DynXML: Safely Programming the Dynamic Web

Joshua Sunshine Jonathan Aldrich
{joshua.sunshine,jonathan.aldrich}@cs.cmu.edu
Carnegie Mellon University

Abstract

A single web page in a complex web application has a huge number of possible runtime states. Functions, like JavaScript event handlers, that operate on such pages are extremely difficult to write correctly, because there are virtually no guaranteed constraints on the page. In this paper we propose DynXML, a new language for the web which safely and naturally mutates XML trees. Any dynamic web application written in DynXML is statically guaranteed to maintain the page in a subtype of a programmer-defined page type. Furthermore, event handlers are guaranteed to receive the page in the state they expect and leave it in the form expected by the next set of handlers—DynXML prevents page-manipulating programs from going wrong.

1. Introduction

The web today relies heavily on client-side code whose primary purpose is to modify a web page represented as an XML document. The major examples of this paradigm are JavaScript changing HTML pages and ActionScript mutating MXML-based Flash interfaces. None of these client side programming languages, nor, as far as we are aware any of the myriad tools, frameworks, and libraries built on top of the native languages (e.g. ASP.NET, Struts, Java Server Faces, Google Web Toolkit, Ruby on Rails) provide any guarantees about the state of the pages they modify. The negative results of this phenomenon are ubiquitous and sometimes severe — runtime errors, browser crashes, unresponsive applications, and lost data.

Think of a user composing an email in a webmail client. He composes an email and clicks send, which triggers the browser onclick event, which is forwarded to a JavaScript function. This function expects the page to contain a certain element, which is missing because of the particular path, chosen possibly from among thousands, he used to travel to the page state. The email isn't sent and the browser may even crash causing him to lose his data. Similar examples happen to every web user constantly and they cause much grief. One can easily see evidence of this problem by looking in the browser's error console after any long browsing session and observing the many hundreds of errors logged there.

Very little research has focused on solutions to this problem. The only work of which we are aware is context logic [6], a Hoare logic-based formal verification technique for reasoning about dynamic updates to tree-like XML documents. While context logic can verify arbitrary correctness properties of programs that manipulate tree structures, this generality also makes it difficult to provide full automation in a tool.

We are currently investigating whether a type-based approach could provide basic safety guarantees to the programmer, while potentially retaining the automatability of type systems. In order to do so, we build on the regular expression types of Hosoya and Pierce [19, 21]. However, this and other prior XML-related language research [1, 4, 8, 20] focused on purely functional query and transformation. While this paradigm is natural in many XML-processing applications, in the setting of dynamic web applications, a functional approach would introduce copying inefficiencies and require a drastically different programming style. Our work, therefore, focuses on reasoning about the imperative modification of XML trees.

Support for imperative update, in turn, underlies the central challenge of this paper: that the structure of an XML tree can change as it is modified. Therefore, the type system must track the state of an XML tree flow-sensitively, reason about the side effects of functions on trees, and control aliasing to ensure that elements of the page are not unexpectedly changed. Furthermore, dynamic web pages contain embedded event handlers that are invoked in response to end-user actions; we must ensure that when these handlers are invoked, the web page is in the expected state, and that the handlers leave the web page in a potentially changed but still well-formed state.

Our paper makes the following contributions to addressing these challenges:

- The design of DynXML, a language for writing Dynamic XML pages, is described with several realistic examples (Section 2). We provide a formal syntax, operational semantics, and typing rules for DynXML(Section 3).
- A permission-based type system is adapted from the setting of object calculi [2] to provide flow-sensitive reasoning and alias control in the setting of the lambda calculus (Section 3.2), ensuring that page-manipulating code does not go wrong.
- An approach to embedding event handler functions within an XML page is developed, where the type of the function expresses its effect on the page. The page is typechecked to ensure that it will be in the correct state whenever the handler is invoked, and that the page the handler produces is itself a valid page that meets the user's specification (Section 3.5).
- An approach for scoping the modifications a function can perform on an XML tree is described, allowing clients to ensure that parts of the tree they depend on are not modified (Section 3.6).

2. Language

In this section we describe the features of DynXML. We will use the book store page displayed in Figure 1, as a running example.

APLWACA 2010 June 6, 2010, Toronto, Canada.
Copyright © 2010 ACM 978-1-60558-913-8/10/06. . . $10.00

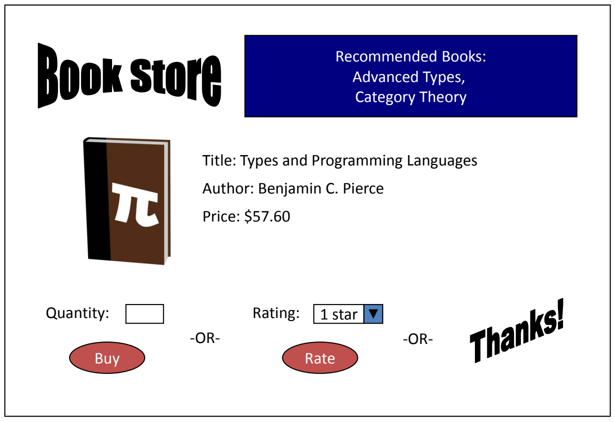
Book Store
Recommended Books: Advanced Types, Category Theory
Title: Types and Programming Languages
Author: Benjamin C. Pierce
Price: $57.60
Quantity:
Buy
-OR-
Rating:
1 star
Rate
-OR-
Thanks!

Figure 1. Schematic of book store page.

```
type α page =
  div[mutable(string)], div[string], α;
type thanks = div[string];
type rating =
  div[dropdown[option[int]*], //rating selector
  button[(rating page)→(thanks page)]];
type quantity =
  div[textbox[], //quantity textbox
  button[(quantity page)→(rating page)]];
type full =
  mutable(thanks | rating | quantity); [1]
```

Listing 1. Code describing type of book store item page.

This page is divided into three parts. The top section lists related books that the store recommends to the user. The middle contains the main content like the book title, description and price. Finally, the bottom section has three alternate states: 1) the purchasing state, which contains a textbox allowing the user to enter the quantity of books to buy and a button to process the purchase; 2) the rating state, which allows a user who has already purchased a book to select a rating from a dropdown list and to push a button to finalize the rating; and 3) the thanks state, which thanks a user who has already purchased and rated the book.

2.1 XML Types

This page can be represented as a type such as a DTD [5] or XML Schema [13]. Our XML description syntax borrows heavily from the regular expression types of Hosoya and Pierce [21]. The type constructor n[. . .] classifies XML nodes with the label n (i.e. <n>. . .</n>). The inhabitants of the type div[string] are div elements containing a string (e.g. <div>Hello World!</div>). Types can also include regular expression operators like * (repetition); ? (option); , (sequence); and | (union). Therefore, the type table[tr[td[$(int|string)$, textarea[]?]]$*$] is the type of all one column tables whose cells each include an integer or a string followed by an optional textarea.

The type of the book store item page in DynXML is in Listing 1. The code in this listing assumes a condensed version of XHTML without XML attributes. For example, a textbox element in html is normally written <input type="text"/>, but instead we write <textbox/>, which corresponds to the type textbox[]. For simplicity's sake we have chosen to follow Hosoya and Pierce [21] and encode attributes as subelements.[1]

The type constructor[2] page takes a type argument α and describes a page that contains a div element for the book recommendation at the top of the page, a div element for the item description in the middle of the page, and a variable element described by the type argument for the bottom section of the page. The types quantity, rating, and thanks are the types for the last element of the page when it is in the purchasing state, the rating state, and the thanks state respectively; we can combine these with the overall page type to describe the full page in the purchasing state as quantity page. Type full represents the type of the bottom section of the page in all three possible dynamic states. It will contain a textbox and button in the purchasing state, a dropdown and button in the rating state, and simply a string in the thanks state.

DynXML supports recursive types, which you can see in action in several places in Listing 1. For example, the button in quantity contains an arrow type which refers to quantity page. We also use recursive types to encode several regular expression types as derived forms.

The arrow types embedded in the buttons in Listing 1 illustrate one of DynXML's most interesting features. They are manifestations of the event handlers introduced in Section 1. We treat event handlers as functions which are passed the page as an argument and return it as the result.

The right and left halves of the arrow type are not fixed types, but instead flow-sensitive *permissions*. We use permissions to track the precise type of XML trees as they are modified. [3] We call this precise type the *current state*; it is the system's best estimate of the most specific current type for the expression referenced by the permission, and is expressed using the regular expression types introduced earlier in this section.

Permissions are linear resources that cannot be duplicated, so they can be used to ensure that an XML tree is not modified through aliased pointers. If a permission to a tree is not passed to a function, we can be sure that function will not modify that tree.

DynXML's type system is richer and more complex than what is illustrated here. Permissions in the full system contain not only the current state, but also the *maximum state*—a programmer specified constraining type. The maximum state is also written using regular expression types. DynXML guarantees that the expression referenced by the permission will *always* conform to the maximum state. The maximum state serves a software engineering purpose. It acts as a global invariant on the page which limits the data a page can contain and thereby shrinks the universe of pages to programmer-understandable size.

By contrast, the current state of an XML document will change from one program points to another. For example, the XML document referred to by the permission when a function is called is guaranteed to conform to the current state given there, but the state of the document may change as the function executes. The maximum state throughout the example in Listing 1 is full page.

In the calculus, a permission is accompanied by *standard types*, which correspond to the types in the lambda calculus. Unlike per-

[1] Many real examples require a more sophisticated approach to attribute types [18, 22]. These examples are important, but the problems they present are orthogonal to those that we're dealing with in this paper.

[2] The focus of this paper is not on parameterized XML types, but as this example shows type parameters enable programmers to reuse the top-level type of an XML tree when one of several subtree types can be plugged into one of its parts

[3] It may be worth exploring purely functional alternatives such as monads in the future, but it was not obvious to us how a monad-based solution could provide the flexibility and reasoning of our permissions system.

```
fn buy = lambda page:
   XML,(full page@quantity page) =>
   //persist purchase
   recommend(page);
   page.3 := <div><dropdown> [2]
               <option>1</option>
               ...
            </dropdown>
            <button>rate</button></div>;
   page; // return the page

fn rate = lambda page:
   XML,(full page@rating page) =>
   //persist rating
   recommend(page);
   var options = page.3*.1*; [3] [4]
   var grade;
   for(option:options)
      if (option selected)
         grade = option*;
   var thankstring =
      ‘‘Thanks for giving the book ’’
         .grade.‘‘ stars!’’;
   page.3 := <div>thankstring</div>;
   page; //return the page
```

Listing 2. Code for button click event handlers.

```
type rec = thanks | rating | quantity; [5]

fun recommend = lambda page:
   XML,(rec page@rec page) =>
   match ((page.3)*).1 with
      textbox[] in //purchasing state
         ((page.1)*).1 := ‘‘Harry Potter’’;
      dropdown[option[int]*] in //rating state
         ((page.1)*).1 := ‘‘Advanced Types’’;
       string in //thanks state
        if(ratedwell)
           ((page.1)*).1 := ‘‘Advanced Types’’;
        else
           ((page.1)*).1 := ‘‘Harry Potter’’;
```

Listing 3. Recommendation engine code.

missions, standard types are not flow sensitive. The operations on types like integers, strings, or functions are static. In the case of XML documents, the standard type is always XML. The standard type prevents us from mixing up XML documents and functions, but ensuring that an XML page is valid or has a particular structure is done using permissions, not types.

Permissions, like types in most programming languages, are stated explicitly in code only at function boundaries. In other words, the programmer is only required to write the permissions of parameters to and values returned by functions. Within a function the system maintains a list of permissions in the *permission context*, each of which is automatically updated as changes are made to the XML documents referred to by the permissions. Therefore, the system is modular— an implementation of DynXML should be able to check each method independently.

In general, the maximum state for any XML document will always remain the same, while the current state may change if an assignment is made to an element of the document. The current state allows programmers to safely use or manipulate the XML document more conveniently then they could if we only maintained the maximum state. For example, imagine we have an XML document containing only an integer, maximum state (int | string), and current state int. If we only maintained the maximum state, the programmer, who knows the value of the document is an integer, would be forced to pattern match against the document, throw an exception if a string is found and only then use the integer. Instead, in our system he can simply use the integer like any other (e.g. by adding it to another one).

2.2 Functions and Event Handlers

The bodies of the buy and rate event handlers are shown in Listing 2. The parameters to both functions are annotated with a fully general type. The parameter to the buy function has type XML,(full page@quantity page), which specifies that page's standard type is XML, its maximum state is full page, and its current state is quantity page. The code at $\boxed{2}$ uses the .n operator which projects out the nth element of a sequence. At $\boxed{3}$ we use a similar operator, *, which projects out the internals of a node. For example, let's say the variable x has current state $a[b, c]$. The current state of x* would be b, c. Both functions call the recommend function which we will discuss at length later. The buy function simply transitions the page to the ratings state. The ratings function determines what rating the user gave the book and thanks the user for his rating — transitioning the page to the thanks state.

The assignment operator := assigns the expression on the right of the operator to the reference cell on the left. Every node in an XML tree in DynXML is conceptually a reference cell. However, the maximum state of the tree must enclose this cell in the mutable keyword for the assignment to be valid. For example, in Listing 1, the mutable keyword at $\boxed{1}$ encloses all three states of the bottom section which allows the event handlers to change this section of the page.

2.3 Maintaining Precise State

The code in Listing 3 adds the recommendation engine facility we mentioned earlier. The type rec page is the current and maximum state of the page parameter passed to the function responsible for generating new recommendations. The body of the function pattern matches against the third node of the first div to determine the state of the page. If this element is a textbox, than the page is in the purchasing state—since the user has not yet purchased the book, the program recommends a general interest book, *Harry Potter*. After the user purchases *Types and Programming Languages*, the page is in the rating state with a dropdown, so the application recommends a related book, *Advanced Types*. Finally, in the thanks state the application recommends related books if the user gave the book a good rating and recommends general interest books otherwise.

Notice that the current state of the recommend function's parameter is rec page, while the parameter of the rate event handler has state rating page. Our subtyping facility, which we describe in detail in Section 3.3, allows the page to be passed from the rate function to the recommend function. For now, think of subtyping in terms of set inclusion. For example, rate page is a subtype of rec page. Therefore a page in the rating state can safely be passed to both the rate event handler and the recommend function.

The type of the recommend function's parameter is tailored to minimize information loss. The mutable keyword that appears at $\boxed{1}$ of Listing 1 does not appear at $\boxed{5}$ of Listing 3. This means that when clients call the recommend function, they can be confident that recommend will not change any part of the third div of the page. This means that even though recommend returns a permission to the page that does not specify if the page's third div is in the thanks, rating, or quantity states, if the client knows that this part of the page was in the rating state before the call, it can conclude that the div is still in the rating state. Essentially, the mutable keyword acts to scope the possible effects that the recommend function may have on the page.

For example, look back at the rate function in Listing 2. The page enters the rate function with current state rating page. Then, the page is passed to the recommend function which mutates the page and returns it. The current state returned by this function is rec page. The system must then merge the current state rec page from recommend with the current state, rating page it held before the event was fired. We want the system to maintain the current state rating page for the page, since this is still its most specific type. It does so by merging only the sections marked as mutable in the returning permission. In this case, the only section that is mutable is the string in the first div. This same string appears in rating page so that the current state is maintained. This allows the programmer to safely write the code at $\boxed{4}$ which accesses the user ratings dropdown list. If the current state were rec page, the programmer would first have to match against all three components of the union type to extract the rating she knows will *always* be there.

The function recommend is written with type rec page so that it can be more generic and can be called with the page in any of 3 states. Permission merging allows the caller to reason about what is changed. This effects a kind of polymorphism, but is more lightweight than adding a polymorphic type for every non-mutable sub-part of the page. In particular, a polymorphic solution does not scale well to pages with many different mutable sections as a type parameter would be required for each one.

3. Formal System

This section formalizes DynXML and it differs in a few ways from the examples given in Section 2. The examples make use several of additions and simplifications to ease understanding [4]. Our type system does not include type abbreviations, looping constructs, or base types (e.g. string, booleans), all of which are well understood and would only add needless complexity. Our type system requires all code to be in let-normal form, but the examples are instead written using the more familiar sequence (;) notation.

3.1 Syntax

Figure 2 shows the syntax of DynXML. We distinguish between pure terms t and possibly effectful expressions e. Arguments to functions and even inputs to built-in operators are restricted to terms. Instead, expressions are bound to variables using the let syntactic form. The entire program is therefore in let-normal form. This serves two purposes — it makes execution order explicit and allows permissions to refer to store locations by name.

Most of the expression forms are standard or were introduced in Section 2. Function application is written t(t), and XML pairs are introduced by writing t t. The expression page t instructs DynXML to check all events embedded in the XML document t, a topic that will be discussed at length in Section 3.5. The only terms are variables, function abstractions, memory locations (ℓ), and unit.

The function abstraction form listed here adds one wrinkle to what was discussed in Section 2. The Δ in the abstraction

$$
\begin{array}{rcll}
\text{(Expressions)} & e & ::= & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
 & & & \mathsf{page}\ t \mid \mathsf{<n>}t\mathsf{</n>} \mid \\
 & & & tt \mid t.1 \mid t.2 \mid \\
 & & & t{*} \mid t := t \mid t(t) \mid t \\
\text{(Values)} & v & ::= & \lambda x{:}\tau.\Delta{\Rightarrow}e \mid () \mid \ell \\
\text{(Terms)} & t & ::= & v \mid x \\
\text{(Static Types)} & T & ::= & T, p{>>}p.\Delta{\rightarrow}\tau.\Delta \mid \\
 & & & \mathsf{Unit} \mid \mathsf{XML} \mid X \mid \\
 & & & \mathsf{rec}X{\Rightarrow}T \\
\text{(XML Types)} & M & ::= & M|M \mid M,M \mid \mathsf{n}[M] \mid \\
 & & & \mathsf{mutable}(M) \mid T \\
\text{(Types w/Holes)} & \mathcal{M} & ::= & \{-\} \mid \mathcal{M}|M \mid M|\mathcal{M} \mid \\
 & & & \mathcal{M},M \mid M,\mathcal{M} \mid \mathsf{n}[\mathcal{M}] \mid \\
 & & & \mathsf{mutable}(\mathcal{M}) \mid M \\
\text{(Permissions)} & p & ::= & M@M \mid \multimap p \\
\text{(Types)} & \tau & ::= & (T,p) \mid \varnothing \\
\text{(Perm. Refs)} & r & ::= & x \mid r{*} \mid r.1 \mid r.2 \mid \ell \\
\text{(Stores Values)} & s & ::= & vv \mid \mathsf{<n>}v\mathsf{</n>} \mid v \\
\text{(Stores)} & \mu & ::= & \mu, \ell{=}s \mid \varnothing \\
\text{(Contexts)} & \Gamma & ::= & \Gamma, x{:}T \mid \varnothing \\
\text{(Perm. Contexts)} & \Delta & ::= & \Delta, r{:}p \mid \varnothing \\
\text{(Store Typings)} & \Psi & ::= & \Psi, \ell{:}T \mid \varnothing
\end{array}
$$

Figure 2. Syntax

form ($\lambda x{:}\tau.\mathbf{\Delta}{\Rightarrow}e$) allows the programmer to specify permissions to non-parameters in the function declaration. This is important to verify functions with free variables, which is necessary to support common idioms like currying and partial evaluation. Similarly, the arrow type contains a delta on both the right and the left of the arrow which contain input and output permissions for free variables. Finally, the $p{>>}p$ in the arrow type specifies the pre and post state of the function's parameter. For example, the full type of the rate function in Listing 2 is XML,full page@rating page>>full page@thanks page.$\varnothing$ $\rightarrow$ Unit,Unit@Unit.$\varnothing$. Notice that the permission to the page is *not* returned as the result of the function. Instead, the result of the function is unit. Nonetheless, as we will explain in Section 3.4, the permission to page winds up in the permission context produced by function application just as it would if it were the function's result.

We also distinguish between the regular expression types introduced in Section 2.1 and called XML Types here, and standard types. The current state and maximum state in a permission are XML types, while regular types are used in the regular way. The standard types include one type, XML, which is the type of any XML document. Permissions primarily refer to expressions of this type.

Permissions refer to locations, variables, and subcomponents of XML trees with r.1, r.2, or r*. These subtree references are used when the system infers permissions to subtrees from permissions to the main tree. We will delve into the details of permissions in Section 3.2.

All of the judgements used in DynXML are listed alongside their defining rules in the figures spread throughout the paper. There are two typing judgments: $\Gamma;\Psi;\Delta \vdash_e e : \tau \dashv \Delta$ for expressions and $\Gamma;\Psi \vdash_t t : T$ for terms. The typing rules used to interpret these judgments will be introduced in Section 3.4. In our typing rules we sometimes need to extract new permission from existing ones and we check that this is safe with $\Gamma;\Psi;\Delta \vdash_\Delta \Delta$ (Section 3.2). Our typing rules also convert XML types to standard types

[4] Two type constructors introduced in Section 2 can be encoded as a combination of other constructors. These derived forms are not present in the syntax of our formal system. The option type, T?, can be rewritten as a union of the type T and the unit type, $T|\mathsf{Unit}$. The list type, T*, can be encoded as using a recursive type, pair type, and unit type: $\mathsf{rec}X{\Rightarrow}T, X|\mathsf{Unit}$

$$\boxed{M \rightsquigarrow T}$$

$$\frac{M = T}{M \rightsquigarrow T}\,\text{C-T} \qquad \frac{M = M_1|M_2}{M \rightsquigarrow \mathsf{XML}}\,\text{C-Union}$$

$$\frac{M = M_1, M_2}{M \rightsquigarrow \mathsf{XML}}\,\text{C-Concat} \qquad \frac{M = \mathsf{n}[M']}{M \rightsquigarrow \mathsf{XML}}\,\text{C-Node}$$

$$\frac{M = \mathsf{mutable}(M')}{M \rightsquigarrow \mathsf{XML}}\,\text{C-Ref}$$

Figure 3. Conversion of expression types and term types.

using $M \rightsquigarrow T$ (Section 3.2). We define DynXML's semantics with a small step operational semantics using, $e\,|\,\mu \mapsto e\,|\,\mu$ (Section 3.7). The types with holes syntactic category, the page t expression form, and the $M;\mathcal{M} \vdash_{\mathcal{M}} M$ judgement all relate to the checking of events embedded in XML documents, and will be discussed in Section 3.5.

3.2 Preliminaries

Figure 3 shows the conversion rules used in the typing rules to convert XML types to regular types. The rules, C-Union, C-Concat, C-Node, and C-Ref simply convert tree structure XML types to the type XML. The rule C-T, converts XML types that are *exact* standard types to the standard type. This rule applies when functions or Unit appear inside an XML tree. It would also apply to the integers, strings, and booleans in our examples in Section 2. In Listing 4, the int type in the current state of x.2 is converted to by C-T to a standard int and added to one.

Many of the typing rules discussed in Section 3.4 rely on the permission context judgement, $\Gamma;\Psi;\Delta \vdash_\Delta \Delta'$. This judgement is defined in Figure 4. The judgment asks what permission context can we derive from an existing permission context. The simplest rule, P-In, says that a permission context containing a single permission p, derives from any permission context containing p. P-Pair derives a pair of permissions p_1 and p_2 from a context that implies both p_1 and p_2. Note that the context that implies p_1 must be distinct from the context that implies p_2.

DynXML often needs to derive a permission for an XML subtree from a permission to the main tree. The three rules Sp-Inner, Sp-Proj1, and Sp-Proj2 provide this facility. They do so by splitting the original permission into a permission to the subtree and a contract to return the original permission if given the subtree permission. This contract is written using the linear implication operator $\multimap$ which should convey the right intuition to readers familiar with linear logic.

When a permission to the main tree is needed again it is recovered with one of the combination rules, namely C-Inner, C-Proj1 or C-Proj2. These rules take the two permissions that were created by the corresponding split rule and combine them into a modified version of the original permission. The current state in the permission to the subtree replaces the corresponding component of the current state of the main tree. This modification is necessary to update the main tree's permission to make any modifications (e.g. assignments) made to the subtree.

The rule Sp-Proj2 is employed at $\boxed{6}$ of Listing 4. In that example we derive two permissions: 1) A permission to x.2 with current state int and maximum state b | int. 2) A contract which allows us to recover the permission to x given in the specification of f from the first permission. Consider adding a new function call bar(x) to the end of the body of f: C-Proj1 will recover the original permission to x to enable the call.

$$\boxed{\Gamma;\Psi;\Delta \vdash_\Delta \Delta}$$

$$\frac{p \in \Delta}{\Gamma;\Psi;\Delta \vdash_\Delta p}\,\text{P-In}$$

$$\frac{\Gamma;\Psi;\Delta_1 \vdash_\Delta p_1 \qquad \Gamma;\Psi;\Delta_2 \vdash_\Delta p_2}{\Gamma;\Psi;\Delta_1, \Delta_2 \vdash_\Delta p_1, p_2}\,\text{P-Pair}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x{:}\mathsf{n}[M_1]@\mathsf{n}[M_2]}{\Gamma;\Psi;\Delta \vdash_\Delta x * {:}M_1@M_2, x{*}{:} \multimap \mathsf{n}[M_1]@\mathsf{n}[M_2]}\,\text{Sp-Inner}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x{:}M_1, M_1'@M_2, M_2'}{\Gamma;\Psi;\Delta \vdash_\Delta x.1{:}M_1@M_2, x.1{:} \multimap M_1, M_1'@M_2, M_2'}\,\text{Sp-Proj1}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x{:}M_1, M_1'@M_2, M_2'}{\Gamma;\Psi;\Delta \vdash_\Delta x.2{:}M_1'@M_2', x.2{:} \multimap M_1, M_1'@M_2, M_2'}\,\text{Sp-Proj2}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x{*}{:}M_1@M', x{*}{:} \multimap \mathsf{n}[M_1]@\mathsf{n}[M]}{\Gamma;\Psi;\Delta \vdash_\Delta x{:}\mathsf{n}[M_1]@\mathsf{n}[M']}\,\text{C-Inner}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x.1{:}M_1@M_1'', x.1{:} \multimap M_1, M_2@M_1', M_2'}{\Gamma;\Psi;\Delta \vdash_\Delta x{:}M_1, M_2@M_1'', M_2'}\,\text{C-Proj1}$$

$$\frac{\Gamma;\Psi;\Delta \vdash_\Delta x.2{:}M_2@M_2'', x.2{:} \multimap M_1, M_2@M_1', M_2'}{\Gamma;\Psi;\Delta \vdash_\Delta x{:}M_1, M_2@M_1', M_2''}\,\text{C-Proj2}$$

Figure 4. Permissions

3.3 Subtyping

Figure 5 shows our subtyping rules. S-Union3 says that union type is symmetrical, and S-Union2 says that each half of a union type is a subtype of the union type. S-Union1 allows union A as a subtype of union B if each of A's subcomponents is a subtype of B's subcomponents. S-Mut says that an XML type surrounded by mutable XML type is a subtype of the same XML type without the mutable keyword. The type system relies on the uniqueness of the permission and its current state to determine the type of a subtree. It does not need to rely on whether a reference is marked mutable to know if it has changed. Therefore, this subtyping rule is safe because any mutable element can also be used as an immutable element. S-Concat specifies that pair A is a subtype of pair B if A's first component is a subtype of B's first component and A's second component is a subtype of B's second component. XML node subtyping is handled in the obvious way by S-XML. The transitivity and reflexivity of subtyping is ensured with rules S-Trans and S-Refl.

One point worth mentioning here is that our subtyping can be used to elegantly guarantee that any page in DynXML is valid HTML. A library writer could encode the type of all valid HTML pages [10] pages as a DynXML XML type.[5] Then, a programmer could specify that all of her page types are subtypes of the valid HTML type. Then, any page generated by DynXML will always be valid HTML.

[5] Our XML types cannot capture the full richness of the XHTML specification. For example, we cannot specify that the value of id attributes must be unique throughout a document. However, an important subset of the specification can be encoded, and more importantly, *enforced* by DynXML.

$\boxed{\Gamma;\Psi\vdash_t t : T}$ $\boxed{\Gamma;\Psi;\Delta\vdash_e e : \tau \dashv \Delta}$

$$\frac{x : T \in \Gamma}{\Gamma;\Psi\vdash_t x : T}\textsc{T-Var} \qquad \frac{}{\Gamma;\Psi\vdash_t () : \mathsf{Unit}}\textsc{T-Unit} \qquad \frac{\Psi(\ell) = T}{\Gamma;\Psi\vdash_t \ell : T}\textsc{T-Loc}$$

$$\frac{\Gamma,x{:}T;\Psi;\Delta_1, x{:}p \vdash_e e : \tau_2 \dashv \Delta_2 \qquad \Delta_2[x] = p'}{\Gamma;\Psi\vdash_t \lambda x{:}T,p.\Delta_1 \Rightarrow e : T, p{>>}p'.\Delta_1 \rightarrow \tau_2.(\Delta_2 \setminus x)}\textsc{T-Abs}$$

$$\frac{\begin{array}{c}\Gamma;\Psi;\Delta_1\vdash_e e_1 : (T_1,p_1) \dashv \Delta_1' \\ \Gamma, x : T_1;\Psi;(\Delta_2,\Delta_1', x : p_1)\vdash_e e_2 : T_2 \dashv \Delta_3\end{array}}{\Gamma;\Psi;(\Delta_1,\Delta_2)\vdash_e \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv [e_1/x]\Delta_3}\textsc{T-Let} \qquad \frac{\Gamma;\Psi\vdash_t t : \mathsf{XML} \qquad \Gamma;\Psi;\Delta\vdash_\Delta t{:}p}{\Gamma;\Psi;\Delta\vdash_e t : \mathsf{XML}, p \dashv \varnothing}\textsc{T-XmlTerm}$$

$$\frac{\begin{array}{c}\Gamma;\Psi;\Delta_1\vdash_e t_1 : T, \mathsf{mutable}(M)@M' \dashv \varnothing \\ \Gamma;\Psi;\Delta_2\vdash_e t_2 : T, M_1@M_1' \dashv \varnothing \qquad M_1' <: M\end{array}}{\Gamma;\Psi;(\Delta_1,\Delta_2)\vdash_e t_1 \coloneqq t_2 : \mathsf{Unit}, (\mathsf{Unit}@\mathsf{Unit}) \dashv t_1{:}\mathsf{mutable}(M)@M_1'}\textsc{T-Asgn} \qquad \frac{\Gamma;\Psi\vdash_t t : T \qquad T \neq \mathsf{XML}}{\Gamma;\Psi;\Delta\vdash_e t : T, T@T \dashv \varnothing}\textsc{T-Term}$$

$$\frac{\Gamma;\Psi;\Delta_1\vdash_\Delta t_1{:}M_1@M_1' \qquad \Gamma;\Psi;\Delta_2\vdash_\Delta t_2{:}M_2@M_2'}{\Gamma;\Psi;\Delta_1,\Delta_2\vdash_e t_1 t_2 : \mathsf{XML}, M_1,M_2@M_1',M_2' \dashv \varnothing}\textsc{T-Pair} \qquad \frac{\begin{array}{c}\Gamma;\Psi\vdash_t t : \mathsf{XML} \\ M_2;\{-\}\vdash_{\mathcal{M}} M_2 \qquad \Gamma;\Psi;\Delta\vdash_\Delta t{:}M_1@M_2\end{array}}{\Gamma;\Psi;\Delta\vdash_e \mathsf{page}\ t : \mathsf{XML}, M_2@M_2 \dashv \varnothing}\textsc{T-Page}$$

$$\frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_\Delta t.1{:}M_1@M_1' \\ M_1 \rightsquigarrow T\end{array}}{\Gamma;\Psi;\Delta\vdash_e t.1 : T, M_1@M_1' \dashv \varnothing}\textsc{T-Proj1} \qquad \frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_\Delta t.2{:}M_2@M_2' \\ M_2 \rightsquigarrow T\end{array}}{\Gamma;\Psi;\Delta\vdash_e t.2 : T, M_2@M_2' \dashv \varnothing}\textsc{T-Proj2} \qquad \frac{\begin{array}{c}\Gamma;\Psi;\Delta\vdash_\Delta t{*}{:}M@M' \\ M \rightsquigarrow T\end{array}}{\Gamma;\Psi;\Delta\vdash_e t{*} : T, M@M' \dashv \varnothing}\textsc{T-Inner}$$

$$\frac{M_1 <: M_3 \qquad \Gamma;\Psi;\Delta\vdash_\Delta t{:}M_2@M_1}{\Gamma;\Psi;\Delta\vdash_e \texttt{<n>}t\texttt{</n>} : \mathsf{XML}, \mathsf{n}[M_3]@\mathsf{n}[M_1] \dashv \varnothing}\textsc{T-Xml} \qquad \frac{\begin{array}{c}\Gamma;\Psi\vdash_t t_2 : T \qquad \Gamma;\Psi;\Delta\vdash_\Delta \Delta_1 \qquad \Gamma;\Psi;\Delta\vdash_\Delta t_2{:}M_1@M_2' \\ \Gamma;\Psi\vdash_t t_1 : T, M_1@M_2{>>}p.\Delta_1\rightarrow\tau_2.\Delta_2 \qquad M_2' <: M_2\end{array}}{\Gamma;\Psi;\Delta\vdash_e t_1(t_2) : \tau_2 \dashv \Delta_2, t_2{:}p}\textsc{T-App}$$

Figure 6. Typing

$\boxed{M <: M}$

$$\frac{M_1 <: M_1' \qquad M_2 <: M_2'}{M_1|M_2 <: M_1'|M_2'}\textsc{S-Union1}$$

$$\frac{}{M_1 <: M_1|M_2}\textsc{S-Union2} \qquad \frac{}{M_1|M_2 <: M_2|M_1}\textsc{S-Union3}$$

$$\frac{M_1 <: M_1' \qquad M_2 <: M_2'}{M_1, M_2 <: M_1', M_2'}\textsc{S-Concat}$$

$$\frac{}{\mathsf{mutable}(M) <: M}\textsc{S-Mut} \qquad \frac{M_1 <: M_1'}{\mathsf{n}[M_1] <: \mathsf{n}[M_1']}\textsc{S-XML}$$

$$\frac{M_1 <: M_2 \qquad M_2 <: M_3}{M_1 <: M_3}\textsc{S-Trans} \qquad \frac{}{M <: M}\textsc{S-Refl}$$

Figure 5. Subtyping

```
fun f = lambda x
    :XML.(a,mutable(b|int))@(a,int) =>
        g(x.2); [6]
        h();
        x.2+1; // valid
fun g = lambda y:XML.int>>int => ... [7]
```

Listing 4. Permissions control aliasing, allowing DynXML to safely track state information.

3.4 Typing Rules

Throughout the remainder of Section 3 we will illustrate the judgements and inference rules that define the system with the code shown in Listing 4. The example illustrates alias control with permissions. The function f takes an XML document, x whose current state is an a node followed in sequence by an integer. The maximum state, however, allows the integer cell to change to a b node. Function f calls another function g, specified but undefined at $\boxed{7}$, and passes x's second subtree. Then, an unspecified function h is called. Finally, x.2 is added to one which is only valid if x.2 is an integer.

This example is not in let-normal-form, to ease reading, but this is otherwise valid DynXMLcode. DynXML verifies both that g doesn't change the memory cell referred to by x.2, and that h doesn't know about x at all.

DynXML's typing rules are shown in Figure 6. As discussed in Section 3.1 we have two typing judgements — one for terms and one for expressions. The first four rules in Figure 6 use only the term typing judgement. These rules are fairly standard: T-Var extracts the type of variables from the context; T-Unit assigns the unit type to unit terms; T-Loc extracts the type of locations from the store typing context; finally, the abstraction rule T-Abs assigns an arrow type to a function. Notice that the permission produced for the function, p', is extracted from the context, Δ_2 produced by checking the function body.

The rest of the typing rules rely on the substantially more complex expression typing judgment, $\Gamma;\Psi;\Delta\vdash_e e : T, M@M' \dashv \Delta'$. A permission context, written Δ, appears on both sides of the judgement. This is because it is a linear context and therefore the judgement both consumes and produces permissions. The type T in the expression type is the same as in the term type. The permission produced for e, written $M@M'$, follows immediately after the type. Permissions to other values produced by checking e appear in Δ'.

One of the primary goals of DynXML is to check that a modification to an XML tree is safe. The three premises of the T-ASGN rule perform checks to this end. The maximum state of the term must be enclosed in a mutable tag for reasons introduced in Section 2.3 and expanded in Section 3.6. We also ensure that the current state M_1' of the term t_2 on the right hand side of the assignment conforms to the maximum state of t_1. Finally, the permissions produced to t_1 retains the current state of t_2.

The let rule, T-LET, is perhaps the most important rule in our system. Notice that the permission context produced by the vast majority of the rules is empty. In general, our rules drop rules that are not explicitly related to the expression being evaluated. In other words we drop unused permissions in the input context. This simplifies our rules because we do not need to separate unused permissions. The let rule splits the permissions into two parts. The first part, Δ_1 is used to check the expression e_1 bound to the variable x. The rule does not specify which permissions are in Δ_1, but it should be clear from this discussion Δ_1 should equal the smallest subset of the input permissions that can safely check e_1. Since all DynXML programs are in let-normal form we avoid losing permissions by carefully splitting at in this manner. Then, the permission Δ_1' generated by checking e_1 is combined with the second part Δ_2 to check the body of the let e_2. Finally, the permission generated by checking e_2 is also produced when checking the whole let expression.

When a new pair is created in the system a new permission is created with the T-PAIR. The rule splits the context used to check the pair $t_1 t_2$ and then checks that it can derive a permission for each of t_1 and t_2. It then converts the XML type containing the pair of the maximum states from the permissions into a standard type. The resultant type is this standard type. The permission formed contains a maximum state which is a pair of t_1 and t_2's maximum state and a current state which is a pair of t_1 and t_2's current state.

New XML documents are checked using the T-XML. This rule creates a new permission for the new XML tree just like the pair rule. The current state of this new permission is simply the current state of enclosed term t, surrounded by the type of the new root node <n></n>. The maximum state is the same tree type, that instead contains a supertype of the current state of t. This rule does not specify a particular supertype, instead it is "chosen" non-deterministically. This supertype can contain mutable constructs and union types (|). In fact, T-XML is the only way to introduce these forms into XML trees.

The three rules T-PROJ1, T-PROJ2, and T-INNER are very similar to each other. Understanding T-PROJ1 should be sufficient to understanding the others. The first premise states that we must be able to derive a permission for $t.1$ from the context. This usually will require permission splitting as discussed in Section 3.2. The second premise says that the maximum state in the permission to $t.1$ converts to type T (Section 3.2). At $\boxed{6}$ of Listing 4, T-PROJ2 is employed when passing x.2 to g.

In T-APP, the permission context used to check the application is derived from the parameter specification provided by the programmer. The argument term is simply substituted for the formal parameter in the specified context. This rule ensures that the permission—which by a typing invariant is the only one in the system—is passed to the called function. Therefore, in Listing 4, f holds the only permission to x. This brings up subtle distinction between permissions and parameters. In our example h is not passed x and therefore cannot manipulate it. However, if h was a lambda nested inside f, then x would be in scope, and if the right permission were passed to h, then h could use x even if x itself were not passed as a parameter. Finally, notice that permission produced by includes both the permission context produced by checking the function and the permission to the argument ($t_2{:}p'$).

The final rule, T-PAGE, checks an XML tree that the programmer has declared to be a webpage with the page keyword. Webpages are treated specially in DynXML because the functions embedded in them are checked to ensure they expect the page as an argument and return it as a result (Section 3.5). Note, that the page keyword has only degenerate operational semantics, its primary purpose is to tell the typechecker to check the validity of events. The third premise invokes the event handling judgement which performs this check. Finally, the permission to the page has the same maximum state and current state. This ensures that the state of the page cannot change after the event checking has taken place.

3.5 Event Handlers

One of the core features of DynXML is that programmers can embed functions inside XML trees. We believe this cleanly encodes the event handlers for particular form elements (e.g. buttons, textboxes). These event handlers often use data from the page or modify the page. Therefore, a helpful intuition is to think of them as functions that are passed the page as an argument by the browser and return the page as a result. We need to check that these functions maintain the page in a valid state. As explained in Section 3.4 this check happens when an XML tree is declared as page with the page keyword. The rules governing these checks are shown in Figure 7.

The syntactic forms in the category $\mathcal{M}$, are XML Types with holes, and are shown in Figure 2. We borrow the hole notation from evaluation contexts [25]. We write a hole {-}, and we "fill" a hole in a $\mathcal{M}$ by writing $\mathcal{M}\{\mathcal{M}'\}$. Filling a hole in $\mathcal{M}$ with $\mathcal{M}'$ simply replaces the hole with $\mathcal{M}'$. For example, say we have a type $\mathcal{M}$ = $\mathsf{n}[\{-\}]$, filling it with unit by writing $\mathcal{M}\{\mathsf{Unit}\}$ gives us $\mathsf{n}[\mathsf{Unit}]$. Several XML types, namely pairs, union types, and trees contain subcomponents. The $\mathcal{M}$ syntactic category contains three types of forms: holes ({-}), XML types with exactly one of the subcomponents replaced with a $\mathcal{M}$, and XML Types (M).

The event checking rules rely on a special judgement, $M;\mathcal{M} \vdash_{\mathcal{M}} M'$. This judgement checks the arrow types representing all event handlers attached to a page. The left side of each arrow type should contain a permission to the page with current and maximum states that conform to the type of the page to which they belong. The rules in Figure 7 are stated declaratively, but here we will describe them algorithmically. The rules construct a type of the page that each event handler should expect. In the event checking judgement used in all of the rules, the M in the hypothesis is the maximum state of the page, the $\mathcal{M}$ is the partially constructed type of the page an event handler should expect, and the M in the conclusion is the subtree being checked.

H-XML, H-MUTABLE, H-UNION, and H-PAIR are responsible for splitting up the tree and recursively calling the judgement on their subcomponents. Notice that the partially constructed page type used to check each subcomponent in H-UNION and H-PAIR are different. An event handler that is in subtree A of a union type $A|B$ need not worry about handling a page that contains B. If the event fires, then we know the page contains A and not B. The rule H-UNIT does nothing as unit contains no subcomponents and therefore the search need not proceed further. If we were to add new types like string or int we would need to add similar rules for these types.

The rule H-ABS checks arrow types. The now fully constructed page type $T, p{>>}p'.\Delta_1{\rightarrow}\tau.\Delta_2$ must be a subtype of the current state stored in parameter's permission. The page must be able to be passed to the event handler. In addition the maximum state in the result permission must be a subtype of the maximum state of the page. We do not want the event handler to change the page beyond

$$\boxed{M;\mathcal{M} \vdash_{\mathcal{M}} M}$$

$$\frac{M_{type};\mathcal{M}\{\mathsf{n}[\{-\}]\} \vdash_{\mathcal{M}} M}{M_{type};\mathcal{M} \vdash_{\mathcal{M}} \mathsf{n}[M]}\text{H-XML}$$

$$\frac{M_{type};\mathcal{M}\{\mathsf{mutable}(\{-\})\} \vdash_{\mathcal{M}} M}{M_{type};\mathcal{M} \vdash_{\mathcal{M}} \mathsf{mutable}(M)}\text{H-MUTABLE}$$

$$\frac{M_{type};\mathcal{M} \vdash_{\mathcal{M}} M_1 \qquad M_{type};\mathcal{M} \vdash_{\mathcal{M}} M_2}{M_{type};\mathcal{M} \vdash_{\mathcal{M}} M_1|M_2}\text{H-UNION}$$

$$\frac{\begin{array}{c} M_{type};\mathcal{M}\{(\{-\}, M_2)\} \vdash_{\mathcal{M}} M_1 \\ M_{type};\mathcal{M}\{(M_1, \{-\})\} \vdash_{\mathcal{M}} M_2 \end{array}}{M_{type};\mathcal{M} \vdash_{\mathcal{M}} M_1, M_2}\text{H-PAIR}$$

$$\frac{}{M_{type};M_{page} \vdash_{\mathcal{M}} \mathsf{Unit}}\text{ H-UNIT}$$

$$\frac{\begin{array}{c} p' = M_2@M_2 \qquad p = M_{type}@M_1 \\ (\mathcal{M}\{T, p{>>}p'.\Delta_1{\to}\tau.\Delta_2\}) <: M_1 \qquad M_2 <: M_{type} \end{array}}{M_{type};\mathcal{M} \vdash_{\mathcal{M}} T, p{>>}p'.\Delta_1{\to}\tau.\Delta_2}\text{H-ABS}$$

Figure 7. Event Checking

the scope of the maximum state. Finally, the permission returned by the event handler must have the same maximum and current states to ensure the page isn't changed further after the page check in the fuction.

3.6 Mutable Keyword

The mutable keyword, as explained at the end of Section 2, reduces the pattern matching required of the programmer. After a function is returned, the permission from the returning function are merged with the permissions stored by the caller. The current states are merged by maintaining any information that doesn't appear within a mutable keyword in the called function. This information is safe to maintain because it cannot be modified inside the function.

The rules in Figure 8 define a new judgement, $M_1;M_2;M_3 \vdash_m M_4$, which supports the mutable keyword. The judgement derives the current state of the function parameter M_4 from the parameter's maximum state, M_1; the current state of the argument at the call site, M_2; and the current state of the parameter after the function has returned M_3. Throughout the rest of this section we will call M_1 the maximum state, M_2 the input, M_3 the output, and M_4 the result. This helper judgement is a new premise in a modified T-APP rule. The result is substituted for the current state of the argument in the permission context Δ_2.

Rules M-1, M-2, and M-3 allow for arbitrary reordering of union types. This corresponds to reordering of union subtyping, S-UNION3, discussed in Section 3.3. The next two rules, M-PAIR and M-XML, apply the judgement recursively to subtrees of the maximum state, input, and output. Any simple type, handled by M-T, is always exactly the same in the maximum state, input, and output, and it is therefore also the result. When a mutable keyword appears in the maximum state, as in M-MUT, the result is the current state of the parameter.

The remaining rules apply when the maximum state is a union type. Remember when reading these rules that ordering within unions is immaterial. When the input, M_2, is *not* a union type, then the result derives from the half of the maximum state corresponding to M_2. This principle is enunciated primarily in M-UNION1, which eliminates M_3' from consideration since it could come about only

$$\boxed{M;M;M \vdash_m M}$$

$$\frac{M_1'|M_1|; M_2;M_3 \vdash_m M_4}{M_1|M_1'|; M_2;M_3 \vdash_m M_4}\text{M-1}$$

$$\frac{M_1;M_2'|M_2;M_3 \vdash_m M_4}{M_1;M_2|M_2';M_3 \vdash_m M_4}\text{M-2} \qquad \frac{M_1;M_2;M_3'|M_3 \vdash_m M_4}{M_1;M_2;M_3|M_3' \vdash_m M_4}\text{M-3}$$

$$\frac{M_1;M_2;M_3 \vdash_m M_4 \qquad M_1';M_2';M_3' \vdash_m M_4'}{M_1, M_1';M_2, M_2';M_3, M_3' \vdash_m M_4, M_4'}\text{M-PAIR}$$

$$\frac{M_1;M_2;M_3 \vdash_m M_4}{\mathsf{n}[M_1];\mathsf{n}[M_2];\mathsf{n}[M_3] \vdash_m \mathsf{n}[M_4]}\text{M-XML}$$

$$\frac{}{T;T;T \vdash_m T}\text{M-T} \qquad \frac{}{\mathsf{mutable}(M_1);M_2;M_3 \vdash_m M_3}\text{M-MUT}$$

$$\frac{\begin{array}{c} M_2 <: M_1 \qquad M_2 \not<: M_1' \qquad M_3 <: M_1 \\ M_1;M_2;M_3 \vdash_m M_4 \end{array}}{M_1|M_1';M_2;M_3|M_3' \vdash_m M_4}\text{M-UNION1}$$

$$\frac{\begin{array}{cc} M_2 <: M_1 & M_2 <: M_1' \\ M_3 <: M_1 & M_3' <: M_1' \\ M_1;M_2;M_3 \vdash_m M_4 & M_1';M_2;M_3' \vdash_m M_4' \end{array}}{M_1|M_1';M_2;M_3|M_3' \vdash_m M_4|M_4'}\text{M-UNION2}$$

$$\frac{M_2 <: M_1 \qquad M_3 <: M_1 \qquad M_1;M_2;M_3 \vdash_m M_4}{M_1|M_1';M_2;M_3 \vdash_m M_4}\text{M-UNION3}$$

$$\frac{\begin{array}{cc} M_2 <: M_1 & M_3 <: M_1 \\ M_2' <: M_1' & M_3' <: M_1' \\ M_1;M_2;M_3 \vdash_m M_4 & M_1';M_2';M_3' \vdash_m M_4' \end{array}}{M_1|M_1';M_2|M_2';M_3|M_3' \vdash_m M_4|M_4'}\text{M-UNIONALL}$$

Figure 8. Mutable

if M_2 were a subtype of M_1', which it is not. To illustrate this, look back at the rate function in Listing 2. The page enters the rate function with current state rating page, and the current state returned by the rate function is rec page. The system merges the three part union in rec page with the ratings element in rating page. It does so by discarding the other two nodes in the union with M-UNION1.

In the rare case that the input is a subtype of both halves of the union, the system conservatively considers both halves of the output in M-UNION2. When both the input and output are not unions then rule M-UNION3 discards half of the union in the maximum state. Finally, M-UNIONALL is the same as M-PAIR, except additional subtyping premises are added to properly align the union type halves. There is no rule for the case when the input is a union type, but the output is not since it is impossible.

An alternative solution to the mutable keyword is to introduce bounded quantification [7, 24] to our system. Bounded quantification can sometimes more powerfully provide opportunities for reuse. However, as explained in Section 2.3, bounded quantification does not scale as well to XML documents with multiple mutable sections. In addition, we believe programmers will better understand the simple semantics of the mutable keyword — that one can assign to the enclosed subtree — than the complex semantics of bounded quantification.

$$\boxed{e \,|\, \mu \mapsto e \,|\, \mu}$$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \,|\, \mu \mapsto [v/x]e \,|\, \mu}\ \textsc{E-LetV}$$

$$\frac{e_1 \,|\, \mu \mapsto e_1' \,|\, \mu'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \,|\, \mu \mapsto \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2 \,|\, \mu'}\ \textsc{E-Let}$$

$$\frac{\ell \notin dom(\mu)}{\texttt{<n>}v\texttt{</n>} \,|\, \mu \mapsto \ell \,|\, [\ell \hookrightarrow \texttt{<n>}v\texttt{</n>}]\mu}\ \textsc{E-XML}$$

$$\frac{\ell \notin dom(\mu)}{v_1 v_2 \,|\, \mu \mapsto \ell \,|\, [\ell \hookrightarrow v_1 v_2]\mu}\ \textsc{E-Pair} \qquad \frac{\mu(\ell) = v_1 v_2}{\ell.1 \,|\, \mu \mapsto v_1 \,|\, \mu}\ \textsc{E-Proj1}$$

$$\frac{\mu(\ell) = v_1 v_2}{\ell.2 \,|\, \mu \mapsto v_2 \,|\, \mu}\ \textsc{E-Proj2} \qquad \frac{\mu(\ell) = \texttt{<n>}v\texttt{</n>}}{\ell{*} \,|\, \mu \mapsto v \,|\, \mu}\ \textsc{E-*}$$

$$\frac{}{\ell := v_2 \,|\, \mu \mapsto () \,|\, [\ell \hookrightarrow v_2]\mu}\ \textsc{E-Asgn} \qquad \frac{}{\mathsf{page}\ t \,|\, \mu \mapsto t \,|\, \mu}\ \textsc{E-Page}$$

$$\frac{}{(\lambda x{:}\tau.\Delta \Rightarrow e)(v) \,|\, \mu \mapsto [v/x]e \,|\, \mu}\ \textsc{E-App}$$

Figure 9. Operational Semantics

3.7 Operational Semantics

We define DynXML with a small step operational semantics. The store, μ, is carried along (possibly changed) after each step. The full operational semantics is shown in Figure 9. The first thing most readers will notice is that there is only one congruence rule, E-LET. This is because expressions must be bound to a let and only terms appear in other syntactic categories. E-XML evaluates a new XML document by creating a new location in the store and storing the document in that location. Similarly, E-PAIR evaluates a pair of XML documents by creating a new location for the pair. Rules E-PROJ1, E-PROJ2, E-* project out the first element of a pair, the second element of a pair, and the internals of a tree respectively. The rest of the rules provide standard semantics for let, assignment and application.

Notably absent from our syntax is an explicit operator for location dereferencing (written ! in most systems). Instead, the three rules E-PROJ1, E-PROJ2, E-* perform both their main functions, described in the previous paragraph, and dereference the location of a pair or tree from the store. Our locations point not only to values as in standard systems, but also to pairs and trees. This supports the common idiom in dynamic web programming to assign directly to the page or subpage. These are not values in the traditional sense because they can take a step, instead they are part of a special category in our syntax, s, of store "values." We use an equi-recursive [12] approach to recursive types and therefore we also do not have any rules for folding and unfolding.

4. Related Work

Context logic [6], a novel spatial logic, which we introduced in Section 1, is also able to check that mutable XML trees *always* conform to a schema. Context logic is used to express the pre and post conditions of Hoare logic. The work has been used to specify the W3C Document Object Model (DOM) [16]. Their logic is more expressive than the specifications in DynXML, but it is more heavyweight than our types-based system and therefore less easily automatable. We have not implemented DynXML, but our group has successfully automated a similar permission-based effect system [3].

Event handlers were first embedded in XML trees in Xdπ [15], an extension of the π-calculus for modeling distributed programs. Their event handlers, which are π-processes extended with location-migration and, like the event handlers in DynXML, XML mutation. However, they do not check if the mutated trees conform to a schema. Their focus is on dynamic web services with an XML data store, not mutable web pages. The other related work falls into three categories: industrial frameworks and tools for constructing web applications, programming language support for XML types, and web application programming languages.

Since the popularization of AJAX in 2004, there has been massive effort in industry aimed at writing client-side code more abstractly. The general strategy employed by industry is to produce a web application framework which allows to write their client-side code as plugins in Java, C# or Ruby. The framework is responsible for generating HTTP responses which include JavaScript code to perform the actions specified by the plugin. Google Web Toolkit[6] generates JavaScript from arbitrary Java code. The framework provides basic syntactic checks to ensure that the code is translatable to JavaScript, but it does not ensure that the page manipulations the programmer writes are valid.

The AJAX support in JavaServer Faces[7] and ASP.NET employ a very different tactic. The frameworks generate a Java or C# class to represent the structure of a page. Client-side code manipulates an object of this class and the compiler checks that these manipulations safely read from and write to the representation. Unfortunately, the object/XML impedance mismatch [22] makes it essentially impossible for the classes generated by the frameworks to accurately represent the page. In JavaServer Faces and ASP.NET, only top level nodes in the XML tree (i.e. those not embedded in lists, union types, etc.) are specified by the classes and therefore all other manipulations use unsafe features of Java or C#.

Programming languages researchers have extensively investigated support for XML within programming languages. However, unlike DynXML, almost all of this work has focused on immutable XML trees. Hosoya and Pierce provide the foundation for our work with regular expression types [21]. Hosoya and Pierce define subtyping in terms of set inclusion. They also define an algorithm for efficiently determining if the set of XML documents that have type A is a subset of the set of XML documents with type B [21]. There system is more expressive in that there are pairs of XML types that are subtypes in their system, but not in ours. We expect that we could easily incorporate their subtyping rules into our system. However, their rules are substantially more complex than ours and including them would needlessly complicate our proof. The regular expression pattern matching employed by Hosoya and Pierce [19] can also be easily adapted to DynXML by modifying their rules to use the types within our permissions instead of regular types. Every place the acceptance relation is used in their matching rules, written $t \in T$, we would replace T with the current state in the permission to t.

CΩ [4] and XStatic [14] attempt to add support for XML types to Java-like languages, however the trees in both of these languages are immutable. Flux [8] supports mutation of XML trees, but it is aimed at update queries of XML data, not direct manipulation of XML webpages. XJ [17] adds supports for both querying and in-place mutation of XML trees to Java. However, it does not constrain these mutations to a type nor does it prevent page manipulating programs from going wrong.

Finally, several researchers have proposed languages for unifying web application development into one language. The Links project [11] compiles an ML-like language into JavaScript on

[6] http://code.google.com/webtoolkit/

[7] http://java.sun.com/javaee/javaserverfaces/

the client, O'Caml on the server, and SQL for database access. ML5 [23] extends ML with a modal logic for reasoning about distributed resources. Swift [9] is a Java-like language for writing web applications that are automatically partitioned to ensure conformance to security policies. None of these languages support sophisticated types for XML, and both ML5 and Links avoid state and mutable data.

5. Conclusion

Dynamic web applications are playing an ever more important role in our lives. Every passing day brings more and more complex and dynamic web applications. We have moved beyond e-commerce, search, and webmail. We are replacing traditional software with web software—Salesforce.com is already the dominant enterprise customer relational management tool and Google Docs is commonly used in place of Microsoft Office. Many popular web applications even allow end users to develop powerful extensions (e.g. Facebook applications).

At the same time, web applications are very brittle. Everyone who uses the web regularly has experienced data loss, browser crashes, button clicks that don't do anything, etc. As we mentioned in the introduction, one can quickly see *a lot* of evidence of this problem simply by checking the browser's error console after long surfing session. This trend of more and more dynamic web applications is not sustainable without significant improvement of programming languages and tools used to build these applications. DynXML provides the foundations for a solution to this important problem. We believe the specification and checking capabilities it provides are powerful enough to guarantee the absence of structural errors and lightweight enough to be practically applied to real applications. We are currently implementing DynXML as part of the Plaid language in order to evaluate the practicality of the mechanisms descripted in this paper.

Acknowledgments

We would like to thank the Éric Tanter and the PLAID group for their helpful feedback and suggestions. This research was supported by DARPA grant #HR00110710019. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an xml-centric general-purpose language. In *ICFP '03: Proceedings of the Eighth ACM International Conference on Functional Programming*, pages 51–63.

[2] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–320.

[3] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical api protocol checking with access permissions. In *ECOOP '09: Object Oriented Programming, 23rd European Conference*, pages 195–219.

[4] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in CΩ. In *ECOOP 2005: Object-Oriented Programming, 19th European Conference*, pages 287–311.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth edition), November 2008. http://www.w3.org/TR/REC-xml/.

[6] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL '05: Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 271–282.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[8] J. Cheney. FLUX: functional updates for XML. In *ICFP '08: Proceeding of the 13th ACM International Conference on Functional Programming*, pages 3–14.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6):31–44, 2007.

[10] W. W. W. Consortium. XHTML 1.0 strict document type definition. W3C Recommendation, 2002. http://www.w3.org/TR/xhtml1/dtd/xhtml1-strict.dtd.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO '06: Formal Methods for Components and Objects, 5th International Symposium*, pages 266–296.

[12] K. Crary, R. Harper, and S. Puri. What is a recursive module? *SIGPLAN Not.*, 34(5):50–63, 1999.

[13] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer, October 2004. http://www.w3.org/TR/xmlschema-0/.

[14] V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP '03: Object-Oriented Programming, 17th European Conference*, pages 469–474.

[15] P. Gardner and S. Maffeis. Modelling dynamic web data. *Theor. Comput. Sci.*, 342(1):104–131, 2005. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2005.06.006.

[16] P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local hoare reasoning about dom. In *PODS '08: Proceedings of the 27th ACM symposium on Principles of Database Systems*, pages 261–270.

[17] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 278–287.

[18] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. *Theor. Comput. Sci.*, 360(1):327–351, 2006.

[19] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *J. Funct. Program.*, 13(6):961–1004, 2003. ISSN 0956-7968.

[20] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Interet Technol.*, 3(2):117–148, 2003.

[21] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[22] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch. In *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 285–364. Springer, 2007.

[23] T. Murphy VII, K. Crary, and R. Harper. Type-safe distributed programming with ml5. In *Trustworthy Global Computing 2007*.

[24] B. C. Pierce. Bounded quantification is undecidable. In *POPL '92: Proceedings of the 19th ACM symposium on Principles of Programming Languages*, pages 305–315.

[25] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.