Analysis of Imperative XML Programs

Christoph Reichenbach[1]

University of Colorado at Boulder

Michael G. Burke

IBM T. J. Watson Research Center

Igor Peshansky

IBM T. J. Watson Research Center

Mukund Raghavachari

Google Inc.

Abstract

The widespread adoption of XML has led to programming languages that support XML as a first class construct. In this paper, we present a method for analyzing and optimizing imperative XML processing programs. In particular, we present a program analysis, based on a flow-sensitive type system, for detecting both redundant computations and redundant traversals in such programs. The analysis handles imperative loops that traverse XML values explicitly and declarative queries over XML data in a uniform framework. We describe two optimizations that take advantage of our analysis: one merges queries that traverse the same set of XML nodes, and the other replaces an XPath expression by a previously computed result. We demonstrate performance improvements for selected XMark benchmark queries and XLinq sample queries.

Key words: Program analysis, XML, Imperative programming

1. Introduction

XML processing applications in imperative languages such as Java and C# use runtime APIs such as DOM [21], or language-based approaches such as XLinq [3], XJ [6], or XAct [9]. In either case, the programmer is provided with an XML data model and navigational constructs. The XML data model is typically an object view, where each element in an XML document is instantiated as an object. The navigational

[1]This work was supported in part by NSF Grant ST-CRTS 0540997

constructs range from library routines that access children of a node in an XML tree, to comprehensions, to queries in declarative query languages such as XPath [20].

The imperative nature of systems such as XLinq and XJ poses challenges that differ from those in declarative languages such as XQuery. Consider the program in Figure 1 written in a language based on XJ. Assume that in Line 1, x is set to refer to some XML value. The XPath expression on Line 2 can be interpreted as computing the set of all descendants of the root of the tree referred to by x such that each member of the result is labeled book and has an attribute author with value 'Poe'. Similarly, the XPath expression on Line 5 can be interpreted as computing the set of all publisher descendants of x. Some challenges in the optimization of such programs are:

- **Query identification:** Queries may be latent in a program where programmers combine imperative traversals (with variable assignment) with declarative queries. Consider the loop that begins on Line 7 of Figure 1. The statement on Line 10 can be interpreted as $k = k \cup \{i\}$—the accumulate operator "$\Leftarrow$" models the invocation of a method such as add on an instance of the Set class in Java. Observe that at the end of the loop, k is guaranteed to contain the same value as y. While the loop itself is not redundant (it has effects), the computation of k certainly is.

- **Optimizations across Multiple Queries:** The detection of two queries (or subqueries) that return the same results could be used to remove redundant computation. The complication in this analysis is that there are many ways of writing equivalent queries (including as explicit loops), which precludes the use of syntactic techniques such as value numbering [1, 8]. In all executions of the program of Figure 1, the variable v on Line 4 will refer to the same value as y—the computation of v is redundant.

  Further, two different computations over an XML tree may visit the same set of nodes, even if they do not produce the same value. If so the two computations could be combined to return the two results in a single traversal. This transformation is called *tupling*. Consider the expressions in Lines 2 and 5. They traverse the same set of nodes (the subtree rooted at x), but filter these sets in different ways—both sets of results can be produced efficiently in one traversal.

This paper studies the analysis of imperative XML processing programs, where traversals over data may be specified in many ways—as explicit loops over data and in terms of XPath expressions. We present a program analysis, based on a flow-sensitive type system, for detecting both redundant computations and redundant traversals in such programs. The analysis handles both loops that traverse XML values explicitly and declarative query expressions in a uniform framework. For exposition, we focus on a core language for XML processing based on the XJ programming language.

The contributions of this paper are an analysis, based on a flow-sensitive type system, that computes a symbolic representation of the values assumed by each XML expression or variable in a program; a proof of correctness of the analysis; a description of transformations enabled by the analysis; experimental results that provide a preliminary demonstration of the effectiveness of the transformations.

```
1  x = ...;
2  y = x//book[@author='Poe'];
3  u = x//book;
4  v = u[@author='Poe'];
5  z = x//publisher;
6  k = ∅;
7  foreach i in u {
8     System.out.println(i);
9     if (i[@author='Poe'])
10       k ⇐ i
11 }
```

Figure 1: Example demonstrating redundant computations.

Structure of the Paper. Section 2 introduces the XML processing language that we use as the basis of the exposition of our analysis. In Section 3 we describe the types that track the values of expressions and variables in programs, and formally define correctness criteria for our analysis. In Section 4 we present a flow-sensitive type system for detecting redundant computations and traversals. We describe the transformations enabled by the analysis in Section 5. Section 6 discusses how our approach can be extended to a full imperative language such as XJ. Section 7 describes our implementation and experimental results. Section 8 presents related work. We conclude in Section 9 and give a proof of correctness of our analysis as an appendix.

2. Syntax and Semantics

We model XML documents as ordered, labeled trees. $\mathfrak{T}$ refers to the set of all such trees, and $\mathcal{N}$ is the (infinite) set of all nodes used in trees in $\mathfrak{T}$. Each node n in each XML tree has unique identity and a label, $\text{LABEL}(n)$, drawn from an infinite alphabet Σ (we use uppercase characters (A, B, C) to represent members of Σ).

We focus on a fragment of XPath 1.0 [20], whose (somewhat non-standard) syntax is listed in Figure 2. The evaluation of an XPath expression is always with respect to a set of nodes in XML trees (the nodes could belong to different XML trees) and the result is another set of nodes. The operators $\downarrow$ and $\downarrow^+$ represent the *child* and *descendant* traversals, that is, they return the union of the set of children and the set of descendants of the nodes in the input node set, respectively. In the syntax, s ranges over Σ and it represents a node test, which filters its inputs with respect to s. The semantics of these expressions is standard and is also provided in Figure 2.

We describe a core imperative language for XML processing that serves as the domain for our static analysis (Figure 3). For simplicity, we have not included XML literal-based construction, XML updates, effects (such as I/O or Java-like constructs), a more expressive XPath fragment, or schema information in our core language. The handling of these constructs is mostly orthogonal to the central ideas of this paper. We use this compact core language for the exposition and proof of soundness of our analysis. A proof of soundness for the core language illustrates interesting features of

$$Xp ::= \epsilon \mid \downarrow \mid \downarrow^+ \mid s \mid Xp/Xp \mid Xp[Xp] \mid Xp[\neg Xp]$$

$$[\![\cdot]\!] : \mathcal{P}(\mathcal{N}) \to \mathcal{P}(\mathcal{N})$$

$$\begin{array}{l}
[\![\epsilon]\!](N) = N \\
[\![\downarrow]\!](N) = \bigcup\{child(n) \mid n \in N\} \\
[\![\downarrow^+]\!](N) = \bigcup\{descendant(n) \mid n \in N\} \\
[\![s]\!](N) = \{n \in N \mid \textsc{label}(n) = s\} \\
[\![Xp_1/Xp_2]\!](N) = [\![Xp_2]\!]([\![Xp_1]\!](N)) \\
[\![Xp_1[Xp_2]]\!](N) = \{n \in [\![Xp_1]\!](N) \mid [\![Xp_2]\!](\{n\}) \neq \varnothing\} \\
[\![Xp_1[\neg Xp_2]]\!](N) = \{n \in [\![Xp_1]\!](N) \mid [\![Xp_2]\!](\{n\}) = \varnothing\}
\end{array}$$

Figure 2: Syntax and semantics of XPath-like expressions. We further use parentheses for disambiguation.

Var	::=	*Id* | *Index* | *Doc*
Expr	::=	*Var* | *Var* / Xp | $\varnothing$
Stmt	::=	*Id* = *Expr*
	|	*Id* $\Leftarrow$ *Expr*
	|	if (*Expr*) then *Stmt* else *Stmt*
	|	foreach *Index* in *Expr* *Stmt*
	|	*Stmt* ; *Stmt*
	|	skip

Figure 3: Language syntax.

a proof for an imperative language such as XJ, and offers a base for understanding how to treat extensions of the core language (Section 6).

In the language, there are three disjoint, finite sets of variables—*Id*, *Index*, and *Doc*. *Index* variables may only appear in foreach statements, where each foreach statement has a unique *Index* variable. The *Doc* variables represent some input XML document or XML construction. Only *Id* variables may be on the left-hand side of assignments or accumulations. *Index* variables are updated implicitly by loops and *Doc* variables remain constant through the program.

The semantics of program execution is provided in Figure 4. A value in the language is a subset of $\mathcal{N}$. A store σ maps each program variable to such a value. $\langle S, \sigma\rangle \Downarrow \sigma'$, where σ, σ' are *stores*, represents that the evaluation of statement S takes the program from store σ to σ'. $\langle$*Expr*$, \sigma\rangle \models value$ states that expression *Expr* evaluates to $value$, given store σ.

In the initial store, each *Id* variable used in the program is mapped to $\varnothing$, and each *Doc* variable used in the program is mapped to the root node of some tree in $\mathfrak{T}$. The expression *Var*/Xp evaluates the XPath expression Xp with respect to the set of nodes specified by *Var*.

$$\text{VAR}\ \frac{}{\langle x, \sigma\rangle \models \sigma(x)} \qquad \text{XPATH}\ \frac{}{\langle x/Xp, \sigma\rangle \models [\![Xp]\!](\sigma(x))} \qquad \text{EMPTY}\ \frac{}{\langle \varnothing, \sigma\rangle \models \varnothing}$$

$$\text{ASSIGN}\ \frac{\langle \mathit{Expr}, \sigma\rangle \models N}{\langle \mathsf{x} = \mathit{Expr}, \sigma\rangle \Downarrow \sigma[x \mapsto N]} \qquad \text{ACCUM}\ \frac{\langle \mathit{Expr}, \sigma\rangle \models N \qquad N' = \sigma(x) \cup N}{\langle x \Leftarrow \mathit{Expr}, \sigma\rangle \Downarrow \sigma[x \mapsto N']}$$

$$\text{IF-THEN}\ \frac{\langle \mathit{Expr}, \sigma\rangle \models N, N \neq \varnothing \qquad \langle S_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}(\mathit{Expr})\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma\rangle \Downarrow \sigma'} \qquad \text{IF-ELSE}\ \frac{\langle \mathit{Expr}, \sigma\rangle \models \varnothing \qquad \langle S_2, \sigma\rangle \Downarrow \sigma'}{\langle \mathsf{if}(\mathit{Expr})\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma\rangle \Downarrow \sigma'}$$

$$\text{FOREACH}\ \frac{\begin{array}{c}\langle \mathit{Expr}, \sigma\rangle \models \{x_1, x_2, \ldots, x_k\} \\ \langle S, \sigma[i \mapsto \{x_1\}]\rangle \Downarrow \sigma_1 \\ \vdots \\ \langle S, \sigma_{k-1}[i \mapsto \{x_k\}]\rangle \Downarrow \sigma_k\end{array}}{\langle \mathsf{foreach}\ i\ \mathsf{in}\ \mathit{Expr}\ S, \sigma\rangle \Downarrow \sigma_k[i \mapsto \varnothing]} \qquad \text{SKIP}\ \frac{}{\langle \mathsf{skip}, \sigma\rangle \Downarrow \sigma}$$

$$\text{COMPOSE}\ \frac{\langle S, \sigma\rangle \Downarrow \sigma' \qquad \langle S', \sigma'\rangle \Downarrow \sigma''}{\langle S; S', \sigma\rangle \Downarrow \sigma''}$$

Figure 4: Language semantics.

A program is a *Stmt*. The foreach loop iterates over the value denoted by its *Expr*, which we call the loop's *iteration space*; for each node in this set, it binds the *Index* variable to a singleton set consisting of that node, and then evaluates the *Stmt* in the new store. Since an index variable is only defined within a loop, it is removed from the result store of the loop. The execution of foreach is non-deterministic (the elements are visited in some unspecified order). The statement skip has no effect on the store. The accumulate statement, x $\Leftarrow$ y, sets x to the equivalent of x $\cup$ y. Observe that one can express general union operations, *i.e.*, x = y $\cup$ z, with a pattern like x = y; x $\Leftarrow$ z.

Consider the code sample in Figure 5. Line 1 sets x to the singleton set containing the root of some XML tree that is referred to by the *Doc* variable d. The foreach loop on lines 3–7 iterates over an XPath expression evaluated with respect to the value referred to by x. This expression returns a set of nodes containing all B descendants of the root node of the tree referenced in Line 1. In each iteration of the loop, if a particular B node has a C child, then the B node is added to y. At the end of the loop, y will refer to the equivalent of the expression x/$\downarrow^+$/(B[$\downarrow$/C]).

3. Types

The types in our type system are the "don't know" type or ξ; the "empty" type or $\varnothing$, which denotes that a variable or expression evaluates to an empty set; types of the form (x, Xp, Ψ), where Ψ is a set $\{\psi_1, \ldots, \psi_k\}$ and each ψ_i is of the form τ or $\neg\tau$ with

```
1  x = d;
2  y = ∅;
3  foreach i in x/↓+/B
4    if (i/↓/C) then
5      y ⇐ i
6    else
7      skip
```

Figure 5: Sample program.

τ a type, and union types, $\tau_1 \oplus \tau_2$. This gives us a type lattice with $\varnothing$ as the bottom and ξ as the top element.

$$\tau ::= \xi \mid \varnothing \mid (\mathsf{x}, Xp, \Psi) \mid \tau \oplus \tau'$$

$$\Psi = \{\psi_1, \ldots, \psi_k\}, \text{where } \psi_i ::= \tau \mid \neg\tau$$

In a type (x, Xp, Ψ), x is either a *Doc* or an *Index* variable, and Xp is an XPath expression. For such a type, we refer to x as the *context variable* of the type, and Ψ as the *filter* of the type. If a variable has the type $(\mathsf{d}, \epsilon, \varnothing)$, under all executions, the variable refers to the value to which the store maps d. The type $(\mathsf{d}, \epsilon, \Psi)$ is equivalent to $(\mathsf{d}, \epsilon, \varnothing)$ if the denotation of each $\psi \in \Psi$ is non-empty, and to $\varnothing$ otherwise.

For example, consider the type $(\mathsf{x}, A, \varnothing)$: This type describes precisely the type of all elements in the XPath expression x/A. If we now add a filter $\{\tau\}$, then $(\mathsf{x}, A, \{\tau\})$ may describe one of three things:

1. If τ does not describe any elements, then neither does $(\mathsf{x}, A, \{\tau\})$.
2. If τ describes some elements, then $(\mathsf{x}, A, \{\tau\})$ again describes all elements in the path x/A.
3. If τ denotes the type of the distinguished 'don't-know' set ξ, then we don't know whether or not it applies as a filter; thus, we must interpret $(\mathsf{x}, A, \{\tau\})$ as ξ also.

More precisely, the denotation of a type τ is defined in terms of a special store D. This denotation, $[\![\tau]\!]_D$, is a subset of $\mathcal{N}$ or a distinguished set ξ. Since $[\![-]\!]_D$ only depends on *Doc* variables, which never change during the course of the program, $[\![-]\!]_D$ is independent of any execution state. Without loss of generality, we assume in the remainder of this document that all stores σ are within *Doc* pointwise equal. We then define the semantics of our types as:

$$[\![\xi]\!]_D = \xi \qquad [\![\varnothing]\!]_D = \varnothing \qquad [\![\tau_1 \oplus \tau_2]\!]_D = [\![\tau_1]\!]_D \cup [\![\tau_2]\!]_D$$

$$[\![(\mathsf{x}, Xp, \Psi)]\!]_D = \begin{cases} [\![Xp]\!](D(\mathsf{x})) & \textsf{satisfied}(\Psi) = true \\ \xi & \textsf{satisfied}(\Psi) = \xi \\ \varnothing & \text{otherwise} \end{cases}$$

The function $\textsf{satisfied}(\Psi)$ is a three-valued logic function:

$$\textsf{satisfied}(\Psi) = \begin{cases} \xi & \exists\tau \in \Psi \vee \neg\tau \in \Psi, [\![\tau]\!]_D = \xi. \\ true & \forall\tau \in \Psi, [\![\tau]\!]_D \neq \varnothing \ \wedge \forall\neg\tau \in \Psi, [\![\tau]\!]_D = \varnothing \\ false & \text{otherwise} \end{cases}$$

A typing environment, Γ, maps program variables to types. Our goal is a type system that ensures that if two variables x and y are assigned equivalent types at a program point, then in all executions of the program, x and y refer to identical values at that program point. Our notion of equivalence here is semantic in nature. We find it sufficient to require this notion of equivalence, $\equiv$, to be a conservative approximation (sound but not necessarily complete) of full semantic equivalence that satisfies certain minimal properties. These properties are relevant for our correctness proof (Appendix 10.4) and can be helpful during implementation. We require that $(\equiv)$ at least (i) relates syntactically identical types; (ii) satisfies the rewriting rules from Table 6; and (iii) satisfies a further non-discrimination property (Definition 17). We later motivate and discuss individual rewriting rules in detail.

A store σ is *consistent* with a typing environment Γ, iff for all $\mathsf{x} : \tau \in \Gamma$, $\tau \equiv \xi$ or $[\![\tau]\!]_D = \sigma(\mathsf{x})$. With this definition of consistency, we define soundness as follows:

Definition 1 (Statement Typing Soundness). If a store σ is consistent with Γ, and $\Gamma\ \{S\}\ \Gamma'$ and $\langle S, \sigma\rangle \Downarrow \sigma'$, then σ' is consistent with Γ'.

By $\Gamma\ \{S\}\ \Gamma'$, we mean that if the type system starts in environment Γ, the environment at the end of S is Γ'. Specifically, if a store σ is consistent with Γ and $\Gamma(\mathsf{x}) \equiv \Gamma(\mathsf{y})$, and $\Gamma(\mathsf{x}) \not\equiv \xi$, then x and y contain the same value at that point.

In the following, we develop a type system with this property.

4. A Flow-Sensitive Type System

We first consider a type system for detecting when variables must refer to the same value in programs *without* loops. We then extend this type system to support loops. The typing judgments for expressions (Figure 7) are of the form $\Gamma \vdash \mathit{Expr} : \tau$.

It is straightforward to show that if a store σ is consistent with respect to an environment Γ, and $\langle \mathit{Expr}, \sigma\rangle \models N$, then $\Gamma \vdash \mathit{Expr} : \tau$ implies that $\tau \equiv \xi$ or $[\![\tau]\!]_D = N$.

4.1. Analyzing Programs Without Loops

Figure 8 lists the judgments of our type system for statements other than foreach. The judgments are of the form $\Gamma\ \{S\}\ \Gamma'$. A program S is well typed if $\Gamma_\emptyset\ \{S\}\ \Gamma'$ is derivable, where $\Gamma_\emptyset$ assigns the $\varnothing$ type to each *Id* variable, and $(\mathsf{d}, \epsilon, \varnothing)$ to each *Doc* variable d.

The rule for accumulation reflects the set-based semantics of the operation — the resulting type is the union of the types of the two expressions in the accumulation.

The IF rule is designed to handle cases such as the following statement:

if c **then** y = c/Xp_2 **else** y = $\varnothing$

If the type of the variable c is $(\mathsf{d}, Xp_1, \varnothing)$, then ideally the analysis should derive the type $(\mathsf{d}, Xp_1/Xp_2, \varnothing)$ for y at the end of the conditional. In any execution of the program, the store would either map c to $\varnothing$ or to a non-empty set of nodes. In the first case, the else branch would be taken, and $[\![(\mathsf{d}, Xp_1/Xp_2, \varnothing)]\!]_D = \varnothing$, which is sound. If c is non-empty, then again $(\mathsf{d}, Xp_1/Xp_2, \varnothing)$ would be an appropriate type according to the x/Xp rule in Figure 7.

$$\tau_1 \oplus \tau_2 \iff \tau_2 \oplus \tau_1 \qquad (\textit{comm})$$

$$(\tau_1 \oplus \tau_2) \oplus \tau_3 \iff \tau_1 \oplus (\tau_2 \oplus \tau_3) \qquad (\textit{assoc})$$

$$\begin{array}{r} (x, Xp, \{\tau\} \cup \Psi) \\ \oplus (x, Xp, \{\neg\tau\} \cup \Psi) \end{array} \iff (x, Xp, \Psi) \quad (\textit{if } \overline{\xi}(\tau)) \qquad (\textit{join})$$

$$(x, Xp, \{\tau_1 \oplus \tau_2\} \cup \Psi) \Longrightarrow (x, Xp, \{\tau_1, \tau_2\} \cup \Psi) \qquad (\textit{flat-0})$$

$$(x, Xp, \{\neg(\tau_1 \oplus \tau_2)\} \cup \Psi) \Longrightarrow (x, Xp, \{\neg\tau_1, \neg\tau_2\} \cup \Psi) \qquad (\textit{flat-0}')$$

$$(x, Xp, \{(y, Xp', \Psi')\} \cup \Psi) \Longrightarrow (x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \Psi') \qquad (\textit{flat-1})$$

$$(x, Xp, \{\neg(y, Xp', \Psi')\} \cup \Psi) \Longrightarrow (x, Xp, \{\neg(y, Xp', \varnothing)\} \cup \Psi \cup (\neg\Psi')) \qquad (\textit{flat-1}')$$

$$\tau \oplus \varnothing \Longrightarrow \tau \qquad (\textit{empty})$$

$$(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi)) \Longrightarrow (\mathsf{x}, Xp, \Psi) \qquad (\textit{selfdep})$$

$$(\mathsf{x}, \epsilon/Xp, \Psi) \Longrightarrow (\mathsf{x}, Xp, \Psi) \qquad (\textit{xp}\epsilon)$$

$$(\mathsf{x}, Xp/\epsilon, \Psi) \Longrightarrow (\mathsf{x}, Xp, \Psi) \qquad (\textit{xp}\epsilon')$$

$$(\mathsf{x}, (Xp/Xp')/Xp'', \Psi) \iff (\mathsf{x}, Xp/(Xp'/Xp''), \Psi) \qquad (\textit{xpassoc})$$

$$\tau \oplus \xi \Longrightarrow \xi \qquad (\oplus\xi)$$

$$(\mathsf{x}, Xp, \{\xi\} \cup \Psi) \Longrightarrow \xi \qquad (\Psi\xi)$$

$$(\mathsf{x}, Xp, \{\neg\xi\} \cup \Psi) \Longrightarrow \xi \qquad (\Psi\neg\xi)$$

Figure 6: Type equivalence rules, expressed as rewriting rules. We require the equivalence relation $\equiv$ to respect the above rules. In the rule (*join*), predicate $\overline{\xi}(\tau)$ holds iff τ does not syntactically contain ξ.

$$\frac{}{\Gamma \vdash \varnothing : \varnothing} \qquad \frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash \mathsf{x} : \tau} \qquad \frac{\Gamma \vdash \mathsf{x} : \tau}{\Gamma \vdash \mathsf{x}/Xp : \tau \circ Xp}$$

$$\xi \circ Xp = \xi \qquad \varnothing \circ Xp = \varnothing \qquad (\mathsf{x}, Xp_1, \Psi) \circ Xp_2 = (\mathsf{x}, Xp_1/Xp_2, \Psi)$$

$$(\tau \oplus \tau') \circ Xp = (\tau \circ Xp) \oplus (\tau' \circ Xp)$$

Figure 7: Expression type system.

The typing rule evaluates the then and else branches of an if statement independently. The **merge** function is used to unify the environments obtained in the two branches. Its definition depends on that of the type constructor, $\tau[\psi]$. For a type τ and

$$\text{ASSIGN}\ \frac{\Gamma \vdash \mathit{Expr} : \tau}{\Gamma\{x = \mathit{Expr}\}\ \Gamma[x \mapsto \tau]} \qquad \text{ACCUM}\ \frac{\Gamma \vdash \mathit{Expr} : \tau \qquad \Gamma \vdash x : \tau'}{\Gamma\{x \Leftarrow \mathit{Expr}\}\ \Gamma[x \mapsto \tau' \oplus \tau]}$$

$$\text{SEQ}\ \frac{\Gamma\ \{S_1\}\ \Gamma' \qquad \Gamma'\ \{S_2\}\ \Gamma''}{\Gamma\ \{S_1; S_2\}\ \Gamma''} \qquad \text{IF}\ \frac{\Gamma \vdash \mathit{Expr} : \tau \qquad \Gamma\ \{S_1\}\ \Gamma' \qquad \Gamma\ \{S_2\}\Gamma'' \qquad \Gamma_f = \textbf{merge}(\Gamma', \Gamma'', \tau)}{\Gamma\ \{\mathsf{if}\ \mathit{Expr}\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\}\ \Gamma_f}$$

$$\text{SKIP}\ \frac{}{\Gamma\ \{\mathsf{skip}\}\ \Gamma}$$

Figure 8: Type system for programs without loops.

ψ, where ψ is of the form τ' or $\neg\tau'$, we define $\tau[\psi]$ as follows:

$$\tau[\psi] = \begin{cases} \xi & \tau = \xi \vee \tau' = \xi \\ \varnothing & \tau = \varnothing \wedge \tau' \neq \xi \\ \tau_1[\psi] \oplus \tau_2[\psi] & \tau = \tau_1 \oplus \tau_2 \wedge \tau' \neq \xi \\ (\mathsf{d}, Xp, \Psi \cup \{\psi\}) & \tau = (\mathsf{d}, Xp, \Psi) \wedge \tau' \neq \xi \end{cases}$$

Definition 2. The **merge**$(\Gamma', \Gamma'', \tau)$ function yields a new environment Γ_f such that:

$$\textbf{merge}(\Gamma', \Gamma'', \tau)(x) = \begin{cases} \Gamma'(\mathsf{x}) & \Gamma'(\mathsf{x}) \equiv \Gamma''(\mathsf{x}) \\ \Gamma'(\mathsf{x})[\tau] \oplus \Gamma''(\mathsf{x})[\neg\tau] & \text{otherwise} \end{cases}$$

In short, the **merge** function encodes the control dependency in the type of a variable to ensure greater precision. In our example, the resulting type for y would be $(\mathsf{d}, Xp_1/Xp_2, \varnothing)$ in Γ', and $\varnothing$ in Γ''. The **merge** function would generate the type $(\mathsf{d}, Xp_1/Xp_2, \{(\mathsf{d}, Xp_1, \varnothing)\}) \oplus \varnothing$, which can be simplified to $(\mathsf{d}, Xp_1/Xp_2, \varnothing)$, which is equivalent to $\Gamma_f(\mathsf{c}) \circ Xp_2$ (where Γ_f is again the environment determined by **merge**).

4.2. Handling Foreach *Loops*

We now consider the one missing part of our earlier specification, namely foreach loops of the form

```
foreach i in PATH { BODY }
```

Our goal is to determine the types of variables assigned to or accumulated on within the loop body. In some cases, we can determine the types simply by observing the kinds of operations performed; we consider these cases first. For example,

```
foreach i in d/Xp {
        a = i;
}
```

Here, our ASSIGN inference rule specifies that the type of a should be whatever the type of i is. While this is appropriate locally, it is inadequate after the loop: a should have the type of the last element in d/Xp. Since we have defined the traversal order of foreach loops as nondeterministic, we cannot express this concept. Consequently, we must assign a the type ξ; this generalizes to all assignments that involve the loop variable on the right-hand side.

Now, consider assignments that do not involve the loop variable:

```
foreach i in d/Xp {
        a = y;
}
```

According to ASSIGN, $\mathsf{a} : (\mathsf{y}, \epsilon, \varnothing)$. However, this type judgment is incorrect outside of the loop: if $[\![\mathsf{d}/Xp]\!] = \varnothing$, then the loop body will never execute — and therefore a would remain unmodified. Thus, we should give the same type we would give in the program

```
if (d/Xp)
        a = y;
else skip;
```

Of the remaining block constructs, both sequencing and conditionals turn out to be innocuous. This leaves us with accumulation. First consider accumulation that does not involve the loop variable:

```
foreach i in d/Xp {
        a ⇐ y;
}
```

This example has the same problems as assignment, in that it may be conditional, and we can handle it equivalently.

Finally, consider accumulation with a loop variable:

```
foreach i in d/Xp {
        a ⇐ i;
}
```

In this case, the a accumulates all elements selected by d/Xp. If $\mathsf{a} : \varnothing$ held before the loop, we therefore expect $\mathsf{a} : (\mathsf{d}, Xp, \varnothing)$ after the loop, otherwise a union type involving whichever type previously populated a.

We have now found type assignments for all interesting scenarios. We now describe how to detect these scenarios and integrate them with our existing typing rules (Fig.8).

For the existing rules to serve as the basis for detection, we assign loop variables to a distinguished type. We assign $\mathsf{i} : (\mathsf{i}, \epsilon, \varnothing)$, allowing us to distinguish types that involve the loop variable i from types that do not. Distinguishing accumulation of loop variables from assignments of loop variables is more difficult. Consider the loop body in the following program:

```
foreach i in d/Xp {
        a ⇐ i;
        b = i;
}
```

Considering only the loop body, we arrive at the distinct typing judgments

$$\begin{array}{lcl} a & : & \varnothing \oplus (\mathsf{i}, \epsilon, \varnothing) \\ b & : & (\mathsf{i}, \epsilon, \varnothing) \end{array}$$

However, union types can arise in other ways also, see the "IF" typing rule. There is one crucial difference between assignments in conditionals and aggregation that we can exploit, though: conditional assignment *sets* types, while aggregation *modifies* types, and (in particular) retains its previous type as part of the resulting union.

In practice, it can be hard to track exactly what this previous type is. Therefore instead of typing the loop body with the concrete environment preceding it, we type it starting with a specially tagged environment that exposes accumulation, and later graft this "special" environment onto the previous environment.

More concretely, we begin with an environment

$$\Gamma_0 = \{j \mapsto (j, \epsilon, \varnothing)\}$$

that maps not only document variables, but also *Index* and *Id* variables to a specially tagged type. We then compute an environment Γ_s via our existing type inference rules, as in

$$\Gamma_0\{S\}\Gamma_s$$

Assume a fixed but arbitrary foreach loop with index variable i. Γ_s then defines the effect of the loop, as follows:

- $\Gamma_s \vdash x : (x, \epsilon, \varnothing)$: means that x was not modified in the loop (or, if it was modified, it was set back to its original value). We can thus leave the type of x alone.
- $\Gamma_s \vdash x : (x, \epsilon, \varnothing) \oplus (i, Xp, \Psi)$: means that x has accumulated i. First, x references the loop variable — this means that x absorbs all elements of the iteration space (modulo Xp and Ψ). Secondly, x references itself, unmodified — this means that, during each iteration, x preserved whatever contents it held in the previous iteration (and before the loop). Correspondingly, we replace it by the iteration space of the loop, processed and filtered as per Xp and Ψ.
- While there are other cases we can handle effectively, we can always default to ξ if we are not sure of the outcome.

This approach works well in most instances, but it leaves open how we should deal with occurrences of i in Ψ. While we can always map such variables to ξ, we can do better. First consider how such types arise:

```
foreach i in d/A {
        if i/↓/B
                x ⇐ i;
        else skip;
}
```

Here, we arrive at the type

$$\Gamma_s \vdash \{\ \mathsf{x}\ : (\mathsf{x}, \epsilon, \{(\mathsf{i}, \downarrow /B, \varnothing)\}) \oplus (\mathsf{i}, \epsilon, \{(\mathsf{i}, \downarrow /B, \varnothing)\}) \oplus (\mathsf{x}, \epsilon, \{\neg(\mathsf{i}, \downarrow /B, \varnothing)\})$$

The type of x is somewhat involved. To understand it properly, we first note three equivalences (in the form of rewriting rules) that allow us to simplify the type:

$$\tau_1 \oplus \tau_2 \iff \tau_2 \oplus \tau_1 \qquad (\textit{comm})$$
$$(\tau_1 \oplus \tau_2) \oplus \tau_3 \iff \tau_1 \oplus (\tau_2 \oplus \tau_3) \qquad (\textit{assoc})$$
$$(x, Xp, \{\tau\} \cup \Psi) \oplus (x, Xp, \{\neg\tau\} \cup \Psi) \iff (x, Xp, \Psi)(\textit{if}\ \overline{\xi}(\tau)) \qquad (\textit{join})$$

where $\overline{\xi}(\tau)$ iff τ does not syntactically contain a ξ. All rewriting rules employed in this section are summarized in Figure 6.

Using the above three rewriting rules, we arrive at the semantically equivalent type

$$\Gamma_s \vdash \mathsf{x} : (\mathsf{x}, \epsilon, \varnothing) \oplus (\mathsf{i}, \epsilon, \{(\mathsf{i}, \downarrow /B, \varnothing)\})$$

Were we to translate this type with the same scheme as above, we would interpret our result as mapping $\mathsf{x} : \tau$ to $\tau \oplus (\mathsf{d}, A, \{(\mathsf{d}, A/\ \downarrow\ /B,)\})$. This type means "if there is any node in $\mathsf{d}/A/\ \downarrow\ /B$, then add d/A to τ, otherwise leave τ alone". But if we examine the loop that defined x, we notice that its semantics are "for each node in d/A, if this node has a child with B, add it" — which we can also express as $\mathsf{d}/A[\downarrow /B]$.

Therefore, our approach gave the wrong result. The reason for this is that each individual instance of adding i coincided with a condition only on that particular i, instead of on the entire iteration space. In our solution below, we address this by rewriting dependences on i in Ψ to conditions on the XPath fragment, in a function **flatten**.

$$\textbf{flatten}_{\mathsf{i}}((\mathsf{i}, A, \{(\mathsf{i}, B, \{(\mathsf{x}, C, \varnothing)\})\})) \quad = \quad (\mathsf{i}, A, \{(\mathsf{i}, B, \varnothing), (\mathsf{x}, C, \varnothing)\})$$

Before this step, we simplify our types. Note that conditions may be nested arbitrarily deeply inside each other, as in

$$(\mathsf{c}, \epsilon, \{(\mathsf{d}, \epsilon, \{\neg(\mathsf{e}, \epsilon, \ldots)\})\})$$

It can be hard to see what the semantics of such types are. Below we show how we can simplify types to eliminate deep nesting.

Definition 3. A type τ is *predicate-free* iff $\tau \in \{\xi, \varnothing\}$ or $\tau = \tau_1 \oplus \tau_2$ and τ_1, τ_2 are both predicate-free, or if $\tau = (x, Xp, \varnothing)$.

Conversely, a type is *predicated* if it is not *predicate-free*.

We have defined the absence of predication as a syntactic property, though there are some types that are inherently predicated (i.e., cannot be rewritten into predicate-free types), such as $(\mathsf{c}, Xp, \{(\mathsf{d}, Xp', \varnothing)\})$. This type depends on two different root nodes, so we cannot simplify it. However, we can reduce them to a single level of nesting, yielding *predicate-normal* types:

Definition 4. A type τ is *predicate-normal* iff $\tau \in \{\xi, \varnothing\}$, $\tau = \tau_1 \oplus \tau_2$ and both τ_1 and τ_2 are predicate-normal, or if $\tau = (x, Xp, \Psi)$, and for all $\tau', \neg\tau'$ in Ψ, τ' is predicate-free.

Lemma 1. *For each type τ, there exists a semantically equivalent type τ' such that τ' is predicate-normal.*

Proof. We can rewrite each type that is not predicate-normal to a type that is predicate-normal by repeatedly applying the following semantics-preserving rewriting rules:

$$(x, Xp, \{\tau_1 \oplus \tau_2\} \cup \Psi) \Longrightarrow (x, Xp, \{\tau_1, \tau_2\} \cup \Psi) \quad (\textit{flat}\text{-}0)$$
$$(x, Xp, \{\neg(\tau_1 \oplus \tau_2)\} \cup \Psi) \Longrightarrow (x, Xp, \{\neg\tau_1, \neg\tau_2\} \cup \Psi) \quad (\textit{flat}\text{-}0')$$
$$(x, Xp, \{(y, Xp', \Psi')\} \cup \Psi) \Longrightarrow (x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \Psi') \quad (\textit{flat}\text{-}1)$$
$$(x, Xp, \{\neg(y, Xp', \Psi')\} \cup \Psi) \Longrightarrow (x, Xp, \{\neg(y, Xp', \varnothing)\} \cup \Psi \cup (\neg\Psi')) \quad (\textit{flat}\text{-}1')$$

where $\neg\{\tau_1, \ldots, \tau_n\} = \{\neg\tau_1, \ldots, \neg\tau_n\}$ and $\neg\neg\tau = \tau$. We defer the proof of correctness for the above rules to Theorem 3. □

With our notion of predicate-normal types, we define a helper function:

Definition 5. Let τ be a type. Then $\mathbf{pn}(\tau)$ is a predicate-normal type such that $\mathbf{pn}(\tau) \equiv \tau$.

As we know from Lemma 1, **pn** is total. However, there may be many such functions; without loss of generality, we can pick any one of them.

We use the notion of predicate-normal types to simplify the definition of **flatten**:

Definition 6.

$$\mathbf{fold}_{\mathsf{i}}(Xp, \Psi) = \begin{cases} \mathbf{fold}_{\mathsf{i}}(\epsilon[Xp'/Xp, \Psi') & \Psi = \{(\mathsf{i}, Xp', \varnothing)\} \cup \Psi' \\ \mathbf{fold}_{\mathsf{i}}(\epsilon[\neg Xp'/Xp, \Psi') & \Psi = \{\neg(\mathsf{i}, Xp', \varnothing)\} \cup \Psi' \\ (\mathsf{i}, Xp, \Psi) & \text{otherwise} \end{cases}$$

$$\mathbf{flatten}_{\mathsf{i}}(\tau) = \begin{cases} \mathbf{flatten}_{\mathsf{i}}(\tau_1) \oplus \mathbf{flatten}_{\mathsf{i}}(\tau_2) & \tau' = \tau_1 \oplus \tau_2 \\ \mathbf{fold}_{\mathsf{i}}(Xp, \Psi) & \tau' = (\mathsf{i}, Xp, \Psi) \\ \xi & \tau' = (x, Xp, \Psi), x \neq \mathsf{i} \\ & \quad \text{and} \begin{pmatrix} \Psi = \{(\mathsf{i}, Xp', \varnothing)\} \cup \Psi' \\ \text{or} \quad \Psi = \{\neg(\mathsf{i}, Xp', \varnothing)\} \cup \Psi' \end{pmatrix} \\ \tau & \text{otherwise} \end{cases}$$

where $\tau' = \mathbf{pn}(\tau)$.

For example, consider the program fragment

```
x = ∅
foreach i in d/X {
        if (i/B) {
                if (k/C) {
                        x ⇐ i/A;
}}}
```

Here, we would infer $\mathsf{x} : (\mathsf{i}, A, \{(\mathsf{i}, B, \varnothing), (\mathsf{k}, C, \varnothing)\})$ to indicate that x accumulates all the elements in d/X that have a B child, or nothing at all if k/C is empty. We flatten this as follows:

$$\begin{aligned}\mathbf{flatten}_{\mathsf{i}}((\mathsf{i}, A, \{(\mathsf{i}, B, \varnothing), (\mathsf{k}, C, \varnothing)\})) &= \mathbf{fold}_{\mathsf{i}}(A, \{(\mathsf{i}, B, \varnothing), (\mathsf{k}, C, \varnothing)\}) \\ &= \mathbf{fold}_{\mathsf{i}}(\epsilon[B]/A, \{(\mathsf{k}, C, \varnothing)\}) \\ &= (\mathsf{i}, \epsilon[B]/A, \{(\mathsf{k}, C, \varnothing)\})\end{aligned}$$

The resultant type expresses our intuitive earlier notion formally. This is not quite the type we ultimately want for x yet, though — to get that type, we need to 'promote' this type to

$$\mathsf{x} : (\mathsf{d}, X/\epsilon[B]/A, \{(\mathsf{k}, C, \varnothing)\}) \equiv (\mathsf{d}, X[B]/A, \{(\mathsf{k}, C, \varnothing)\})$$

We describe our notion of 'promotion' below, though we first need a helper function **rec** that distinguishes variables that are potential accumulators (i.e., that reference themselves in their result type) from others:

Definition 7.

$$\mathbf{rec}(x, \tau) = \begin{cases} \mathit{true} & \tau = (x, \epsilon, \varnothing) \\ \mathit{true} & \tau = \tau_1 \oplus \tau_2 \\ & \quad \text{and } \mathbf{rec}(x, \tau_i) = \mathit{true}, \text{ for some } i \in \{1, 2\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Now we are ready to formalize loop body treatment and 'promotion':

Definition 8. Let $\tau \equiv (c, Xp, \Psi)$. Then

$$\mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(x, r, \tau') = \begin{cases} \Gamma(x) & \tau' = (x, \epsilon, \varnothing) \\ (\tau \circ Xp')[\mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(\bot, \mathit{true}, \Psi')] & \tau' = (\mathsf{i}, Xp', \Psi') \\ & \quad \text{and } r = \mathit{true} \\ \varnothing & \tau' = \varnothing \\ \quad \mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(x, r, \tau_1) & \tau' = \tau_1 \oplus \tau_2 \\ \quad \oplus \mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(x, r, \tau_2) & \\ (\mathsf{d}, Xp', \mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(\bot, \mathit{true}, \Psi')) & \tau' = (\mathsf{d}, Xp', \Psi') \\ & \quad \text{and } \mathsf{d} \in \mathit{Doc} \\ \xi & \text{otherwise} \end{cases}$$

where $\bot$ is a fresh identifier (i.e., an identifier that is not part of the program), and $\tau[\{\psi_1, \psi_2, \ldots, \psi_n\}] = \tau[\psi_1][\psi_2, \ldots, \psi_n]$.

We extend the definition of **promote** to environments as follows:

$$\mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma') \vdash x : \tau'' \iff \Gamma' \vdash x : \tau' \text{ and } \mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(x, \mathbf{rec}(x, \tau')\tau') = \tau''$$

promote permits assignments within the loop body. Since these assignments are only executed if the loop body is executed at least once, we must treat the resulting environment as we would treat the "then" branch of a conditional. For this purpose, we recycle our function **merge** that we previously defined for handling conditionals. Putting everything together, we arrive at our typing rule:

$$\frac{\Gamma \vdash \mathsf{XP} : \tau \qquad \Gamma_0\{\mathsf{S}\}\Gamma_s \qquad \Gamma_f = \textbf{flatten}(\Gamma_s)}{\Gamma\ \{\ \mathsf{foreach\ i\ in\ XP\ \{\ S\ \}\ }\}\ \textbf{merge}(\textbf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma_f), \Gamma, \tau)}$$

To see that this rule is correct, consider the following examples:

Example 1 (Simple loop accumulation).

```
a = ∅;
foreach i in d/Y {
        a ⇐ i;
}
```

$$\Gamma_s \vdash \{\ \mathsf{a} \ \mapsto\ (\mathsf{a}, \epsilon, \varnothing) \oplus (\mathsf{i}, \epsilon, \varnothing)\}$$

flatten is a no-op here, so $\Gamma_f = \Gamma_s$. **promote** then maps a to the entire iteration space (in union with its previous contents). After **merge**, we arrive at the type

$$\mathsf{a} : (\mathsf{d}, Y, \{(\mathsf{d}, Y, \varnothing)\}) \oplus \varnothing$$

While this type is correct, it is unnecessarily complex. We therefore introduce two more rewriting rules based on semantic equivalence:

$$\begin{aligned} \tau \oplus \varnothing &\Longrightarrow \tau && (\textit{empty}) \\ (\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi)) &\Longrightarrow (\mathsf{x}, Xp, \Psi) && (\textit{selfdep}) \end{aligned}$$

We then find the desired type judgment

$$\mathsf{a} : (\mathsf{d}, Y, \varnothing)$$

Example 2 (Nested loop accumulation).

```
x = e/B;
foreach i in d/A {
        foreach j in i/↓/B {
                x ⇐ j;
        }
}
```

First consider the inner loop:

$$\Gamma_s = \{\ \mathsf{x} \ \mapsto\ (\mathsf{x}, \epsilon, \varnothing) \oplus (\mathsf{j}, \epsilon, \varnothing)\}$$

analogously to our previous example. This type we map to $\tau \oplus (\mathsf{i}, \downarrow /B, \varnothing)$, where $x : \tau$ held before the inner loop. To determine what this τ is, now consider the outer

loop: In the outer loop, we (at this point) are using Γ_0, and $\Gamma_0 \vdash x : (x, \epsilon, \varnothing)$. Thus, the outer loop yields

$$\Gamma_s = \{ \ \mathsf{x} \ \mapsto \ (\mathsf{x}, \epsilon, \varnothing) \oplus (\mathsf{i}, \downarrow / B, \varnothing)\}$$

flatten is again a no-op, and **promote** maps the type of x to

$$\mathsf{x} : \tau' \oplus (\mathsf{d}, A/ \downarrow / B, \varnothing)$$

where τ' is the type we have for x prior to the outer loop body. This type is $(\mathsf{e}, B, \varnothing)$, so the final type judgment is

$$\mathsf{x} : (\mathsf{e}, B,) \oplus (\mathsf{d}, A/ \downarrow / B, \varnothing)$$

as we expected.

4.3. Correctness

As our examples illustrate, our algorithm is precise in many interesting cases. It is also sound, as we show in Appendix 10.4. Our proof is mostly technical. We first show that our algorithm is sound in the absence of loops. We find it useful to introduce a relation $(\sqsubseteq)$ that expresses that a type approximates another– $\tau \sqsubseteq \tau'$ iff $[\![\tau]\!]_D = \xi$ or $[\![\tau]\!]_D = [\![\tau']\!]_D$. We extend this definition to environments.

Next, we introduce a notion of substitution via a function **subst** (Section 10.3), wherein we substitute environments into other environments with holes derived from 'abstract environments' Γ_0. These abstract environments are precisely the specially tagged environments we used in Section 4.2.

In Theorem 1 we then show the following: Let $\Gamma\{S\}\Gamma'$, and $\Gamma_0\{S\}\Gamma_1$. Then $\textbf{subst}(\Gamma, \Gamma_1) \sqsubseteq \Gamma'$, where $\textbf{subst}(\Gamma, \Gamma_1)$ substitutes mappings from Γ into all 'holes' in Γ left over from the initial abstract environment Γ_0.

This property is then fundamental in our proof for correctness in the presence of loops: we show that all types that our algorithm does not map to ξ are either preserved or accumulate inductively, resulting in Lemma 16. Theorem 2 then collects our results and proves correctness.

4.4. Algorithmic complexity

The algorithm for assigning types to variables according to the typing rules specified in Sections 4.1 and 4.2 is fairly straightforward. It starts with an empty environment, $\Gamma_\varnothing$, and applies the typing rules to each statement to produce the resulting environment. When a loop is encountered, the algorithm applies the rules to the abstract environment, and then promotes the results. Each statement is visited only once.

The complexity of the algorithm depends on efficient mechanisms for simplifying union types, for detecting the equivalence of types, and an efficient representation of sets.

Independently of those factors, the number of operations on types is linear in the size of the program. The size of the predicate set is also linear with respect to program size. Union types make the worst-case complexity exponential, since they can potentially double in length after each conditional. However, in practice the length has a

much lower bound, as shown in [7]. We can perform simplifications that in practice reduce the length of union types. Further, we can guarantee a linear bound on the length of union types by selecting a size limit and setting a type to ξ whenever that bound is reached.

There are several techniques for determining the equivalence of XPath expressions [12, 5]. Our analysis is orthogonal to the equivalence test used; an appropriate test could be chosen depending on the fragment of XPath supported. Some XPath equivalence tests are non-linear in the length of the types compared. In our algorithm, we use a straightforward technique based on matching the syntactic structure of types. Two types $(\mathsf{x}, Xp_1, \Psi_1)$ and $(\mathsf{x}, Xp_2, \Psi_2)$ are equivalent if Xp_1 is equivalent to Xp_2 and one can match each element in Ψ_1 with an element in Ψ_2. Xp_1 and Xp_2 are equivalent if the tree representations of Xp_1 and Xp_2 are identical modulo commutativity of predicates, that is, $\tau[\tau_1][\tau_2]$ is equivalent $\tau[\tau_2][\tau_1]$. While this syntactic matching is incomplete, it allows us in practice to detect equivalences in the presence of data value comparisons, count, and other functions that more complete techniques do not handle [5]. Our technique for determining XPath equivalence is linear in the length of the types compared.

5. Transformations

The analysis described in Section 4 computes a symbolic representation of all possible values assumed by each XML expression or variable in the program. This section describes how this symbolic representation is used to optimize programs. We describe three transformations enabled by our analysis. The first is *common subexpression elimination* [8], which replaces an XPath expression by a previously computed result. The second, *XPath extraction* allows for the treatment of loops as XPath expressions; while it is not an optimization in itself, it enables other optimizations. The third, *common traversal elimination* is an optimization across multiple queries; if two XPath evaluations traverse a common set of nodes (though they might return different results), the XPath engine could optimize the computation by evaluating both queries in parallel. We describe these transformations below. Subsection 5.4 discusses how the above transformations are applied in concert.

5.1. Common Subexpression Elimination (CSE)

Common subexpression elimination (CSE) replaces an XPath expression by a previously computed result. Common subexpression elimination has been extensively covered in the literature [8]. The symbolic representation resulting from our analysis provides a basis for applying traditional CSE algorithms to XPath expressions. For example, given a statement "y = x/XP", if the analysis were to discover that the type of some variable z after the statement is equivalent to that of y, then we could replace the statement with "y = z".

We now give a general description of the CSE transformation:

$$\begin{array}{l} \texttt{//}\ \exists z : z \equiv e \\ \texttt{y = } e \end{array} \quad \Longrightarrow \quad \begin{array}{l} \texttt{//}\ \exists z : z \equiv e \\ \texttt{y = z} \end{array}$$

For CSE our analysis must determine that $\Gamma(z)$ and $\Gamma(y)$ map to equivalent types, i.e., $\Gamma(z) = \tau$ and $\Gamma(y) = \tau'$ and $\tau \equiv \tau'$. z is an *available expression* at the point of the assignment to y.

Consider the following transformation example, which combines XPath extraction with CSE.

```
// ∃z : z ≡ e/Xp1                    // ∃z : z ≡ e/Xp1
foreach i in e {                     y = z;
    Stmt1;                    ⇒      foreach i in e {
    y ⇐ i/Xp1;                           Stmt1; // accumulate statement
    Stmt2;                               Stmt2; // removed from loop body
}                                    }
```

Before we can perform CSE in this example, we must extract an XPath expression out of the foreach loop. We describe this transformation in Section 5.2.

5.2. XPath Extraction

This transformation extracts XPath expressions out of loops that accumulate values. It consists of two steps: *loop splitting* and *XPath conversion*. If, using algorithms such as loop reordering analysis [13], we can detect that splitting a loop preserves semantics, then we can isolate accumulate operations by splitting the loop. The essence of the transformation can be described through the following example:

```
foreach i in x/XP {                  // Loop 1
    y ⇐ i/...;                       foreach i in x/XP {
    Stmt                       ⇒         y ⇐ i/...;
}                                    }
                                     foreach i in x/XP {
                                         Stmt // y ⇐ ... removed
                                     }
```

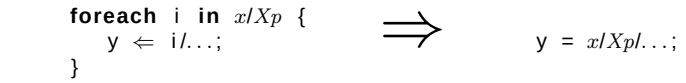
The XPath conversion step replaces loops of the form of Loop 1 in the previous example with the statement "y = x/XP/...". Such a transformation may enable further optimizations such as CSE and common traversal elimination.

In certain cases some of the code preceding the accumulate statement needs to be duplicated because *Stmt* includes a read of i. For example:

```
foreach i in x/XP1 {                 foreach i in x/XP1 {
    j = i/XP2;                           j = i/XP2;
    y ⇐ j/...;                           y ⇐ i/...
    Stmt(j);                     }
}                                    foreach i in x/XP1 {
                                         j = i/XP2;
                                         Stmt(j); // accumulate statement
                                                  // removed from loop body
                                     }
```

In the transformation example in Section 5.1, we need loop splitting prior to loop to XPath conversion. In the transformation example in Section 5.2, loop splitting is unnecessary.

We now give a general description of the XPath conversion transformation:



5.3. Common Traversal Elimination

Consider two XPath expressions over the same document whose evaluation would traverse the same set of nodes. The analysis results described in Section 4 implicitly encode the sets of nodes traversed by XPath evaluations. Common traversal elimination, or *tupling*, merges XPath expressions that traverse the same set of XML nodes. Intuitively, the tupling optimization represents simultaneous computation of multiple results over the same data set. For example, consider two XPath expressions a = x/↓/B/↓/C and b = x/↓/B/↓/D. The tupling transformation takes advantage of the fact that the evaluation of both XPath expressions would visit the B children of x and all the children of those nodes. Rather than evaluating the two XPath expressions separately, one could compute the two solutions in parallel. To support this optimization, we add a new operator "⊗" to our XPath syntax. In our XPath engine, the two XPath expressions would be represented as x/↓/B/↓/(C ⊗ D). The denotation of the ⊗ operator, $[\![\tau \otimes \tau']\!](N)$ is defined to be the tuple $\langle [\![\tau]\!](N), [\![\tau']\!](N)\rangle$. Consider a statement of the form y = x/XP_1/XP_2. If some variable z at that statement has type (x, $Xp_1/Xp_3, \Psi$), we identify the definition of z to see if the computation of z and y are amenable to common traversal elimination. The transformation detects whether the computation of y can be safely hoisted to the point where z is computed.

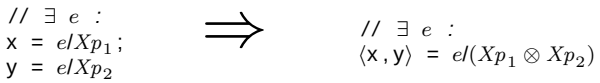
For example, consider the following instance of the transformation:

In this example, we first perform XPath extraction to move the assignment to y out of the loop. We can then tuple the computation of x and y. If $\Gamma(x) = (\mathsf{d}, Xp_1/Xp_1', \Psi_1)$ and $\Gamma(y) = (\mathsf{d}, Xp_2/Xp_2', \Psi_2)$, our implementation searches for an expression e, where e is a "common prefix" of x and y, i.e., for $\Gamma \vdash e : \tau'$, where

$$\tau' \equiv (\mathsf{d}, Xp_1, \Psi_1) \equiv (\mathsf{d}, Xp_2, \Psi_2)$$

The implicit encoding of traversals in the analysis results provides the information needed to find a common traversal for x and y. More elaborate matching is possible, but would require a more complex transformation than the tupling described above.

We now give a general description of the tupling transformation:



For tupling, our analysis must determine:

$$\Gamma(x) = \sigma_1$$

$$\Gamma(y) = \sigma_2$$

$\exists\, (\sigma_1', \sigma_2')$ such that $\sigma_1 \equiv \sigma_1'/Xp_1$, $\sigma_2 \equiv \sigma_2'/Xp_2$, $\sigma_1' \equiv \sigma_2'$, and $\exists\, e$: $\Gamma(e) = \sigma_1'$

The main challenge for our analysis then is to find a matching e, where e is a "common prefix" of x and y. The complexity of this task largely depends on the power of our equivalence relation $(\equiv)$.

5.4. Bringing it all together

Our general transformational approach is then the following:

Given the results of our analysis, we find all pairs of variables that are associated with the same symbolic value at a program point. For each such pair of variables, we perform the CSE transformation. If the value that is computed later is a result of loop accumulation, we precede CSE by XPath extraction.

After the CSE phase, we find all pairs of variables whose analysis results have a common prefix. For each such pair of variables, we perform the tupling transformation. If either value is a result of loop accumulation, we precede tupling by XPath extraction.

All transformations are subject to the constraints described in the preceding sections. If we cannot perform XPath extraction (due to dependencies, for example), we cannot perform the subsequent transformation (CSE or tupling) on that pair of variables.

6. Extensions for Generality

For simplicity, we have focused on a core fragment of an XML-based language. Here we discuss the extension of our analysis to the richer set of constructs available in an imperative language such as XJ. We have implemented the analysis in the context of a larger subset of XJ than the core language. Our implementation of the analysis handles more general control constructs, e.g., `switch` and `while`.

The interaction between XML values and non-XML values, in both our core language and XJ, occurs in a constrained manner, namely, references are allowed from objects of Java types to objects of XML types, but not in the other direction. For example, XML values can be stored in fields of Java objects. Thus, traditional alias/points-to analyses or value numbering algorithms could be applied to the non-XML (Java) subset of the imperative language prior to the execution of our analysis. Our analysis can then use the results of these pre-pass analyses as inputs. The implemented analysis integrates the computation of single-level Java reference aliasing.

To extend our analysis to handle updates to XML values, we have to detect all values that are or could be modified by each update statement. Implementing such a strategy requires two parts: (1) a general update mechanism that describes what it means for one type to be substituted by another, and (2) inference rules that tell us for each update primitive what type they should substitute by what other type.

First, consider the general update mechanism. For all assignments of one set of values of type τ_s to a storage location represented by a type τ_d, we must identify all type assignments $\mathsf{x} : \tau$ in the current environment and determine whether τ overlaps with τ_d (for this we can use existing algorithms [14]). If τ and τ_d do not overlap, the assignment does not affect x. If τ and τ_d partially overlap or could overlap, we set

$\mathsf{x} : \xi$. If τ and τ_d are equal and not ξ, we set $\mathsf{x} : \tau_s$. Note that if $\tau_d = \xi$, we must map the entire environment to ξ since any value may be affected.

Second, consider update primitives in general. Each update replaces one set of values (of type τ_s) by another (of type τ_d). Thus, all we need are inference rules for each update primitive that uses the above mechanism and construct τ_s and τ_d accordingly.

Supporting method calls would be relatively straightforward in the absence of imperative updates. A method may act as a query function and (in that case) will return the union of the types that all of its return statements return. For recursive calls, the matter is more complex and may require handling similar to loops — though we can always default to ξ. The only complication here arises when we accumulate on formal parameters, but we can initially approximate this by setting their types to ξ after the call.

Adding both methods and imperative updates to our core language has a more profound impact. Since we assume reference semantics, updates can affect the entire heap and must therefore be considered globally. We can accomplish this either by employing a context-sensitive analysis, or by computing a rewrite mapping for each function that directly or indirectly updates XML values. Each rewrite mapping then updates our environment at the corresponding method call, thereby requiring only one analysis pass per method (as for our loops). We can use analogous techniques to provide more precise typing judgments for formal parameters.

The type system we described is mostly orthogonal to the fragment of XPath used - the framework depends essentially on an efficient algorithm for detecting the equivalence of XPath expressions. Recently, Geneves *et al.* [5] have presented an engine that in practice can detect equivalences between XPath expressions efficiently. We could adapt our analysis to support a larger fragment by taking advantage of their equivalence checker. XML Schema information can be incorporated into our analysis by performing a preprocessing pass, where XPath expressions are rewritten using schema information. For example, $(\mathsf{a}, \downarrow^+ /A, \Psi)$ could be rewritten into $(\mathsf{a}, \downarrow /B/ \downarrow /A, \Psi)$ if appropriate schema information states that A elements only occur as children of B elements.

7. Experiments

The implementation of type assignment used for our experiments is based on a type assignment algorithm described in [2], which differs in some details from the type assignment algorithm described in Section 4.4. The algorithm described in this paper is more efficient than the one described in [2]. The algorithm in this paper is also more precise in terms of handling a number of cases, such as loop variable accumulation on nonempty initial variables, that were mapped to ξ or even undefined in [2]. However, the [2] algorithm does compute the correct type in the presence of a loop-carried dependence [19] on assignment to a variable, whereas the algorithm formulated in Section 4.4 does not. Detecting this case requires at least two passes over the loop body (since the type of that variable might be different on the second pass). The algorithm in this paper is a one-pass solution, and misses this case, but could readily be modified to handle it without an additional pass. The precision of the two algorithms is identical

XLinq	*34*	: Union two sets of nodes: books authored by Anders and/or Peter.
XLinq	*35*	: Intersect two sets of nodes: books that are common for both authors.
XLinq	*36*	: All nodes in first set except the nodes in the second set: books that are authored by one without other as co-author.
XLinq	*38*	: Check if two sets of nodes are equal: did the two authors co-author all of their the books?
XMark	*Q7*	: How many pieces of prose are in the auction database?
XMark	*Q20*	: Group customers by their income and output the cardinality of each group.
XMark	*Q3*	: Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.

Figure 9: Benchmark descriptions.

with respect to the benchmarks used in this section, and so their differences are not relevant to the measurements given here.

As discussed in Section 6, our implementation is a fuller subset of XJ than our core language, but this difference too is not relevant to our measurements.

We compare the execution times achieved by code emitted by our AXIL backend [15] with and without the transformations described in the paper. The benchmarks for our experiments are based on queries drawn from the XMark XML Benchmark project [16] and the XLinq [3, 11] 101 samples. We selected queries that have potential for significant redundancy in terms of the computation of XPath expressions or of the traversal of XPath nodes. The speedups obtained do not represent average expected speedups for the full XMark or XLinq benchmark suites. Further, they do not indicate expected speedups for a full application.

In addition to the type assignment algorithm, our compiler implements the tupling optimization from Section 5.

We provide the performance comparisons for the tupling optimization on XLinq34, XLinq35, XLinq36, XLinq38 (from the XLinq samples) and XMarkQ7 and XMarkQ20 (from the XMark benchmark suite) [2].

We ran our experiments on the data sets provided by the XMark benchmarks and the XLinq samples. We measured the execution times with and without the tupling optimization on an IBM Intellistation with 3.0 GHz processor and 3GB of memory, running the IBM J9 VM 1.5.0 on top of a GNU/Linux 2.6.15-28 system. We ran each query 10 times, discarding the first run and picking the median result for each query. Before measuring, we removed all text output from the benchmarked code. Our results are summarized in Table 1. The results of the tupling optimization are shown in the column "Tupling". For the queries testing tupling, the introduction of tupling produces an improvement of 20.5% to 47.5%.

[2] Our implementations of the benchmarks are available at
`http://www-plan.cs.colorado.edu/creichen/xj/`.

Table 1: Performance results, in microseconds, median out of 9 consecutive executions.

Benchmark	Unopt	Tupling
XLinq34	4181	2195 / 47.5%
XLinq35	3282	2609 / 20.5%
XLinq36	3794	2297 / 39.5%
XLinq38	2581	1898 / 26.5%
XMark7	16545	11749 / 29.0%
XMark20	1235	873 / 29.3%

We implemented the CSE optimizations by hand using our analysis results. We provide results for XMarkQ3 and XMarkQ20. We manually modified *XMark20.xj* into *XMark20opt.xj*, eliminating the redundant traversal in the same way the tupling optimization did, and applying manual CSE to an XPath expression. *XMark3opt.xj* is a manual modification of *XMark3.xj* that eliminates the redundant computation of two XPath expressions. The improvements on other applications that have the same pattern is similar. *XMark20opt.xj* achieves a 50.04% reduction in the runtime of *XMark20.xj* , while the tupling optimization alone achieves a 29.3% reduction. This difference is due to the hand-coding of XPath expression CSE in *Xmark20opt.xj*. *XMark3opt.xj* achieves an 9.4% reduction in runtime with respect to *XMark3.xj* by eliminating the redundant computation of two XPath expressions.

8. Related Work

The work closest to ours is that of XAct [9], which defines a static analysis for typechecking XML processing programs, where types are used to verify statically that constructed XML data satisfy a specified schema. Their analysis computes a summary graph for every XML variable and expression in the program. It is a *may* analysis, computing all XML templates that may occur in some program execution. Our analysis, in detecting equivalences of values of variables and expressions, is of necessity a *must* analysis, that is, we compute the values to which an XML variable must refer under all program executions. Secondly, in our language, as in XQuery, XJ, and the XPath 1.0 standard, and unlike XAct, XML values have identity. In other words, two nodes are not equivalent unless they refer to the *same* node in the same tree in memory. This distinction results in a different flavor to the analysis and the values computed.

The problem studied in this paper is similar to the inference of relational queries and optimizations from imperative programs. For example Lieuwen and Dewitt [10] analyze database programming languages to detect whether optimizations such as reordering loops can improve performance. Recently, Wiedermann and Cook studied the inference of queries in a language with orthogonal persistence [18]. The motivation in this paper is similar — understanding accesses to a different data model in the scope of an imperative language. We, however, focus on the XML data model, and the XPath querying language, with the incident challenges these bring.

Genevès *et al.* have developed a framework for analyzing XPath expressions (with or without schema information). They provide a uniform representation capable of

answering questions such as equivalence, containment, and satisfiability of XPath expressions. Our types fit well into their framework, and it would be interesting to use their engine as the underlying basis of our analysis.

The problem we study in this paper is closely related to that of value numbering [1, 8], which attempts to discovers those expressions that are Herbrand equivalent: *i.e.*, use the same operator applied to equivalent operands, where the operators are treated as uninterpreted functions. In our context, however, it is necessary to take advantage of known algorithms for detecting equivalences of XPath expressions, and not treat them as uninterpreted functions. Moreover, we wished to be able to deduce the values computed by loops in the same framework.

Steensgard [17] presents an interprocedural flow-insensitive points-to analysis for a small imperative pointer language, based on type inference methods. He uses types to model how storage is used in a program at runtime, where typing rules specify when a program is well-typed. In some sense, the problem addressed in this paper can be considered a points-to analysis problem. We wish to derive some notion of the relationships between nodes in a tree when the tree is accessed using complex "pointer" expressions such as XPath expressions.

9. Conclusions

In this paper, we have studied the analysis of embedded XPath queries in an imperative language. We have described a flow-sensitive type system that takes into account the equivalence properties of XPath expressions and that can detect when a loop produces values equivalent to XPath expressions. While we have motivated this analysis using the example of redundant computation removal, such an analysis is essential for many purposes — for example, if we can infer that the values computed by a loop are equivalent to an XPath expression, then, in certain circumstances we can replace a loop with a direct invocation to an XPath engine that could implement the query more efficiently (in a sense, performing strength reduction).

An interesting area of future work is to generalize the analysis to other imperative languages that support XML queries. Languages to consider would include imperative derivatives of XQuery, such as XQueryP [4]. One could also consider runtime APIs such as DOM, if the compiler detects invocations of XPath expressions on DOM objects as special operations.

Acknowledgements

We would like to thank Kris Rose, John Fields, and the anonymous DBPL 2007 and ISJ reviewers for their valuable insights and feedback.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[2] Michael G. Burke, Igor Peshansky, Mukund Raghavachari, and Christoph Reichenbach. Analysis of Imperative XML Programs. In *Database Programming Languages, 11th International Symposium, DBPL 2007*, September 2007.

[3] Charlie Calvert. Linq samples update. `http://blogs.msdn.com/charlie/ archive/2007/03/04/samples-update.aspx`, 2007.

[4] Don Chamberlin, Michael Carey, Daniela Florescu, Donald Kossman, and Jonathan Robie. XQueryP: Programming with XQuery. In *XIME-P*, 20606.

[5] Pierre Genevès, Nabil Layaida, and Alan Schmitt. Efficient static analysis of XML paths and types. In *Conference on Programming Language Design and Implementation*, June 2007.

[6] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proceedings of World Wide Web (WWW)*, pages 278–287, May 2005.

[7] P. C. Kanellakis and J. C. Mitchell. Polymorphic unification and ml typing. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–115, New York, NY, USA, 1989. ACM.

[8] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Symposium on Principles of Programming Languages*, pages 194–206, 1973.

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

[10] Daniel F. Lieuwen and David J. DeWitt. Optimizing loops in database programming languages. In *DBPL*, pages 287–305, 1991.

[11] Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored (The Return of the Monoids). In *XML 2005 Proceedings*, 2005.

[12] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[13] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[14] Mukund Raghavachari and Oded Shmueli. Conflicting XML updates. In *Proceedings of the 10th International Conference on Extending Database Technology*, volume 3896 of *LNCS*. Springer-Verlag, March 2006.

[15] Christoph Reichenbach, Michael Burke, Igor Peshansky, Mukund Raghavachari, and Rajesh Bordawekar. AXIL: An XPath Intermediate Language. IBM Research Report RC24075, 2006.

[16] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 974–985, 2002.

[17] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[18] Benjamin A. Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *Proceedings of the 34th Symposium on Principles of Programming Languages*, January 2007.

[19] Michael Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley Publishing Co., 1996.

[20] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, 1999.

[21] World Wide Web Consortium. *Document Object Model Level 2 Core*, 2000.

10. Appendix

We now show that our type analysis results (Figure 8 and foreach rule in Section 4.2) match the results of our evaluation rules (Figure 4). For our proof, we are forced to constrain our equivalence relation $(\equiv)$ to ensure that it behaves correctly (Definition 18) and does not artificially discriminate simple types when it can see the equivalence between more complex types of the same structure (Definition 17).

We separate our proof into two parts: First, we introduce basic properties for stores and environments (Section 10.1), then we use these definitions to show soundness in programs that do not contain loops (Section 10.2).

For the second part, we begin by introducing an apparatus to better deal with loop bodies. Specifically, we introduce a notion of *substitution*, implicit in our earlier description of **promote** and Γ_0, and show several properties of this notion, most prominently a substitution theorem (Section 10.3). With the substitution theorem, we then extend our earlier proof to include loops (Section 10.4).

10.1. Stores and Environments

Definition 9. A *store* σ is *initial* iff for all $x \notin \mathit{Doc}$, $\sigma(x) = \varnothing$. An *environment* Γ is initial iff for all $x \notin \mathit{Doc}$, $\Gamma \vdash x : \xi$ or $\Gamma \vdash x : \varnothing$.

Definition 10. A type τ syntactically contains a type τ' (a variable v) iff

1. $\tau = \tau'$ ($\tau = (v, Xp, \Psi)$ for some Xp, Ψ), or
2. $\tau = \tau_1 \oplus \tau_2$ and τ_1 or τ_2 syntactically contains τ' (or v), or

3. $\tau = (x, Xp, \Psi)$ and, for some τ'' or $\neg\tau''$ in Ψ, τ'' syntactically contains τ' (or v).

An environment Γ syntactically contains η iff, for some x, $\Gamma(x)$ syntactically contains η.

Definition 11. A type or environment is *loop-dependent* iff it syntactically contains a variable $i \in$ *Index*.

Lemma 2. *For all S with $\Gamma\{S\}\Gamma'$, Γ' is loop-dependent only if Γ is loop-dependent.*

Proof. Proof by structural induction on program structure. The only interesting case is that of loops, which both introduce and (via **promote**) eliminate *Index* variables (and, thereby, loop dependence). Syntactically, a foreach loop with index variable i introduces i. All types from the loop body are filtered using the **promote** function. Assume that τ may syntactically contain the loop variable i. We now inductively show that **promote**(τ) eliminates such loop variables.

The cases $\tau = \varnothing$ and $\tau = \xi$ are trivial. $\tau = \tau_1 \oplus \tau_2$ follows from the induction hypothesis. The only interesting case then is (v, Xp, Ψ). If $v \neq i$, the result is trivially not loop dependent (since **flatten** eliminates all i in Ψ). If $v = i$, then we eliminate loop dependence locally (and again have no i in Ψ).

We must further consider our set of rewriting rules (listed e.g. in Theorem 3), though it is easy to see that those never introduce any new loop dependences. □

We can now see the following:

Corollary 1. *Let Γ be initial. Then for all S with $\Gamma\{S\}\Gamma'$, Γ' is not loop dependent.*

Before proceeding to consistency, a notion we previously described in Section 3, we define a more basic notion:

Definition 12. A type τ *describes a value* $v \in \mathfrak{T}$, notation $v :: \tau$, iff $v = [\![\tau]\!]_D$ or $\tau \equiv \xi$.

We define consistency as pointwise description, mirroring our earlier definition from Section 3:

Definition 13. We say that a store σ and an environment Γ are *consistent* iff, for all $\mathsf{x} : \tau \in \Gamma$, $\sigma(x) :: \tau$.

10.2. Correctness without Loops

For exposition, we again begin with a discussion that only considers programs without loops. Section 10.4 then extends our discussion to prove that our treatment of loops is also correct.

Before we can move on to the correctness proof, we require a small set of auxiliary lemmata:

Lemma 3. *For all Xp, the function $[\![Xp]\!] : \mathcal{P}(\mathcal{N}) \to \mathcal{N}$ is (a) a homomorphism, i.e.,*

$$[\![Xp]\!](A) \cup [\![Xp]\!](B) = [\![Xp]\!](A \oplus B)$$

and (b) has a fixpoint at $\varnothing$, i.e.,

$$[\![Xp]\!](\varnothing) = \varnothing$$

Proof. Straightforward (see Figure 2). □

Lemma 4. *For all stores D the function $[\![-]\!]_D : \tau \to \mathcal{P}(\mathcal{N})$ is a homomorphism, i.e.,*

$$[\![\tau_1]\!]_D \cup [\![\tau_2]\!]_D = [\![\tau_1 \oplus \tau_2]\!]_D$$

Proof. The proof follows from the definition of $[\![-]\!]_D$. Note the special-case treatment of ξ, which yields ξ if either case is ξ. □

We now show soundness in loop-free programs, first by considering XPath evaluation (Lemma 5), then by considering arbitrary expressions (Lemma 6), and finally by considering all non-loop statements (Lemma 7).

Lemma 5 (XPath Selection Soundness)**.** *Let $N :: \tau$, and Xp one of our XPath-like expressions (Figure 2). Then $[\![Xp]\!](N) :: \tau \circ Xp$.*

Proof. We show the property by structural induction over the structure of τ. If $\tau = \xi$, the statement holds trivially. Otherwise, we have $N = [\![\tau]\!]_D$.

Empty type.. With $\tau = \varnothing$, $N = \varnothing$, so $[\![Xp]\!](N) = \varnothing = [\![\tau \circ Xp]\!]_D$.

Triple type.. Assume $\tau = (x, Xp', \Psi)$. If $\mathsf{satisfied}(\Psi)$ is not true, then by definition of $[\![-]\!]$ for triple types and our induction hypothesis, $N = \xi$ or $N = \varnothing$ and the property trivially holds. Otherwise, we know

$$[\![Xp']\!](\sigma(x)) = N$$

and can show $[\![\tau \circ Xp]\!]_D =$

$$\begin{aligned}[\![(x, Xp'/Xp, \Psi)]\!]_D &= [\![Xp'/Xp]\!](\sigma(x)) \\ &= [\![Xp]\!]([\![Xp']\!](\sigma(x))) \\ &= [\![Xp]\!](N)\end{aligned}$$

Union type.. $\tau = \tau_1 \oplus \tau_2$. Thus we know that

$$N = [\![\tau_1]\!]_D \cup [\![\tau_2]\!]_D$$

We can therefore find N_i ($i \in \{1, 2\}$) such that $N_i = [\![\tau_i]\!]_D$ and $N = N_1 \cup N_2$. By Lemma 3, it is now sufficient to show

$$[\![Xp]\!](N_i) = [\![\tau_i \circ Xp]\!]_D$$

but this we know from the induction hypothesis. □

Lemma 6 (Expression Typing Soundness)**.** *Let σ be consistent with Γ. Then for all expressions e, with $\langle e, \sigma \rangle \models N$ and $\Gamma \vdash e : \tau$ the property $N :: \tau$ holds.*

Proof. By induction.

Empty set.. $\langle \varnothing, \sigma \rangle \models \varnothing$, and $\Gamma \vdash \varnothing : \varnothing$, with $[\![\varnothing]\!]_D = \varnothing$.

Variable.. $\langle x, \sigma \rangle \models \sigma(x)$, and $\Gamma \vdash x : \tau$. Since σ, Γ are consistent, $\sigma(x) :: \tau$ by definition.

XPath selection.. $\langle x/Xp, \sigma \rangle \models [\![Xp]\!](\sigma(x))$ where $\langle x, \sigma \rangle \models N$. In the type system, $\Gamma \vdash x/Xp : \tau \circ Xp$, where $\Gamma \vdash x : \tau$.

First observe that $N :: \tau$. Then $[\![Xp]\!](N) :: \tau \circ Xp$ follows directly from Lemma 5. □

Lemma 7 (Partial Statement Typing Soundness)**.** *Let S be a program such that S is loop-free (i.e., S does not contain the keyword* foreach*), and let σ be a store and Γ an environment such that σ and Γ are consistent. Now let $\langle S, \sigma \rangle \Downarrow \sigma'$ and $\Gamma\{S\}\Gamma'$. Then σ' and Γ' are also consistent.*

Proof. By structural induction. First observe that we explicitly exclude the case of loops. This leaves us with five cases to consider:

Skip.. $\langle \mathsf{skip}, \sigma \rangle \Downarrow \sigma$ and $\Gamma\{\mathsf{skip}\}\Gamma$, which trivially preserves consistency.

Composition.. $\langle S;S', \sigma \rangle \Downarrow \sigma''$, where $\langle S, \sigma \rangle \Downarrow \sigma'$ and $\langle S', \sigma' \rangle \Downarrow \sigma''$. Similarly, $\Gamma\{S;S'\}\Gamma''$ where $\Gamma\{S\}\Gamma'$ and $\Gamma'\{S'\}\Gamma''$. Using our induction hypothesis, σ' is consistent with Γ', and therefore σ'' with Γ''.

Assignments.. $\langle \mathsf{x} = \mathit{Expr}, \sigma \rangle \Downarrow \sigma[x \mapsto N]$ where $\langle \mathit{Expr}, \sigma \rangle \models N$. In the type system, $\Gamma\{\mathsf{x} = \mathit{Expr}\}\Gamma[x \mapsto \tau]$, where $\Gamma \vdash \mathit{Expr} : \tau$. By Lemma 6, $N :: \tau$.

Accumulation.. $\langle \mathsf{x} \Leftarrow \mathit{Expr}, \sigma \rangle \Downarrow \sigma[x \mapsto \sigma(x) \cup N]$ where $\langle \mathit{Expr}, \sigma \rangle \models N$. In the type system, $\Gamma\{\mathsf{x} \Leftarrow \mathit{Expr}\}\Gamma[x \mapsto \tau' \oplus \tau]$, where $\Gamma \vdash \mathit{Expr} : \tau$ and $\Gamma \vdash x : \tau'$. Again we know that $N :: \tau$ from Lemma 6, and we know $\sigma(x) :: \tau'$ by the assumption that σ and Γ are consistent. With Lemma 4, we then know that $\sigma(x) \cup N :: \tau' \oplus \tau$.

Conditionals.. The evaluation rules for conditional statements handle two cases. We first consider the "else" branch case: if $\langle S_2, \sigma \rangle \Downarrow \sigma''$ and $\langle \mathit{Expr}, \sigma \rangle \models \varnothing$, then

$$\langle \mathsf{if}\ (\mathit{Expr})\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, \sigma \rangle \Downarrow \sigma''$$

Meanwhile, our typing rules specify, with $\Gamma \vdash \mathit{Expr} : \tau$, $\Gamma\{S_1\}\Gamma'$ and $\Gamma\{S_2\}\Gamma''$,

$$\Gamma\{\mathsf{if}\ (\mathit{Expr})\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\}(\mathbf{merge}(\Gamma', \Gamma'', \tau))$$

By Lemma 6, we know that $\varnothing :: \tau$. By our induction assumption, we further know that σ'' and Γ'' are consistent. We must now show that $\mathbf{merge}(\Gamma', \Gamma'', \tau) = \Gamma_f$ is consistent with σ''.

For $\tau = \xi$, this is trivial, since Γ_f is pointwise either equal to Γ' or ξ.

For $\tau = \varnothing$, first note that **merge** preserves entries in Γ' and Γ'' if they are equivalent (thereby preserving their semantics). For any x on which Γ' and Γ'' diverge, we construct the type

$$\tau_x = \Gamma'(x)[\tau] \oplus \Gamma''(x)[\neg\tau]$$

We show that $\sigma(x) :: \tau_x$ as follows: First, note that $[\![\Gamma'(x)[\tau]]\!]_D \in \{\xi, \varnothing\}$, since we know that $\varnothing :: \tau$. If $[\![\Gamma'(x)[\tau]]\!]_D = \xi$, then $[\![\tau_x]\!]_D = \xi$ (due to the semantics of $(\oplus)$ on ξ). Otherwise

$$\begin{aligned}[\![\tau_x]\!]_D &= [\![\Gamma'(x)[\tau]]\!]_D \cup [\![\Gamma''(x)[\neg\tau]]\!]_D \\ &= \varnothing \cup [\![\Gamma''(x)[\neg\tau]]\!]_D\end{aligned}$$

All that remains now is to show $[\![\Gamma''(x)]\!] = [\![\Gamma''(x)[\neg\tau]]\!]$, with $\tau \neq \xi$. This we show by structural induction over $\Gamma''(x)$:

- If $\Gamma''(x) = \xi$, then $\Gamma''(x)[\neg\tau] = \xi$
- If $\Gamma''(x) = \varnothing$, then $\Gamma''(x)[\neg\tau] = \varnothing$
- If $\Gamma''(x) = \tau_1 \oplus \tau_2$, we have identical semantics by the induction hypothesis and by Lemma 4.
- If $\Gamma''(x) = (x, Xp, \Psi)$, consider

  $$[\![\Gamma''(x)[\neg\tau]]\!]_D = [\![(x, Xp, \Psi \cup \{\neg\tau\})]\!]_D = [\![(x, Xp, \Psi)]\!]_D$$

  by the definition of **satisfied**, since $[\![\neg\tau]\!] \neq \varnothing$.

The "then" branch case (where $\tau \notin \{\varnothing, \xi\}$) is analogous. □

10.3. Abstract Environments and Substitution

Definition 14. An environment Γ is *abstract* iff it syntactically contains a type (x, Xp, Ψ) with x in *Index* $\cup$ *Id*. Whenever Γ is not abstract, then Γ is *basic*.

Corollary 2. *Any initial environment is basic.*

Next we define a special null environment Γ_0 whose purpose it is to help us in handling loops. We do not use this environment outside of loop analysis.

Definition 15. The abstract *null-environment* Γ_0 is

$$\Gamma_0 = \{j \mapsto (j, \epsilon, \varnothing) | j \in \textit{Index} \cup \textit{Id}\}$$

For abstract environments, we describe a notion of substitution:

Definition 16. We define the substitution of an environment Γ into a type τ, denoted $\textbf{subst}(\Gamma, \tau)$, as follows:

$$\textbf{subst}(\Gamma, \tau) = \begin{cases} \textbf{substv}(Xp, \textbf{subst}(\Gamma, \Psi), \Gamma(x)) & \tau = (x, Xp, \Psi), \text{ if } x \in \textit{Id} \cup \textit{Index} \\ (d, Xp, \textbf{subst}(\Gamma, \Psi)) & \tau = (d, Xp, \Psi), \text{ if } d \in \textit{Doc} \\ \textbf{subst}(\Gamma, \tau_1) \oplus \textbf{subst}(\Gamma, \tau_2) & \tau = \tau_1 \oplus \tau_2 \\ \varnothing & \tau = \varnothing \\ \xi & \tau = \xi \end{cases}$$

with **subst** suitably extended on sets of types that may be prefixed by $\neg$, and

$$\textbf{substv}(Xp, \Psi, \tau) = \begin{cases} (y, Xp'/Xp, \Psi' \cup \Psi) & \tau = (y, Xp', \Psi') \\ \quad \textbf{substv}(Xp, \Psi, \tau_1) & \\ \quad \oplus \textbf{substv}(Xp, \Psi, \tau_2) & \tau = \tau_1 \oplus \tau_2 \\ \varnothing & \tau = \varnothing \\ \xi & \tau = \xi \end{cases}$$

We further extend **subst** to operate on environments, pointwise.

We eliminate meaningless syntactic differences introduced by the above by introducing the rewriting rules

$$(\mathsf{x}, \epsilon/Xp, \Psi) \Longrightarrow (\mathsf{x}, Xp, \Psi) \qquad (xp\epsilon)$$
$$(\mathsf{x}, (Xp/Xp')/Xp'', \Psi) \Longleftrightarrow (\mathsf{x}, Xp/(Xp'/Xp''), \Psi) \qquad (xpassoc)$$

To show the substitution theorem, we must make an assumption about our equivalence operator $(\equiv)$. The assumption is that the relation is *non-discriminating* and *correct*, as defined below:

Definition 17. An equivalence relation $(\equiv)$ on types is *non-discriminating* iff

$$\tau \equiv \tau' \implies \textbf{subst}(\Gamma, \tau) \equiv \textbf{subst}(\Gamma, \tau')$$

for any Γ.

This property is a relatively weak restriction and depends on our notion of substitution, which is why we could not define it earlier. In practice, it forbids non-equality on semantically irrelevant properties such as the presence of certain variable names or the size of a type.

Definition 18. An equivalence relation $(\equiv)$ on types is *correct* iff

$$\tau \equiv \tau' \implies [\![\tau]\!]_D = [\![\tau']\!]_D$$

Our first goal is the *substitution theorem*, which states that type inference on statements starting with Γ, and type inference starting with Γ_0 and followed up by substituting Γ via **subst** , give similar results in the sense that the latter is a conservative approximation of the former. We define

Definition 19. A type τ *approximates* a type τ', notation $\tau \sqsubseteq \tau'$, iff $[\![\tau]\!]_D = \xi$ or $[\![\tau]\!]_D = [\![\tau']\!]_D$. Similarly, an environment Γ approximates an environment Γ' ($\Gamma \sqsubseteq \Gamma'$) iff for each x, $\Gamma(x) \sqsubseteq \Gamma'(x)$.

Furthermore,

$$\tau_1 \sqsubseteq_\Gamma \tau' \iff \textbf{subst}(\Gamma, \tau_1) \sqsubseteq \tau'$$
$$\Gamma_1 \sqsubseteq_\Gamma \Gamma' \iff \textbf{subst}(\Gamma, \Gamma_1) \sqsubseteq \Gamma'$$

Observe that $(\sqsubseteq)$ is transitive.

Lemma 8. *Let* $\Gamma \vdash x : \tau$ *and* $\Gamma_0 \vdash x : \tau_0$. *Then* $\mathbf{subst}(\Gamma, \tau_0) \equiv \tau$.

Proof. $\Gamma_0 \vdash x : (\mathsf{x}, \epsilon, \varnothing)$ by definition. Assume $\Gamma(x) = (\mathsf{v}, Xp, \Psi)$ (the other cases follow directly from the definition of **subst**). The case of $\mathsf{v} \in$ *Doc* is straightforward. Otherwise:

$$\begin{aligned}
\mathbf{subst}(\Gamma, (\mathsf{x}, \epsilon, \varnothing)) &= \mathbf{substv}(\epsilon, \varnothing, \tau) \\
&= \mathbf{substv}(\epsilon, \varnothing, (\mathsf{v}, Xp, \Psi)) \\
&= (\mathsf{v}, \epsilon/Xp, \Psi) && (\textit{epsilon}) \\
&\equiv (\mathsf{v}, Xp, \Psi) \\
&= \tau
\end{aligned}$$

□

Lemma 9 (Path Appendage Substitution)**.** *Let* $\tau \sqsubseteq_\Gamma \tau'$. *Then, for any* Xp, $\tau \circ Xp \sqsubseteq_\Gamma \tau' \circ Xp$.

Proof. We use structural induction over τ'.

empty set.. Let $\tau' = \varnothing$. By definition, $\mathbf{subst}(\Gamma, \varnothing) = \varnothing$, and $\varnothing \circ Xp = \varnothing$ both for τ and τ'. The same result applies to ξ.

union type.. Let $\tau' = \tau_1 \oplus \tau_2$. By translation, $\tau = \mathbf{subst}(\Gamma, \tau_1) \oplus \mathbf{subst}(\Gamma, \tau_2)$, which follows from the induction hypothesis.

triple type.. Let $\tau' = (\mathsf{y}, Xp', \Psi)$. The case of $\mathsf{y} \in$ *Doc* is trivial. Otherwise we must show from $\mathbf{subst}(\Gamma, (\mathsf{y}, Xp', \Psi)) = \tau''$ that $\mathbf{subst}(\Gamma, (\mathsf{y}, Xp'/Xp, \Psi)) = \tau'' \circ Xp$. This requires further induction, this time over $\Gamma(\mathsf{y})$. If $\Gamma(\mathsf{y}) = \varnothing$ or $\Gamma(\mathsf{y}) = \xi$, we are done. Union types follow from the induction hypothesis and the definition of **substv**. Otherwise, $\Gamma(\mathsf{y}) = (\mathsf{d}, Xp'', \Psi')$ for some d, Xp'', Ψ'. Therefore

$$\begin{aligned}
\mathbf{subst}(\Gamma, (\mathsf{y}, Xp', \Psi)) &= \mathbf{substv}(Xp', \mathbf{subst}(\Gamma, \Psi), \Gamma(\mathsf{y})) \\
&= (\mathsf{d}, Xp'/Xp'', \Psi' \cup \mathbf{subst}(\Gamma, \Psi)) \quad (*) \\
&= \tau''
\end{aligned}$$

From this, we can show

$$\begin{aligned}
\mathbf{substv}(Xp'/Xp, \Psi, \Gamma(\mathsf{y})) &= \mathbf{substv}(Xp'/Xp, \mathbf{subst}(\Gamma, \Psi), (\mathsf{d}, Xp'', \Psi')) \\
&= (\mathsf{d}, Xp''/(Xp'/Xp), \Psi' \cup \mathbf{subst}(\Gamma, \Psi)) && (\textit{xpassoc}) \\
&\equiv (\mathsf{d}, Xp''/Xp', \Psi' \cup \mathbf{subst}(\Gamma, \Psi)) \circ Xp && (*) \\
&= \tau'' \circ Xp
\end{aligned}$$

□

Lemma 10 (Expression Substitution)**.** *Let Expr an expression,* Γ' *an environment, and* Γ_1 *and* Γ *such that for all* x, $\Gamma' \vdash x : \tau$ *with* $\Gamma_1 \vdash x : \tau_1$, $\mathbf{subst}(\Gamma, \tau_1) \sqsubseteq \tau$.

Now let $\Gamma' \vdash$ *Expr* $: \tau$ *and* $\Gamma_1 \vdash$ *Expr* $: \tau_1$. *Then* $\mathbf{subst}(\Gamma, \tau_1) \sqsubseteq \tau$.

Proof. By case distinction over *Expr*. The case $\varnothing$ is trivial, and the case of variables holds by assumption. We only need to consider the case of XPath attachment, where $\mathit{Expr} = \mathsf{x}/Xp$, but this follows from Lemma 9. □

In the following, we utilize another set of rewriting rules:

$$\begin{aligned} \tau \oplus \xi &\Longrightarrow \xi \quad (\oplus\xi) \\ (\mathsf{x}, Xp, \{\xi\} \cup \Psi) &\Longrightarrow \xi \quad (\Psi\xi) \\ (\mathsf{x}, Xp, \{\neg\xi\} \cup \Psi) &\Longrightarrow \xi \quad (\Psi\neg\xi) \end{aligned}$$

Lemma 11. *Assume types τ, τ'. Then*

1. *If $[\![\tau']\!]_D = \xi$, then $[\![\tau]\!]_D[\tau'] = \xi$*
2. *If $[\![\tau']\!]_D = \varnothing$, then $[\![\tau]\!]_D[\tau'] = \varnothing$*
3. *If $[\![\tau']\!]_D \notin \{\xi, \varnothing\}$, then $[\![\tau]\!]_D[\tau'] = [\![\tau]\!]_D$*

Proof. Straightforward. □

Lemma 12 (Conditional Appendage Substitution)**.** *Let $\tau \sqsubseteq \tau'$ and $\tau_1 \sqsubseteq \tau_1'$. Then*

$$\tau[\tau_1] \sqsubseteq_\Gamma \tau'[\tau_1']$$

and

$$\tau[\neg\tau_1] \sqsubseteq_\Gamma \tau'[\neg\tau_1']$$

Proof. Straightforward by Lemma 11.

□

Lemma 13. *Let $[\![\tau]\!]_D = [\![\tau']\!]_D$ and $[\![\tau'']\!]_D \neq \xi$.*

$$[\![\tau[\tau''] \oplus \tau[\neg\tau'']]\!]_D = [\![\tau]\!]_D = [\![\tau']\!]_D$$

Proof. First, consider $[\![\tau]\!]_D = \xi$; this case is trivial. Next, we distinguish

1. $[\![\tau'']\!]_D = \varnothing$. In this case $[\![\tau[\tau'']]\!]_D = \varnothing$ but $[\![\tau[\neg\tau'']]\!]_D = [\![\tau]\!]_D$.
2. $[\![\tau'']\!]_D \neq \varnothing$, $[\![\tau'']\!]_D \neq \xi$. This is the inverse of the above.

□

Lemma 14 (Merge Substitution)**.** *Assume $\Gamma_1' \sqsubseteq_\Gamma \Gamma'$ and $\Gamma_1'' \sqsubseteq_\Gamma \Gamma''$. Then, for all $\mathsf{subst}(\Gamma, \tau_1) \sqsubseteq \tau$,*

$$\mathsf{subst}(\Gamma, \mathsf{merge}(\Gamma_1', \Gamma_1'', \tau_1)) \sqsubseteq \mathsf{merge}(\Gamma', \Gamma'', \tau)$$

Proof. We show this property by showing the computed environments pointwise equivalent. First, observe that since we require $\equiv$ to be non-discriminating (see Definition 17), we know that $\Gamma_1'(x) \equiv \Gamma_1''(x)$ implies $\Gamma'(x) \equiv \Gamma''(x)$ and $\Gamma'(x) \sqsubseteq_\Gamma \Gamma_1'(x)$ by our precondition.

Otherwise, $\Gamma'(x) \not\equiv \Gamma''(x)$. We must then show that

1. $\Gamma_1'(x)[\tau_1] \oplus \Gamma_1''(x)[\neg\tau_1] \sqsubseteq_\Gamma \Gamma'(x)[\tau] \oplus \Gamma''(x)[\neg\tau]$ It is sufficient to show that

$$\Gamma_1'(x)[\tau_1] \sqsubseteq_\Gamma \Gamma'(x)[\tau] \quad \text{and}$$
$$\Gamma_1''(x)[\neg\tau_1] \sqsubseteq_\Gamma \Gamma'(x)[\neg\tau]$$

   which follow from Lemma 12.
2. $\Gamma_1'(x)[\tau_1] \oplus \Gamma_1''(x)[\neg\tau_1] \sqsubseteq_\Gamma \Gamma'(x)$ where $\Gamma'(x) \equiv \Gamma''(x)$. If any of τ_1, $\Gamma_1'(x)$, $\Gamma_1''(x)$ have the semantics of ξ after **subst**, then so has the result and we are done. Otherwise the desired property follows from Lemma 13.

□

Theorem 1 (Substitution)**.** *Let* $\Gamma\{S\}\Gamma'$*, and* $\Gamma_0\{S\}\Gamma_1$*. Then* $\textbf{subst}(\Gamma, \Gamma_1) \sqsubseteq \Gamma'$.

Proof. Proof by structural induction over S. First, observe that $\textbf{subst}(\Gamma, \Gamma_0) = \Gamma$ (via extension of Lemma 8, thus handling skip). Sequencing is trivial from the induction hypothesis. This leaves four cases: assignment and accumulation follow directly from Lemma 10, while conditionals follow from Lemma 14.

loop.. Consider the case $S =$ foreach i in *Expr* $\{ B \}$. Let $\Gamma_0\{B\}\Gamma_1$ and assume (without loss of generality) that all types in Γ_1 are predicate-normal. Let $\Gamma_f = \textbf{flatten}_{\mathsf{i}}(\Gamma_1)$. Furthermore let

$$\Gamma \vdash \mathit{Expr} : \tau$$
$$\Gamma_0 \vdash \mathit{Expr} : \tau_0$$

By Lemma 8, we know $\tau_0 \sqsubseteq_\Gamma \tau$. We must now show, for each x, r, τ' with $\Gamma_f \vdash x : \tau'$,

$$\textbf{promote}_{\Gamma_0,\tau_0,\mathsf{i}}(x, r, \tau') \sqsubseteq_\Gamma \textbf{promote}_{\Gamma,\tau,\mathsf{i}}(x, r, \tau')$$

These only differ in the indices to **promote**. Thus, we only need to consider (via induction) two cases:

1. $\tau' = (\mathsf{x}, \epsilon, \varnothing)$. Here we must show

$$\Gamma_0(x) \sqsubseteq_\Gamma \Gamma(x)$$

   which we know from Lemma 10.
2. $\tau' = (\mathsf{i}, Xp', \Psi')$ follows directly from Lemma 10 and Lemma 13.

Finally, by Lemma 14,

$$\textbf{merge}(\textbf{promote}_{\Gamma_0,\tau_0,\mathsf{i}}(\Gamma_1), \Gamma_0, \tau_0) \sqsubseteq_\Gamma \textbf{merge}(\textbf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma_1), \Gamma, \tau)$$

□

10.4. Correctness with Loops

Definition 20. A variable x *has a fixpoint* τ *in* S iff with $\Gamma_0\{S^k\}\Gamma_k$, $\Gamma_k \vdash x : \tau$, for any $k \geq 1$, where $S^1 = S$ and $S^{n+1} = S; S^n$.

Definition 21. A type τ is *stable* iff it falls into one of the following categories:

1. $\tau = \varnothing$
2. $\tau = (\mathsf{d}, Xp, \Psi)$, $\mathsf{d} \in$ *Doc*, and for all τ', $\neg\tau'$ in Ψ, τ' is stable
3. $\tau = \tau_1 \oplus \tau_2$ and τ_1 and τ_2 are both stable

Lemma 15. *Assume* $\Gamma_0\{S\}\Gamma_1$ *and* $\Gamma_1 \vdash x : \tau$. *If* τ *is stable,* x *has a fixpoint* τ *in* S.

Proof. With $\Gamma_k\{S\}\Gamma_{k+1}$, we know from the substitution theorem that

$$\mathbf{subst}(\Gamma_1, \Gamma_k) \sqsubseteq \Gamma_{k+1}$$

so it is sufficient to show that $\mathbf{subst}(\Gamma, \tau) = \tau$. This follows from straightforward structural induction over the structure of τ. □

Definition 22. A type τ is i-stable iff one of the following cases holds:

1. τ is stable
2. $\tau = \tau_1 \oplus \tau_2$ and both τ_1 and τ_2 are i-stable
3. $\tau = (\mathsf{i}, Xp, \Psi)$ and for all τ', $\neg\tau'$ in Ψ, τ' is i-stable.

Further, a type τ is i,x-stable iff one of the following cases holds:

1. τ is i-stable
2. $\tau = \tau_1 \oplus \tau_2$ and both τ_1 and τ_2 are i,x-stable
3. $\tau = (\mathsf{x}, \epsilon, \varnothing)$

Lemma 16 (Statement Typing Soundness)**.** *Let* σ *be a store consistent with* Γ, *and let* $\Gamma\{S\}\Gamma'$ *and* $\langle S, \sigma\rangle \Downarrow \sigma'$. *Then* σ' *is consistent with* Γ'.

Proof. We follow the proof of Lemma 7, except for also handling loops.

Consider the following:

- a loop S = foreach i in *Expr* { B }
- a typing environment Γ consistent with a store σ,
- σ' as per $\langle S, \sigma\rangle \Downarrow \sigma'$,
- $\Gamma_0\{S\}\Gamma_s$, and
- $\Gamma_f = \mathbf{flatten}_{\mathsf{i}}(\Gamma_s)$.

Also assume $\Gamma \vdash$ *Expr* $: \tau$ and $\langle$*Expr*$, \sigma\rangle \models N$. If N is empty, then Γ and σ' are consistent. Otherwise, let $N = \{n_1, \ldots, n_l\}$ and define

- $\sigma_u^0 = \sigma$

- $\langle B, \sigma^k \rangle \Downarrow \sigma_u^{k+1}$ for all $k \in \{1, \ldots, l\}$
- $\sigma^k = \sigma_u^k[i \mapsto \{n_k\}]$
- $\sigma' = \sigma_u^{l+1}[i \mapsto \varnothing]$

Note that $\langle S, \sigma \rangle \Downarrow \sigma'$. The σ^k give us a means for referring to intermediate computational stages, but we do not have a typing environment that is consistent with them. To achieve this, we define a family of fresh *Doc* variables I_k such that $D(I_i) = \{n_i\}$ (without loss of generality) for suitable i, and define

- $\Gamma_u^0 = \Gamma$
- $\Gamma^k\{B\}\Gamma_u^{k+1}$ for all $k \in \{1, \ldots, l\}$
- $\Gamma^k = \Gamma_u^k[i \mapsto (I_k, \epsilon, \varnothing)]$

Observe:

1. For $k \in \{1, \ldots, l\}$, Γ^k and σ^k are consistent by our induction hypothesis
2. $[\![(I_1, \epsilon, \varnothing)]\!]_D \cup \ldots \cup [\![(I_l, \epsilon, \varnothing)]\!]_D = N$

and, by Lemma 6, $N :: \tau$.

To show our typing treatment of loops correct, it is therefore (by Observation 1 above) sufficient to show

$$\mathsf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma_f) \sqsubseteq \Gamma^l$$

For each variable x, consider $\Gamma_s \vdash x : \tau'$ and $\Gamma^l \vdash x : \tau^l$. We show that

$$\mathsf{promote}_{\Gamma,\tau,\mathsf{i}}(x, \mathsf{rec}(x, \tau'), \mathsf{flatten}_{\mathsf{i}}(\tau')) \sqsubseteq \tau^l$$

Observe that τ' is i,x-stable iff $\mathsf{flatten}_{\mathsf{i}}(\tau')$ is i,x-stable, and that **promote** maps all τ' that are not i,x-stable to ξ. Without loss of generality, we then only need to prove the above property for predicate-normal i,x-stable τ'. We employ structural induction over $\tau'' = \tau'$. If τ'' is stable, we have the desired property by Lemma 15; with union, the property follows from the induction hypothesis. We are then left with the following cases:

1. $\tau'' = (x, \epsilon, \varnothing)$. From the substitution lemma we see by straightforward induction that $\tau^l = (x, \epsilon, \varnothing)$.
2. $\tau'' = (\mathsf{i}, Xp, \Psi)$ and τ' syntactically contains $(\mathsf{x}, \epsilon, \varnothing)$ (i.e., $\mathsf{rec}(x, \tau')$). To simplify our exposition, we consider two separate cases for Ψ:
   (a) All types in Ψ are stable. In that case, the substitution lemma shows us that, inductively,
   $$\tau^l = (I_1, Xp, \Psi) \oplus \ldots \oplus (I_l, Xp, \Psi)$$
   which (by Observation 2) is equivalent to $(\tau \circ Xp)[\Psi]$, so the desired property follows from Lemmas 12 and 9.

(b) All types in Ψ are i-stable but not all are stable. Then, without loss of generality, $\Psi = \{(\mathsf{i}, Xp', \varnothing)\} \cup \Psi'$ (analogously with negation).

$$\tau^l = (I_1, Xp, \Psi' \cup \{(I_1, Xp', \varnothing)\}) \oplus \ldots \oplus (I_l, Xp, \Psi' \cup \{(I_l, Xp', \varnothing)\})$$

which is (by Observation 2) equivalent to $(\tau \circ [Xp'] \circ Xp)[\Psi']$; again the desired property follows from Lemmas 12 and 9.

If $N \neq \varnothing$, then Γ and σ' are consistent, otherwise $\mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma_f)$ and σ' are consistent. As we showed in the correctness proof for conditionals in Lemma 7, this implies that $\mathbf{merge}(\mathbf{promote}_{\Gamma,\tau,\mathsf{i}}(\Gamma_f), \Gamma, \tau)$ and σ' are consistent. □

Typing soundness is then straightforward:

Theorem 2 (Typing Soundness)**.** *Assume a program P, an initial store σ and an initial environment Γ.*

Now let $\langle P, \sigma\rangle \Downarrow \sigma'$ and $\Gamma\{P\}\Gamma'$. Then

1. *σ' is consistent with Γ'*
2. *Γ' is not loop-dependent.*

Proof. By Lemma 16 and Corollary 1. □

Further, all of our rewriting rules preserve semantics:

Theorem 3. *The rewriting rules from Figure 6 are correct.*

Proof. (*comm*), (*assoc*), (*empty*), (*idem*) arise trivially from the semantics of (lifted) set union. The others follow from the definition of XPath semantics, with (*join*) following directly from Lemma 13. We show two of the more interesting cases below:

Consider (*selfdep*):

$$(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi)) \Longrightarrow (\mathsf{x}, Xp, \Psi)$$

We must consider three cases:

1. $[\![(\mathsf{x}, Xp, \Psi)]\!]_D = \varnothing$, but then $[\![(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi))]\!]_D = \varnothing$.
2. $[\![(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi))]\!]_D = \xi$. Observe that this means that, for τ or $\neg\tau$ in Ψ, $[\![\tau]\!]_D = \xi$. But then $[\![(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi))]\!]_D = \xi$.
3. $[\![(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi))]\!]_D \notin \{\varnothing, \xi\}$, then $[\![(\mathsf{x}, Xp, (\mathsf{x}, Xp, \Psi))]\!]_D = [\![(\mathsf{x}, Xp, \varnothing)]\!]_D$.

Consider (*flat*-1):

$$(x, Xp, \{(y, Xp', \Psi')\} \cup \Psi) \Longrightarrow (x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \Psi')$$

Again we consider three cases:

1. $\mathbf{satisfied}(\Psi') = \mathit{true}$. Then

$$\begin{aligned}[\![(x, Xp, \{(y, Xp', \Psi')\} \cup \Psi)]\!]_D &= [\![(x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi)]\!]_D \\ &= [\![(x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \varnothing)]\!]_D \\ &= [\![(x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \Psi')]\!]_D\end{aligned}$$

2. $\textsf{satisfied}(\Psi) = \textit{false}$. Then

$$\begin{aligned}
[\![(x, Xp, \{(y, Xp', \Psi')\} \cup \Psi)]\!]_D &= [\![(x, Xp, \{(y, Xp', \{\varnothing\})\} \cup \Psi)]\!]_D \\
&= [\![(x, Xp, \{(y, Xp', \{\varnothing\})\} \cup \Psi \cup \{\varnothing\})]\!]_D \\
&= [\![(x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \{\varnothing\})]\!]_D \\
&= [\![(x, Xp, \{(y, Xp', \varnothing)\} \cup \Psi \cup \Psi')]\!]_D
\end{aligned}$$

3. $\textsf{satisfied}(\Psi) = \xi$. Then both expressions have the semantics of ξ, analogously to the previous points.

□