologies falls far short of what is possible. In particular, schemas play little part in the static analysis of programs that operate on XML structures: they are not used for checking code for inconsistencies at compile time, or for optimization—in short, they are not used as types in the usual programming-language sense of the term. Taking advantage of this missed opportunity, and thereby improving both the robustness and the efficiency of XML-based information systems, is the long-range goal of the Xtatic project at the University of Pennsylvania.

The key technology for this project is *regular expression types*. Regular expression types are based on well-known constructions from automata theory—they are a mild generalization of nondeterministic tree automata. Their basic constructors (union, concatenation, repetition, etc.) are similar to those found in existing XML schema formalisms such as DTDs [35] and XML-Schema [36]. In a programming language based on regular expression types, however, XML trees are built-in values and static analysis of the shapes of trees that may appear at run time (as values of variables, parameters to methods, results of complex expressions, etc.) becomes part of the ordinary behavior of the typechecker.

Past work on regular expression types led to a language prototype called XDuce [16, 18, 15, 17]. XDuce is a statically typed language for writing recursive tree transformers—roughly, a statically typed fragment of the popular XSLT language [37]. Beyond regular expression types, its main innovation is a powerful form of *regular expression pattern matching*—a statically typed "tree grep" primitive that arises naturally from types [15]. The XDuce implementation demonstrates efficient algorithms for subtyping and typechecking [18].

XDuce has had a significant impact in parts of the XML world; in particular, its influence can be seen in the type system of the XML Query Algebra [11], the core of the W3C standard query language for XML, as well as newer schema languages such as TREX [8] and Relax NG [9]. However, significant work remains before the benefits of regular expression types can be made available to the vast majority of XML programmers. In particular, the simple tree-data model of XDUCE must be enriched to include objects.

We have begun a new phase of the XDUCE project—a redesign and re-implementation along more ambitious lines, dubbed XTATIC, whose main focus is on inter-operability both at source level and at run time with an established, object-oriented host language. We have chosen compatibility with C# as our immediate target; a similar exercise could easily be carried out for a related language such as Java. The goal is to make XTATIC as lightweight an extension of C# as possible, smoothly merging the tree values and types of XDUCE with the familiar object model of C# and re-using existing C# features wherever possible in the design, rather than introducing new, XML-specific mechanisms.

This paper develops the formal core of the XTATIC design—a combination of the tree-structured data model of XDUCE with the classes-and-objects model of C#. Our tool for this investigation is a tiny language called FX, which combines Featherweight Java [19] with the core features of XDUCE. The main points of interest in FX may be summarized as follows.

- The two original data models are tightly interwoven in FX. On one hand, the subtype hierarchy of tree types is grafted into the class hierarchy, allowing tree values to be passed to generic library facilities (e.g., collection classes), stored in fields of objects, etc. Conversely, the role of labels and label types in XDUCE is played by objects and classes in FX.

- Subtyping in FX is a natural extension of both the object-oriented subclass relation and the richer subtype relation of regular expression types. XDUCE's simple "semantic" definition of subtyping (sans inference rules) is extended to objects and classes.

- FX enriches XDUCE's regular expression pattern matching construct with a natural form of type-based pattern-matching on objects.

The paper is organized as follows. Section 2 gives a brief illustrative example of XTATIC code. In Section 3, we review some details of XDUCE and FJ's data models. Section 4—the heart of the paper—combines these to produce the data model of FX. The remainder of the the FX language is informally described in Section 5; standard soundness properties are sketched in Section 6. In Section 7, we show how the FX data model encodes XML types and values. Section 8 discusses related work, and Section 9 sketches our plans for the future development of XTATIC.

2 Example

XTATIC provides general mechanisms for manipulating tree-structured data. In Section 7, we will show in detail how this mechanism can be used to represent XML. Here, we use a preliminary sketch of the idea from Section 7 to illustrate the features of the language.

Assume we have a class `Tag` whose objects encode XML tags. Let classes `Person`, `Name`, `Email`, `Phone` be descendants of `Tag` intended to encode XML tags `<person>`, `<name>`, `<email>`, `<phone>`, and let `person`, `name`, `email`, `phone` be variables containing objects of the corresponding classes. Then the expression

```
<person>[
    <name>[<"Queen Elisabeth">[]]
    <email>[<"queen@buckingham.uk">[]] ]
<person>[
   <name>[<"Tony Blair">[]]
   <phone>[<"+44 34 3456">[]] ]
```

can be thought of as representing an XML document of a similar structure. A possible type for this expression is the sequence type

```
<Person>[
   <Name>[<String>[]]
   (<Email>[<String>[]]
    | <Phone>[<String>[]])
]*.
```

A tree is constructed using the form `<...>[...]` where `<...>` contains the tree's label and `[...]` contains a sequence of child trees. A sequence is built by placing trees adjacent to each other. The type constructor "`|`" is type alternation (union), and "`*`" is repetition. Note that native C# values—the tag objects `person`, `name`, etc., and strings—occur only as labels.

Sequence values can be examined using type-based pattern matching. For example, assuming the variables `list` and `phonebook` each contain a sequence of the type given above and `spamlist` holds a string, the code fragment

```
match (list) {
  case [ <Person>[<Name>[String]
                  <Email>[String e]]
         Any ]:
```

```
      spamlist = spamlist + "," + e;
      // ...
   case [ (<Person>[String] p) Any ]:
      phonebook = [ phonebook p ];
      //...
   case [ Any ]:
      //...
}
```

inspects the first tree in the sequence `list` and, if the corresponding person has an email, extracts the address into a string variable `e` and uses it to extend the string `spamlist`; otherwise, the person must have a phone, and the second `case` branch handles this case by binding the whole entry to the variable `p` and adding it to the end of the sequence stored in the variable `phonebook`. The pattern `Any` in each `match` clause matches an arbitrary sequence of trees. Tree values, types, and patterns are enclosed in square brackets, explicitly signaling the shift from the world of host language ($C^{\#}$) values and types to the world of trees.

3 Technical Background

The *data model* of a language is the collection of *values* that programs in the language manipulate, their *types*, and fundamental relations such as *value typing* and *subtyping*. The data model is the bedrock on which the full language definition (the syntax, typing rules, and evaluation rules for expressions) rests. Because the primary topic of this paper is the combination of trees and objects (and their types), the data model of FX is where we will concentrate our attention. As background for this development, we begin in this section by sketching the data models found in XDuce and in FJ.

3.1 The XDuce Data Model

The data model of XDuce is parameterized on a language of *labels*. The details of these labels can vary (and do vary, across the several published XDuce papers and implementations), but all variations offer the following common structure:

- a set L of *label values*, ranged over by `l`,
- a set of *label types*, ranged over by `L`,
- a denotation function $[\![\cdot]\!]$ giving the set $[\![\mathtt{L}]\!] \subseteq L$ of label values that are members of each label type `L`.

The subtyping relation on label types, written $\mathtt{L}_1 <: \mathtt{L}_2$, is generated by $[\![\cdot]\!]$—that is $\mathtt{L}_1 <: \mathtt{L}_2$ iff $[\![\mathtt{L}_1]\!] \subseteq [\![\mathtt{L}_2]\!]$.

One simple choice of label language is to select an arbitrary set of identifiers as the set L of label values; for each value $\mathtt{l} \in L$, we consider `l` to be a label type as well (i.e., `l` is the *singleton type* whose denotation contains just `l`); we also introduce the wildcard label type `~`, denoting the whole set L. A yet simpler choice would be to omit `~`, but having a maximal label type turns out to be quite useful in pattern matching, where it functions as a "don't care" pattern.

Having selected the language of labels, the XDuce data model can be defined in a uniform way. First, a *tree value* `t` consists of a label value and a sequence of children tree values:

$$\mathtt{t} \quad ::= \quad \mathtt{<l>[t_1 \ldots t_n]} \qquad \text{where } n \geq 0$$

Now, a *sequence value* is a sequence $\mathtt{t}_1 \ldots \mathtt{t}_n$ of zero or more tree values placed next to each other. (We use the shorthand notation $\overline{\mathtt{t}}$ throughout the paper for sequences, and `()` to denote the empty sequence. We write $\overline{\mathtt{s}}\ \overline{\mathtt{t}}$ for the concatenation of the sequences $\overline{\mathtt{s}}$ and $\overline{\mathtt{t}}$.)

XDuce types—*regular expression types*—are built from tree types and references to type names `X` from a globally defined collection of type definitions:

`T`	::=	
	`X`	type name
	`<L>[T]`	tree
	`()`	empty sequence
	`T T`	concatenation
	`T|T`	union
	`T*`	repetition

We write *def* for the function that maps each `X` to its definition `T`. The global definitions given by the function *def* may be recursive or mutually recursive, but (to limit the expressive power of the type language to describing regular, rather than context-free, sets of trees), we impose the condition that all "loops" from a variable `X` back to itself must pass through the body of at least one `<L>[T]` construct—i.e., "top-level" recursion is not allowed (see [18] for details).

Next, the denotation function $[\![\cdot]\!]$ mapping types `T` to sets of sequence values $\overline{\mathtt{t}}$ is defined as the least solution of the following equations:

$$\begin{array}{lcl}
[\![\mathtt{X}]\!] & = & [\![\mathit{def}(\mathtt{X})]\!] \\
[\![\mathtt{<L>[T]}]\!] & = & \{\, \mathtt{<l>}[\overline{\mathtt{t}}] \mid \mathtt{l} \in [\![\mathtt{L}]\!] \text{ and } \overline{\mathtt{t}} \in [\![\mathtt{T}]\!] \,\} \\
[\![\mathtt{()}]\!] & = & \{\, \mathtt{()} \,\} \\
[\![\mathtt{T}_1\ \ \mathtt{T}_2]\!] & = & \{\, \overline{\mathtt{t}}_1\ \ \overline{\mathtt{t}}_2 \mid \overline{\mathtt{t}}_1 \in [\![\mathtt{T}_1]\!], \overline{\mathtt{t}}_2 \in [\![\mathtt{T}_2]\!] \,\} \\
[\![\mathtt{T}_1 | \mathtt{T}_2]\!] & = & [\![\mathtt{T}_1]\!] \cup [\![\mathtt{T}_2]\!] \\
[\![\mathtt{T*}]\!] & = & \{\, \overline{\mathtt{t}}_1 \ldots \overline{\mathtt{t}}_n \mid n \geq 0 \wedge \forall k.\ \overline{\mathtt{t}}_k \in [\![\mathtt{T}]\!] \,\}
\end{array}$$

The subtyping relation for regular expression types is defined in the simplest imaginable way: $\mathtt{T}_1 <: \mathtt{T}_2$ iff $[\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$. The fact that subtyping can be defined in this "semantic" fashion is actually quite important in XDUCE. The alternative—writing down a collection of inference rules characterizing the same relation inductively—would be much heavier and harder to understand than the subtype relations of most languages, since the regular expression type constructors satisfy many algebraic laws arising from the associativity of comma and the associativity, commutativity, and distributivity (over sequencing and `<L>[...]`) of the (*non*-disjoint!) union. An inference-rule presentation of the subtyping relation can certainly be given—indeed, it must be, since it is the basis for the algorithm for subtype checking [18]—but it is not pretty.

3.2 The FJ Data Model

Featherweight Java (or FJ) is a tiny calculus designed to capture the essential typing mechanisms of class-based object-oriented languages such as Java and $C^{\#}$. It was first used by Igarashi, Pierce, and Wadler [19] to formalize the GJ [5] type system, and has since formed the basis of numerous formal studies of Java and related languages [20, 30, 2, 3, 38, 1, 28, 22, etc.]. FJ embodies the core mechanisms of object creation, field access, method invocation, and inheritance (and—in the most common presentation, though not here—casting) in exactly the same form as they are found in Java, while omitting everything else... from reflection and concurrency to interfaces, overloading, static members, and even assignment.

An FJ program consists of a collection of class declarations plus a single expression to be evaluated. The types in an FJ program are just class names `C`. FJ values are objects, which (since FJ is a declarative language, the only things that distinguish one object from another are its class and the arguments passed to its constructor) are simply identified with `new` expressions.

$$\mathtt{o} \quad ::= \quad \mathtt{new\ C(o_1, \ldots, o_n)}$$

The constructor arguments $\mathtt{o}_1, \ldots, \mathtt{o}_n$ (usually written just $\overline{\mathtt{o}}$) are required to correspond exactly to the fields of the class `C`. For example, if `C` has fields `a` and `b` and its immediate subclass `D` has fields `e` and `f`, then an instance of `D` will have the form $\mathtt{new\ D(o_1,o_2,o_3,o_4)}$, where $\mathtt{o}_1$ is the value for the `a` field of the new object, $\mathtt{o}_2$ is the value of the `b` field, $\mathtt{o}_3$ of the `e` field, and $\mathtt{o}_4$ of the `f` field.

The global set of class definitions in an FJ program is formalized as a *static context*—a collection of sets, relations, and functions summarizing different aspects of the class definitions: the set of all defined classes (which always includes the special class `Object`); the immediate-subclass relation, which must be tree-structured with `Object` at the root; the list of field names and types in each class; the method names and signatures in each class; and the method bodies for each class. This static context is used to define the typing and evaluation relations. For purposes of discussing the FJ data model, we can restrict attention to the part of the static context comprising just the set of class names and the immediate-subclass relation; we call this the *static data context*.

The subtype relation in FJ, written $\mathtt{C}_1 \sqsubset: \mathtt{C}_2$, is the reflexive and transitive closure of the immediate-subclass relation. Like XDUCE's, this definition of subtyping is pleasingly simple; however, it has a completely different—more syntactic—character. In order to combine the two data models, we need to look for a more "semantic" presentation of this one (as we remarked above, a syntactic presentation of XDUCE subtyping is an unattractive alternative). This can be achieved as follows.

We say that a value $\mathtt{new\ C}(\overline{\mathtt{o}})$ is an *instance* of the class `C`. That is, an object is an instance of the class from which it was created. The *denotation* of a class `C` is then the set of all instances of this class and all its subclasses:

$$[\![\mathtt{C}]\!] \quad = \quad \{\, \mathtt{o} \mid \mathtt{o} \text{ is an instance of } \mathtt{D}, \text{ for some } \mathtt{D} \sqsubset: \mathtt{C} \,\}$$

Note that this does *not* require that the constructor arguments $\overline{\mathtt{o}}$ belong to the types of the fields in class `C`. This may appear overly permissive, but it has some useful implications:

1. It is obvious from the definition that the "semantic" subtyping relation derived from it coincides exactly with the syntactic subclass relation: $\mathtt{C}_1 \sqsubset: \mathtt{C}_2$ iff $[\![\mathtt{C}_1]\!] \subseteq [\![\mathtt{C}_2]\!]$.

2. This definition requires no changes if we enrich the language with imperative features. A more precise definition ("the values in type `C` are objects of the form $\mathtt{new\ D}(\overline{\mathtt{o}})$, where $\mathtt{D} \sqsubset: \mathtt{C}$ and, $\mathtt{o}_i \in [\![\mathtt{F}_i]\!]$, where $\mathtt{F}_i$ is the type of the i^{th} field of class `D`") would require a co-inductive reading (cf. [33]) to make sense in the imperative setting.

Intuitively, the reason we can get away with this permissive interpretation of object types is that later, e.g., in the proof of soundness, we will never deal with arbitrary elements of $[\![\mathtt{C}]\!]$, but only with elements that we also know are well-typed (according to the expression typing relation, which *does* ensure that constructor arguments have the right types).

Values			**Types**		
		Full FX language			
`a`	::=	FX value	`A`	::=	FX type
	`new C(ā)`	object		`C`	class type
	`[t̄]`	delimited sequence		`[T]`	delimited RE type
		Regular expression sublanguage			
`t`	::=		`T`	::=	RE type
	`<a>[t̄]`	tree value		`X`	RE type name
				`<C>[T]`	tree type
				`()`	empty sequence
				`T T`	concatenation
				`T|T`	union
				`T*`	repetition

Figure 1: FX values and types.

$$\mathit{Instances}(\mathtt{C}) = \begin{cases} \{\, \mathtt{[\overline{t}]} \mid \overline{\mathtt{t}} \text{ arbitrary } \} & \text{if } \mathtt{C} = \mathtt{Seq} \\ \{\, \mathtt{new\ C(\overline{a})} \mid \overline{\mathtt{a}} \text{ arbitrary} \,\} & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl}
[\![\mathtt{C}]\!] & = & \bigcup\{\, \mathit{Instances}(\mathtt{D}) \mid \mathtt{D} \sqsubset: \mathtt{C} \,\} \\
[\![\mathtt{[X]}]\!] & = & [\![\mathtt{[}\mathit{def}(\mathtt{X})\mathtt{]}]\!] \\
[\![\mathtt{[<C>[T]]}]\!] & = & \{\, \mathtt{[<a>[\overline{t}]]} \mid \mathtt{a} \in [\![\mathtt{C}]\!] \text{ and } \mathtt{[\overline{t}]} \in [\![\mathtt{[T]}]\!] \,\} \\
[\![\mathtt{[()]}]\!] & = & \{\, \mathtt{[]} \,\} \\
[\![\mathtt{[T_1\ T_2]}]\!] & = & \{\, \mathtt{[\overline{t}_1\ \overline{t}_2]} \mid \mathtt{[\overline{t}_1]} \in [\![\mathtt{[T_1]}]\!], \mathtt{[\overline{t}_2]} \in [\![\mathtt{[T_2]}]\!] \,\} \\
[\![\mathtt{[T_1|T_2]}]\!] & = & [\![\mathtt{[T_1]}]\!] \cup [\![\mathtt{[T_2]}]\!] \\
[\![\mathtt{[T*]}]\!] & = & \{\, \mathtt{[\overline{t}_1 \ldots \overline{t}_n]} \mid \forall k \in 1 \ldots n.\ \overline{\mathtt{t}}_k \in [\![\mathtt{T}]\!],\ \text{for some } n \geq 0 \,\}
\end{array}$$

Figure 2: Type denotations.

4 The FX Data Model

The interweaving of XDUCE's and FJ's data models in FX is founded on two observations.

1. We can treat sequences of trees as objects simply by "grafting" the whole collection of regular expression types into the class hierarchy, inventing a special *class* `Seq` whose sub*types* are all the regular expression types. This grafting is justified by our intended compilation model—reminiscent of GJ's homogeneous translation [5, 19]—in which all regular expression types in an FX program are "erased" to the single class type `Seq` and all tree values are translated into objects of class `Seq`.

2. The data model of objects and classes qualifies as a "label language" in the sense discussed in Section 3.1, so we can use arbitrary objects as the labels in XDUCE trees and classes as label types.

Formally, the data model is defined in three steps. First, we give the syntax of values and types. Next, we give the notion of a static context, which summarizes the type-related information defined in a program. Finally, fixing a particular static context, we define the membership relation for values in types.

Figure 1 defines the syntax. An FX value `a` can have one of two forms: it is either an object `new C(ā)` or a sequence `[t̄]` delimited by brackets. Observe that, inside an object `new C(ā)`, the values of fields may be arbitrary FX values `ā`; in particular, they can be sequences. The organization of FX types `A` is similar, combining class types `C` and regular types `[T]`.

Regular values $\overline{\mathtt{t}}$ and regular types $\mathtt{T}$ are essentially those of XDUCE, where any FX value can be used as a label in a tree value and any class type $\mathtt{C}$ can be used as a label in a tree type.[1]

A *static data context* is a tuple $DatCtx = \langle Classes, \sqsubset:, Typenames, def\rangle$, where

- *Classes* is a set of class names, ranged over by $\mathtt{C}$ and containing special names `Object` and `Seq`;
- $\sqsubset:$ is a binary relation on *Classes*, generated as a reflexive and transitive closure from a relation corresponding to an "immediate predecessor" function $Parent : Classes \setminus \{\mathtt{Object}\} \to Classes$;
- *Typenames* is a set of type names, ranged over by $\mathtt{X}$;
- *def* is a function from *Typenames* to types, that maps each type name $\mathtt{X}$ to a regular expression type $\mathtt{T}$ (its *definition*);

and such that

1. for each $\mathtt{C} \in dom(Parent)$ there is $n \geq 1$ such that $Parent^n(\mathtt{C}) = \mathtt{Object}$;
2. $Parent(\mathtt{Seq}) = \mathtt{Object}$;
3. for every $\mathtt{C} \in Classes$, $Parent(\mathtt{C}) \neq \mathtt{Seq}$;
4. if a type name $\mathtt{X}'$ appears in $def(\mathtt{X})$, then $\mathtt{X}' \in Typenames$; and
5. a grammar obtained from *def* by considering variables from *Typenames* as non-terminals generates a regular language (cf. Sect. 3.1).

The semantics of types is given by the denotation function $[\![\cdot]\!]$, which maps each type $\mathtt{A}$ to its set of inhabitants $\mathtt{a}$. This function is the least solution of the equations in Figure 2. Note the special role of the class `Seq`, whose denotation does not contain objects (`new Seq(`$\overline{\mathtt{a}}$`)` is not in the denotation of any type), but instead contains all sequence values.

Subtyping on FX types is defined semantically:

$$\mathtt{A}_1 <: \mathtt{A}_2 \qquad \Longleftrightarrow \qquad [\![\mathtt{A}_1]\!] \subseteq [\![\mathtt{A}_2]\!].$$

The XDUCE subtyping algorithm [18] can be used to decide this relation, since it is parameterized by the subtyping relation for tree label types (called there "subtagging"), which corresponds in FX to the subclass relation $\mathtt{C}_1 \sqsubset: \mathtt{C}_2$.

[1] The careful reader may note a small discrepancy here: a sequence can be used as a label in another tree, as in `<[`$\overline{\mathtt{s}}$`]>[`$\overline{\mathtt{t}}$`]`, but a regular expression *type* cannot be similarly used as a label. This raises the question of what type can be given to a value of the above form. As we shall see soon, the type would have the form `<Seq>[T]`.

5 The FX Language

The FX data model described in the previous section establishes a skeleton, on which a full-blown programming language can be constructed—providing ways of interrogating and destructing values, as well as abstraction mechanisms and all the other usual apparatus. Naturally, FX's value-destruction mechanisms are contributed by the corresponding sublanguages: FJ provides field projection on objects and XDUCE brings in regular-expression pattern matching on sequences and trees. The abstraction mechanisms of FX—classes, methods, and inheritance—are taken entirely from FJ.

Figure 3 gives the syntax of FX expressions and their constituent patterns. The behavior of most of these constructs is standard; therefore we discuss the language semantics mostly informally, commenting in more detail on the issues that are novel in FX.

We do not describe concrete syntax for class and method declarations: for the present discussion it is more convenient to think about an FX program as an abstract *static context* *Ctx* defined along the lines, and as an extension of, the static data context *DatCtx* of Section 4. Namely, in addition to the items from *DatCtx*, the full context *Ctx* associates with each class a collection of typed fields and a collection of methods available for calling on the objects of the class. For each method, *Ctx* provides its signature (types of the arguments and the return type), the list of argument variables, and the expression of the body. Additionally, *Ctx* must obey constraints on method types in subclasses, stemming from the $\mathrm{C}^{\#}$ inheritance rules.

The only significant difference of an FX context *Ctx* from the information provided by an FJ program is that the types of fields and the types appearing in method signatures are arbitrary FX types, i.e. they can be regular types as well as classes. Consequently, the subtyping relation used for checking the method overriding constraints (as part of the process of checking that a class is well-formed) is the semantic subtyping relation <:. Similarly, FX variables `x` (which can only originate in FX as method argument names or as binders in patterns) can hold any FX values, either objects or sequences. As in FJ, there is a variable `this` that can be used in expressions to refer to the current object. The typing and evaluation rules treat this variable specially.

The FX data model permits only tree values to be members of sequences. That is, something like `[ [`$\overline{\mathtt{t}}$`] (new C(`$\overline{\mathtt{a}}$`)) [`$\overline{\mathtt{s}}$`] ]` is not a well-formed value. The syntax of *expressions*, however, does allow nested sequences. The reason is that we want an expression

e, d	::=	expression
	x	value variable
	new C($\overline{\texttt{e}}$)	new object creation
	e.f	field access
	e.m($\overline{\texttt{e}}$)	method call
	<e>[$\overline{\texttt{e}}$]	tree
	[$\overline{\texttt{e}}$]	sequence
	match(e){case [$\overline{\texttt{P}}$]: $\overline{\texttt{e}}$}	pattern match

R	::=	FX pattern
	Q	class pattern
	[P]	delimited RE pattern
Q	::=	class pattern
	C	class
	C x	object binding

P	::=	RE pattern
	X	RE type name
	<Q>[P]	tree
	()	empty sequence
	P P	concatenation
	P|P	alternative
	T*	type repetition
	P x	RE value binding

Figure 3: FX language syntax.

like

```
[db.getPapers("POPL") db.getPapers("ICFP")]
```

to be legal—provided the method `getPapers()` returns values of a sequence type—and to mean the concatenation of the sequences returned by the two calls. Therefore, a nested sequence `[` [$\overline{\texttt{e}}$] [$\overline{\texttt{d}}$] `]` is a valid FX expression, which evaluates to the same value as `[` $\overline{\texttt{e}}$ $\overline{\texttt{d}}$ `]`. Generally, FX typing rules ensure that an expression [$\texttt{e}_1$ $\texttt{e}_2$] is legal only when $\texttt{e}_1$ and $\texttt{e}_2$ both have valid regular types.

On the other hand, an object is never legal as a member of a sequence and, symmetrically, a tree expression `<e>`[$\overline{\texttt{d}}$] is never allowed outside the sequence parentheses `[...]`. Since both are permitted syntactically, this condition is checked by the typing rules.

Deconstruction of sequence values is done by matching them against patterns using the `match` construct, which syntactically resembles C# `switch` statement but behaves more like XDuce's `match`. That is, the behavior of an expression

```
match (d) {
    case [P1]:  e1;
    case [P2]:  e2;
...
    case [Pn]:  en;
}
```

is to evaluate `d` and match the result against each of the patterns in turn until the first one, say [$\texttt{P}_i$], matching the value is encountered. The successful match produces an environment that maps variables declared in [$\texttt{P}_i$] to the appropriate portions of the value computed from `d`. The result of the whole expression is the result of evaluating $\texttt{e}_i$, assuming variable mappings from the environment. So, the case bodies do not have the "fall through" behavior of `switch`. The value of `match`'s input `d` must be a sequence, and all `case` patterns must be sequence patterns.

The syntax of FX sequence patterns `[P]` is essentially that of XDuce: a pattern is just a type annotated with variable binders.[2] This intuition is extended in FX to class types. A *class pattern* has the form `C x`, where `C` is a class name and `x` is a variable to be bound. Correspondingly, the pattern-matching relation $\texttt{a} \in \texttt{R} \Rightarrow \Sigma$, which defines when a value `a` matches a pattern `R` to produce a value environment Σ, is based on the pattern-matching relation $[\overline{\texttt{t}}] \in [\texttt{P}] \Rightarrow \Sigma$ of XDuce. Additional rules are needed only to handle class patterns:

$$\frac{\texttt{D} \sqsubset : \texttt{C}}{\texttt{new D}(\overline{\texttt{a}}) \in \texttt{C} \Rightarrow \bullet} \qquad \frac{\texttt{Seq} \sqsubset : \texttt{C}}{[\overline{\texttt{t}}] \in \texttt{C} \Rightarrow \bullet}$$

$$\frac{\texttt{a} \in \texttt{C} \Rightarrow \Sigma}{\texttt{a} \in \texttt{C x} \Rightarrow \texttt{x:a}, \Sigma}$$

[2]As in XDuce, we demand that each pattern `P` satisfy the same regularity constraint as for types, and that it be *linear*. Intuitively, linearity means that no variable is bound in `P` twice, except in alternation patterns, where branches must bind exactly the same variables (see [15], appendix A.2, for the formal definition).

Observe that the rules agree with the denotation function $[\![\cdot]\!]$ in the sense that $\mathtt{a} \in [\![\mathtt{C}_1]\!]$ and $[\![\mathtt{C}_1]\!] \subseteq [\![\mathtt{C}_2]\!]$ imply $\mathtt{a} \in \mathtt{C}_2 \Rightarrow \bullet$. Also note that, like the definition of $[\![\cdot]\!]$, the class pattern matching relation does not examine the types of an object's fields. This "permissiveness" is safe because the FX expression typing relation guarantees that no object with ill-typed fields can ever be created (see Prop. 6.3).

Since classes are types of labels in tree types, it is natural to use a class pattern in the label position in a tree pattern. This allows one to extract a label from a tree as an object for later use in the program. It is worthwhile noticing that this is a benefit of our goal to use, whenever possible, $C^{\#}$ features for the needs of regular types. To compare, in XDUCE a label is an integral part of a tree and cannot be extracted from it as a first-class value.

The typing of `match` depends on the type inference for variables bound in its patterns. In XDUCE the type inference is formalized as the judgment $\mathtt{T} \triangleright \mathtt{P} \Rightarrow \Gamma$, relating an input type T, a pattern P and a typing environment Γ. This judgment is precise: for each type $\Gamma(\mathtt{x})$, each value from its denotation can be possibly bound to x at run time as a result of matching some value from T's denotation against P, and $[\![\Gamma(\mathtt{x})]\!]$ does not contain values that cannot be thus obtained. The precision is achieved thanks to the availability of unrestricted union operation on XDUCE types. In FX, however, we cannot have union for class types, and have to use an upper bound instead, sacrificing the precision of type inference. For example, suppose class D has A, B, and C as its direct subclasses, and consider matching values of type `[<D>[]]` against pattern `[<A x>[] | <B x)[]]`. Since there is no class whose denotation is an exact union of the denotations of A and B, the only reasonable type assignment for x is D, which is not precise—x can never be bound to an object of C, another D's descendant. Therefore, we decided to formalize FX type assignment for pattern variables by a simpler relation, $\triangleright R \Rightarrow \Gamma$, which does not take the input type into account and assigns $\Gamma(\mathtt{x})$ to be the type on which x appears as an annotation in the pattern R (and, in the case of the alternation pattern like in the above example—the *join* of the types of the alternatives, defined as their smallest expressible common upper bound).

In XDUCE, precise pattern type inference extends to the whole collection of patterns of a `match`: the input type for the i^{th} pattern $\mathtt{P}_i$ is not the type of input to the whole `match`, but rather the input restricted to those values that could not be matched by any of the previous patterns. Implementation of this feature depends on the fact that the *difference* of two regular expression types is also a regular expression type. Since this property fails for classes (the "semantic difference" of two classes is not necessarily a class), we cannot guarantee precise type inference in XTATIC.[3] However, we are still able to check `match` expressions for exhaustiveness, by checking that the input type is a subtype of the union of the types of all the patterns.[4] We can also provide a restricted form of pattern redundancy checking by comparing, for each prefix of the pattern list, the union of the prefix's patterns and the input type.

6 Properties

We can now state for FX the standard results of static type safety ("preservation" and "progress"). The proofs (which are straightforward inductive arguments) are omitted. All the results are stated assuming there is a well-formed static context corresponding to an FX program.

Value environments Σ and typing environments Γ are mappings from variables x to values a, and, correspondingly, to types A. An environment with the empty domain is written $\bullet$.

The following three relations formalizing FX operational semantics can be obtained by adopting the corresponding relations from FJ and XDUCE, taking into account the comments in Section 5 (we have to omit their formal definitions from the paper for the lack of space):

- $\Gamma \vdash \mathtt{e} \in \mathtt{A}$, "in the typing environment Γ, expression e gets type A",
- $\Sigma \vdash \mathtt{e} \Downarrow \mathtt{a}$, "in the value environment Σ, expression e evaluates to value a",
- $\Sigma \vdash \mathtt{e} \not\Downarrow$ "evaluation of e gets stuck in finite number of steps" (this relation is specific to big-step semantics, the analogous property for small-step semantics says that e gets reduced to a non-value, an expression to which none of the evaluation rules is applicable).

[3] We are currently exploring (with Alan Schmitt) some ideas on how to eliminate this limitation. It appears possible to extend the type system of FJ with the boolean operations of intersection, union, and difference on *class* types, with the restriction that these extended types are used only in variable and method declarations and in patterns, not in `new` expressions. This would re-open the possibility of precise type inference for XTATIC.

[4] Note that this method does not allow us to check the exhaustiveness of a sequence of *class* patterns, due to the unavailability of precise unions of class types (the "semantic union" of two classes does not necessarily correspond to a class). This is the reason why our `match` construct is restricted to sequence patterns only.

Write $\mathtt{a} \in: \mathtt{A}$ to mean that $\bullet \vdash \mathtt{a} \in \mathtt{A}'$ for some $\mathtt{A}' <: \mathtt{A}$. A value environment Σ *conforms* to a typing environment Γ, written $\Sigma \in: \Gamma$, if $dom(\Sigma) = dom(\Gamma)$ and $\Sigma(\mathtt{x}) \in: \Gamma(\mathtt{x})$, for all $\mathtt{x}$.

6.1 Theorem [Preservation]: For $\Sigma \in: \Gamma$, if $\Gamma \vdash \mathtt{e} \in \mathtt{A}$ and $\Sigma \vdash \mathtt{e} \Downarrow \mathtt{a}$, then $\mathtt{a} \in: \mathtt{A}$.

6.2 Theorem [Progress]: If $\bullet \vdash \mathtt{e} \in \mathtt{A}$, then not $\bullet \vdash \mathtt{e} \not\Downarrow$.

Both of the standard theorems depend (in the parts of their proofs corresponding to the `match` construct) on the following property of pattern matching, which is interesting in itself. Recall that the object-against-pattern case in our pattern matching relation $\mathtt{a} \triangleright \mathtt{R} \Rightarrow \Sigma$ does not check for the well-typedness of object fields. The property says that, despite of this, if pattern matching is done against a well-typed value a, any binding in the resulting environment is also well-typed.

6.3 Proposition: Let a and A be such that $\bullet \vdash \mathtt{a} \in \mathtt{A}$. If $\mathtt{a} \in \mathtt{R} \Rightarrow \Sigma$ and $\triangleright R \Rightarrow \Gamma$, then $\mathtt{A} <: tyof(\mathtt{R})$ and $\Sigma \in: \Gamma$. (We write $tyof(\mathtt{R})$ for the type obtained from the pattern R by erasing value binding annotations.)

7 XML in FX

So far, none of the mechanisms we have described have been especially tied to XML—we have simply established a generic foundation for representing and manipulating ordered, labeled tree structures in an object-oriented setting. Our final job is to show how this foundation supports a natural encoding of (most of) XML itself, based on a simple form of singleton types and a modicum of syntactic sugar.

We begin by explaining how the textual "leaf data" of XML documents, known as PCDATA (parsed character data), can be treated. Our first step is to extend, conceptually, the C$^{\#}$ data model by introducing singleton classes for individual characters. We assume that the data context *DatCtx* provides a class `Char`, corresponding to the standard C$^{\#}$ character class, plus, for each character c, a class $\mathtt{Char}_\mathtt{c}$ extending `Char`. All these classes have no fields and have nullary constructors—thus, each class $\mathtt{Char}_\mathtt{c}$ contains only a single object, $\mathtt{new\ Char}_\mathtt{c}\mathtt{()}$, which we can identify with the character c itself. Now, a C$^{\#}$ character literal, say 'a', is considered as syntactic sugar for either the object $\mathtt{new\ Char}_\mathtt{a}\mathtt{()}$, when used in an expression, or for the class $\mathtt{Char}_\mathtt{a}$, when used in a type.

We can now define a regular expression type `PCDATA` for representing XML character data:

$$def(\mathtt{PCDATA}) \quad = \quad (\ \mathtt{<Char>[]}\ )*$$

That is, an XML text value is represented by a sequence of trees, where each tree has no children and has a character object as its label. The type `PCDATA` contains arbitrary text strings, so we can write patterns like `<Object>[PCDATA]`, which matches a tree whose body contains only character data.

Why we did not adopt the more obvious choice of using C$^{\#}$'s `String` class to hold XML character data? One reason is that the `PCDATA` representation opens the way to interesting uses of pattern matching for string regular expression processing. Since each $\mathtt{Char}_\mathtt{a}$ is a subtype of `Char`, we can write types that restrict text to a particular form. For example, all character sequences starting with 'a' and ending with 'b' belong to the type

$$\mathtt{<'a'>[]\ PCDATA\ <'b'>[]}.$$

This type, like any XTATIC type, can be annotated with variable binders to obtain a pattern. The general pattern-matching facility, then, offers functionality somewhat similar to that of Perl's regular expression string patterns, but with static typing support. (See [32] for a deeper exploration of this idea.)

Another reason for using `PCDATA` instead of `String` is that, in XML, two character sequences following each other are indistinguishable from a single larger character sequence. The `PCDATA` type satisfies this requirement,

$$\begin{aligned} [\![\mathtt{PCDATA\ PCDATA}]\!] &= [\![\mathtt{<Char>[]*\ <Char>[]*}]\!] \\ &= [\![\mathtt{<Char>[]*}]\!] \\ &= [\![\mathtt{PCDATA}]\!] \end{aligned}$$

but a `String`-based representation does not, since $[\![\mathtt{<String>[]\ <String>[]}]\!] \neq [\![\mathtt{<String>[]}]\!]$.

The encoding of XML documents in XTATIC now follows naturally—all we need is an encoding for XML tags, and this can be obtained by following exactly the same intuitions that we used for characters. We assume that the data context *DatCtx* contains a special class `Tag` and, for each XML tag `<g>`, a singleton class $\mathtt{Tag}_{\mathtt{<g>}}$ (with the object $\mathtt{new\ Tag}_{\mathtt{<g>}}\mathtt{()}$ as its only inhabitant) as an immediate subclass of `Tag`. Then, for an XML fragment

```
<basket> <apple/> <banana/> </basket>
```

the corresponding XTATIC value is

$$\mathtt{<new\ Tag_{<basket>}()>[\ <new\ Tag_{<apple>}()>[]} \\ \mathtt{<new\ Tag_{<banana>}()>[]\ ]}$$

and the corresponding type is

$$\texttt{<Tag}_{\texttt{<basket>}}\texttt{>[<Tag}_{\texttt{<apple>}}\texttt{>[] <Tag}_{\texttt{<banana>}}\texttt{>[]]}$$

Of course, an implementation needs special syntax that makes these values and types readable (and even writable!). The concrete syntax in our current prototype implementation looks very close to standard XML.

Together, the encodings of character data and tags allow a good-size fragment of XML to be represented very directly in FX. (There are still important parts missing, though. Most urgently, we still lack a good treatment of attributes which, until very recently [14], was also lacking in XDuce.)

The only basic data type provided by the XML standard is character sequences. Some schema formalisms, however, introduce *datatypes*—a set of conventions by which a schema can specify that a particular textual fragment in an XML document is supposed to represent a non-textual value, e.g. a float or a date. Some of these datatype descriptions can be captured using subtypes of `PCDATA` built from regular expression operators to mimic the string regular expressions that describe particular datatype formats. Alternatively, the FX framework could accommodate a Schema-datatype-aware encoding of XML, when a text representing a Schema datatype value gets translated directly into a value of an appropriate C$^{\#}$ type (placed as a label of a childless tree), bypassing the `PCDATA` representation.

8 Related Work

There is a substantial literature (and many formalisms and tools) addressing dynamic validation of XML documents against expected schemas, either by stand-alone processors or during document construction, as has been proposed for DOM Level 3 [10]. While Xtatic shares some formal background with these techniques, its central goal—to support *static* checking of XML-manipulating code—falls completely outside their purview.

Among static approaches, there are two overlapping kinds of work that are directly relevant to ours: (1) work on providing XML processing capabilities in pre-existing programming languages with static guarantees of correctness, and (2) work on combining object-orientation with XML-like data models.

A popular direction for work of the first kind is to provide a translation that generates type definitions (and value constructors) in the original language corresponding to XML types of interest. Examples include JAXB [31], Relaxer [27], HaXML [34], and XMλ [23, 29]. One disadvantage of these translations is that they tend to introduce "spurious structure," destroying some useful flexibility in the subtype relation. This point is discussed in detail in [17] and [13].

There can be varying degrees of integration of a "foreign" data model into the object-oriented data model. One is creating a combined data model that incorporates the features of both on the equal level. A successful example is the ODMG data model [6], an accepted standard for object-oriented databases, which offers a class-based object-oriented type system analogous to that of programming languages like C$^{\#}$, together with a few other built-in type constructors: records, sets, bags, lists and arrays (all of them typed).

A greater degree of integration can be achieved by taking the object-oriented data model as primary and the other data model as subsidiary, in the sense that its values can also be viewed as objects. This approach has the advantage of better integration with legacy software written entirely under the original object-oriented model. Examples of this approach can be found in both the programming language and database communities.

The Pizza [25] project extended Java with parametric polymorphism, higher-order functions, and tagged union types with pattern matching. (The polymorphism component became the basis for the GJ [5] proposal for adding generic types for Java.) All these features where implemented by translation into pure Java in such a way that the extended data model is used to typecheck Pizza source, while run-time representations of the additional features are objects of either pre-determined Java classes (for their *homogeneous* translation), or of classes generated from the Pizza source (for the *heterogeneous* translation). We plan to use a scheme analogous to the homogeneous translation in the final implementation of Xtatic. Current work on the programming language Scala [24] is aimed at incorporating many of the same ideas as Pizza in a language aimed at programming Web services, including XML processing.

Even before XML became popular, the database community was actively investigating the management of "semistructured" data; the Object Exchange Model (OEM) [26] is a popular formalism in this area. An OEM data value is a directed graph (often just a tree) with edges labeled by tags, internal nodes containing unique identifiers, and leaf nodes containing atomic values (integers, strings, etc.).

The combination of ordinary algebraic types with objects in the ODMG data model proves to be rather inflexible for working with semistructural data, as it involved encodings within the structural ODMG model, which are usually complex and difficult to manage and evolve. The Ozone project [21] approached this problem by integrating the OEM data model into the ODMG data model. Their solution is similar to ours at the level of values: first, the OEM model is generalized to allow arbitrary ODMG values, including objects and structural values as leaves; second, a special ODMG class, `OEM`, is designated to hold all OEM values. The OEM values are ultimately implemented as objects of `OEM` subclasses. The OEM data model, however, is not statically typed. The motivation for Ozone was to allow convenient manipulation of semistructured data in an object-oriented database while avoiding the overly strict ODMG typing restrictions. Our contribution can be viewed as proceeding from the observation that an Ozone-like integration of objects and semistructured data can be carried out in a fully typed way, once an appropriate alternative to algebraic types (i.e., regular expression types) is identified.

Two ongoing language design efforts that are very close to XTATIC in intentions and approach are CDUCE and JWIG. JWIG [7] is an extension of Java intended for programming interactive sessions between Web servers and clients. Although quite different in style from XTATIC (it uses data flow analysis to check well-formedness of XML expressions constructed by filling in "templates", rather than a conventional type system and tree expression language), the basic expressive power of Jwig's analysis is close to that of XDUCE's type system; see [7] for a detailed discussion of this point.

CDUCE [4], similarly to XTATIC, aims to introduce XDUCE features into an object-oriented type system. The latter, in case of CDUCE, is λ-&—a variant of λ-calculus with overloading, commonly used as a formalism for multiple-dispatch OO languages. As XTATIC, CDUCE adopts the semantic interpretation of subtyping to a larger type system [12]. The major difference is the need to give semantic interpretation of arrow types, which does not arise in XTATIC. Assuming that functional types are to be inhabited by function closures, a natural equation for them (in the spirit of equations in Fig. 2) would be

$$[\![\mathtt{A}\to\mathtt{B}]\!] = \{\, \mathtt{fix\ f(x:A'):B'.e} \mid [\![\mathtt{A}]\!] \subseteq [\![\mathtt{A'}]\!], [\![\mathtt{B'}]\!] \subseteq [\![\mathtt{B}]\!] \,\}$$

However, it does not give rise to a monotone operator on "denotation candidates", so the denotation function $[\![\cdot]\!]$ cannot be guaranteed to exist as the least fixed point of the operator. In contrast to XDUCE and XTATIC, CDUCE has to justify existence of $[\![\cdot]\!]$ by other means. A further complexity dimension in CDUCE is the support for the full set of boolean type constructors (including intersection and difference).

9 Future Work

We are currently experimenting with a prototype interpreter for XTATIC. Though it still lacks most of the features of $C^{\#}$, the language implemented by the interpreter goes quite a bit beyond the simple FX core described here—in particular, it includes imperative features, interfaces, and overloading; we have used it to experiment with a number of small demos. Our immediate goals include handling a larger fragment of $C^{\#}$, building more ambitious demos, and replacing the simple interpreter by a back end targeting the .NET Common Language Runtime.

Another important near-term goal is to extend the type system to encompass a larger part of XML—most urgently, attributes. Hosoya and Murata [14] have recently proposed a typing mechanism and corresponding algorithms based on the attribute-element constraints of Relax NG; we hope to be able to adapt this proposal to XTATIC. We also plan to implement translators from standard XML schema languages (in particular, a subset of the XML-Schema standard) into XTATIC.

Our longer-term goals concentrate in two major areas: improving the efficiency of the underlying algorithms and run-time representations, and refining and extending the design of the core language. On the efficiency side, the main development currently in the works is high-performance compilation of pattern matching. We also need to come up with better run-time representations for certain special cases, while keeping compliance with the basic data model. One case in point is the `PCDATA` type. The typing and pattern-matching properties of the `PCDATA` definition given in Section 7 are attractive, but the naive representation that we sketched is clearly too heavy to perform well; something more clever will be needed. At the level of the core language design, there are also numerous questions to be considered. Can objects and trees be further unified? E.g., could pattern matching be used to extract object fields? Could attributes and fields be unified? Can we offer other kinds of pattern matching primitives, e.g. support for XPath? And, last but not least, can the XTATIC design be extended to cope with parametric polymorphism ("generics" in $C^{\#}$ parlance)?

Acknowledgements

Michael Levin has been closely involved in the XTATIC design and contributed many ideas to this formalization of the core. We are also grateful to Haruo Hosoya, originator of XDUCE, and to Alan Schmitt for numerous conversations, to Matthias Zenger, Vincent Cremet, and Martin Odersky for helping work out some of the basic insights in the XML encoding, to Todd Proebsting, Eric Meijer, and Wolfram Schulte for insights into the world of XML processing in C# and its implications on language design, to Peter Buneman for the name XTATIC, and to Arnaud Sahuguet for making us aware of Ozone and for general encouragement and enthusiasm.

Our work on XDUCE and XTATIC has been supported by the National Science Foundation under Career grant CCR-9701826 and ITR CCR-0219945 and by a gift from Microsoft.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Nov. 2002.

[2] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for java exceptions. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 16–30, 2001.

[3] D. Ancona and E. Zucca. True modules for java-like languages: Design and foundations, Aug. 2000. Technical Report DISI-TR-00-12, Dipartimento di Informatica e Scienze dellInformazione, Università di Genova.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: a white paper. `ftp://ftp.ens.fr/pub/di/users/castagna/cduce-wp.ps.gz`, 2002. Workshop on Programming Language Technologies for XML (PLAN-X).

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, Oct. 1998.

[6] R. Catell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[7] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Extending Java for high-level web service construction. `http://www.brics.dk/~mis/jwig.ps`, 2002.

[8] J. Clark. TREX: Tree Regular Expressions for XML. `http://www.thaiopensource.com/trex/`, 2001.

[9] J. Clark and M. Murata. RELAX NG. `http://www.relaxng.org`, 2001.

[10] Document object model (dom) level 3 validation specification, w3c working draft. `http://www.w3.org/TR/DOM-Level-3-Val`, 2002.

[11] M. F. Fernández, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In J. V. den Bussche and V. Vianu, editors, *Proceedings of 8th International Conference on Database Theory (ICDT 2001)*, volume 1973 of *Lecture Notes in Computer Science*, pages 263–300. Springer, 2001.

[12] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *LICS*, 2002.

[13] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, 2000.

[14] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2002.

[15] H. Hosoya and B. Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version to appear in *Journal of Functional Programming*.

[16] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[17] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology*, 2002. Submitted for publication.

[18] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2001. To appear; short version in ICFP 2000.

[19] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[20] A. Igarashi and B. C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). To appear in *Information and Computation*.

[21] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: integrating structured and semistructured data. In *International Workshop on Database Programming Languages*. 1999.

[22] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, 2002.

[23] E. Meijer and M. Shields. XMλ: A functional programming language for constructing and manipulating XML documents. Submitted for publication, 1999.

[24] M. Odersky. Report on the programming language scala. `http://lamp.epfl.ch/~odersky/scala/reference.ps`, 2002.

[25] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

[26] Y. Papaconstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *International Conference on Data Engineering*, Mar. 1995.

[27] Relaxer. `http://www.asahi-net.or.jp/~dp8t-asm/java/tools/Relaxer/index.html`.

[28] U. P. Schultz. Partial evaluation for class-based object-oriented languages. In *Programs as Data Objects (PADO), Aarhus, Denmark*, volume 2053 of *Lecture Notes in Computer Science*, pages 173–197, 2001.

[29] M. Shields and E. Meijer. Type-indexed rows. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, Jan 2001.

[30] T. Studer. Constructive foundations for featherweight java. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*. Springer-Verlag, 2001. Lecture Notes in Computer Science, volume 2183.

[31] I. Sun Microsystems. The Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb`, 2001.

[32] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In J. V. den Bussche and V. Vianu, editors, *Proceedings of Workshop on Types in Programming (TIP)*, pages 1–18, July 2002.

[33] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), Nov. 1990.

[34] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9 of *ACM SIGPLAN Notices*, pages 148–159, N.Y., Sept. 27–29 1999. ACM Press.

[35] Extensible markup language (XML™), Feb. 1998. XML 1.0, W3C Recommendation, `http://www.w3.org/XML/`.

[36] XML Schema Part 0: Primer, W3C Working Draft. `http://www.w3.org/TR/xmlschema-0/`, 2000.

[37] XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[38] M. Zenger. Type-safe prototype-based component evolution. In *European Conference on Object-Oriented Programming (ECOOP), Malaga, Spain*, June 2002.