A formal semantics of patterns in XSLT

Philip Wadler
Bell Labs, Lucent Technologies

March 29, 2000

1 Introduction

> Semanticists should be obstetricians, not coroners of programming languages. — John Reynolds

Programming language theorists have developed a number of techniques to formally capture the meaning of programming languages. As our epigram insists, these theories are best applied not when a language is too old for help, but when it is aborning and subject to guidance.

XSLT is a language for transforming XML documents into other XML documents [XSLT]. A key element of XSLT is the sub-language of patterns, which is used for matching and selection. The pattern language of XSLT has recently evolved into XPath [XPath], a language of selection paths and expressions that performs core functions of both XSLT and XPointer [XPointer].

This note presents a formal semantics of the pattern language from the 16 December 1998 draft of XSLT [XSL-Dec98]. The semantics is clear and concise, summarizing in one page of formulas what required about ten pages of prose to describe. With the aid of the semantics one can rigorously state and prove properties of the language; these properties helped to guide future development of the XSLT design; thus fullfilling Reynolds's prescription.

The semantics was developed using standard techniques from the programming language community, and this article provides a tutorial introduction to these techniques. While little here will be new to the language theorist, some of what is here may be of use to the markup technologist.

A formal semantics brings to light issues that can be hard to spot in an english language description. For instance, matching is a central concept in XSLT. Here is how it was described in [XSL-Dec98].

> The result of the *MatchExpr* is true if, for any node in the document that contains the context of the *MatchExpr*, the result of evaluating the *SelectExpr* with that node as context contains the context of the *MatchExpr*. Otherwise the result is false.

The attempt to make this sentence precise has rendered it almost unreadable. Even so, as we shall later see, the sentence is ambiguous, so even a careful reading can leave one unsure of the intention. In contrast, the formal specification given here presents the same information in just one line, is easier to read than the english, and avoids the ambiguity. Partly as a result of insight gleaned from the semantics, the definition of match patterns in XSLT was simplified and made easier to implement.

The formal semantics is given in a style known as *denotational semantics*. There are several textbook introductions to this subject, including those by Schmidt [Schmidt] and Allison [Allison]. We will be able to get by using some of the most basic ideas from semantics. Many of the tricky corners is semantics arise from possibly infinite behaviour, such as a program that may enter an infinite loop. Since in our case we deal with finite documents and finite sets of nodes, these complexities can be avoided.

The formal semantics also draws upon techniques from the functional programming community. Again, there are several fine introductions, including those by Bird [Bird] and Paulson [Paulson]. The semantics was developed and debugged by transliterating it into the functional language Haskell [Haskell], and a copy of the Haskell program may be had by contacting the author. In related work, Haskell programs for manipulating XML have been developed by Wallace and Runciman [Wallace and Runciman].

The same techniques used here can be extended to give a denotational semantics of the entire XPath language, and such a semantics has been written. However, XPath is considerably more powerful than the pattern language of the December 1998 XSLT, and the semantics is correspondingly more complex. The semantics given here seems more appropriate for a gentle introduction.

This paper is organized as follows. Section 2 reviews the relevant mathematical theory of sets and relations, showing how it can be applied to model XML documents. Section 3 fleshes out the XML data model. Section 4 gives a tutorial introduction to denotational semantics, giving two simple semantics for arithmetic expressions. Section 5 gives a denotational semantics of patterns in the 16 December 1998 draft of XSLT. Section 6 draws some lessons from this semantics. Section 7 applies the semantics to prove some propositions about patterns. Section 8 concludes.

2 Sets and relations

We review here standard operations on sets and relations, and shows how we might use these to model XML documents and operations upon them.

Here is part of your family tree, represented as an XML document.

```
<Adam>
  <Cain>
    <Enoch/>
  </Cain>
  <Abel/>
  <Seth>
    <Enosh/>
  </Seth>
</Adam>
```

Let $\{\,\mathsf{Root}, \mathsf{Adam}, \mathsf{Cain}, \mathsf{Enoch}, \mathsf{Abel}, \mathsf{Seth}, \mathsf{Enosh}\,\}$ be the set of nodes in this tree, where Root is a distinguished root node. The key relationships in the tree can be modeled with two functions.

$$\begin{array}{lcl} \mathit{children} & : & \mathit{Node} \to \mathit{Set}(\mathit{Node}) \\ \mathit{children}(\mathsf{Root}) & = & \{\,\mathsf{Adam}\,\} \\ \mathit{children}(\mathsf{Adam}) & = & \{\,\mathsf{Cain}, \mathsf{Abel}, \mathsf{Seth}\,\} \\ \mathit{children}(\mathsf{Cain}) & = & \{\,\mathsf{Enoch}\,\} \\ \mathit{children}(\mathsf{Enoch}) & = & \emptyset \\ \mathit{children}(\mathsf{Abel}) & = & \emptyset \\ \mathit{children}(\mathsf{Seth}) & = & \{\,\mathsf{Enosh}\,\} \\ \mathit{children}(\mathsf{Enosh}) & = & \emptyset \end{array}$$

$$\begin{array}{lcl} \mathit{parent} & : & \mathit{Node} \to \mathit{Set}_1(\mathit{Node}) \\ \mathit{parent}(\mathsf{Root}) & = & \emptyset \\ \mathit{parent}(\mathsf{Adam}) & = & \{\,\mathsf{Root}\,\} \\ \mathit{parent}(\mathsf{Cain}) & = & \{\,\mathsf{Adam}\,\} \\ \mathit{parent}(\mathsf{Enoch}) & = & \{\,\mathsf{Cain}\,\} \\ \mathit{parent}(\mathsf{Abel}) & = & \{\,\mathsf{Adam}\,\} \\ \mathit{parent}(\mathsf{Seth}) & = & \{\,\mathsf{Adam}\,\} \\ \mathit{parent}(\mathsf{Enosh}) & = & \{\,\mathsf{Seth}\,\} \end{array}$$

We write $\mathit{Set}(A)$ for the type of a set where each element is of type A, and $\mathit{Set}_1(A)$ for the subtype of sets with at most one element, and $\emptyset$ for the emptyset. We write $\in$ for membership, $\subseteq$ for subset, and $\cup$ for set union. For example, $\mathsf{Cain} \in \mathit{children}(\mathsf{Adam})$, $\{\,\mathsf{Cain}, \mathsf{Abel}\,\} \subseteq \mathit{children}(\mathsf{Adam})$, and $\{\,\mathsf{Cain}, \mathsf{Abel}\,\} \cup \{\,\mathsf{Seth}\,\} = \mathit{children}(\mathsf{Adam})$.

We will make extensive use of comprehension notation, as in the following example.

$$\begin{array}{lcl} \mathit{siblings} & : & \mathit{Node} \to \mathit{Set}(\mathit{Node}) \\ \mathit{siblings}(x) & = & \{\, z \mid y \in \mathit{parent}(x),\ z \in \mathit{children}(y)\,\} \end{array}$$

This is read 'the siblings of node x is the set of all nodes z such that some node y is a parent of x and z is a child of y'. With this definition, we have $\mathit{siblings}(\mathsf{Cain}) = \{\,\mathsf{Cain}, \mathsf{Abel}, \mathsf{Seth}\,\}$. (Rather than 'Am I my brother's keeper?', the apposite question here is 'Am I my brother?')

To further demonstrate the power of the comprehension notation, here is a definition that corrects the shortcoming of the first.

$$\begin{array}{lcl} \mathit{properSiblings} & : & \mathit{Node} \to \mathit{Set}(\mathit{Node}) \\ \mathit{properSiblings}(x) & = & \{\, y \mid y \in \mathit{siblings}(x),\ x \neq y\,\} \end{array}$$

This is read 'the proper siblings of node x is the set of all nodes y such that y is a sibling of x, and y is distinct from x'. With this definition, we have $\mathit{properSiblings}(\mathsf{Cain}) = \{\,\mathsf{Abel}, \mathsf{Seth}\,\}$.

Here is another example.

$$\begin{array}{lcl} \mathit{grandparent} & : & \mathit{Node} \to \mathit{Set}_1(\mathit{Node}) \\ \mathit{grandparent}(x) & = & \{\, z \mid y \in \mathit{parent}(x),\ z \in \mathit{parent}(y)\,\} \end{array}$$

This is read 'the grandparent of node x is the set of all nodes z such that some node y is a parent of x, and z is a parent of y'. Thus, $\mathit{grandparent}(\mathsf{Enosh}) = \{\,\mathsf{Adam}\,\}$ and $\mathit{grandparent}(\mathsf{Adam}) = \emptyset$. Note how this definition deals neatly with the case of a node that has no grandparent — instead of an error, one simply gets an empty set.

We represent a relation on nodes by a function $r : \mathit{Node} \to \mathit{Set}(\mathit{Node})$. The functions *children*, *parent*, *siblings*, and *grandparent* are all examples of relations. If r_1, r_2 are two relations, we write $r_1 \,;\, r_2$ for their composition, defined as follows.

$$(r_1 \,;\, r_2)(x) \quad = \quad \{\, z \mid y \in r_1(x), z \in r_2(y)\,\}$$

Thus, relation $r_1 \,;\, r_2$ relates node x to node z if relation r_1 relates x to some y, and relation r_2 relates y to some z. For example, $\mathit{siblings} = \mathit{parent} \,;\, \mathit{children}$ and $\mathit{grandparent} = \mathit{parent} \,;\, \mathit{parent}$.

The identity relation *self* is the one that relates a node solely to itself.

$$\mathit{self}(x) \quad = \quad \{\, x \,\}$$

Observe that the identity relation is indeed a left and right identity for composition, that is $r \,;\, \mathit{self} = r = \mathit{self} \,;\, r$. Further, composition is associative, so $(r_1 \,;\, r_2) \,;\, r_3 = r_1 \,;\, (r_2 \,;\, r_3)$.

If r is a relation, we define the transitive closure r^+ and the reflexive and transitive closure r^* in the usual way.

$$\begin{array}{lcl} r^+(x) & = & \{\, z \mid y \in r(x),\ z \in r^*(y) \,\} \\ r^*(x) & = & \{\, x \,\} \cup r^+(x) \end{array}$$

Thus $\mathit{children}^+(x)$ returns all proper descendants of x (not including itself), and $\mathit{children}^*(x)$ returns all descendants of x (including itself), and $\mathit{parent}^+(x)$ returns all proper ancestors of x. (We use the same symbols as with regular expressions, because in both cases $+$ corresponds one or more repetitions, and $*$ corresponds to zero or more repetitions.)

Recall that the document order is a total order on all the nodes in a document; for element nodes, it is the order of their start tags. We write

$$\mathit{first}, \mathit{last} : \mathit{Set}(\mathit{Node}) \rightarrow \mathit{Set}_1(\mathit{Node})$$

for the functions that take a set and return the singleton set containing just its first or last node in document order, or return the empty set if the given set is empty. For instance, $\mathit{first}(\mathit{children}(\mathsf{Adam})) = \mathsf{Cain}$, $\mathit{last}(\mathit{children}(\mathsf{Adam})) = \mathsf{Seth}$, and $\mathit{first}(\mathit{children}(\mathsf{Abel})) = \emptyset$.

Having established these preliminaries, we can now proceed to establish a more complete formal model for XML documents.

3 A data model for XML

This section presents a mathematical data model for XML. It is based on the data model given in Section 2.6 of [XSL-Dec98], which is about 4 pages long. Even this simple model revealed some small ambiguities in [XPath], which we discuss at the end of this section. For simplicity, we will ignore namespaces; although a full treatment of them along similar lines is possible.

The model we give is, unsurprisingly, quite similar to the Document Object Model (DOM), a standard interface for accessing and manipulating representations of XML trees, which works with a number of programming languages [DOM]. There are a few differences from the DOM (which we will note parenthetically).

The basic datatype is *Node*. Each *Node* is one of six kinds: root, element, attribute, text, comment, or processing instruction. We indicate this with six boolean functions; for each node, exactly one of these functions yields true.

$$\begin{array}{lcl} \mathit{isRoot} & : & \mathit{Node} \rightarrow \mathit{Boolean} \\ \mathit{isElement} & : & \mathit{Node} \rightarrow \mathit{Boolean} \\ \mathit{isAttribute} & : & \mathit{Node} \rightarrow \mathit{Boolean} \\ \mathit{isText} & : & \mathit{Node} \rightarrow \mathit{Boolean} \\ \mathit{isComment} & : & \mathit{Node} \rightarrow \mathit{Boolean} \\ \mathit{isPI} & : & \mathit{Node} \rightarrow \mathit{Boolean} \end{array}$$

(DOM differs in that root nodes are called document nodes, and instead of six boolean functions there is a single *nodetype* function.)

We have the following functions that relate nodes to other nodes in the document.

$$\begin{array}{lcl} \mathit{parent} & : & \mathit{Node} \rightarrow \mathit{Set}_1(\mathit{Node}) \\ \mathit{children} & : & \mathit{Node} \rightarrow \mathit{Set}(\mathit{Node}) \\ \mathit{attributes} & : & \mathit{Node} \rightarrow \mathit{Set}(\mathit{Node}) \\ \mathit{root} & : & \mathit{Node} \rightarrow \mathit{Node} \end{array}$$

Function *parent* returns the parent of the node, or returns the empty set if the node is the root. Function *children* returns the set of children of a node, and is empty if the node has no children. Attribute, text, comment, and PI nodes never have chidren. An element node may also have related attribute nodes, these are not counted as children, but instead are returned by *attributes*. Function *root* returns the root node of the document.

These functions are related by various laws. Only a root or element can have children, and only an element can have attribute nodes.

$$\begin{array}{lll} \mathit{children}(x) \neq \emptyset & \text{implies} & \mathit{isRoot}(x) \vee \mathit{isElement}(x) \\ \mathit{attributes}(x) \neq \emptyset & \text{implies} & \mathit{isElement}(x) \end{array}$$

Every child is an element, text, comment, or processing instruction. Unsurprisingly, the attributes of a node are attribute nodes, and the root of a node is the root node.

$$\begin{array}{lll} y \in \mathit{children}(x) & \text{implies} & \mathit{isElement}(y) \vee \mathit{isText}(y) \vee \mathit{isComment}(y) \vee \mathit{isPI}(y) \\ y \in \mathit{attributes}(x) & \text{implies} & \mathit{isAttribute}(y) \\ y = \mathit{root}(x) & \text{implies} & \mathit{isRoot}(y) \end{array}$$

Define the subnodes of a node to be the union of its children and attributes.

$$\begin{array}{lcl} \mathit{subnodes} & : & \mathit{Node} \rightarrow \mathit{Set}(\mathit{Node}) \\ \mathit{subnodes}(x) & : & \mathit{children}(x) \cup \mathit{attributes}(x) \end{array}$$

Then one node is the subnode of another exactly when the parent of that node is the other.

$$y \in \mathit{subnodes}(x) \quad \text{if and only if} \quad \mathit{parent}(y) = \{\, x \,\}$$

Every node in the document is descended via the subnode relation from the root.

$$x \in \mathit{subnodes}^*(\mathit{root}(x))$$

(DOM differs in that attribute nodes are considered not to have any parent.)

Other functions on a node access content, with the meaning changing for different node types.

$$\begin{array}{lcl} \mathit{name} & : & \mathit{Node} \rightarrow \mathit{String} \\ \mathit{value} & : & \mathit{Node} \rightarrow \mathit{String} \end{array}$$

The meanings are best given by a table.

	name	*value*
Root	empty	value of children
Element	tag name	value of children
Attribute	attribute name	attribute value
Text	empty	content of text node
Comment	empty	content of the comment
PI	target	content excluding target

Here 'empty' denotes the empty string, and 'value of children' means the result of concatenating the values of the children nodes, taken in ascending document order. (DOM has a similar table, but the name of root, text, and comment nodes is taken to be `'#document'`, `'#text'`, and `'#comment'`, respectively, rather than empty; while the value of root and element nodes is taken be empty, rather than the concatenation of the values of the children.)

Recall that the DTD of an XML document may declare an element to have an ID attribute; a string uniquely identifies the element in the document with that string as its ID attribute. To model this, we introduce the function *id*, which takes a string and returns the singleton set containing the corresponding element, or returns the empty set if no such element exists. (If we were dealing with multiple documents, then *id* should have an additional parameter to specify which document.) Sometimes multiple ids are contained in a single attribute. The function *split* takes a string into the set of space-separated identifiers within it.

$$\begin{array}{lcl} id & : & String \rightarrow Set_1(Node) \\ split & : & String \rightarrow Set(String) \end{array}$$

(There is nothing in DOM that corresponds to these functions. By the way, note that XSL allows any string to be taken as an identifier, not just strings in 'idref' attributes.)

The above model already reveals some small ambiguities in [XPath], which does not say what the name of a root, text, or comment node should be. In fact, most readers of the document guess (correctly) that they should be empty, but this is not entirely obvious since the DOM makes a different definition. Indeed, when this point was brought to the attention of the XSL committee, some members argued that the DOM definition was better, although (at this writing) the definition given here appears to prevail. At any event, the act of writing a formal model revealed the ambiguity, which should now be eliminated in the next XPath draft.

We now have all the tools we need to write the denotational semantics, so the next step is to review how such semantics are constructed.

4 Denotational semantics

This section introduces the standard techniques of denotational semantics, by giving a trivial semantics for arithmetic expressions, and a simple semantics for expressions with variables. Anyone already familiar with denotational semantics may skip (or skim) this section; anyone not familiar may wish to consult standard texts like [Schmidt] or [Allison] for further details.

The starting point for *semantics* is *syntax* — we require expressions to which meaning can be attached. Fortunately, the state of the art for dealing with syntax is far superior to that for dealing with semantics, as indicated by the widespread use of formal notations such as BNF and regular expressions. Imagine what it would be like if we were forced to resort to english to describe the syntax of our programs as well as their semantics!

Much of the complication in syntax arises from the need to establish precedence of operators. For instance, here is a typical *concrete syntax* for arithmetic expressions.

$$\begin{array}{lcl} expr & ::= & expr \texttt{ + } term \\ & | & expr \texttt{ - } term \\ & | & term \\ \\ term & ::= & term \texttt{ * } primary \\ & | & term \texttt{ / } primary \\ & | & primary \\ \\ primary & ::= & number \\ & | & \texttt{(} \ expr \ \texttt{)} \end{array}$$

This syntax specifies that addition binds less tightly than multiplication, and that all the operations are right associative, so that `1+2*3+4` is parsed the same as `((1+(2*3))+4)`.

The purpose of the concrete syntax is to specify a mapping from strings to parse trees, and it is to the parse trees rather than the strings that we wish to attach a meaning. Therefore, we take as our starting point for semantics an *abstract syntax* that corresponds directly to the underlying parse tree. Here is an abstract syntax for expressions.

$$\begin{array}{lll} i & : & \mathit{Number} \\ e & : & \mathit{Expr} \qquad e \ ::= \ e \,\texttt{+}\, e \mid e \,\texttt{-}\, e \mid e \,\texttt{*}\, e \mid e \,\texttt{/}\, e \mid i \mid \texttt{(}\, e\, \texttt{)} \end{array}$$

Here for convenience we use e as shorthand for *expr*, *term*, and *primary*, and i as shorthand for *number*. We write *Expr* as the type of the parse trees for expressions. Numbers are sufficiently well understood that we don't bother to distinguish between the syntax and semantics, and take *Number* as both the type used to represent numbers in the parse tree and the type used for the value of expressions. If we want to be precise, we might say that *Number* corresponds to the 64 bit floating point numbers in IEEE 754 format (which is how numbers are represented in XPath).

All information about precedence and associativity of operators has been removed from the abstract syntax above. Some authors of denotational semantics would go even further and remove the case for parentheses, which serves only to identify grouping and have no role in the underlying parse tree.

The denotational semantics of expressions is a function $\mathcal{E}$ from expressions to numbers.

$$\begin{array}{lcl} \mathcal{E} & : & \mathit{Expr} \to \mathit{Number} \\ \mathcal{E}[\![e_1 \,\texttt{+}\, e_2]\!] & = & \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!] \\ \mathcal{E}[\![e_1 \,\texttt{-}\, e_2]\!] & = & \mathcal{E}[\![e_1]\!] - \mathcal{E}[\![e_2]\!] \\ \mathcal{E}[\![e_1 \,\texttt{*}\, e_2]\!] & = & \mathcal{E}[\![e_1]\!] \times \mathcal{E}[\![e_2]\!] \\ \mathcal{E}[\![e_1 \,\texttt{/}\, e_2]\!] & = & \mathcal{E}[\![e_1]\!] \div \mathcal{E}[\![e_2]\!] \\ \mathcal{E}[\![i]\!] & = & i \\ \mathcal{E}[\![\texttt{(}\, e\, \texttt{)}]\!] & = & \mathcal{E}[\![e]\!] \end{array}$$

We write $\mathcal{E}[\![e]\!]$ to denote the meaning of the expression e, where by convention syntax is distinguished from semantics by writing syntactic entities inside special semantic brackets. The meaning of each expression is given in terms of the meanings of its subexpressions. Thus, the first line says that the value of the expression $e_1 \,\texttt{+}\, e_2$ is computed by finding the value of expression e_1 and the value of expression e_2 and adding the results.

The semantics is pretty trivial, amounting to little more than a change of symbols. But that is because we have chosen to explain arithmetic in terms of arithmetic, so there is little useful work to do.

For a less trivial example, consider a semantics for expressions augmented with variable names. Here is the new abstract syntax.

$$\begin{array}{lll} i & : & \mathit{Number} \\ n & : & \mathit{Name} \\ e & : & \mathit{Expr} \qquad e \ ::= \ n \mid \texttt{let}\ n\ \texttt{=}\ e\ \texttt{in}\ e \mid \\ & & \qquad\qquad\quad\ \ e \,\texttt{+}\, e \mid e \,\texttt{-}\, e \mid e \,\texttt{*}\, e \mid e \,\texttt{/}\, e \mid i \mid \texttt{(}\, e\, \texttt{)} \end{array}$$

The grammar for expressions has two new lines, to define variables and bindings. For example, `let a=1+2 in a*a` evaluates to `9`.

The syntax is given in terms of an environment, which maps variables to their value. We let ρ range over environments.

$$\rho \ : \ \mathit{Environment} \ = \ \mathit{Name} \to \mathit{Number}$$

We need two operations on environments. The first takes an environment and a name and returns the corresponding number. The second takes a name, a number, and an environment, and returns a new environment that is the same as the first, but also maps the given name to the given number.

$$
\begin{array}{lcl}
\mathit{lookup} & : & (\mathit{Name}, \mathit{Environment}) \to \mathit{Number} \\
\mathit{lookup}(v, \rho) & = & \rho(v) \\
\\
\mathit{extend} & : & (\mathit{Name}, \mathit{Number}, \mathit{Environment}) \to \mathit{Environment} \\
\mathit{extend}(v, i, \rho) & = & \rho_1 \quad \text{where} \quad \rho_1(v_1) \;=\; \text{if } (v_1 = v) \text{ then } i \text{ else } \rho(v_1)
\end{array}
$$

These definitions exhibit the utility of *higher-order* functions. Both *lookup* and *extend* take a function (ρ) as an argument, and *extend* also returns a function (ρ_1) as its result. For example, let ρ_0 be the environment that maps every name to the IEEE 754 floating point number NaN (Not a Number). Then

$$\rho_1 = \mathit{extend}(\mathtt{b}, 4, \mathit{extend}(\mathtt{a}, 3, \rho_0))$$

is the environment such that $\mathit{lookup}(\mathtt{a}, \rho_1) = \mathtt{3}$. According to the definition, a new binding may *shadow* the old binding of a variable. For example, if $\rho_2 = \mathit{extend}\mathtt{a}, 5, \rho_1$ then $\mathit{lookup}(\mathtt{a}, \rho_2) = \mathtt{5}$, not `3`.

The augmented denotational semantics takes an expression and an environment and returns a number.

$$
\begin{array}{lcl}
\mathcal{E} & : & \mathit{Expr} \to \mathit{Environment} \to \mathit{Number} \\
\mathcal{E}[\![v]\!]\rho & = & \mathit{lookup}(v, \rho) \\
\mathcal{E}[\![\mathtt{let}\ v\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!](\mathit{extend}(v,\ \mathcal{E}[\![e_1]\!]\rho,\ \rho)) \\
\mathcal{E}[\![e_1 \mathbin{\mathtt{+}} e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 \mathbin{\mathtt{-}} e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho - \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 \mathbin{\mathtt{*}} e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \times \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![e_1 \mathbin{\mathtt{/}} e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \div \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![i]\!]\rho & = & i \\
\mathcal{E}[\![\mathtt{(}\ e\ \mathtt{)}]\!]\rho & = & \mathcal{E}[\![e]\!]\rho
\end{array}
$$

Now we write $\mathcal{E}[\![e]\!]\rho$ for the meaning of expression e in environment ρ. Again, the meaning of each expression is given in terms of the meaning of its components. The first line says that the value of name n in environment ρ is given by looking up the name in the environment. The second line says that the value of the expression $\mathtt{let}\ n\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ in environment ρ is given by evaluating e_1 in environment ρ, extending environment ρ by binding name n to the value just computed, then evaluating e_2 in the new environment. The rest is much as before, passing the environment unchanged to each subexpression. (If you find this paragraph hard to read, you may find it easier to just look at the formulas — that is the point of using a formalism!)

By convention, the semantic function $\mathcal{E}$ is given a higher-order type: it takes an expression, and returns a function from environments to values. Thus, for instance, the meaning of $\mathcal{E}[\![\mathtt{a+b}]\!]$ is the function $f : \mathit{Environment} \to \mathit{Number}$ such that $f(\rho) = \mathit{lookup}(\mathtt{a}, \rho) + \mathit{lookup}(\mathtt{b}, \rho)$. Also by convention the parentheses may be dropped around the second argument, so we write $\mathcal{E}[\![e]\!]\rho$ rather than $\mathcal{E}[\![e]\!](\rho)$.

The semantics we have given answers some subtle questions about the meaning of expressions. For instance, consider the expression

```
let a=3 in (let a=4 in a+a)+a
```

Is the value of this expression `11` or `12`? In a language like C, the value of the similar expression `(a=4, a+a)+a` is `12`. But it should be clear from looking at the semantics that the value of the above expression is `11`, because the rightmost `a` is evaluated in a different environment than the other two occurrences of `a`. (To learn how to give a denotational semantics for an imperative language like C, consult any standard text on semantics, such as [Schmidt] or [Allison].)

This concludes our semantics tutorial. We are now ready to look at a denotational semantics for part of XSL.

5 A semantics for patterns

This section presents the semantics of matching and selection in the 16 December 1998 draft of XSLT [XSL-Dec98]. What we describe here corresponds to Section 2.6 of that draft. That section is 11 pages long (including two pages for concrete syntax), as compared to the one page of text and one page of formulas given here (excluding concrete but including abstract syntax).

The abstract syntax of patterns and qualifiers is as follows.

$$
\begin{array}{lllll}
n & : & \mathit{Name} & & \\
s & : & \mathit{String} & & \\
p & : & \mathit{Pattern} & p ::= & p_1 \,|\, p_2 \mid \texttt{/}p \mid \texttt{//}p \mid p_1\texttt{/}p_2 \mid p_1\texttt{//}p_2 \mid p\texttt{[}q\texttt{]} \mid \\
 & & & & n \mid \texttt{*} \mid \texttt{@}n \mid \texttt{@*} \mid \texttt{text()} \mid \texttt{comment()} \mid \texttt{pi(}n\texttt{)} \mid \texttt{pi()} \mid \texttt{id(}p\texttt{)} \mid \texttt{id(}s\texttt{)} \mid \\
 & & & & \texttt{ancestor(}p\texttt{)} \mid \texttt{ancestor-or-self(}p\texttt{)} \mid \texttt{.} \mid \texttt{..} \\
q & : & \mathit{Qualifier} & q ::= & q_1 \texttt{ and } q_2 \mid q_1 \texttt{ or } q_2 \mid \texttt{not(}q\texttt{)} \mid \texttt{(}q\texttt{)} \\
 & & & & \texttt{first-of-type()} \mid \texttt{last-of-type()} \\
 & & & & \texttt{first-of-any()} \mid \texttt{last-of-any()} \mid p\texttt{=}s \mid p
\end{array}
$$

The abstract syntax is much more permissive than the corresponding concrete syntax. For instance, in the concrete syntax the pattern union operator $p_1 \,|\, p_2$ is constrained to appear at the top level of a pattern, while here we allow it at any level in the syntax tree. As we will see, there is no problem whatsoever in giving a syntax for the more general language, and the XSL document does not explain why the syntax is restricted, though presumably the designers believed the restrictions make the language easier and more efficient to implement.

The semantics is given in Figure 1. A pattern may be used either for matching or selection. We write $\mathcal{M}[\![p]\!]x$ to denote whether the pattern p matches against the node x, we write $\mathcal{S}[\![p]\!]x$ to denote the set of nodes selected by the pattern p when the context node is x, and we write $\mathcal{Q}[\![q]\!]x$ to denote whether the qualifier q is satisfied when the context node is x. Thus, the semantic functions have the following types.

$$
\begin{array}{lll}
\mathcal{M} & : & \mathit{Pattern} \to \mathit{Node} \to \mathit{Boolean} \\
\mathcal{S} & : & \mathit{Pattern} \to \mathit{Node} \to \mathit{Set}(\mathit{Node}) \\
\mathcal{Q} & : & \mathit{Qualifier} \to \mathit{Node} \to \mathit{Boolean}
\end{array}
$$

Again, the semantics of each pattern is defined in terms of the semantics of its subpatterns, and similarly for quantifiers.

We give a reading for a few lines in the definition. The first line of $\mathcal{S}$ says that the pattern $p_1 \,|\, p_2$ with current node x selects the union of the nodes selected by p_1 and p_2, each with current node x. The second line says that the pattern $\texttt{/}p$ with current node x selects the same nodes as selected by the pattern p with current node $\mathit{root}(x)$, the root of the document containing node x. The fourth line says that the pattern $p_1\texttt{/}p_2$ with current node x is the set of all nodes x_2 selected as follows: let x_1 be any node selected by p_1 with current node x, and then let x_2 be any node selected by p_2 with current node x_1. The seventh line says that the name n with current node x selects all subnodes of x that are elements and have the name n. The eigth line says that the pattern `*` with current node x selects all the subnodes of x that are elements. The ninth line says that the pattern $\texttt{@}n$ with current node x selects all subnodes of x that are attributes and have the name n. The penultimate line says that the pattern `.` with current node x selects the set of nodes containing just x. The readings for the other formulas are similar. (Again, if you find this paragraph hard to read, you may prefer to just look at the formulas — and, again, that is the point!)

If p is a pattern, then $\mathcal{S}[\![p]\!] : \mathit{Node} \to \mathit{Set}(\mathit{Node})$ is a relation. In other words, the meaning of a pattern is a relation between the current node and all the nodes the pattern selects. Many of the operations on patterns can be re-expressed directly as operations on relations. In particular, it is easy to check that p_1/p_2 is relational composition and . is the identity relation.

$$\begin{array}{rcl} \mathcal{S}[\![p_1/p_2]\!] & = & \mathcal{S}[\![p_1]\!]\,;\mathcal{S}[\![p_2]\!] \\ \mathcal{S}[\![\,.\,]\!] & = & \mathit{self} \end{array}$$

It follows immediately that the patterns p/. and p and ./p are all equivalent.

The semantics clarifies a number of points that may not be obvious on a first reading of the XSL draft specification [XSL-Dec98]. We'll explore some of these in the next section, and see how the formal semantics aided the development of XSL.

6 Lessons from the semantics

Now that we've defined the semantics, what can we do with it? In this section, we make use of the semantics to illustrate some subtle points of the XSL design. And we will show how the semantics helped us to obey Reynold's dictum — rather than merely cataloguing a dead design, insight from the semantics contributed to the growth of a living language.

We begin by examining some key features of the structure of the semantics. For purpose of symmetry, it is written entirely using the *subnodes* function defined in Section 3, and never uses *children* or *attributes* directly. Note that *subnodes* does not appear in the defintion of p_1/p_2, but instead appears in the definition of n, `*`, `@`n, `@*`, `text()`, `comment()`, `pi()`, and `pi(`n`)`. It is very tempting to give a different definition where *subnodes* appears only once in the definition of p_1/p_2 and does not appear in the definitions of n, `*`, `@`n, `@*`, and the like. However, that definition does not work — there is then no possible way to define the . pattern! One consequence of this choice is that the element selection patterns n and `*` always step down to a subnode, and therefore it is crucial that our data model defines a root node which has the document element as a subnode.

These architectural points are not obvious from the english draft of the specification. Discussions revela that people often think of p_1/p_2 as performing a subnode step; that they don't think of n and `*` as performing a subnode step; or that they don't understand why the root is different from the document element. Wallace and Runciman [Wallace and Runciman] have even published a set of Haskell functions for XSL-style pattern matching that makes the opposite choice, where the operation corresponding to p_1/p_2 does perform the subnode step, and hence no operation corresponding to . can be defined.

Another subtlety is that their are two quite distinct ways in which a pattern is converted to a boolean — the definition of $\mathcal{M}[\![p]\!]x$ is quite different from the definition of $\mathcal{Q}[\![p]\!]x$. The latter states that a pattern p acting as a qualifier is satisfied in context node x if selecting from pattern p with context node x yields a non-empty set of nodes. The former is discussed in detail next.

Recall the description of matching in [XSL-Dec98], quoted in the introduction.

> The result of the *MatchExpr* is true if, for any node in the document that contains the context of the *MatchExpr*, the result of evaluating the *SelectExpr* with that node as context contains the context of the *MatchExpr*. Otherwise the result is false.

This is summarized in the one line describing match patterns, which says that a pattern p matches in a context x if there is *any* node x_1 in the document such that selecting with pattern p in context x_1 yields a set that contains the original node x. (Since it is allowed to mention variables, the above sentence is perhaps a little clearer than the XML document, and arguably the one-line formula is clearer still.)

Note that the original paragraph is ambiguous. Should the phrase 'any node in the document that contains the context of the *MatchExpr*' be read as 'any node in (the document that contains the context of the *MatchExpr*)', meaning any node in the document whatsoever? Or should it be read as (any node in the document) that contains the context of the *MatchExpr*', meaning any node in the document that is an ancestor of (i.e., that contains) the context node of the *MatchExpr*? As we can see from the formal specification the intended meaning is the former, since $subnodes^*(root(x))$ generates all nodes in the document containing the context node x. The other interpretation could be formalized by replacing that expression by $parent^*(x)$, which generates all ancestors of x.

Matching against a pattern as defined in this way can potentially be quite expensive. For instance, consider the following document, where the `id` attribute is declared in the DTD to be unique identifiers as used in the `id()` pattern.

```
<doc>
  <cite>ref1</cite>
  <cite>ref2</cite>
  <reference id='ref1'/>reference one</reference>
  <reference id='ref2'/>reference two</reference>
  <reference id='ref3'/>reference three</reference>
</doc>
```

The pattern `id(cite)` will match against any node that is cited in the document. But this means that when checking a reference node for a match, one must scan the entire document to check for a citation; the document may be rather larger than seven lines, so this could be expensive. To avoid repeatedly scanning the document, one might be clever and try to build a reverse index, showing which nodes might be referenced by which other pieces of text, but this won't be easy. (Especially since XSL does not guarantee that id will be contained in an idref attribute, so the entire text must be indexed.) Furthermore, users are unlikely to exploit this sort of pattern, so the effort of implementation is not commensurate with the benefit. Indeed, it turns out that neither XT nor Microsoft IE5 properly implemented this aspect of the December 1998 XSL draft.

Partly as a result of writing the formal semantics, this problem came to light. It was proposed to solve the problem by restricting the patterns that could be used as match patterns to a subset of those that could be used as select patterns. In particular, only `id(`s`)` can be used as a match pattern, yielding nodes identified by a literal string s, but `id(`p`)` cannot. Other restrictions were to outlaw the use of `ancestor(`p`)`, `ancestor-or-self(`p`)`, and `..` in match patterns, all of which were thought to be of little use, hard to implement, and confusing.

One consequence of these restrictions is that the ambiguity noted previously becomes irrelevant: matching from any node in the document becomes equivalent to matching from any ancestor of the context node! This isn't just an idle guess, in the next section we will see how we can state and prove this as a formal proposition.

One benefit of a formal semantics is that it becomes easier to prove equivalences between patterns. We have already observed that p`/.` and `./`p and p are all equivalent patterns. More interestingly, one can also show that p_1`/id(`p_2`)` is equivalent to `id(`p_1`/`p_2`)`. Once this equivalence was formulated, it was decided that it was confusing to have two forms of expression that do the same thing, so patterns were further restricted to eliminate the first form in favor of the second.

7 Some propositions about patterns

When it comes to formalism, the proof of the pudding is in, well, the proof. Here we state and prove some propositions about patterns referred to in the preceding section.

We previously asserted that for a restricted notion of match pattern, matching against any node in the document is equivalent to matching against any ancestor. To formally state this, we first define a second semantics of match patterns, which matches against ancestors rather than against any node.

$$\begin{array}{lcl} \mathcal{M}' & : & \mathit{Pattern} \times \mathit{Node} \to \mathit{Boolean} \\ \mathcal{M}'[\![p]\!]x & = & x \in \{\, x_2 \mid x_1 \in \mathit{parent}^*(x),\ x_2 \in \mathcal{S}[\![p]\!](x_1)\,\} \end{array}$$

Then we have a proposition asserting the two semantics are equivalent for the restricted patterns.

Proposition. For any pattern p (not containing the forms `id(`p`)` `ancestor(`p`)`, `ancestor-or-self(`p`)`, or `..`) and any node x we have

$$\mathcal{M}[\![p]\!]x = \mathcal{M}'[\![p]\!]x.$$

The proof requires two lemmas. With the restricted patterns removed, every remaining pattern is either absolute (that is ignores the context node) or descending (that is, any nodes it yields are descendants of the context node).

Lemma. If p is a pattern of the form $/p_1$ or `id(`s`)`, then

$$\mathcal{S}[\![p]\!]x \quad = \quad \mathcal{S}[\![p]\!](\mathit{root}(x))$$

Lemma. For any pattern p (not containing the forms $/p$, `id(`s`)`, `ancestor(`p`)`, `ancestor-or-self(`p`)`, or `..`) and any node x we have

$$\mathcal{S}[\![p]\!]x \subseteq \mathit{subnodes}^*(x)$$

The first lemma is proved by case analysis on the two possibilities, the second lemma is proved by case analysis on the remaining possibilities and structural induction. The proposition then follows easily, since $y \in \mathit{subnodes}^*(x)$ is equivalent to $x \in \mathit{parent}^*(y)$. QED.

We skip the details of the proof — further explanation of the relevant techniques can be found in any good text on functional programming or denotational semantics, such as [Bird], [Paulson], [Schmidt], and [Allison]. However, just to give the flavor, the next example does present one simple proof in detail.

As we mentioned earlier, the patterns p_1`/id(`p_2`)` and `id(`p_1/p_2`)` are equivalent. Here we give the proof in some detail. We begin with the special case where p_2 is taken to be the `.` patttern.

Proposition. For any pattern p and for any node x, we have

$$\mathcal{S}[\![p/\texttt{id(.)}]\!]x \quad = \quad \mathcal{S}[\![\texttt{id}(p)]\!]x$$

The proof is a simple calculation, expanding definitions and simplifying.

$$\begin{array}{cl}
 & \mathcal{S}[\![p/\texttt{id(.)}]\!]x \\
= & \qquad \text{definition of } \mathcal{S}[\![p_1/p_2]\!] \\
 & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ x_2 \in \mathcal{S}[\![\texttt{id(.)}]\!]x_1 \,\} \\
= & \qquad \text{definition of } \mathcal{S}[\![\texttt{id}(p)]\!] \\
 & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ x_2 \in \{\, x_4 \mid x_3 \in \mathcal{S}[\![\texttt{.}]\!]x_1,\ s \in \mathit{split}(\mathit{value}(x_3)),\ x_4 \in \mathit{id}(s)\,\}\,\} \\
= & \qquad \text{definition of } \mathcal{S}[\![\texttt{.}]\!] \\
 & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ x_2 \in \{\, x_4 \mid x_3 \in \{\, x_1 \,\},\ s \in \mathit{split}(\mathit{value}(x_3)),\ x_4 \in \mathit{id}(s)\,\}\,\} \\
= & \qquad \text{simplify} \\
 & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ x_2 \in \{\, x_4 \mid s \in \mathit{split}(\mathit{value}(x_1)),\ x_4 \in \mathit{id}(s)\,\}\,\} \\
= & \qquad \text{simplify} \\
 & \{\, x_4 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ s \in \mathit{split}(\mathit{value}(x_1)),\ x_4 \in \mathit{id}(s)\,\} \\
= & \qquad \text{definition of } \mathcal{S}[\![\texttt{id}(p)]\!] \\
 & \mathcal{S}[\![\texttt{id}(p)]\!]x
\end{array}$$

(This style of proof is widely used, see [Bird] for examples of its power.)

Using the fact that p_1/p_2 corresponds to relational composition, we can rewrite the above proposition as follows.

$$\mathcal{S}[\![\mathtt{id(}p\mathtt{)}]\!] \quad = \quad \mathcal{S}[\![p]\!]\,;\mathcal{S}[\![\mathtt{id(.)}]\!]$$

Using the fact that relational composition is associative, it is then easy to prove the more general result.

Proposition. For any patterns p_1 and p_2 and for any node x, we have

$$\mathcal{S}[\![p_1\mathtt{/id(}p_2\mathtt{)}]\!]x \quad = \quad \mathcal{S}[\![\mathtt{id(}p_1\mathtt{/}p_2\mathtt{)}]\!]x$$

Here is the calculation, spelled out in some detail.

$$\begin{array}{cl}
 & \mathcal{S}[\![p_1\mathtt{/id(}p_2\mathtt{)}]\!] \\
= & \qquad \mathtt{/}\text{ is relational composition} \\
 & \mathcal{S}[\![p_1]\!]\,;\mathcal{S}[\![\mathtt{id(}p_2\mathtt{)}]\!] \\
= & \qquad \text{previous proposition} \\
 & \mathcal{S}[\![p_1]\!]\,;(\mathcal{S}[\![p_2]\!]\,;\mathcal{S}[\![\mathtt{id(.)}]\!]) \\
= & \qquad \text{relational composition is associative} \\
 & (\mathcal{S}[\![p_1]\!]\,;\mathcal{S}[\![p_2]\!])\,;\mathcal{S}[\![\mathtt{id(.)}]\!] \\
= & \qquad \mathtt{/}\text{ is relational composition} \\
 & \mathcal{S}[\![p_1\mathtt{/}p_2]\!]\,;\mathcal{S}[\![\mathtt{id(.)}]\!] \\
= & \qquad \text{previous proposition} \\
 & \mathcal{S}[\![\mathtt{id(}p_1\mathtt{/}p_2\mathtt{)}]\!]
\end{array}$$

QED.

8 Conclusions

We have seen that a formal semantics can be more concise and readable than english; that it lets one formulate and prove precise statements about the meaning of programs; and that such statements can aid further development of a language.

The techniques described in this paper have been applied to derive a denotional semantics for the entire XPath language. Unfortunately, that semantics is significantly more complex than the one given here. Will the insight gleaned from the new semantics aid in further simplification of XPath? Only time (and the workings of W3C committees) will tell.

Acknowledgements James Clark provided much useful feedback on the formal semantics, as well as providing an excellent informal semantics in the XSLT draft documents. Catherine Lyons made several suggestions that improved the presentation of this paper. Matias Giovannini and Dan Connolly spotted a number of typos.

References

[Allison] Lloyd Allison, *A Practical Introduction to Denotational Semantics*, Cambridge University Press, 1987.

[Bird] Richard Bird, *Introduction to Functional Programming, 2nd edition*, Addison-Wesley, 1998.

[DOM] Lauren Wood, editor, *Document Object Model (DOM)*, Version 1.0, W3C Recommendation, 1 October 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`

[Haskell] The Haskell home page, `www.haskell.org`.

[Paulson] Larry Paulson, *ML for the Working Programmer, 2nd edition*, Cambridge University Press, 1998.

[Schmidt] David A. Schmidt, *The Structure of Typed Programming Languages*, MIT Press, Cambridge, MA, 367 pages, 1994.

[Wallace and Runciman] Malcolm Wallace and Colin Runciman, Haskell and XML: Generic Combinators or Type-Based Translation? *4'th International Conference on Functional Programming (ICFP 99)*, Paris, ACM Press, September 1999.

[XML] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors, *Extensible Markup Language (XML) 1.0*, W3C Recommendation, 10 February 1998. `http://www.w3.org/TR/1998/REC-xml`

[XPointer] Steve DeRose, editor, *XML Pointer Language (XPointer)*, W3C Working Draft. `http://www.w3.org/TR/WD-xptr`

[XSL-Dec98] James Clark and Stephen Deach, editors, *Extensible Stylesheet Language (XSL)*, W3C Working Draft, 16 December 1998. `http://www.w3.org/TR/1998/WD-xsl-19981216`

[XSLT] James Clark, editor, *XSL Transformations (XSLT)*, W3C Working Draft. `http://www.w3.org/TR/WD-xslt`

[XPath] James Clark and Steve DeRose, editors, *XSL Path Language (XPath)*, W3C Working Draft, 9 July 1999. `http://www.w3.org/1999/07/WD-xpath-19990709`

LaTeX generated from XML via XSLT.

$$
\begin{array}{lcl}
\mathcal{M} & : & \mathit{Pattern} \to \mathit{Node} \to \mathit{Boolean} \\
\mathcal{M}[\![p]\!]x & = & x \in \{\, x_2 \mid x_1 \in \mathit{subnodes}^*(\mathit{root}(x)),\ x_2 \in \mathcal{S}[\![p]\!](x_1)\,\} \\
 & & \\
\mathcal{S} & : & \mathit{Pattern} \to \mathit{Node} \to \mathit{Set}(\mathit{Node}) \\
\mathcal{S}[\![p_1 \,|\, p_2]\!]x & = & \mathcal{S}[\![p_1]\!]x \cup \mathcal{S}[\![p_2]\!]x \\
\mathcal{S}[\![\texttt{/}p]\!]x & = & \mathcal{S}[\![p]\!](\mathit{root}(x)) \\
\mathcal{S}[\![\texttt{//}p]\!]x & = & \{\, x_2 \mid x_1 \in \mathit{subnodes}^*(\mathit{root}(x)),\ x_2 \in \mathcal{S}[\![p]\!]x_1\,\} \\
\mathcal{S}[\![p_1\texttt{/}p_2]\!]x & = & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p_1]\!]x,\ x_2 \in \mathcal{S}[\![p_2]\!]x_1\,\} \\
\mathcal{S}[\![p_1\texttt{//}p_2]\!]x & = & \{\, x_3 \mid x_1 \in \mathcal{S}[\![p_1]\!]x,\ x_2 \in \mathit{subnodes}^*(x_1),\ x_3 \in \mathcal{S}[\![p_2]\!]x_2\,\} \\
\mathcal{S}[\![p\texttt{[}q\texttt{]}]\!]x & = & \{\, x_1 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ \mathcal{Q}[\![q]\!]x_1\,\} \\
\mathcal{S}[\![n]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isElement}(x_1),\ \mathit{name}(x_1) = n\,\} \\
\mathcal{S}[\![\texttt{*}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isElement}(x_1)\,\} \\
\mathcal{S}[\![\texttt{@}n]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isAttribute}(x_1),\ \mathit{name}(x_1) = n\,\} \\
\mathcal{S}[\![\texttt{@*}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isAttribute}(x_1)\,\} \\
\mathcal{S}[\![\texttt{text()}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isText}(x_1)\,\} \\
\mathcal{S}[\![\texttt{comment()}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isComment}(x_1)\,\} \\
\mathcal{S}[\![\texttt{pi(}n\texttt{)}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isPI}(x_1),\ \mathit{name}(x_1) = n\,\} \\
\mathcal{S}[\![\texttt{pi()}]\!]x & = & \{\, x_1 \mid x_1 \in \mathit{subnodes}(x),\ \mathit{isPI}(x_1)\,\} \\
\mathcal{S}[\![\texttt{id(}p\texttt{)}]\!]x & = & \{\, x_2 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ s \in \mathit{split}(\mathit{value}(x_1)),\ x_2 \in \mathit{id}(s)\,\} \\
\mathcal{S}[\![\texttt{id(}s\texttt{)}]\!]x & = & \{\, x_1 \mid s_1 \in \mathit{split}(s),\ x_1 \in \mathit{id}(s_1)\,\} \\
\mathcal{S}[\![\texttt{ancestor(}p\texttt{)}]\!]x & = & \mathit{last}(\{\, x_1 \mid x_1 \in \mathit{parent}^+(x),\ \mathcal{M}[\![p]\!]x_1\,\}) \\
\mathcal{S}[\![\texttt{ancestor-or-self(}p\texttt{)}]\!]x & = & \mathit{last}(\{\, x_1 \mid x_1 \in \mathit{parent}^*(x),\ \mathcal{M}[\![p]\!]x_1\,\}) \\
\mathcal{S}[\![\texttt{.}]\!]x & = & \{\, x\,\} \\
\mathcal{S}[\![\texttt{..}]\!]x & = & \mathit{parent}(x) \\
 & & \\
\mathcal{Q} & : & \mathit{Qualifier} \to \mathit{Node} \to \mathit{Boolean} \\
\mathcal{Q}[\![q_1 \ \texttt{and}\ q_2]\!]x & = & \mathcal{Q}[\![q_1]\!]x \wedge \mathcal{Q}[\![q_2]\!]x \\
\mathcal{Q}[\![q_1 \ \texttt{or}\ q_2]\!]x & = & \mathcal{Q}[\![q_1]\!]x \vee \mathcal{Q}[\![q_2]\!]x \\
\mathcal{Q}[\![\texttt{not(}q\texttt{)}]\!]x & = & \neg\mathcal{Q}[\![q]\!]x \\
\mathcal{Q}[\![\texttt{(}q\texttt{)}]\!]x & = & \mathcal{Q}[\![q]\!]x \\
\mathcal{Q}[\![\texttt{first-of-type()}]\!]x & = & x \in \mathit{first}(\{\, x_1 \mid x_1 \in \mathit{siblingElements}(x),\ \mathit{name}(x_1) = \mathit{name}(x)\,\}) \\
\mathcal{Q}[\![\texttt{last-of-type()}]\!]x & = & x \in \mathit{last}(\{\, x_1 \mid x_1 \in \mathit{siblingElements}(x),\ \mathit{name}(x_1) = \mathit{name}(x)\,\}) \\
\mathcal{Q}[\![\texttt{first-of-any()}]\!]x & = & x \in \mathit{first}(\mathit{siblingElements}(x)) \\
\mathcal{Q}[\![\texttt{last-of-any()}]\!]x & = & x \in \mathit{last}(\mathit{siblingElements}(x)) \\
\mathcal{Q}[\![p\texttt{=}s]\!]x & = & \{\, x_1 \mid x_1 \in \mathcal{S}[\![p]\!]x,\ \mathit{value}(x_1) = s\,\} \neq \emptyset \\
\mathcal{Q}[\![p]\!]x & = & \mathcal{S}[\![p]\!]x \neq \emptyset \\
 & & \\
\mathit{siblingElements}(x) & = & \{\, x_2 \mid x_1 \in \mathit{parent}(x),\ x_2 \in \mathit{subnodes}(x_1),\ \mathit{isElement}(x_2)\,\}
\end{array}
$$

Figure 1: Semantics of patterns in December 1998 draft of XSLT