Compilation of Generic Regular Path Expressions Using C++ Class Templates

Compilation of Generic Regular Path Expressions Using C++ Class Templates

Luca Padovani[⋆]

University of Bologna, Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, Italy
`lpadovan@cs.unibo.it`

Abstract. Various techniques for the navigation and matching of data structures using path expressions have been the subject of extensive investigations. No matter whether such techniques are based on type information, indexing, automata, it is desirable to synthesize implementations automatically, starting from a high-level description of the path expressions to be traversed.
In this paper we present a library of C++ templates for the representation of regular path expressions and their compilation into efficient backtracking algorithms. The resulting code can be used to implement visitors, pattern matchers, node collectors on regular paths over possibly heterogeneous, linked data structures.
The point of the paper is on the path compilation technique, which was inspired by a continuation-passing, functional semantics of the path expressions. We rely on some peculiar aspects of C++ templates to create a compilation framework that closely follows the given semantics.

1 Introduction

The traversal of linked data structures along paths with a certain pattern is an operation that underlies many kinds of more complex queries: the recognition of a context in the proximity of a node in the data structure, the selection, the iteration, the visit of a set of nodes that are related by a pattern.

Regular path expressions extend the concept of plain regular expressions to the traversal of linked data structures. The recent diffusion of XML technologies has made path expressions the subject of a renewed interest. Among proposed standards we cite XPath [11,17] and XQuery [19], but a whole plethora of languages and techniques have been studied and developed. Optimized implementations are possibly based on type information, indexing, finite state automata [20], tree automata [12,15].

There are contexts, however, where optimizations are not applicable or give little advantage if compared to backtracking algorithms for path traversal. These contexts are often characterized by the following aspects:

– the data structure evolves under the effect of frequent editing operations;
– operations may change the structure arbitrarily, making it infeasible to keep track of the state of the structure if not by considering the whole structure as "the state";

[⋆] This work has been supported by the European Project IST-2001-33562 MoWGLI.

– the patterns to be matched are sparse or linear, that is we are interested in checking only a limited number of objects around a focused node (which is sometimes called *cursor*), rather than performing an exhaustive query on whole documents.

The problem of backtracking algorithms is that their complexity grows unmanageably as the path expressions become more complicated, the strive to make them efficient contributes significantly to their complexity, they are seldom generic, reusable, and they cannot be easily composed together. It is generally desirable to be able to specify the path expressions to be traversed in a high-level language which would be compiled into efficient traversal code.

In this paper we describe a framework for the compilation of generic regular path expressions into backtracking algorithms. The framework is based on C++ class templates that represent path expressions. The same class templates are able to synthesize the traversal code following the functional semantics of the path expressions being compiled. The bottom line is that there is no need for any tool other than the C++ compiler itself, and the generated algorithm integrates seamlessly with the rest of the code.

The structure of the paper is as follows: in Section 2 we overview the basic constructs of generic regular path expressions and define their set-based semantics in a way similar to what was done for XPath in [10]. In Section 3 we define a continuation-passing, functional semantics for path expressions that is equivalent to the set-based semantics. Section 4 gives the stateless implementation of a simple but limited compilation framework that closely follows the functional semantics. Section 5 elaborates the stateless implementation into a stateful one and shows how the library can be adapted so as to accomplish specific tasks. Section 6 gives a brief account of related work. We conclude in Section 7 with some performance comparisons. The source code of the PET library is publicly available at `http://www.cs.unibo.it/~lpadovan/PET/index.html`.

Some knowledge of C++ templates and of the λ-calculus notation is assumed.

2 Syntax and set-based semantics

We consider a linked data structure where links are represented as labelled arcs. For the sake of simplicity, in this paper we assume that the data structure is uniform and that its elements have all the same type Object. This assumption is relaxed in the implementation of the library. We consider generic regular path expressions generated by the grammar of Table 1. The expressions are generic in the sense that the finite set of *atomic expressions*, denoted by the nonterminal $\langle \mathit{atom} \rangle$, is left unspecified. Conceptually atoms can be classified as *selectors* and *filters*. Selectors follow labelled arcs in the data structure: for each $x, y \in \mathit{Object}$, we say that $s \in \langle \mathit{atom} \rangle$ selects y from x if there is an arc $x \xrightarrow{s} y$ in the data structure. For example, in a tree data structure we may have the “first child of” or “parent of” selectors (links), with their usual meaning. Filters can be thought of as *predicates* on objects in the Object domain. In data structures with labeled objects a typical filter is “has label l” (examples of “labels” are names, types, identifiers). Only two atoms are pre-defined, they are the identity selector $\mathbf{1}$ and the null selector $\mathbf{0}$ (equivalently the filters for the always-true and always-false predicates respectively).

Table 1. Abstract syntax of generic regular path expressions.

$\langle atom\rangle$ ::=	$\mathbf{0}$			
	$\mid$	$\mathbf{1}$		
	$\mid$	$\ldots$		
$\langle expr\rangle$ ::=	a			$a \in \langle atom\rangle$
	$\mid$	$e_1 \mid e_2$		
	$\mid$	$e_1\, e_2$		
	$\mid$	e^*		
	$\mid$	$e?$	$\mathbf{1} \mid e$	
	$\mid$	e^+	ee^*	
	$\mid$	e^n	$\underbrace{ee\cdots e}_{n}$	$0 \leq n$
	$\mid$	$e^{n,m}$	$e^n \mid e^{n+1} \mid \cdots \mid e^m$	$0 \leq n \leq m$

A *core* path expression can be a single atom a, the alternative $e_1 \mid e_2$ between two path expressions e_1 and e_2, the composition $e_1\, e_2$ of two path expressions e_1 and e_2, or the closure e^* of a path expression e. The remaining regular path expressions can be expressed in terms of these core expressions as shown in Table 1, hence they will not be considered any further; the implementation, however, supports them. Note that given two filters p_1 and p_2, the composition $p_1\, p_2$ represents the conjunction $p_1 \wedge p_2$ and $p_1 \mid p_2$ represents the disjunction $p_1 \vee p_2$.

We define the set-based semantics of a path expression e focused on an object x as the finite set of objects selected (or reached) by e starting from x (Table 2). Borrowing the notation from [10], we write $Set(Object)$ for the type of a set where each element is of type $Object$ and $Set_1(Object)$ for the subtype of sets with at most one element. Each atom is modelled as a function $a : Object \to Set_1(Object)$. A selector s is modelled as a function s such that, given an object x, $s(x) = \{y\}$ if $x \xrightarrow{s} y$, and $s(x) = \emptyset$ if there is no arc from x labelled s. A filter p is modelled as a function p such that, given an object x, $p(x) = \{x\}$ if x satisfies p, and $p(x) = \emptyset$ otherwise.

Table 2. Set-based semantics of regular path expressions.

$$\mathcal{S}[\![]\!] \;:\; Expr \to Object \to Set(Object)$$

$$\begin{aligned}
\mathcal{S}[\![\mathbf{0}]\!]x &= \emptyset\\
\mathcal{S}[\![\mathbf{1}]\!]x &= \{x\}\\
\mathcal{S}[\![a]\!]x &= a(x)\\[1ex]
\mathcal{S}[\![e_1 \mid e_2]\!]x &= \mathcal{S}[\![e_1]\!]x \cup \mathcal{S}[\![e_2]\!]x\\
\mathcal{S}[\![e_1\, e_2]\!]x &= \mathcal{S}[\![e_2]\!]\mathcal{S}[\![e_1]\!]x = \textstyle\bigcup_{y \in \mathcal{S}[\![e_1]\!]x} \mathcal{S}[\![e_2]\!]y\\
\mathcal{S}[\![e^*]\!]x &= \textstyle\bigcup_{i=0}^{\infty} \mathcal{S}[\![e^i]\!]x
\end{aligned}$$

Example 1 (plain regular expressions). Regular expressions over sequences of symbols in a finite vocabulary V can be seen as a special case of regular path expressions where there is only one selector "next character" and there is one filter for each symbol $a \in V$ with the meaning "the current character is a". As there is only one selector, the syntax of plain regular expression does not normally have any notation for it, juxtaposition of symbols has the meaning of concatenation.

Example 2 (XPath expressions). The subset of XPath (version 1.0 [11]) expressions without qualifiers can be encoded as regular path expressions. Assuming that the parent, $\mathit{first\_child}$, $\mathit{next\_sibling}$ selectors are defined, we encode XPath axes as follows:

`self`	$\Rightarrow$	$\mathbf{1}$
`child`	$\Rightarrow$	$\mathit{first\_child}\ \mathit{next\_sibling}^*$
`descendant`	$\Rightarrow$	$(\mathit{first\_child}\ \mathit{next\_sibling}^*)^+$
`ancestor`	$\Rightarrow$	parent^+
`following_sibling`	$\Rightarrow$	$\mathit{next\_sibling}^+$
`following`	$\Rightarrow$	$\mathit{parent}^*\ \mathit{next\_sibling}^+ (\mathit{first\_child}\ \mathit{next\_sibling}^*)^*$

The remaining XPath axes are symmetric to the shown ones.

3 Functional semantics

The set-based semantics is very concise and clear in giving the meaning of path expressions, and its naive implementation is naturally based on object sets and union operations. If we were to compile an expression e to a function that way, the type of such functions would be $\mathcal{F}_0[\![e]\!] : \mathit{Object} \to \mathit{Set}(\mathit{Object})$ and, for instance, the compilation rule for a compound expression $e_1\, e_2$ would look something like

$$\mathcal{F}_0[\![e_1\, e_2]\!] = \lambda x.\, \cup_{y \in (\mathcal{F}_0[\![e_1]\!]\ x)} (\mathcal{F}_0[\![e_2]\!]\ y)\,.$$

This implementation, which is typical for several XPath engines, is unsatisfactory when

1. we are not interested to knowing which nodes have been selected, but only if at least one node was selected, or
2. we are interested to bind only a limited subset of the selected objects, or
3. we want to bind different objects to different names (perhaps because objects have different types), or
4. we want to visit the selected objects as they are discovered.

The heart of the problem is the composition $e_1\, e_2$, in particular when the path e_1 selects more than one object. We cannot avoid the set-union operation as long as e_1 and e_2 are completely evaluated in isolation. What we need is a way to tell $\mathcal{F}[\![e_1]\!]$ that, whenever it selects a node, it should proceed along the path e_2 from this node and, if this path fails, search for the next object selected by e_1. We do this by adding to $\mathcal{F}[\![e_1]\!]$ one more parameter, called the *continuation*.

Table 3 shows the continuation-passing, functional semantics of regular path expressions. We adopt the λ-calculus notation with the following extensions. We use $\{x\}$

Table 3. Functional semantics of regular path expressions.

$$
\begin{aligned}
\mathsf{null} &\equiv \lambda x.\emptyset \\
\mathsf{id} &\equiv \lambda x.\{x\} \\
\mathsf{fork} &\equiv \lambda k_1.\lambda k_2.\lambda k_3.\lambda x.\mathbf{match}\ (k_1\ x)\ \mathbf{with} \\
&\qquad\qquad\qquad\qquad\quad \{y\} \to (k_2\ y) \\
&\qquad\qquad\qquad\qquad\quad\ \ \emptyset \to (k_3\ x) \\
\mathcal{A}[\![\mathbf{0}]\!] &= \mathsf{null} \\
\mathcal{A}[\![\mathbf{1}]\!] &= \mathsf{id} \\
\mathcal{A}[\![a]\!] &= \lambda x.a(x) \\
\mathcal{F}[\![]\!] &: \mathit{Expr} \to (\mathit{Object} \to \mathit{Set}_1(\mathit{Object})) \to \mathit{Object} \to \mathit{Set}_1(\mathit{Object}) \\
\mathcal{F}[\![a]\!] &= \lambda k.(\mathsf{fork}\ \ \mathcal{A}[\![a]\!]\ \ k\ \ \mathsf{null}) \\
\mathcal{F}[\![e_1 \mid e_2]\!] &= \lambda k.(\mathsf{fork}\ \ (\mathcal{F}[\![e_1]\!]\ k)\ \ \mathsf{id}\ \ (\mathcal{F}[\![e_2]\!]\ k)) \\
\mathcal{F}[\![e_1\, e_2]\!] &= \lambda k.(\mathcal{F}[\![e_1]\!]\ (\mathcal{F}[\![e_2]\!]\ k)) \\
\mathcal{F}[\![e^*]\!] &= \lambda k.(\mathsf{fix}\ \lambda f.(\mathsf{fork}\ \ k\ \ \mathsf{id}\ \ (\mathcal{F}[\![e]\!]\ f)))
\end{aligned}
$$

to denote "some object x" and $\emptyset$ to denote "no object". These correspond to the `Some` x and `None` values in languages like ML or OCaml. We destruct optional values using a simplified form of pattern matching which is only capable of discriminating a singleton and the empty set and optionally binding a name to the value of the singleton.

The terms null, id, fork, and fix are called *basic terms*. The first two terms correspond to the compiled versions of the $\mathbf{0}$ and $\mathbf{1}$ paths respectively. The fork term is the basic term for backtracking: $(\mathsf{fork}\ e_1\ e_2\ e_3\ x)$ tries to follow the path e_1 with focus x. If it succeeds (thus reaching an object y), it continues along e_2 with focus y. If it fails, it continues along e_3 with focus x. The composition of two paths $e_1\, e_2$ is translated as the function corresponding to e_1 to which the function corresponding to e_2 is passed as continuation. The compilation of e^* makes use of the fix-point basic term, fix, which is left unspecified in the semantics as it will be implemented using a peculiar feature of C++ class templates. As usual, we require that $(\mathsf{fix}\ F) = (F\ (\mathsf{fix}\ F))$ for all functions F.

Note that the function we obtain from a path expression e expects a continuation k that "receives" the selected objects. We will see in Section 5.3 that, by varying k, one can use the path expression to accomplish different tasks.

Let us conclude this section with a proposition stating that the set-based semantics and the functional semantics are in a way equivalent:

Proposition 1. *Let*

$$\mathsf{eq} = \lambda x.\lambda y.\begin{cases} \{x\}, & \textit{if } x = y \\ \emptyset, & \textit{otherwise} \end{cases}$$

be the filter that tests whether two objects are the same object, then we have

$$\forall x, y \in \mathit{Object}, x \in \mathcal{S}[\![e]\!]y \iff (\mathcal{F}[\![e]\!]\ (\mathsf{eq}\ x)\ y) = \{x\}\,.$$

The proof follows by a structural induction on the path expression e.

4 Implementation

If we were to generate code for a functional language, the rules of the functional semantics in Table 3 could be used directly. As a matter of fact, when we first attacked the problem it seemed that the only way of producing a code close to the functional semantics was to use a functional language as target (a concrete attempt was done with Objective Caml [21]). In C++ functions are not first-class entities (let alone continuations) and moreover they can only be declared at the top-level or inside a class, hence the compilation process would not be compositional. In particular, it would not be possible to compile a local path expression within another C++ function, for instance as a test expression of an `if` statement. We can circumvent these limitations using C++ class templates.

C++ class templates allow the programmer to abstract the definition of a class with respect to one or more *template parameters*. This way it is possible to design generic classes that can be used with arbitrary data types. For example, the type

```
template <typename A, typename B>
struct Pair {
  A first;
  B second;
};
```

defines a generic `Pair` structure with two template parameters `A` and `B` for representing pairs of values of arbitrary types. In order to be used, parameterized classes must be *instantiated* with the appropriate types. The instantiation acts by substituting the abstracted types with the provided ones. For example `Pair<int, float>` represents the type of `Pair`s where the `first` component has type `int` and the `second` component has type `float`. Roughly speaking, class templates can be thought of as functions operating on and returning types.

In addition to member fields, class templates can contain member methods, member types, and member templates as well, all of which may depend on template parameters. Member methods of class templates are instantiated (that is, their actual object code is output by the compiler) on demand when they are invoked. Upon instantiation of a method of a class template, the compiler may decide to do code inlining for the method's body, or to output a standalone instance of the method to be called one or more times.

Going back to our problem, we can let the C++ compiler output functions on demand using templates, thus: for each basic term we define a template that has as many template parameters as the continuation parameters of the term (none for null and id, three for fork) and has a static member function `walk` accepting one parameter (the cursor) and implementing the term semantics. This approach is possible as long as all the continuation parameters are known at compile-time, because template instantiation must be resolved by the C++ compiler. This condition holds: given a compiled path expression $\mathcal{F}[\![e]\!]$ and a statically-known receiving term t, no redex in $(\mathcal{F}[\![e]\!]\ t)$ involving continuations does depend on runtime information.

The use of templates also allows us to relax the constraint of working on homogeneous data structures. As anticipated in the introduction, we will focus on the compilation framework rather than on these details of the implementation. From now on we assume that the objects of the data structure being traversed are accessed by a pointer `Object*`: the `walk` method will accept and return a `Object*` value.

4.1 Basic terms

The user has to provide a class (template) for each atom occurring in the path expressions. Typically, selectors corresponds to accessors in the `Object` class and filters are predicates over `Object` objects. The class for an atom a must provide a static `walk` method accepting the current object in the path (the x variable in the specification, the `x` parameter in the C++ code below) and returning a possibly null object.

Following these guidelines, the id and null atoms are implemented as follows:

```
struct IdTerm
{ static Object* walk(Object* x) const { return x; } };
struct NullTerm
{ static Object* walk(Object*) const { return 0; } };
```

According to Table 3 the term fork has four parameters. However, the first three parameters (k_1, k_2, and k_3) are continuations so they are represented by template parameters:

```
template <typename K1, typename K2, typename K3>
struct ForkTerm {
  static Object* walk(Object* x) const {
    if (Object* y = K1::walk(x)) return K2::walk(y);
    else return K3::walk(x);
  }
};
```

Finally, we need to implement the fix-point operator and we do so by exploiting a special case of parameterized inheritance. The function for which we have to compute the fixed point is represented by the template template parameter `F`:

```
template <template <typename> class F>
struct FixTerm : public F<FixTerm<F> > { };
```

A variant of this construct, in which the class itself is a parameter of the class it derives from, is known under the name of Curiously Recurring Template Pattern [7,5]. Although there are other slightly more compact ways of implementing recursive types, this one closely resembles the functional semantics and, anyway, it introduces no overhead if compared to the equivalent variants (however see note 1 in Section 4.2).

Example 3. The path expression *parent* (*parent* $\mid \mathbf{1}$), which selects both the parent and the grandparent of the cursor, is represented by the type

```
ForkTerm<ParentTerm,
         ForkTerm<ForkTerm<ParentTerm, k, NullTerm>,
                  IdTerm, ForkTerm<IdTerm, k, NullTerm> >,
         NullTerm>
```

where `Parent` implements the *parent* atom and k is the atom that is supposed to receive the selected objects.

4.2 A template-based compiler

It is clear from Example 3 that types representing path expressions are not readable and handy to work with: it would be better to use some syntax that is more closely related to the structure of the path expressions those types derive from. Although the concrete syntax of path expressions is affected by the application domain (as the two examples at the end of Section 2 have shown), we can lift from the level of basic terms to the level of the path structure. The idea, the same used in expression templates [5,18], is to encode the structure of a path expression using types so that, for instance, the type

```
SeqPath<AtomPath<ParentTerm>,
        OrPath<AtomPath<ParentTerm>, AtomPath<IdTerm> > >
```

would represent the path expression $\mathit{parent}\ (\mathit{parent} \mid \mathbf{1})$.

C++ classes (and class templates) may contain other class template declarations. We exploit this feature for implementing the compilation rules shown in Table 3. To this aim, each template representing the structure of a path expression defines a member class template `Compile` with a template parameter `K` for the continuation. The inner `Compile` class must define a member type `RES` representing the basic term resulting from the compilation. A look at the actual code for the `AtomPath`, `OrPath`, and `SeqPath` templates should clarify the basic idea:

```
template <typename A> struct AtomPath {
  template <typename K> struct Compile
  { typedef ForkTerm<A, K, NullTerm> RES; };
};

template <typename P1, typename P2> struct OrPath {
  template <typename K> struct Compile {
    typedef typename P1::template Compile<K>::RES T1;
    typedef typename P2::template Compile<K>::RES T2;
    typedef ForkTerm<T1, IdTerm, T2> RES;
  };
};

template <typename P1, typename P2> struct SeqPath {
  template <typename K> struct Compile {
    typedef typename P2::template Compile<K>::RES T1;
    typedef typename P1::template Compile<T1>::RES RES;
  };
};
```

For the compilation of the `StarPath` construct we first define an auxiliary template `F` representing $\lambda f.(\mathsf{fork}\ \ k\ \ \mathsf{id}\ \ (\mathcal{F}[\![e]\!]\ f))$ and then we apply the fix-point operator `FixTerm` to `F`:[1]

[1] At the time of this writing not every C++ compiler is capable of handling correctly the `StarPath` template. An alternative formulation for the `RES` member type which can be successfully compiled with GCC version 3.3.3 is `struct RES : public ForkTerm<K, IdTerm, typename P::template Compile<RES>::RES> { };` Note that this

```
template <typename P> struct StarPath {
  template <typename K> struct Compile {
    template <typename f>
    struct F : public ForkTerm<K, IdTerm,
                   typename P::template Compile<f>::RES> { };
    typedef FixTerm<F> RES;
  };
};
```

4.3 Code improvement

The uniform treatment of selectors and filters simplifies the framework but can result into inefficient code. When the first template argument of `ForkTerm` is a filter, `ForkTerm`'s `walk` method is exceedingly complex because a filter never returns an object different from the cursor. Depending on the type of the `walk`'s parameter (which need not necessarily be an actual pointer), a compiler may be unable to optimize the code by itself. Fortunately we can help the compiler improving the generated code in such cases by using partial template specialization. Below we show an example of such optimization:

```
template <typename Object, typename K2, typename K3>
struct ForkTerm<IdTerm, K2, K3> {
  static Object* walk(Object* x) { return K2::walk(x); }
};
```

Any time a `ForkTerm` is instantiated with `IdTerm` as its first template argument, this specialization, which defines a shorter and more efficient implementation of the `walk` method, will be "preferred" by the C++ compiler to the more general one.

5 Stateful implementation

The library developed so far is relatively simple and clean and the compilation scheme closely follows the functional semantics of path expressions. However, the use of static walking methods prevents methods from accessing any data that is not constant or at the global scope. For example, if we were to design a *sink* atom collecting any object that it is passed to, we would have to declare a global container and access that container from the `walk` method of the `Sink` class. More generally, terms can only be parameterized by values that are constants with internal static linkage, because this is the only category of values that can be passed as template parameters. Not even constant strings nor floating point numbers, for example, can be used as template parameters.

It is possible to extend the implementation seen so far to a stateful one, that is an implementation where term classes are allowed to have non-static member fields that can affect (or can be affected by) the evaluation of the `walk` method. Since most of the changes needed to the classes already seen are either trivial or technical, in the section that follows we only give a few examples and a brief account for them. The interested reader can have a look at the source code for the details.

type too is defined using a variant of the CRTP even if the `FixTerm` template is not needed anymore.

5.1 Stateful basic terms

The changes required to implement stateful terms are the following:

- terms containing subterms (like fork) must have a constructor that accepts objects representing the compiled subterms and stores them as member fields;
- the `walk` method must no longer be static;
- calls to the continuations must no longer be static but rather are proper method invocations on the continuation member fields.

In `IdTerm` and `NullTerm` only the `walk` method changes, which is no longer static. The stateful variant of `ForkTerm` is as follows, with the relevant changes underlined:

```
template <typename K1, typename K2, typename K3>
struct ForkTerm {
  ForkTerm(const K1& _k1, const K2& _k2, const K3& _k3)
    : k1(_k1), k2(_k2), k3(_k3) { }
  Object* walk(Object* x) const {
    if (Object* y = k1.walk(x)) return k2.walk(y);
    else return k3.walk(x);
  }
  const K1 k1;
  const K2 k2;
  const K3 k3;
};
```

The instances corresponding to the three continuations are passed to the constructor and embedded in the instance of `ForkTerm`. Embedding the subterms (as opposed to referencing them via pointers) is necessary because most of the time terms will be instantiated by the C++ compiler into temporaries, and storing references to such temporaries would likely result into dangling pointers.

Not surprisingly, the most delicate class to change is `FixTerm`. The problem arises because not only the type of the stateful `FixTerm` must be circular (which, as we have seen in Section 4.1, can be achieved in a relatively easy way), but also its instance as well. The circularity of the instance must be broken somehow using a reference for otherwise we end up with the paradoxical situation of an instance object containing a proper copy of itself. Also, such a circular term must be constructed in "one shot" by the default C++ constructor mechanism for we do not want the user to have to manually patch circular terms after their compilation!

The initialization of a recursive term is possible because during the instantiation of a class, and precisely when the constructor of a derived class is initializing the base class, it is already possible to refer to `this`, since the memory for the object is allocated *before* initialization takes place. In particular, in the initialization of the base class it is possible to pass `*this` as a constant reference to the object being instantiated. If the base class, or any other class this reference is passed to, stores it somewhere we have the desired circularity. To make sure that no attempt is done to copy `*this`, we introduce a new term, which we call `WeakRefTerm`, that stores its child term as a reference rather than as an embedded object. The `WeakRefTerm`'s `walk` method just forwards the invocation to the child term hence it is semantically transparent:

```
template <typename K> struct WeakRefTerm {
  WeakRefTerm(const K& _k) : k(_k) { }
  Object* walk(Object* x) const { return k.walk(x); }
  const K& k;
};
```

It is sufficient to pass `WeakRefTerm(*this)` to the base class to have the desired effect of creating a finite circularity. This solution is still problematic, though, because if a circular object gets copied (this eventually happens as continuations may be duplicated) the default field-by-field copy constructor will break the circularity. Nor it is possible to define a specific copy constructor that restores the circular references in the new copy.

The only possibility is to forbid the copy of circular objects, and to actually share them when continuations are duplicated. This implies that circular objects must be allocated in the heap, that they must provide a reference-counter for keeping track of their sharing, and that they must be managed by a special `RefTerm` class which stores a pointer to such a heap allocated object forwarding any call to it. The `RefTerm` is similar to `WeakRefTerm` as far as the expression semantics is concerned. In addition, it acts as a smart pointer that increases and decreases the reference counter appropriately and eventually releases the circular object when it is no longer used.

5.2 Expression templates

The stateful implementation also allows us to define a set of overloaded operators that can be used to construct complex path expression types (and corresponding instances) in a transparent way. In our implementation we have overloaded the infix operators >> and `|` to be used for composition and alternatives, respectively, and the prefix operators `*` and + to be used for "zero or more" and "one or more" closure operators, respectively. We have also overloaded the bracketing operator `[]` to implement qualifiers (these are special filters that verify a structural predicate, similar to XPath qualifiers) and the function application operator `()` that, given a path expression e and a node x, starts up the compilation process and evaluates e starting from x.

Example 4. The following C++ expression implements the `following` axes as defined in the example 2, assuming that the `Parent`, `NextSibling`, and `FirstChild` terms have been defined with their intuitive meaning, and evaluates the expression from the node x:

```
(*atom(Parent()) >> +atom(NextSibling())
 >> *atom(FirstChild()) >> *atom(NextSibling()))(x)
```

5.3 Usage patterns

Once the basic framework has been designed and implemented, it is possible to add atoms to perform more specific operations.

Example 5 (pattern matching). To test whether the data structure matches a pattern (specified as a regular path expression) p from a node x, we just write the statement

```
if (p(x)) { /* there is a match */ }
```

Example 6 (collecting). In order to collect all the nodes selected by a regular path expression from a node x, we introduce a filter atom `Sink` that stores each node that has been encountered. The filter does not propagate nodes encountered in an earlier traversal:

```
struct Sink {
  Object* walk(Object* x) const {
    if (sink.find(x) == sink.end()) {
      sink.add(x);
      return x;
    } else return 0;
  }
  std::set<Object*> sink;
};
```

After the visit is completed, the `Sink` term can be queried to retrive the set of collected nodes:

```
Sink sink;
(p >> atom(sink) >> empty())(x);
/* do something with sink */
```

where the expression `empty()` is the library's implementation of the **0** atom.

Example 7 (visitor). To perform a user-provided operation on each node selected by a regular path expression from a node x (without necessarily collecting the visited nodes), a visitor class is implemented as an atom that always fails, thus forcing the backtracking algorithm to search for any possible alternative path:

```
template <typename Object> struct Visitor {
  bool walk(Object* x) const {
    /* visit x */
    return false;
  }
};
```

Example 8 (unique visitor). If the data structure contains cycles, and more generally if one wants to be sure that the each selected node is not visited more than once, a sink term can be composed just before the `Visitor` atom:

```
Visitor visitor;
(p >> atom(Sink()) >> atom(visitor)))(x);
```

6 Related work

The compilation framework that we have presented builds on top of a standard C++ compiler and heavily relies on template metaprogramming, with no need for external tools. Although this work does not introduce new concepts, the used techniques have been applied in original ways.

Continuations are well-known and date back to the construction of compilers based on Continuation Passing Style (CPS, see [3]) and also to the field of denotational semantics [1,2]. In the latter case, continuations become critical in specifying the semantics of the sequential composition of commands in imperative programming languages with `gotos`. In [1], the semantics of a construct $S_1; S_2$ in an environment ρ and with continuation θ is defined to be $\mathcal{C}[\![S_1; S_2]\!]\ \rho\ \theta = \mathcal{C}[\![S_1]\!]\ \rho\ (\mathcal{C}[\![S_2]\!]\ \rho\ \theta)$ which basically is the same rule for path composition of Table 3, except that in our context the environment ρ plays no role.

The use of C++ templates for metaprogramming is also well-known [4,18], and the synthesis of types from other types using template classes is related to C++ *traits* [6]. In our development it is crucial the capability of class templates to be instantiated everywhere in the source program. The compiler keeps track of which templates have been instantiated, hence it can decide whether to do instantiation or to retrieve a previous instantiation. Compared to higher-order functions in a functional programming language, templates have the advantage that they can be expanded by inlining. The Curiously Recurring Template Pattern (CRTP [7]), which occurs in the bibliography mainly as a twisted mix of genericity and inheritance, is also crucial since it represents the only way to generate implicitly recursive functions (where the recursive nature is not apparent from the source code). Other approaches that relate templates and functional programming (like the FC++ library [14,16] or the BOOST Lambda library[2]) do not address recursion but rather rely on explicitly recursive functions.

Our use of class templates is just an application of offline partial evaluation [8,9]: C++ may be regarded as a two-level language where template parameters represent statically known values, and method parameters represent dynamically known values. A path expression compiled using the rules in Table 3 results into a function where all the continuation parameters are statically known. The C++ compiler partially evaluates the functions obtained from the compilation process by unfolding the continuations and recursively evaluating the resulting code.

7 Final remarks

There are several contexts where it is desirable to use backtracking algorithms for the evaluation of regular path expressions. The code generated by the PET library is efficient and modern C++ compilers (such as the latest versions of GCC[3] or LLVM[4]) are capable of tail-optimizing function calls and simple (but not trivial) path expressions are compiled as loops instead of recursive function calls. The library is generic in that it makes no assumptions on the data structures being traversed. The user is free to add atoms to fit her own needs, while the library of basic terms can be written once and for all without loosing genericity.

[2] `http://www.boost.org/libs/lambda/doc/`
[3] `http://gcc.gnu.org/`
[4] `http://llvm.cs.uiuc.edu/`

Table 4. Comparing PET against other query engines. The times are in milliseconds and refer to 20 evaluations of the indicated paths, excluding parsing time. The factor f is the ratio $matching\ time/(parsing\ time + matching\ time)$.

	Nodes	PET		Xalan		`libxml2`		`Fxgrep`	
XPath expression	n	ms	f	ms	f	ms	f	ms	f
`//node()`	33806	238	.079	100	.008	18368	.869	4102	.054
`//mrow[@xref]`	750	158	.054	120	.010	3807	.579	4007	.052
`//mrow[@xref]/text()`	3162	161	.054	190	.015	5435	.661	3942	.053
`//text()[../mrow[@xref]]`	3162	202	.068	930	.068	8298	.750	-	-
`//*[@xref][text()]`	2486	147	.050	510	.042	5634	.671	3603	.054
`//text()/../*[@xref]`	2486	175	.059	1220	.092	14729	.824	-	-

We have made some (non-exhaustive) comparative tests of the PET library against the implementations of XPath provided by Xalan[5] and `libxml2`[6] and against the `Fxgrep` XML querying tool [20]. It should be kept in mind that the compared libraries have very different architectures. While PET produces native code, `Fxgrep` translates paths into finite state automata, and `libxml2` and Xalan provide interpreters for XPath expressions. Table 4 shows the absolute times spent for the matching phase, as well as the ratio given by the matching time over the total time (parsing and matching). This way, we have tried to give a performance score that roughly measures the matching algorithm regardless of the implementation language and architecture.

By looking at the absolute times PET outperforms the other tools in most cases. We have to remark that while PET, Xalan, and `libxml2` are C/C++ libraries, `Fxgrep` is written in SML/NJ (Standard ML of New Jersey) and, as the author of `Fxgrep` has recognized, this might be the main cause of its modest absolute performance. By looking at the f ratio, PET performances are rather good. The important thing to notice in this case is how much the ratio f varies, that is how much the absolute time depends on the particular query. On one side we have `Fxgrep`, which is almost unaffected by the kind of query, and this is consequence of the fact that `Fxgrep` uses an optimized automaton that scans the whole document. On the opposite side, Xalan and particularly `libxml2` show a significant variance. In these tools the time spent on union operations on node-set becomes predominant in certain queries. In some cases `libxml2` spends more than 90% of the total time merging node sets. The cost of computing a node set is also relative to whether we are actually interested to know *which* nodes have been selected, or just if *some* node has been selected (this is relevant in XPath when a path occurs within a qualifier). The PET library roughly sits in between these two situations, and proves to be an effective and cheap tool for the programmer.

It is also important to remark that while the other tools are targeted to XML processing, PET is completely generic and provides an easily extensible compilation framework that can be adapted to specific tasks.

[5] C++ version, see `http://xml.apache.org/xalan-c/`
[6] `http://xmlsoft.org/`

References

1. Stoy, J. E. "*Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*", MIT Press, Cambridge, Massachusetts, 1977.
2. Schmidt, D. A. "*Denotational Semantics: A Methodology for Language Development*", Wm. C. Brown Publishers, 1988.
3. Appel, A. "*Compiling with Continuations*", Cambridge University Press, 1992.
4. Veldhuizen, T. "*Using C++ Template Metaprograms*", C++ Report Vol. 7 No. 4, pp. 36–43, May 1995.
5. Veldhuizen, T. "*Expression Templates*", C++ Report, Vol. 7 No. 5, pp. 26–31, June 1995.
6. Myers, N. "*A new and useful template technique: Traits*", C++ Report, Vol. 7 No. 5, pp. 33–35, June 1995.
7. Coplien, J. O. "*Curiously Recurring Template Patterns*", in Stanley B. Lippman, editor, C++ Gems, 135–144. Cambridge University Press, New York, 1996.
8. Jones, N. D. "*An introduction to partial evaluation*", ACM Computing Surveys 28, 3, pp. 480–503, September 1996.
9. Veldhuizen, T. "*C++ templates as partial evaluation*", in ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipula-tion (PEPM 1998), pp. 13–18, San Antonio, TX, USA, January, 1999.
10. Wadler, P. "*A formal semantics of patterns in XSLT*", in B. Tommie Usdin, D. A. Lapeyre, and C. M. Sperberg-McQueen, editors, Proceedings of Markup Technologies, Philadelphia, 1999.
11. Clark, J., and DeRose, S. "*XML Path Language (XPath)*", W3C Recommendation, 1999, `http://www.w3.org/TR/xpath`
12. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. "*Tree Automata Techniques and Applications*", 1999, `http://www.grappa.univ-lille3.fr/tata`
13. Eisenecker, U. W., Czarnecki, K. "*Generative Programming: Methods, Tools, and Applications*", Addison-Wesley, 2000.
14. Mcnamara, B., Smaragdakis, Y. "*Functional Programming in C++ using the FC++ Library*", ACM SIGPLAN Notices, 36(4), pp. 25–30, April 2001.
15. Hosoya, H., Pierce, B. "*Regular expression pattern matching for XML*", in Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 67–80, 2001.
16. Mcnamara, B., Smaragdakis, Y., "*Functional Programming with the FC++ Library*", under consideration for publication in Journal of Functional Programming, July 2002.
17. Clark, J., and DeRose, S. "*XML Path Language (XPath) 2.0*", W3C Working Draft, 2002, `http://www.w3.org/TR/xpath20`
18. Vandevoorde, D., Josuttis, N. M. "*C++ Templates: The Complete Guide*", Addison-Wesley, 2002.
19. Boag S. et al., "*XQuery 1.0: An XML Query Language*", W3C Working Draft, November 2003, `http://www.w3.org/TR/xquery/`
20. Neumann, A., Berlea, A., and Seidl, H. "*`fxgrep`, The Functional XML Querying Tool*", `http://www.informatik.uni-trier.de/~aberlea/Fxgrep/`
21. Leroy X. et al., "*Objective Caml*", `http://caml.inria.fr/ocaml/`