

The Path Expression Template Library

<http://www.sti.uniurb.it/padovani/Software/PET/index.html>

Luca Padovani
University of Urbino
padovani@sti.uniurb.it

```
struct Node
{
    Node(int _label, Node* _first_child,
         Node* _prev, Node* _next)
        : label(_label),
          first_child(_first_child),
          prev(_prev), next(_next)
    { }

    int label;
    Node* first_child;
    Node* prev;
    Node* next;
};
```

Figure 1: A node in a linked data structure.

```
void
search_leaves_aux(Node* p,
                  std::set<Node*>& leaves)
{
    if (p) {
        if (!p->first_child) leaves.insert(p);
        for (Node* q = p->first_child; q;
             q = q->next)
            search_leaves_aux(q, leaves);
    }
}

std::set<Node*>
search_leaves(Node* p)
{
    std::set<Node*> leaves;
    search_leaves_aux(p, leaves);
    return leaves;
}
```

Figure 2: Handmade traversal function.

1 Introduction

The traversal of linked data structures underlies operations like pattern recognition, node visiting, iteration, querying. The Path Expression Template library (PET for short) is a library of C++ templates for the representation of regular path expressions and their evaluation by means of a backtracking algorithm.

Consider a linked data structure, like for instance a generic tree, whose nodes may be represented by the `Node` structure shown in Figure 1 and suppose that you want to collect all the leaves of a tree whose root node is stored in a variable `x`. You might write a function like `search_leaves` in Figure 2, and call it with `x` as argument.

The `search_leaves` function is not particularly complex, because the nodes you are looking for – the leaves – are easily identifiable in a local way, just by looking at the `first_child` field. There are many cases, however, when the nodes one wants to find are identified by a more complex relationship with their neighborhood. Consider for example the set of nodes that are grandparents of a leaf: the code for this query is substantially more intricate, although structurally the query does not involve any notion that has not been considered for the imple-

mentation of `search_leaves`.

As the structure of each `Node` is fixed, and thus there is a finite number of ways you may move from a `Node` to one of its neighbors, it would be nice to have a modular way to specify complex traversal operations, without having to write explicit code for each one of them.

2 Regular expression paths

You can think of a regular path expression as a declarative specification of a traversal in a linked data structure. The evaluation of a path expression determines the nodes reached (or “selected”) by a path expression when starting from a given node `x`.

The most basic path expressions are *selectors* and *filters*. Selectors represent the basic moves, that is how the structure can be navigated by following one of the labelled links available at its nodes. In `Node` structure of Figure 1 the selectors are `first_child`, `next`, and `prev`. Clearly, the set of available selectors depends on the particular data structure, but there are two selectors that can always

be provided. The self selector, notation 1, is the “don’t move” path that selects the current node. On the other hand, the null selector, notation 0, selects nothing. Later sections show why these two apparently useless selectors are necessary.

Filters behave like the 1 or 0 selectors, depending on whether a node of the data structure satisfies a condition or not. Examples of such conditions are “being a leaf”, “having label x ”, “having a left sibling”, and so on. Here too the set of available filters depends on the data structure and on the kind of queries that one is interested to specify.

Compound path expressions build more complicated paths on top of simpler ones. The compound path `(next next)` specifies a traversal operation that follows the `next` link twice. In other words, when evaluated from a node x , it is equivalent to

```
x->next ? x->next->next : NULL
```

A choice path expression `(next | prev)` specifies a path that selects both the left and the right siblings of a given node. Choice paths are a little more complicated to understand as they give the impression that a path expression yields more than one result, something which is nonsense for plain arithmetic expressions. There are two interpretations for these paths. The first interpretation is set-oriented: the evaluation of a path expression from a starting node x yields the whole set of nodes that are reachable from x through the specified path. The second interpretation is operational: whenever there is a choice between two paths, the evaluation tries to follow one of them and, in case of failure (that is, if no node is selected through the first choice), it backtracks and tries the second path. Both interpretations are correct and, in fact, strictly related to each other.

According to the set-oriented interpretation, the evaluation of `(next | prev)` selects the immediately following and immediately preceding siblings of a node x , or only one of them or none, depending on the actual siblings of x . According to the operational interpretation, the evaluation of `(next | prev)` tries to follow the `next` path and succeeds if this leads to a node, otherwise it tries the `prev` path.

One can mix sequential and choice paths arbitrarily. For instance

```
first_child (1 | next | (next next))
```

selects the first three child nodes of a node x .

The main limitation of the constructs described so far is that it is not possible to specify a (sub)path that repeats itself an arbitrary number of times. It is not possible, for instance, to specify a path expression

that selects all the child nodes of a `Node`, unless a `Node` can only have a fixed number of children, like in a binary tree. The last form of compound path expression, repetition, enables this possibility. A path expression of the form

```
first_child (next)*
```

says to follow the `first_child` link and then to follow the `next` link zero or more times. One way to think of the `*` operator is that it finitely describes an infinite sequence of choice paths, that is `(next)*` is equivalent to

```
1 | next | (next next) | (next next next)
| ...
```

This observation may help understanding the compact path

```
(first_child (next))*
```

which stands for

```
1 | (first_child (1 | next | ...))
| (first_child (1 | next | ...))
| first_child (1 | next | ...)
| ...
```

that is a path that selects every node that is a descendant of a given node x .

A variant of `*` is the `+` operator, which says to repeat a path one or more times, thus

```
first_child (next)+
```

selects all the child nodes of a given node x , but the first one (the `next` link must be followed at least once). Note that `+` can be derived from `*` and sequential composition, as `(next)+` is equivalent to `(next (next)*)`.

3 Specifying generic path expressions with the PET library

The PET library provides a set of template classes and overloaded operators for specifying regular path expressions directly in the C++ source code of your programs. The library takes care of evaluating the expressions, leaving the programmer free of thinking to the traversal in terms of high level operations, without having to manually code the operations themselves.

The PET library is completely generic: it works with any data structure that you may possibly define, and the data structure need not be homogeneous (that is, it can consist of nodes with different types). Of course, the PET library cannot automatically figure out how to move across your structure:

```

namespace pet {
    template <> struct Traits<Node> {
        typedef Node* type;
        typedef Node* type_opt;
        static type_opt none(void)
        { return 0; }
        static type_opt some(type x)
        { return x; }
        static bool is_none(type_opt x)
        { return x == none(); }
        static type_opt of_opt(type_opt x)
        { return x; }
    };
}

```

Figure 3: Trait for Node.

```

Node* first_child(Node* p)
{ return p->first_child; }

```

```

Node* next(Node* p)
{ return p->next; }

```

```

Node* prev(Node* p)
{ return p->prev; }

```

Figure 4: Definition of selectors as functions.

you, as the programmer, have to provide information about the selectors and filters that are available. Also, the library needs to know what it means for a reference to a given node x to be meaningful or “null”. In some cases, e.g. when smart pointers are used, it is not always the case that a null link is represented as the `NULL` constant.

Figure 3 shows how to instruct the PET library about the nature of your data structure’s nodes. You define, in the `pet` namespace, one trait class for each kind of node found in the structure. You have to provide two member types: `type` for the type of valid (non-null) references, and `type_opt` for the type of possibly null references. These two types need not coincide: for instance, the default traits set `type` to `Object&` and `type_opt` to `Object*`. You also have to provide actual representations for possibly null references: `none()` returns a null reference, and `some(x)` builds an optional reference from a non-null one. In Figure 3 `some` is the identity function because `type` and `type_opt` coincide. The `is_none(x)` method returns true if x happens to be a null reference, while the `of_opt(x)` method returns a non-null reference of a possibly null reference x which is known not to be null. Data structures that use plain C++ pointers, like `Node`, may safely adopt this trait class.

There are two different ways to specify selectors: you may declare a bunch of functions whose only purpose is to select one of the links of the data struc-

```

struct Atom
{
    typedef Node in;
    typedef Node out;
};

struct FirstChild : public Atom
{
    Node* walk(Node* p) const
    { return p->first_child; }
};

struct Next : public Atom
{
    Node* walk(Node* p) const
    { return p->next; }
};

struct Prev : public Atom
{
    Node* walk(Node* p) const
    { return p->prev; }
};

```

Figure 5: Definition of selectors as classes.

```

bool is_leaf(Node* p)
{ return !p->first_child; }

struct IsLeaf : public Atom
{
    Node* walk(Node* p) const
    { return p->first_child ? 0 : p; }
};

```

Figure 6: Definition of filters.

ture (Figure 4) or you may define, for each selector, an atom class with a `walk` method that does essentially the same thing (Figure 5). Atom classes have to provide two member types `in` and `out` specifying what is the type of the objects “entering” the atom, and what is the type of objects “selected” by the atom. Since in this example the types are all the same, I preferred factoring out these definitions in a common superclass called `Atom`.

The two ways of specifying selectors are not entirely equivalent. Selector classes may be optimized more efficiently by some C++ compilers and they may be used for specifying stateful selectors more conveniently. More details on statefulness are given in a later section.

Likewise selectors, filters too can be specified as functions or as atom classes. Figure 6 shows the implementation of an “is leaf” filter, either as a function or as a class.

4 Building path expressions with PET

For a selector function to be used like an atomic path expression, one writes

```
pet::fun<Node>(next)
```

assuming that `next` accepts and returns references to the same structure `Node`. The more general version

```
pet::fun<NodeIn,NodeOut>(...)
```

is also available. For filter functions one writes instead

```
pet::when<Node>(is_leaf)
```

For an atom class to be used like an atomic path expression, one writes

```
pet::atom(Next())
```

or

```
pet::atom(IsLeaf())
```

The PET library provides overloaded operators for specifying compound path expressions. The operator `>>` denotes sequential composition so that

```
pet::fun<Node>(next) >> pet::fun<Node>(next)
```

represents the expression `(next next)`. The operator `|` creates choice paths:

```
pet::fun<Node>(next) | pet::fun<Node>(prev)
```

Path repetition is accomplished by using the unary `*` and `+` operators. The difference with respect to the syntax of path expressions introduced previously is that these are prefix C++ operators. The expression `(first_child (next) *) *` is thus represented as

```
*(pet::fun<Node>(first_child)
  >> *(pet::fun<Node>(next)))
```

or equivalently, using atom classes, as

```
*(pet::atom(FirstChild())
  >> *(pet::atom(Next())))
```

Once a path expression `e` is created, its evaluation is triggered by supplying the starting node. Thus `e(x)` is a valid C++ expression whose result is a possibly null reference to a node selected by `e` starting from `x`. The evaluation of `e` stops as soon as it reaches a node, and a reference to that node is returned. If no node can be reached from `x` through `e`, `e(x)` returns a null reference.

```
bool has_label_7(Node* p)
{ return p->label == 7; }

bool has_label_4(Node* p)
{ return p->label == 4; }

struct HasLabel
{
  HasLabel(int _label) : label(_label) { }

  Node* walk(Node* p) const
  { return (p->label == label) ? p : 0; }

  int label;
};
```

Figure 7: Stateless versus stateful atoms.

```
std::set<Node*>
search_leaves_with_regular_path(Node* p)
{
  pet::Sink<Node> leaves;
  (* (pet::atom(FirstChild())
    >> * (pet::atom(Next()))
    >> pet::atom(IsLeaf())
    >> pet::sink(leaves)) (p);
  return leaves.getSink();
}
```

Figure 8: Stateless versus stateful atoms.

5 Stateful atoms

The atoms seen so far are stateless in the sense that they do not depend on anything but the node of the data structure they receive as argument. It is often desirable to access information that is not directly available from the node itself. Imagine that you need a set of filters, each selecting nodes with a particular label. One possibility is to provide one separate filter for each label, but this solution does not scale up. It is more convenient to design a generic filter atom that tests whether the label is equal to some constant, and have such constant stored as a parameter of the atom. Figure 5 compares the two alternatives.

Using `HasLabel` it is now possible to write

```
pet::atom(HasLabel(7))
  | pet::atom(HasLabel(4))
```

for the path expression that selects those nodes whose label is either 7 or 4.

Stateful atoms can also affect the outer state as a consequence of their being traversed. Say you need to keep track of any node that is selected at some point in a path expression: you may design an atom that stores the reference that is passed to its walk method in some global variable. Even better, you may let the atom encapsulate a container that

```

std::set<Node*>
search_leaves_with_regular_path(Node* p)
{
    pet::Sink<Node> leaves;
    (*pet::atom(FirstChild()))
    >> *(pet::atom(Next()))
    >> pet::atom(IsLeaf())
    >> pet::sink(leaves)
    >> pet::empty<Node>()(p);
    return leaves.getSink();
}

```

Figure 9: Implementation of `search_leaves` in PET.

is filled with the visited nodes as evaluation proceeds. As this capability is frequently useful, the PET library provides a `Sink` atom that does exactly this. Figure 8 shows a first attempt to implement an alternative version of the `search_leaves` function that uses regular path expressions with a `Sink` atom. There is only a slight problem: the evaluation of the path expression stops as soon as it finds a leaf. So, the sink will always contain only one node (the leaf that has been reached), and not the set of all the leaves of the tree. This is a consequence of the operational interpretation of path expressions.

To recover the set-based interpretation, one can place a null selector at the end of the path expression. The null selector will cause the evaluation to ultimately fail, but as a side effect it will force it try any possible path from the root `x` to one of the leaves, and all the visited leaves will in fact be remembered in the sink. Figure 9 shows the correct implementation of `search_leaves` with a regular path.

6 Tips and pitfalls

The PET library makes heavy use of template meta-programming techniques. While the code generated from path expressions can be very efficient (modern C++ compilers are capable of optimizing evaluation of several classes of path expressions so that they are compiled as inline loops instead of recursive function calls), the compilation of C++ source code using path expressions can take quite some time. A related problem is that error messages from the compiler are somewhat hard to decipher and may end up not being very informative.

It is also mandatory a word of caution on stateful atoms: the order in which a path is traversed cannot be always easily predicted. Stateful atoms, and in particular stateful atoms with side effects, should be designed and used carefully in order not to intro-

duce subtle behaviors during the evaluation of path expressions.

Last but not least, be careful when specifying traversals across cyclic data structures, as these may end up into infinite loops. In this respect, the `Sink` atom is not only useful for optimization purposes, avoiding multiple traversals of a common subpath, but also to break up cycles in the data structure thus ensuring termination of the evaluation.

7 Concluding remarks

As a fully generic library, PET lends itself to be instantiated in the following situations:

Pattern matching. The evaluation of an expression `e` on a node `x` verifies whether the data structure matches the pattern represented by `e` in the proximity of `x`. According to the operational interpretation of path expressions, evaluation stops as soon as a node is selected. This represents a major performance improvement with respect to similar languages (like `XPath`) that usually require the whole set of selected nodes to be computed.

Regular visitor. Given a path expression `e` and a visitor atom class `V`, the path expression

```

(e >> pet::atom(V()))
>> pet::empty<Node>()(x)

```

visits any node that is selected by `e`.

Node selection. The set-based interpretation of regular path expression can be recovered straightforwardly by using the special `Sink` visitor, which stores any visited node in a `std::set` container.

The Appendix shows a full example where a PET path expression is used to query an XML document looking for any text node that is child of an `mrow` element having an `xref` attribute. In other words, the path expression in the query function implements the `XPath` expression

```
//mrow[@xref]/text()
```

The nodes of the XML document have an hypothetical type `XMLNode` and XML strings are represented by the `XMLString` data type.

A detailed description of the internals of the PET library can be found in “Compilation of Generic Regular Path Expressions Using C++ Class Templates”, Proceedings of 14th International Conference on Compiler Construction (CC 2005) LNCS 3443, pp. 27–42, Edinburgh, Scotland, April 2005.

Appendix

```
#include <pet.hh>

namespace pet {
    template <> struct Traits<XMLNode> {
        typedef XMLNode* type;
        typedef XMLNode* type_opt;
        static type_opt none(void)
        { return NULL; }
        static type_opt some(type x)
        { return x; }
        static bool is_none(type_opt x)
        { return x == none(); }
        static type of_opt(type_opt x)
        { return x; }
    };
}

struct Atom
{
    typedef XMLNode in;
    typedef XMLNode out;
};

struct FirstChild : public Atom
{
    XMLNode* walk(XMLNode* x) const
    { return x->children; }
};

struct NextSibling : public Atom
{
    XMLNode* walk(XMLNode* x) const
    { return x->next; }
};

struct NameIs : public Atom
{
    NameIs(const XMLString& _name)
        : name(_name) { }

    XMLNode* walk(XMLNode* x) const
    { return (name == x->name) ? x : 0; }

private:
    XMLString name;
};

struct NamespaceIs : public Atom
{
    NamespaceIs(const XMLString& _ns)
        : ns(_ns) { }

    XMLNode* walk(XMLNode* x) const
    { return (x->ns && ns == x->ns->href)
        ? x : 0; }

private:
    XMLString ns;
};

template <int type>
struct TypeIs : public Atom
{
    XMLNode* walk(XMLNode* x) const
    { return (x->type == type) ? x : 0; }
};

struct HasProp : public Atom
{
    HasProp(const XMLString& _prop) : prop(_prop)
    { }

    XMLNode* walk(XMLNode* x) const
    { return xmlHasProp(x, prop) ? x : 0; }

private:
    XMLString prop;
};
```

```
std::set<XMLNode*>
query(XMLNode* root)
{
    pet::Sink<XMLNode> nodes;
    ((*((pet::atom(FirstChild()))
    >> *(pet::atom(NextSibling()))))
    >> pet::atom(TypeIs<XML_ELEMENT_NODE>())
    >> pet::atom(NamespaceIs("..."))
    >> pet::atom(NameIs("mrow"))
    >> pet::atom(HasProp("xref"))
    >> pet::sink(nodes)
    >> pet::empty<XMLNode>())(root);
    return nodes.getSink();
}
```