Statically Typed Document Transformation: An XTATIC Experience

François Garillot

Rapport de stage

20 septembre 2005

Résumé

XTATIC est une extension du langage C♯ qui introduit un support natif pour le traitement de XML statiquement typé. Il possède ainsi :

- Un système de types basé sur des expressions régulières, généralisation naturelle des DTDs, permettant de programmer des types à l'aide de d'opérateurs courants de ce domaine.
- Un moteur de *pattern-matching* correspondant à ces expressions, permettant de capturer des valeurs d'un type donné à l'aide d'expressions "à la grep".

XTATIC est une extension de C# à la syntaxe légère, étroitement intégrée à son langage d'origine, représentée pratiquement par un compilateur de source à source produisant du code C# natif, complètement compatible avec les APIs et binaires usuels de ce langage. Elle permet de manipuler les arbres de XML comme des valeurs intégrées, et possède un système de types basé sur des types réguliers à la XDUCE, ainsi que des expression régulières conçues pour l'investigation et la manipulation de XML.

Nous fournissons ici un rapport de notre expérience d'utilisation de XTATIC dans une application à l'échelle réelle : un programme pour la transformation de spécifications de XML au format XMLSPEC (utilisé pour l'édition des rapports techniques et recommendations du *World Wide Web Consortium*) en XHTML. Notre implémentaiton suit de très près l'implémentation existante, écrite en XSLT, facilitant ainsi la comparaison entre ces deux langages, et l'analyse des coûts et bénéfices d'un typage statique riche pour du code comportant une manipulation intensive de XML.

Après avoir décrit le contexte de Xtatic dans une introduction, notre première partie, *The* XTATIC *Language*, est une reprise intégrale de l'introduction donnée à XTATIC par V. Gapeyev, M.Y. Levin et B. Pierce dans [1]. Nous décrivons ensuite le problème précis qui fut le centre du travail effectué durant ce stage : la transformation d'un programme non typé d'envergure, dont le source est rédigé en XSLT, un langage spécialisé de traitement de XML, en un programme typé rédigé en XTATIC. Après avoir décrit les spécficités et le contexte de cette transformation, et en quoi elle a ambition à servir de modèle de

référence pour une tâche de conversion de fichiers XML, nous précisons la définition de la correspondance établie entre le programme non typé originel et notre programme typé final, abstraction qui servit de fil directeur à notre travail.

La description, dans la partie 3, d'un procédé de typage aussi systématique et rigoureux que possible nous permet ainsi de singulariser les éléments distinctifs d'une transformation d'un programme XSLT non typé à un programme XSLT typé : l'inférence de type (3.4, 3.5) et la résolution des bugs de typage (3.6) sont abordés dans le but de pouvoir adapter aisément notre approche à un programme XSLT quelconque, et nous permettent aussi de définir quelques conditions du typage statique d'un programme non typé (3.3) Nous décrivons par ailleurs dans la partie 4 notre façon de réimplémenter les fonctions prédéfinies de XSLT, en montrant que les fonctionnalités que nous pouvons émuler égalent pleinement celles du langage d'origine que nous aovns choisi.

Enfin, notre conslusion évoque la possibilité et les conditions d'une généralisation de notre travail à n'importe quel programme défini en XSLT, et une analyse des avantages et inconvénients de l'utilisation d'un système de types strict. Elle esquisse également les perspectives de XTATIC comme langage de traitement de XML.

Le programme implémenté et le thème de ce travail servent actuellement de fondement à un rapport technique de l'université de Pennsylvanie, en cours de préparation.

1 Introduction

The recent rush to adopt XML can be attributed in part to the hope that the static typing provided by DTDs (or more sophisticated mechanisms such as XML-Schema) will improve the robustness of data exchange and processing. However, although XML documents can be checked for conformance with DTDs, current XML processing languages offer no way of verifying that programs operating on XML structures will always produce conforming outputs. In earlier work at Penn,a domain-specific language for XML processing, called XDuce has been designed and implemented. The main novelties of XDuce are :

1. A type system based on regular expression types. Regular expression types are a natural generalization of DTDs, describing structures in XML documents using regular expression operators (*, ?, |, etc.) and supporting a powerful form of subtyping.
2. A corresponding mechanism for regular expression pattern matching, which supports concise "grep-style" patterns for extracting information from inside structured sequences.

The lessons learned from XDuce have recently been incorporated in a new language, called Xtatic, whose design focuses on smooth integration of these novel XML-processing features into mainstream, object-oriented languages such as C♯. Xtatic is a lightweight extension to C♯, offering native support for regular expression types and patterns and completely interoperable at the binary level with ordinary C♯ programs and APIs.

In this report, we will start by outlining the main aspects of the Xtatic language, summarising its features and strenghts, before reporting on our personal experience using Xtatic in a real-world application : a program for transforming the "XMLSpec" format (used for authoring W3C technical reports) into HTML. Our implementation closely follows an existing one written in XSLT, facilitating comparison of the two languages and analysis of the costs and benefits of rich static typing for XML-intensive code.

2 The Xtatic Language

This section, borrowed from [1], sketches the aspects of the Xtatic design. More details can be found in [2, 3].

Consider the following document fragment—a sequence of two entries from an address book—given here side-by side in XML and Xtatic concrete syntax.

```
<person>
  <name>Haruo Hosoya</name>
  <email>hahasoya</email>
</person>
<person>
  <name>Jerome Vouillon</name>
  <tel>123</tel>
</person>
```

```
[[ <person>
     <name>‘Haruo Hosoya‘</name>
     <email>‘hahasoya‘</email>
   </person>
   <person>
     <name>‘Jerome Vouillon‘</name>
     <tel>‘123‘</tel>
   </person> ]]
```

Xtatic's syntax for this document is very close to XML, the only differences being the outer double brackets, which segregate the world of XML values and types from the regular syntax of C♯, and backquotes, which distinguish PCDATA (XML textual data) from arbitrary Xtatic expressions yielding XML elements.

One possible type for the above value is a list of persons, each containing a name, an optional phone number, and a list of emails :

```
<person> <name>pcdata</> <tel>pcdata</>? <email>pcdata</>* </person>*
```

The type constructor "`?`" marks optional components, and "`*`" marks repeated sub-sequences. Xtatic also includes the type constructor "`|`" for non-disjoint unions of types. The shorthand `</>` is a closing bracket matching an arbitrarily named opening bracket. Every regular type in Xtatic denotes a set of sequences. Concatenation of sequences (and sequence types) is written either as simple juxtaposition or (for readability) with a comma. The constructors "`*`" and "`?`" bind stronger than "`,`", which is stronger than "`|`". The type "`pcdata`" describes sequences of characters.

Types can be given names that may be mentioned in other types. E.g., our address book could be given the type `APers*` in the presence of definitions

```
regtype Name  [[ <name>pcdata</> ]]
regtype Tel   [[ <tel>pcdata</>  ]]
regtype Email [[ <email>pcdata</> ]]
regtype TPers [[ <person> Name Tel </> ]]
regtype APers [[ <person> Name Tel? Email* </> ]]
```

A *regular pattern* is just a regular type decorated with variable binders. A value `v` can be matched against a pattern `p`, binding variables occurring in `p` to the corresponding parts of `v`, if `v` belongs to the language denoted by the regular type obtained from `p` by stripping variable binders. For matching against multiple patterns, XTATIC provides a `match` construct that is similar to the `switch` statement of C♯ and the `match` expression of functional languages such as ML. For example, the following program extracts a sequence of type `TPers` from a sequence of type `APers`, removing persons that do not have a phone number and eliding emails.

```
static [[ TPers* ]] addrbook ([[ APers* ]] ps)
 [[ TPers* ]] res = [[ ]];    bool cont = true;
 while (cont)
   match (ps)
    case [[<person> <name>any</> n, <tel>any</> t, any </>, any rest]] :
      res = [[ res, <person> n, t </> ]];   ps = rest;
    case [[ <person> any </person>, any rest ]] :     ps = rest;
    case [[ ]] :    cont = false;
 return res;
```

An XTATIC pattern is actually just a regular type decorated with variable binders; the run-time behavior of pattern matching is just regular tree language membership testing plus subtree (and sub-sequence) extraction.

The XTATIC type system ensures that each pattern match is complete (some branch will always succeed). The compiler can also infer types for variables bound by patterns. For instance, in the first `case` above, the variable `t` is given the type `<tel>pcdata</>`, even though its associated sub-pattern matches the more general type `<tel>any</>`, the type "`any`" being the least precise sequence type. Similarly, the rest of the sequence is matched against the type `any`, but may be stored into the variable `ps`, which has type `APers*`, as the type checker infers its type precisely.

Compared with the facilities available in pure C♯ (such as the raw DOM API), regular pattern matching allows much cleaner and more readable implementations of many tree investigation and transformation algorithms. However, compared with other native XML processing languages, XTATIC's pattern matching primitives are still fairly low-level : for example, no special syntax is provided for collecting *all* sub-trees matching a given pattern, or for iterating over sequences. We are currently investigating how best to add more powerful pattern matching; for now, our implementation efforts are concentrated on achieving good performance for low-level XML processing code.

The integration of XML sequences with C♯ objects is accomplished in two steps. First, XTATIC introduces a special class named `Seq` that is a supertype of every XML type—i.e., every XML value may be regarded as an object this class. The regular type `[[any]]` is equivalent to the class type `Seq`. This approach is justified by the homogeneous translation used by our compiler. Second, XTATIC allows any object—not just an XML tag—to be the label of an element. For instance, we can write `<(1)/>` for the singleton sequence labeled with the integer `1` (the parentheses distinguish an XTATIC expression from an XML tag); similarly, we can recursively define the type `any` as `any = [[ <(object)>any</>* ]]`.

We close this overview by describing how XTATIC views textual data. Formally, the type `pcdata` is defined by associating each character with a singleton class that is a subclass of the C♯ `char` class[1] and taking `pcdata` to be an abbreviation for `<(char)/>*`. In the concrete syntax, we write `'foo'` for the sequence type $\texttt{<(char}_f\texttt{)/><(char}_o\texttt{)/><(char}_o\texttt{)/>}$ and for the corresponding sequence value. This treatment of character data has two advantages. First, there is no need to introduce a special concatenation operator for `pcdata`, as the sequence `'ab','cd'` is identical to `'abcd'`. This can also be seen at the type level :

```
pcdata,pcdata = <(char)/>*,<(char)/>* = <(char)/>* = pcdata
```

[1]These singleton classes do not actually correspond to anything in the C♯ runtime, where the class `char` cannot have any subclasses. They are compiled away by the XTATIC compiler.

Equating `pcdata` with `string` would not allow such a seamless integration of the string concatenation operator with the sequence operator. Second, singleton character classes can be used in pattern matching to obtain functionality very similar to string regular expressions [8]. For instance, the XTATIC type `‘a‘,pcdata,‘b‘` corresponds to the regular expression `a.*b`.

The current XTATIC design adopts a very simple, untyped view of the attributes attached to XML elements : attributes may appear in tree values and in regular patterns, but they are not tracked by the type system. This is an interim solution : a language with XTATIC's goals should clearly treat attributes statically. At the moment, however, a fully satisfactory type system for attributes remains a matter of ongoing research (designs based on conventional record types, such as CDUCE's, do not easily support dynamic transformation of documents with unknown or partially known types ; Hosoya and Murata [4] have a proposal with most of the properties that we want, but of daunting complexity).

3 Typing an XSLT stylesheet

3.1 The XMLSPEC formatting problem

The application we've chosen to implement is the translation documents conforming to the the XMLSPEC DTD into XHTML, as defined by the W3C in an XSLT stylesheet. This DTD is the framework for the W3C's XML Recommendation 1.0 (February 1998) and several related classes of W3C XML-related specifications documents.

This application was chosen as a showcase of the generally-believed idea that statically typed programming and algebraic pattern matching are ideas that have practical benefits in XML programming, as a real-world case of a practical application. Its solution in a widely understood technology, XSLT, has already been explored in [5], and is larger than a toy example : the input DTD defines 102 elements and 57 type-like entities, while the output DTD defines 89 elements and 65 type-like entities.

The bulk of this report discusses a few observations that we found to be most interesting from the point of view of comparing relative strengths and weaknesses of XSLT and XTATIC for the XMLSPEC formatting task.

3.2 The XSLT processing model

An XSLT program, commonly called a stylesheet, is a collection of *templates*, each implementing the part of the transformation applicable to fragments of the input document matching an XPATH expression specified in the template.

Example:`<abstract>`. The abstract element is, in typical XTATIC type syntax :

```
regtype s_abstract [[<abstract>s_hdr_mix</>]]
```

Its template has the following code in XSLT :

```
<xsl :template match="abstract">
   <h2><a name="abstract">Abstract</a></h2>
   <xsl :apply-templates/>
</xsl :template>
```

Informally, a stylesheet is a "soup" of templates, where each template is a self-contained entity : it carries information under what conditions it can be applied, and tools for extracting information form its context, in addition to the content creation code that it executes. The execution environment traverses the document and selects, at each node, a template applicable for doing a transformation based on the templates' declared applicability conditions. The flow of control is usually implicit in an XSLT program : the run-time control flow is based on the traversal of the input in document order and automatic selection of applicable templates. For example, the above template contains operation `<xsl :apply-templates>` which, for each child of the current list element, looks for applicable templates, selects the most appropriate among them, executes it, and inserts the result into the current output. So, an XSLT program defines a document traversal in the form of structural recursion that has fixed shape.

A	::=		XTATIC *type*	T	::=		*regular type*
	|	C	class type		|	()	empty sequence
	|	[[T]]	regular type		|	<(A)> T </>	tree
					|	T,T	concatenation
d	::=		*regular type declaration*		|	T|T	alternative
	|	regtype X [[T]]			|	T*	repetition
					|	X	type name

FIG. 1 – Syntax of regular object types

The bulk of the XMLSPEC stylesheet (starting with the template for `<body>`) is made of "context-unaware" templates that perform simple element-to-element transformations, which basically amount to styling, i.e. interpreting logical document markup via suitable display markup. This is typical document-oriented styling at which XSLT is good. ("Styling" being understood as "keeping the structure intact, but format by small shifts in the output".)

We observe that, for quite a few elements in the XMLSPEC DTD, the sole effect of the stylesheet is to transform the input element's tag into a fixed HTML tag, e.g. `<slist>` into `<ul>`, and invoke recursive travesal. Sometimes there is a little bit more output inserted between the output tag and the result of the resursive traversal. Both cases are easy to re-implement in XTATIC and to augment with precise input and output types (which come in this case directly from the DTDs.). The re-implementation can be seen as fairly literal translation between the two languages.

We have therefore chosen to import the type definitions present in both the input and output DTDs and to proceed by reimplementing each template in XSLT with an XTATIC method, while explicitly writing the recursive traversal that is implicit in XSLT.

Example:for `abstract`. According to this organization, our template for the `abstract` element will look like this :

```
static [[h_block+]] TemplateAbstract ([[s_abstract]] markup)
  [[<abstract> s_hdr_mix* hdrmix</>]] = markup ;
  return [[<h2><a name='abstract'>'Abstract'</></>,
           DispatchContentBlockable(hdrmix)]] ;
```

3.3 Providing a Translation Scheme

Manipulating regular types

The XSLT stylesheet we are trying to emulate is organized in implicitly recursive templates that can be seen as forming a *mapping of XML elements*, where different XML elements from the source DTD can be translated to different XML elements from the destination DTD.

We have seen that we could import type definitions as a grammar with the syntax given in figure 1, where basic textual data is a terminal node, and all the elements of the DTD are non-terminal nodes.

What we have to provide in our Xtatic program, however, is a *typed* heterogeneous translation scheme from XMLSPEC to XHTML that respects the value mapping provided by the XSLT stylesheet.

Our tyed program can therefore be defined in a way similar to data bindings, as :

1. a mapping f from XMLSPEC types to XHTML types, and
2. an injective mapping g from XMLSPEC values (indexed by their types) to XHTML values, such that $g(t)_T \in f(\mathtt{T})$ for each XMLSPEC value `t` of type `T`.

That is, it is a way of embedding XMLSPEC types and values into the types and values of XHTML. Our XSLT stylesheet already provides the value mapping, and what is missing is the type mapping.

The template-based organisation, and its resulting production rules

Let's consider the grammar $G = (A, V, P)$ of the types that can be specified with the operations listed in the figure 1. Let's define the $\leq$ relation as :

$$(\mathtt{S} \leq \mathtt{T}) \Leftrightarrow \exists\ \alpha, \beta \in (A + V)^*, \mathtt{S} \rightarrow^* \alpha \mathtt{T} \beta$$

Since our syntax for specifying types is the same for both XMLSpec types and XHTML types, we will note this relation as being the same over both grammars (XMLSpec and XHTML).

We have noticed that the organisation of this particular XSLT stylesheet provides us with two general rules :

1. For each element `S` of the XMLSpec DTD, there is a corresponding template. In this `TemplateS`, we format the elements that compose `S` using their templates, and not through explicit formatting : templates only deal with recursive calls, and terminal nodes of the XHTML grammar. That means that for all t of type T, the type of $g(t)_T$ is unique and equal to $f(T)$.
2. We do simple formatting, which means that the operations on values we do in g are the same as those we od on types in our grammars.

In that case, we can see templates as production rules over our final grammar. Indeed let's consider for example an template defined over the input a. It will state :

$$g(a)_{\mathtt{A}} = h_a, g(b)_{\mathtt{B}}, g(c)_{\mathtt{C}}$$

From which we can infer the following rule :

$$f(\mathtt{A}) = h_a, f(\mathtt{B}), f(\mathtt{C})$$

Providing a type mapping therefore means :
– Considering the typing rules of our destination DTD as a grammar $G = (A, V, P)$.
– The images of the XMLSpec types, $f(T)$ for all T is in the set of the non-terminals nodes, but we add the equations provided by the templates to the production rules. We then obtain $G' = (A, V', P')$
– Typechecking the set of equations corresponding to G'

We can note that in particular, if $\mathtt{A} \leq \mathtt{B}$ and $g(x)_{\mathtt{B}}$ is mentioned in the template associated with `A`, then $f(\mathtt{A}) \leq f(\mathtt{B})$: our type mapping is compatible with the $\leq$ relation.

Coercion functions

Additionally, we need to specify a way of translating source-language type derivations into *coercion functions* .

Essentially, for every pair of source types `S` and `T` with $\mathtt{S} \leq \mathtt{T}$, there is a coercion function $\varphi(\mathtt{S} \leq \mathtt{T}) \in \varphi(\mathtt{S}) \longrightarrow f(\mathtt{T})$. Sometimes (when we already have $f(\mathtt{S}) \leq f(\mathtt{T})$ in the target type tree) these coercions are just identity functions ; but in other cases they perform real, run-time changes of data. The coercion functions are inserted into the program by the programmer wherever it sees that an (implicit or explicit) upcast is needed during typechecking.

The emulation of our XSLT stylesheet into an Xtatic program therefore consists of two things :

1. providing a type mapping from XMLSpec to XHTML.
2. providing, when necessary, the coercion functions that allow us to translate subtyping in the source DTD.

What we are going to try to show, in this application, is firstly that the use of this coercion functions (i.e. when the occurence where they are not the identity function) exactly match the occurences of typing bugs in the XSLT stylesheet's translation. Secondly, we have noticed how such a coercion function, by making real changes in the value mapping given by our representation, prevents us from exactly reproducing what our XSLT stylesheet does. Finally, we show how this representation problem can be solved using the flexibility of type definitions offered by the Xtatic language.

3.4 Type inference

As explained in 3.2, we have decided to mimick the organization of XSLT's processing by writing methods matching every XSLT template. Since C#'s typing is explicitly declarative, we have to give an output type to every template of the XSLT stylesheet.

Fortunately, the XSLT stylesheet provides us with a very large set of fixed constraints, allowing us to type the desired output of most templates without any ambiguity. However, its specifications are sometimes insufficient to be assertive about some templates, such as in the template corresponding to the spec element `abstract` :

```
<xsl :template match="abstract">
  <h2><a name="abstract">Abstract</a></h2>
  <xsl :apply-templates/>
</xsl :template>
```

This template simply adds an `<h2>` header at the beginning of its output, and one would be hard-pressed to find its exact return type without knowing all the possible formattings of an `abstract`'s contents.

The approach to solve this type inference problem usually fits the description of a type inference algorithm based on the unification of types, such as the bottom-up $\mathcal{W}$ algorithm proposed by Milner in [7], or the top–down $\mathcal{M}$ algorithm proposed by Lee and Yi in [6] Our work was an informal composition of both methods, but allowed us to comment on both :

- The top-down approach allowed us to start writing programs following a depth-first traversal of the call graph, without knowing the return types of all the calls made in a given method, by temporarily assigning them loose types. This property was extremely valuable, since the two very large DTDs we were dealing with are hard to keep in mind. However, the type incompatibilities signaled on a given method by the XTATIC compiler could then result in a recursive type modification of all the methods that depended on it, a very time-wasting process since our XSLT stylesheet contained quite a few typing bugs.
- The bottom-up approach, however, was safer and allowed us to deal more coherently with the error messages of XTATIC's typechecker. However, it needed nothing less than a mental idea of the topological sort of the call graph (see figure2), in order to know what methods to write first, a task rendered very hard by the numerous union types in the `spec.dtd`.

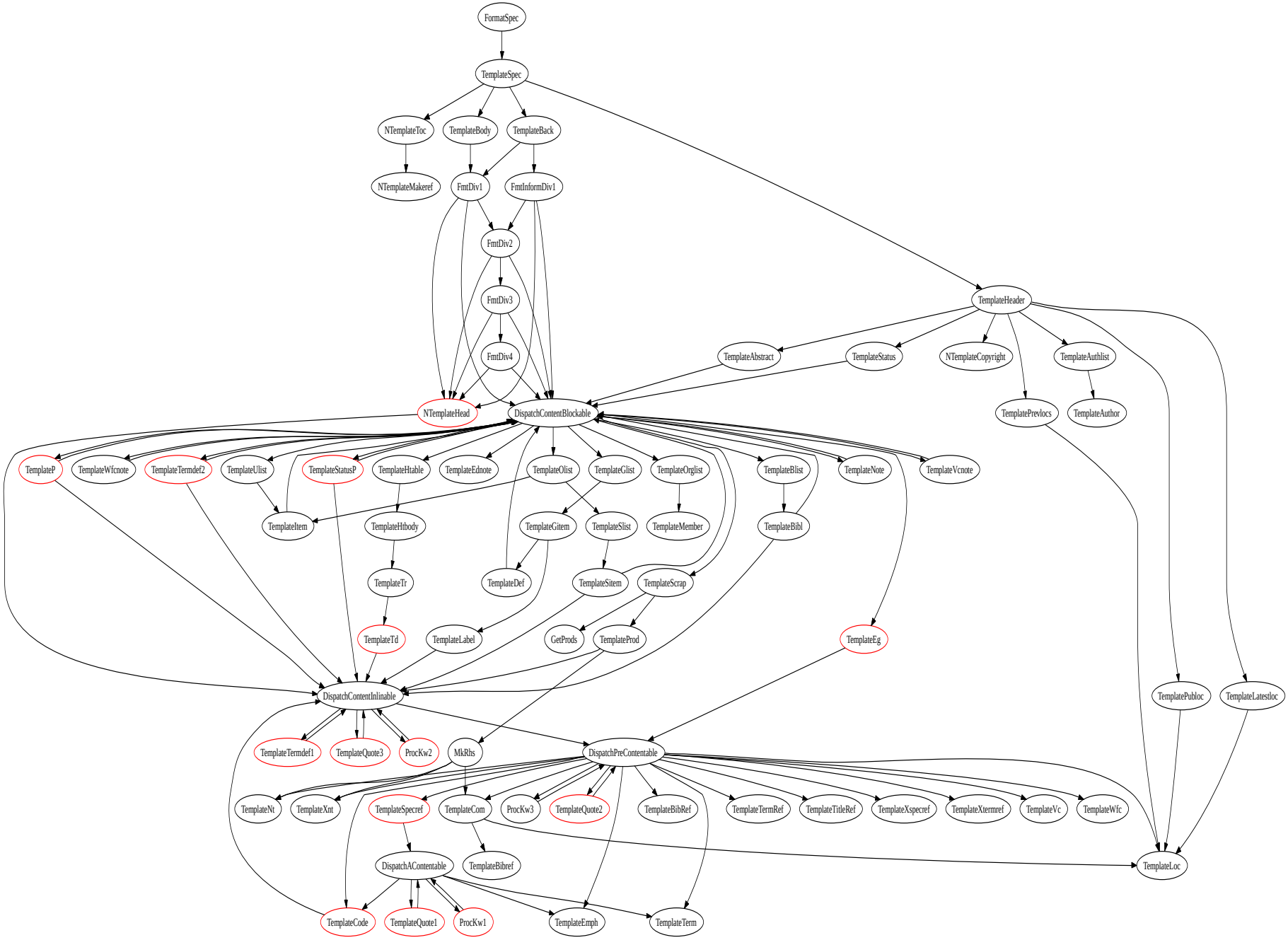

Fig. 2 – A preview of our application's final call graph

3.5 Typed manipulation of passthrough templates

While typing the methods corresponding to XSLT templates, we encountered a number of passthrough templates. One example is the `abstract` template mentioned in 3.4 another is the `quote` template :

```
<xsl :template match="quote">
  <xsl :text>"</xsl :text>
  <xsl :apply-templates/>
  <xsl :text>"</xsl :text>
</xsl :template>
```

This template simply embeds the contents of a quote element inside double quotes. its polymorphic type would therefore be something very close to :

```
[[<quote>'a</>]] → [[pcchar, 'b, pcchar]]
```

Where `'b` is the type of the result of the translatioon of `'a` elements into XHTML. We must thereofre assign *most general type* of all the outputs obtainable from `'a` in such a translation : it that case it is the `Inline` XHTML type. However, if this works well in most cases, it sometimes breaks typing in templates whose contents include the quote element.

For example, the template corresponding to the `eg` element has the following XSLT source code :

```
<xsl :template match="eg">
  <pre>
    <xsl :if test="role='error'">
      <xsl :attribute name="style">color : red</xsl :attribute>
    </xsl :if>
    <xsl :apply-templates/>
  </pre>
</xsl :template>
```

The contents of an `eg` element can, according to the `spec.dtd`, include a `quote` element, but an XHTML `pre` element can only contain a strict subset of `Inline` elements, a type called `pre_content` by the XHTML DTD.

While our template for `eg` could reject all `quote` elements in its contents in order to typecheck correctly, it seems hardly appropriate to reject `quote` elements that get formatted to perfectly appropriate `pre_content`-typed XHTML. We could think of trying to cast the result of the processing of `quote` elements to the `pre_content` type, but XTATIC doesn't support error handling, resulting in a runtime fatal error on unsafe casts. We could also use our match statement to recognize particular types, but it would fail in the `termdef` template, where the *output* needs to be of a different type based on the recognized input.

Our solution therefore consists of writing several methods to represent this single XSLT template, allowing us to manipulate not only `quote` elements, but `quote` elements *whose content is of a certain type*. The constraints imposed by generalization tell us that we could theoretically write a method for every `quote` content that simultaneously fits the two following conditions :

– it gets processed to a subtype of `Inline`.
– it gets processed to a type of which `pcchar` is a subtype.

However, we will only write separate methods for the distinct types required by the templates refering to `quote` elements. In our case, this only amounts three methods, one for the `quote` elements whose contents get processed to three distinct XHTML types : `a_content`, `pre_content`, and `Inline`. Our final program will then include :

```
TamplateQuote1 : [[<quote> a_contentable+ </>]] → [[a_content]]
TemplateQuote2 : [[<quote> pre_contentable+ </>]] → [[pre_content]]
TemplateQuote3 : [[<quote> Inlinable+ </>]] → [[Inline]]
```

Fortunately, this necessity of multiplying different methods with the same processing but with different signatures only arose in the translation of three templates to XTATIC : the templates for the `quote`, `kw` and `termdef` spec elements.

3.6 The typing bugs we encountered

We have found eleven occurences of templates that broke typing, returning invalid XHTML on some combination of their contents. While a complete rundown of their characteristics has been left to annexes, let us discuss a few of their most interesting aspects :

Most were easily dealt with by raising errors on offending content, such as on the various places an editorial note whas authorized by the spec DTD, but led to spurious HTML `blockquote` elements. This particular `ednote` element, whose ubiquity made perfect sense in a document in progress - an usage it was, according to the spec DTD, designed for - was less suitable in an XHTML conversion designed to operate on final documents.

The `code` template is an example of those templates who didn't plan for some nonsensical elements authorized by the spec DTD :

```
<xsl :template match="code">
  <code>
    <xsl :apply-templates/>
  </code>
</xsl :template>
```

The XHTML `code` element must contain `Inline` content only, so in our program, the method corresponding to this template was simply modified to raise an error on an `ednote` element as part of the contents of the `code` element.

Let us note, though, that the design described in 3.5 allowed us to make type distinctions based on the *possible outcomes* of the transformation of a given spec element, which is a many-to-one relation. This thus allowed us to be more subtle than what we would have done by simply basing ourselves on the spec DTD : the content sequences on which our methods raised errors were as minimal as possible.

However, on three instances, some type-breaking elements could not simply be discarded : namely, the `block` content that can occur in spec *paragaphs*, *status paragraphs* and *term definitions* was not only permitted by the spec DTD, but was also legitimized by actual data in the XML specification. Unfortunately, according to the XSLT stylesheet, the templates corresponding to those three elements were designed to assemble their output from HTML `Inline` elements.

Our solution was, in those cases, to understand the processing made by those templates as an accumulation of the succession of outputs created by the processing of their contents. This sequential approach, mimicking XSLT's behavior, allowed us to assemble the longest possible sequences of inlinable content in our elements into single HTML paragraph elements, which are of type `block`, while interrupting this gathering when we encountered genuine `block` elements returned by the formatting of our content.

One may notice the particular difficulty represented by the `termdef` element, since it is not only an instance of this particular type of bug where we must intertwine the output planned by the XSLT stylesheet with txhe one required by typechecking rules, but also a passthrough template, as described in 3.5. The behavior of the templates calling the processing of a termdef element change based on wether it outputs `Inline` content, or actual `block` content. The design we described above allowed us to make the distinction between those two behaviors : for the `termdef` element, we had the following signatures :

```
TemplateTermdef1 : [[<termdef> inlinable+ </>]] → [[a,Inline]]
TemplateTermdef2 : [[<termdef> (inlinable|blockable)* </>]] → [[block*]]
```

In conclusion, we have seen that every time we encountered a break in the subtyping rule, we had to implement a coercion function in a template, wether to reestablish subtyping by lowering the type of our resulting child element (and in our case, that meant equalling it to the empty element), or by extending the type of our resulting parent element (and in our case, that meant upgrading our existing elements to XHTML `blocks`, by specifying new formatting rules). However, we have discovered precise cases where those coercion funcitons did not scale well, especially when their application is nested in a call graph.

One of the major strenghts of XTATIC is therefore to make our input and output types malleable : the types we want to map in our application are not limited to what is imported from both the XMLSPEC and XHTML DTDs. We can in fact act and recognize types as subtle as our pattern matching can define. Manipulating algebraic expressions was a key element to our effortless success in fixing those typing bugs.

4 Other aspects of the Xtatic transformation

This application, as well as others developed during this internship, has proven XTATIC to be a robust language, with an extremely pleasant XML manipulation syntax. It regular pattern syntax is both intuitive and efficient, especially when treating repetitive sequences of heterogeneously typed contents, and the guarantees brought by typed XML manipulation conveniently eliminate the necessity of XML validation.

However, as a compiler, XTATIC still shows some surprising behavior in its optimisation and syntax, with its parser and typechecker choking on minor but common elements such as the '`for`' statement, or array-based C♯ syntax. Moreover, some extremely common object-oriented features such as exception handling, private methods, and properties still go unsupported, preventing the use of some typical patterns an object programmer would often like to use.

Numerous misfeatures, such as the loss of in-code comments chen going from XTATIC to C♯, or character encoding problems still persisting in spite of the author's efforts in the area, are technicalities that still impede a more widespread use of XTATIC as the natural object-oriented XML processing tool.

However, XTATIC's versatility is such that the author is personnaly convinced it has the potential for a considerably increased use once those issues are fixed.

5 Conclusions and future work

We have successfully reimplemented the conversion task implemented in the XSLT language in the W3C's XSLT sytlesheet, to an XTATIC program, with near-identical results over the W3C's XML recommendation. Our processing only differs by some particularities over special characters and HTML entities, a topic on which the Xtatic compiler is particulary -but hopefully, temporarily- fragile. Our reimplementation is about 2000 lines long, a decent result over XSLT's 750, considering that XTATIC is a much lower-level language. Finally, this real-world task has allowed us to explore numerous aspects of XTATIC typing manipulation of XML, particularly on the topic of creating a type mapping for two predefined type trees, based on a predefined untyped value mapping.

We have notably noticed that our explanation of the typing mechanism we applied is strongly tied with the devising of a heterogeneous translation scheme for our formatting application, relying on the application of two typically compiler-related algorithms : type inference and the construction of a coercion function. One would want to check wether the two properties we have noticed in the typing of this stylesheet hold across numerous XSLT applications.

The impression that the most difficult aspects of the writing of this application rely on compiler-related material is in fact so strong that it is the intuition of the author that his behavior during the making of this application should be automated for the XSLT language, by writing a partial or complete interpreter, that would help a programmer accomplish a similar task to what was tackled in this report. One could imagine such a program would suggest output types for unbound template in an XSLT stylesheet, and redefine the types of the input DTD so as to allow the user to write minimally intrusive coercion functions when type restrictions impose it.

6 Acknowledgements

I am grateful to Benjamin C. Pierce, Vladimir Gapeyev, and Michael Levin for their directions, advice and patience, and thank them for making this internship a wonderful en extremely enriching experience.

Références

[1] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native : Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, Apr. 2005.

[2] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[3] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[4] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003. Preliminary version in PLAN-X 2002.

[5] M. Kay. *XSLT programmer's reference*. Programmer to programmer. Second edition, 2001.

[6] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4) :707–723, July 1998.

[7] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, Aug. 1978.

[8] 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.