A New Approach to Efficient XML-Processing by Lazy Functional Programming

Pok-Son Kim, Arne Kutzner
Kookmin University, Department of Mathematics, Seoul 136-702, Korea
pskim@kookmin.ac.kr
Seokyeong University, Department of E-Business, Seoul 136-704, Korea
kutzner@skuniv.ac.kr

Abstract

We present the design and syntax of a lazy evaluating functional programming language for XML-transformations. The introduced language is in the tradition of Haskell and allows the handling of XML-transformations in a descriptive style. The kernel of our approach is a sophisticated pattern matching mechanism, that performs a translation of patterns to core language expressions. The proposed language and techniques allow highly efficient XML-processing, even for huge documents. We also report about a prototype-implementation for our language and compare our approach to others.

Introduction

During the last decade the Extensible Markup Language (XML)[3] has become a standardized medium for the platform-independent representation of structured data. Its transparent and self-describing nature makes XML well suitable for the data-exchange between distinct systems. Such a data exchange usually causes the need for document transformations, a standard case is the visualization of XML-documents in the context of WEB-applications.
Functional programming languages are well applicable in the field of transformations because of their characteristic "to map things". Inside the area of functional programming languages different evaluation strategies for program execution have been devised. One of theses strategies is lazy evaluation, also known as normal order evaluation. When using lazy evaluation, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. This allows programs to act in an "on demand"-manner. A popular lazy evaluating functional programming language is Haskell[7].

In this article we present a functional programming language, called XTL (**X**ML **T**ransformation **L**anguage), which has been designed especially for efficient XML transformations. XTL comprises a powerful pattern matching language adapted to the needs of XML processing. The operational semantics of XTL implements lazy evaluation. Almost all existing systems rely on some tree representation of the XML-document to be transformed, and apply some patterns-based mechanism to achieve a transformation. This design decision leads to difficulties regarding the handling of huge documents, because such a tree representation must be generated and hold somewhere. Laziness offers an elegant solution to overcome these disadvantages. XTL has been designed for usage in high load environments and for the efficient processing of huge documents. XTL-programs can be compiled to native code or the virtual machine code like for the Java-Virtual Machine. XTL conceptually doesn't need any intermediate interpretation for pattern matching or tree processing and knows layout-rules like Haskell for improving the readability of programs.

First we give an overview about XTL and show some code-example. Subsequently a short introduction to the XTL-language is given, where we particularly present an operational semantics for our pattern language. We inform about the compilation process with XTL and introduce our prototype implementation. Finally we remark some open points of XTL in its current form and give ideas for future improvements.

Overview of XTL

XTL borrows a lot of syntactical elements as well as their semantic interpretation from Haskell [7]. Roughly spoken XTL is a functional core-language in Haskell-tradition combined with a special purpose pattern-matching mechanism, all together adapted to the needs of XML-processing.

We now give a short introduction to XTL by showing how to produce a table-based HTML-representation for a XML-document that contains information about some CD-collection.
Assume our CD-collection is described by the following XML-document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
   <cd>
      <title>Greatest Singers - Vol 1</title>
      <artist>Enrico Caruso</artist>
   </cd>
   <cd>
      <title>La Traviata</title>
      <artist>Maria Callas</artist>
   </cd>
</catalog>
```

Further assume that we want to turn the above XML-document into the following HTML-representation:

```
<html> <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="green">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <tr> <td>Greatest Singers - Vol 1</td>
         <td>Enrico Caruso</td>
    </tr>
    <tr> <td>La Traviata</td>
         <td>Maria Callas</td>
    </tr>
  </table>
</body> </html>
```

We can use the code shown below, to achieve the necessary transformation in XTL:

```
tmap f (x:Any, xs) = let
                        y = case x of
                               Tag -> f x
                               _   -> x
                     in y, (tmap f xs)
tmap _ () = ()

mapCD <cd> <title> t </title>,
           <artist> a </artist>, _ </cd>
    = <tr>
         <td> t </td>,
         <td> a </td>
      </tr>

main <catalog> xs </catalog>
    = <html>
      <body>
         <h2> "My CD Collection" </h2>,
         <table border="1">
            <tr bgcolor="green">
                <th> "Title" </th>,
                <th> "Artist" </th>
            </tr>,
            tmap mapCD xs
         </table>
      </body>
      </html>
```

The above XTL-code consists of three function definitions, where `main` is the starting point of the evaluation. Each function definition describes some pattern based mapping, where the syntax for patterns is XML adapted. For example, the pattern `<catalog> xs </catalog>` of the function `main` checks if the actual argument of `main` is a `catalog` element and if so, it binds `xs` to the body of that element. Altogether the `main`-function extracts the sequence of all items comprised by a `catalog` element, puts them into a HTML-template and applies `tmap` in combination with `mapCD` for creating the table-rows inside the template. `tmap` is a higher order function and applies the function given as first argument to all elements of the sequence given as second argument. It is similar to the standard map function with classical functional programming languages. `mapCD` places `title` and `artist` inside a table-row.

XTL is a lazy evaluating language, therefore during pattern matching expressions are only evaluated as far as necessary for making a match-decision. In this context the special pattern _ is of particular interest. The pattern _ matches immediately without any further evaluation of the expression under inspection. So it even matches expressions, without any reduction to some head normal form[1]. XTL also knows patterns for sets of equivalent XML-items. The pattern `Tag` in the function `tmap` is such a pattern, it matches every XML-element no matter which tag name. For improving the code legibility XTL knows the layout rules of Haskell.

The above XTL program is identical to the XSLT style sheet shown below, both generate the same HTML-code presented before.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="green">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
      <tr>
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="artist"/></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

We would like to remark, that the above XSLT style sheet depends on the predefined operations `xsl:value-of` and `xsl:for-each`. The XTL-code doesn't rely on such predefined functionality, it comprises all necessary code for the transformation.

[1] "head normal form" is a notion often used in the context of normal order reduction. Roughly spoken an expression is in head normal form if there is some not reducible element (e.g. a data item or a variable bound to a data item) at top level.

Language Definition

We will now give a formal language definition for XTL. We start with the definition of values and will continue with the introduction of the pattern language and term language.

Values

Values are abstract representations of XML-documents. They are defined inductively as follows:

1. The empty sequence, denoted by $()$, is a value.
2. Each character c is a value.
3. Let v be some value and n be some XML-compliant Tag-name, then $<n>v</n>$ is a value.
4. Let v and v' two values, then the sequence v, v' is a value.

We use the above definition for giving patterns a operational semantics. Because there is no attribute handling on pattern level, the above definition doesn't comprise any representation for attributes.

Pattern Language

We assume some countably infinite set of variables X. x, t shall denote some element of X and n shall denote some XML-compliant tag-name. XTL pattern obey the following generalized syntax:

P	$::=$	$x : P$	Pattern Variable Binding
		`A`	Abstract Pattern
		_	Top
		$'c'$	Character
		$P_1 \mid P_2$	Choice
		$P_1\ ,\ P_2$	Concatenation
		$P*$	Repetition
		$<n>P</n>$	Named Tag
		$<(t)>P</>$	Tag Variable Binding
		$()$	Empty Sequence

Additionally we allow the constructs $P+$ and $P\,?$, where $P+$ is an abbreviation for $P\ ,\ P*$ and $P\,?$ is an abbreviation for $P|()$. A Abstract Pattern `A` is one of the reserved words `Any`, `Tag` or `Char`, where in turn `Any` is an abbreviation for `Tag` | `Char`. Patterns of the form $<n>\ ()\ </n>$ can be abbreviated to $<n/>$. The specification of x : _ can be abbreviated to x merely.

To avoid semantic problems pattern must fulfill some additional restrictions:

(R1) Top[2] must neither occur inside a repetition nor be part of the left sub-pattern of a concatenation. The only exception is that Top is covered by some named-tag-pattern.

(R2) Variable bindings (Pattern Variable Bindings as well as Tag Variable Bindings) inside repetitions or choices are prohibited.

We now describe the semantics of patterns by means of values. For this purpose we define a relation $v \in P \Rightarrow (\mathrm{B}, \mathrm{T})$, read "$v$ is matched by P, yielding (B, T)", where B is an environment that maps variables to values and T is an environment that maps variables to tag-names.

$$() \in () \Rightarrow (\emptyset, \emptyset) \tag{P1}$$

$$() \in P* \Rightarrow (\emptyset, \emptyset) \tag{P2}$$

$$\text{for all values } v : v \in _ \Rightarrow (\emptyset, \emptyset) \tag{P3}$$

$$\text{for all characters } c : \ c \in 'c' \Rightarrow (\emptyset, \emptyset) \tag{P4}$$

$$\frac{v \in P \Rightarrow (\mathrm{B}, \mathrm{T}), \quad v' \in P' \Rightarrow (\mathrm{B}', \mathrm{T}')}{v, v' \in P, P' \Rightarrow (\mathrm{V} \cup \mathrm{V}'\ , \mathrm{T} \cup \mathrm{T}')} \tag{P5}$$

$$\frac{v \in P \Rightarrow (\emptyset, \emptyset)}{v \in P \mid P' \Rightarrow (\emptyset, \emptyset)} \tag{P6}$$

$$\frac{v \in P' \Rightarrow (\emptyset, \emptyset)}{v \in P \mid P' \Rightarrow (\emptyset, \emptyset)} \tag{P7}$$

$$\frac{n \geq 1, \text{ for all } v_i \in \{v_1, \ldots, v_n\} \ : \ v_i \in P \Rightarrow (\emptyset, \emptyset)}{v_1, \ldots, v_n \in P* \Rightarrow (\emptyset, \emptyset)} \tag{P8}$$

$$\frac{v \in P \Rightarrow (\mathrm{B}, \mathrm{T})}{v \in x : P \Rightarrow (\mathrm{B} \cup \{(x, v)\}, \mathrm{T})} \tag{P9}$$

$$\frac{v = <n>v'<n>,\ v' \in P \Rightarrow (\mathrm{B}, \mathrm{T})}{v \in <n>P<n> \Rightarrow (\mathrm{B}, \mathrm{T})} \tag{P10}$$

$$\frac{v = <n>v'</n>,\ v' \in P \Rightarrow (\mathrm{B}, \mathrm{T})}{v \in <(t)>P</> \Rightarrow (\mathrm{B}, \mathrm{T} \cup \{(t, n)\})} \tag{P11}$$

$$\frac{P \in \mathrm{A},\ v \in P \Rightarrow (\mathrm{B}, \mathrm{T})}{v \in \mathrm{A} \Rightarrow (\mathrm{B}, \mathrm{T})} \tag{P12}$$

[2] The notion 'Top' was chosen due to the semantics of the underscore that matches every expression (even expressions that trigger a non-terminating evaluation) without any further evaluation.

The abstract pattern `Char` is the set of pattern consisting of character patterns `'`c`'` for every character c. The abstract pattern `Tag` is the set pattern consisting of patterns $<n>$_$<n>$ for every XML-compliant tag name n. Hence according to the above semantics `Char` matches one character and `Tag` matches some arbitrary XML-element.

The two syntactical restrictions (R1) and (R2) for patterns avoid ambiguities in the above semantics regarding variable bindings. The first restriction assures, that the decomposition of sequences by a concatenation or repetition stays unique. If we omit this restriction, patterns like `x` : _ , `y` : _ would raise an ambiguity regarding the composition of sequences. By denying the specification of variable bindings inside choices and repetitions we assure that the arising sets of bindings stay empty. Without this restriction we would get the following problem: In the special case that both alternatives of some choice are matching and additionally express different sets of bindings, there is an ambiguity with respect to the decision which set of bindings shall be regarded as the result of the choice.

The above semantics bases on an inductively defined set of values. However, patterns are applied to expressions and, unless values, expressions may result in an infinite computation without providing any result. It is a standard technique to represent this kind of "no information" by the symbol $\perp$ [3], pronounced 'bottom'. We will now give some notes on $\perp$-handling in XTL:

We start with a deeper inspection of the top-pattern '_'. The exact operational semantics of the top-pattern is an immediate match without any further evaluation of the expression under inspection. Hence, from a value domain point of view, top matches all values including $\perp$. This has a significant impact regarding the two patterns `Any`$*$ and _. Although at first glance semantically identical they show a different behavior in the context of $\perp$. The first pattern results in an infinite computation, the second one in an immediate match.
There is a further impact of $\perp$ on choices. Internally XTL evaluates choice-patterns in a left-right manner, so the second alternative is only touched if the first alternative doesn't match. This is a side-effect of the construction principle used for the translation of patterns into core language expressions. However, this evaluation strategy leads to a lack of commutativity with choice patterns in the context of $\perp$. An example give the two patterns _ | `Any` and `Any` | _. They deliver different results if they are matched against a$\perp$-expression. The first one provides an immediate match, the second one triggers an infinite computation.

Finally we would like to remark two points. XTL is a lazy evaluating language, even on pattern level. So during pattern matching the expression under inspection is only evaluated so far like absolute necessary for making the match-decision. At the moment XTL doesn't comprise a concept for the handling of attributes in patterns.

[3]A extensive introduction to the semantics of the $\perp$ symbol can be found in [6].

Term Language

An XTL-program consist of one or several declarations, where each declaration D has the following structure:

$$D \quad ::= \quad f\, P_1 \,\cdots\, P_n \;=\; E \qquad \text{Declaration}$$

In the above definition E is some expressions that obeys the following syntax:

E	::=	`let` $D_1 \cdots D_n$ `in` E	Let Clause
		where $n \geq 1$	
		`case` E `of`	Case Clause
		$P_1 \rightarrow E_1$	
		$\vdots$	
		$P_n \rightarrow E_n$	
		where $n \geq 1$	
		E_1 , E_2	Concatenation
		$E_1\ E_2$	Application
		x	Variable
		`'`c`'`	Character
		$<n\ A_1 \cdots A_z> E </n>$	Fixed Element
		where $z \geq 0$	
		$<(x)\ A_1 \cdots A_z> E </>$	Variable Element
		where $z \geq 0$	
		()	Empty Sequence

Each $A_i (i = 0, 1, \cdots, z)$ is an attribute of the form n = " v ", where n is some XML-compliant attribute name and v the attribute's value.
The Let-clause binds the named functions defined be the local declarations $D_1 \cdots D_n$ in the expression E. A case clause checks whether the expression E matches on of the patterns P_1until P_n. The process starts by the first pattern $P_1$and proceeds pattern by pattern until one pattern matches. Two expressions are concatenated by the comma-operator, where the operational semantics of the comma-operator is similar to a classical string-concatenation. The fixed element and the variable element are for constructing new elements, where the syntax is adapted to the tag-syntax of XML.

Compilation Process

Conceptually XTL programs are compiled to either native code or some virtual machine code by a sequence of separated translations. A graphical description of the stepwise compilation process gives Fig. 1. In a first step XTL programs are compiled to a special intermediate code, called enriched core code. Expressions of the enriched core code obey the following syntax:

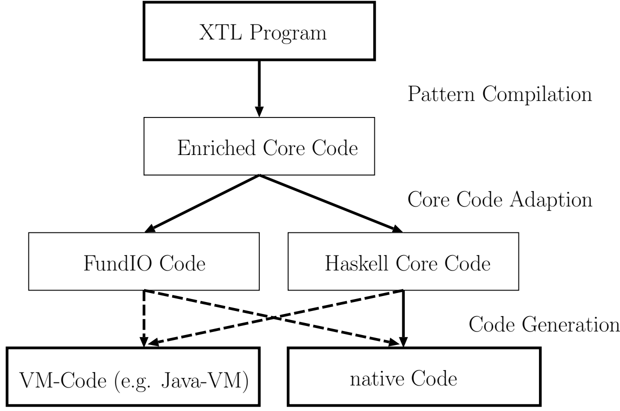

Figure 1: The XTL compilation process

$$
\begin{array}{lcll}
e & ::= & \textbf{letrec}\ f_1\, x_1^1 \cdots x_1^{n_1} \ = e_1 & \text{LetRec} \\
 & & \quad\quad\quad \vdots & \\
 & & \quad f_m\, x_m^1 \cdots x_m^{n_m} = e_m & \\
 & & \textbf{in}\ e & \\
 & & <n\, A_1 \cdots A_n> \ \ e & \text{Element} \\
 & & e_1 e_2 & \text{Application} \\
 & & \lambda v.e & \text{Abstraction} \\
 & & x & \text{Variable} \\
 & & \textbf{'}c\textbf{'} & \text{Character} \\
 & & p & \text{Combinator} \\
 & & \textbf{err} & \text{Error}
\end{array}
$$

The enriched core code comprises a set of 13 combinators with a predefined operational semantics each. 6 of these combinators perform some kind of atomic pattern matching on enriched core code level. 4 combinators are responsible for sequence construction and concatenation, the remaining 3 are the standard combinators I, K, B [4]. Conceptually all these combinators have a concrete implementation in the next stage of the compilation process (Haskell core code, FundIO).

The kernel of the first compilation step is a sophisticated pattern compilation. The generation of enriched core code for a XTL-pattern happens in a two step process. In a first step a special kind of nondeterministic finite automaton (NFA) for the XTL-pattern is generated. Subsequently the behavior of this NFA is implemented by an enriched core code expression. So patterns are really compiled at compile time, there is no interpretation during runtime, the generation of NFA's is only a intermediate step during compilation.

The enriched core code is translated either to Haskell core code or FundIO code. This intermediate step consist of a rather easy code adaption. By producing Haskell core code we can take advantage of existing Haskell compilers like the Glasgow Haskell Compiler (GHC).

The last step of compilation, the generation of native code for some target platform, is the responsibility of the Haskell compiler.

[4]The definition of the combinators I, K, B can be found in [4].

Some arrows in Fig. 1 have broken lines. These arrows represent translations that are impossible in the moment, either because there's no concept available or existing concepts are not practically usable. Particularly the production of Java-VM code for Haskell code is such a point.

Prototype

We have developed a prototype implementation for XTL for getting an impression of the practical usability of our approach. The prototype can compile some XTL-program to standard Haskell code which in turn can be compiled by any standard Haskell compiler to some executable file. First practical experimentation provided very promising results. On basis of this prototype we plan to provide concrete benchmarks soon.

Related work

Transformation of XML documents is a topic with a long tradition of inspection. Numerous tools and approaches for XML transformations have been developed as result of the research efforts in this area. We will pick some of these approaches which we think are interesting in the context of XTL.

The style sheet language XSLT [8] has gained wide attention during the last decade. XSLT processes a tree representation of a XML-document by applying template-based mapping rules, so from its basic nature it is functional programming language. Despite its powerful expressibility XSLT comprises some significant drawbacks. First of all XSLT style sheets are interpreted from conceptual point of view, which results in rather poor performance of most XSLT-engines. A further drawback is the representation of XML-documents as a tree shaped data structure. Huge XML-documents cause huge tree models, probably too huge to keep them in the internal memory. And worse, if we need only few elements of some huge XML-document a lot of unnecessary parsing can be caused by the generation of untouched segments of the tree structure. The only way to overcome such problems is to act in an "on demand manner", this means to keep only vital parts of a huge data structure in memory. Laziness does this in a very natural style. So XTL opens a way for processing huge documents efficiently.

Another well known language for XML-transformations is XQuery [2]. XQuery has shared several design details with XSLT, e.g. the tree shaped interpretation of XML-documents (XSLT and XQuery rely on XPath). But the syntax of XQuery has been designed for queering databases formed by XML-documents.

Other approaches for processing XML are the languages XDuce [5] and CDuce[1], either of them have a strong relationship to the functional programming language ML. But the focus of both languages is different to ours. XDuce applies the concept of regular expression types

for static type checking at compile time. Another feature of XDuce is some form of pattern matching derived from that type system, called regular expression pattern matching. Unlike XTL there's no compilation of patterns into core language, instead there's an intermediate interpretation at runtime. CDuce is a development motivated by XDuce and aims to be an "XML-centric general purpose language". It overcomes several limitations of XDuce regarding the type system, language design and run-time system. The unpublished language XMλ from Meijer and Shields [9] was further work with the focus on types as XDuce.
Finally we want to mention that there are approaches to handle XML directly in Haskell, for example the concept of HaXML as introduced in [11]. However, such approaches are pure libraries, where the decomposition of XML-documents is described by special combinators.

Conclusion

We presented a functional programming language called XTL designed for efficient processing of XML documents. XTL comprises a comfortable pattern language that is adapted to the needs of XML-document decomposition. Patterns are compiled to enriched core code during the compilation process. There is no intermediate interpretation of patterns at runtime as with most other approaches for XML processing. Lazy evaluation in XTL causes a processing of XML-documents in an "on demand manner". This aspect is especially important for the efficient processing of huge documents.
First practical experimentation with a prototype of XTL delivered encouraging results. We plan to provide benchmarks soon in order to show that our approach delivers significant performance advantages compared to similar languages like XSLT or XDuce.

The design work on XTL has not been finished yet, there are still several open points. I/O is such a point. I/O with lazy evaluating functional languages is a nontrivial problem, several solutions have been proposed and implemented in the last decade like e.g. monadic I/O in todays Haskell. We plan to include the concept of unsafe I/O into XTL. For this purpose we intent to rely on the work of Schmidt-Schauß [10] about unsafe I/O. Another point is a type model on XML-level like for example the regular expression types in XDuce. A lot of work has been done in this area, in [5] Hosoya and Pierce give an overview about the recent results . We plan to include one of the existing concepts into XTL but in the style of optional assertions, that are statically checked at compile time. Further we intend to develop a standard prelude for XTL that delivers a wide range of predefined functions for document-processing including a tiny XML-parser.
Altogether our future work aims at a further development of XTL so that it becomes publicly accepted alternative to the widespread XSLT.

References

[1] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, volume 38, 9 of *ACM SIGPLAN Notices*, pages 51–63, New York, August 25–29 2003. ACM Press.

[2] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, November 2003.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/2004/REC-xml11-20040204`, April 2004.

[4] J. R. Hindley and Seldin J. P. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge, 1986.

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

[6] Simon L. Peyton Jones. *The Implementation of functional programming languages.* Prentice-Hall, 1987.

[7] Simon L. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003.

[8] Michael Kay. XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/xslt20/`, November 2003.

[9] Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[10] Manfred Schmidt-Schauß. FUNDIO: A Lambda-Calculus with a `letrec`, `case`, Constructors, and an IO-Interface: Approaching a Theory of `unsafeperformio`. Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, September 2003.

[11] Malcolm Wallace and Colin 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, pages 148–159, N.Y., 27–29 1999. ACM Press.