

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

Luca Padovani

`lpadovan@cs.unibo.it`

Department of Computer Science, University of Bologna

Outline

- ⊗ motivation
- ⊗ syntax and semantics of regular path expressions
- ⊗ compilation
- ⊗ stateful path expressions
- ⊗ examples
- ⊗ concluding remarks

<http://www.cs.unibo.it/~lpadovan/software/PET/>

Problem scenario

Goal: enriching C++ with a DSL for the **evaluation of regular path expressions**

- ✿ **library of classes and objects** in the host language: interpretation versus compilation, performance issues
- ✿ **external** feature: external code generator, constrained by C++ syntax, loose integration between the DSL and the host language
- ✿ **internal** feature: let the C++ compiler translate it using templates (partial evaluation)

Generic regular paths: syntax

$$\begin{array}{l}
 \langle expr \rangle ::= a \qquad a \in \langle atom \rangle \\
 \quad | e_1 | e_2 \\
 \quad | e_1 e_2 \\
 \quad | e^* \\
 \quad | e? \quad \mathbf{1} | e \\
 \quad | e^+ \quad ee^* \\
 \quad | e^n \quad e^n \qquad 0 \leq n \\
 \quad | e^{n,m} \quad e^n | e^{n+1} | \dots | e^m \quad 0 \leq n \leq m \\
 \\
 \langle atom \rangle ::= \mathbf{0} \\
 \quad | \mathbf{1} \\
 \quad | \dots
 \end{array}$$

Examples

Plain regular expressions:

$$ab^* \Rightarrow \text{char_is_a (next_char char_is_b)^*}$$

XPath axes:

`self` \Rightarrow `1`

`child` \Rightarrow `first_child next_sibling*`

`descendant` \Rightarrow `(first_child next_sibling*)+`

`ancestor` \Rightarrow `parent+`

`following_sibling` \Rightarrow `next_sibling+`

`following` \Rightarrow `parent* next_sibling+ (first_child next_sibling*)*`

Generic regular paths: semantics

$\mathcal{S}[e]$: *Object* \rightarrow *Object set*

$$\mathcal{S}[\mathbf{0}]x = \emptyset$$

$$\mathcal{S}[\mathbf{1}]x = \{x\}$$

$$\mathcal{S}[a]x = a(x)$$

$$\mathcal{S}[e_1 \mid e_2]x = \mathcal{S}[e_1]x \cup \mathcal{S}[e_2]x$$

$$\mathcal{S}[e_1 e_2]x = \mathcal{S}[e_2]\mathcal{S}[e_1]x = \bigcup_{y \in \mathcal{S}[e_1]x} \mathcal{S}[e_2]y$$

$$\mathcal{S}[e^*]x = \bigcup_{i=0}^{\infty} \mathcal{S}[e^i]x$$

Towards a functional semantics

$\mathcal{F}_0[e] : Object \rightarrow Object \text{ set}$

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

Cannot avoid the set-union operation as long as e_1 and e_2 are evaluated in complete isolation

$$(e_{11} \mid e_{12}) e_2$$

Idea: whenever e_1 selects a node proceed straight along e_2
if this fails, **backtrack** and search the next node selected by e_1

Functional semantics with continuations

$$\mathcal{F}[e] \quad : \quad (\text{Object} \rightarrow \text{Object option}) \rightarrow \text{Object} \rightarrow \text{Object option}$$

$$\mathcal{F}[e] \ k \ x \quad = \quad \text{“if } x \xrightarrow{e} \text{ Some } y \text{ then } (k \ y) \text{ else None”}$$

$$\text{null} \quad \equiv \quad \lambda x. \text{None}$$

$$\text{id} \quad \equiv \quad \lambda x. \text{Some } x$$

$$\text{fork} \quad \equiv \quad \lambda k_1. \lambda k_2. \lambda k_3. \lambda x. \text{match } (k_1 \ x) \text{ with}$$

$$\qquad \qquad \text{Some } y \quad \rightarrow \quad (k_2 \ y)$$

$$\qquad \qquad \text{None} \quad \rightarrow \quad (k_3 \ x)$$

$$\mathcal{A}[\mathbf{0}] \quad = \quad \text{null}$$

$$\mathcal{A}[\mathbf{1}] \quad = \quad \text{id}$$

$$\mathcal{A}[a] \quad = \quad \lambda x. a(x)$$

$$\mathcal{F}[a] \quad = \quad \lambda k. (\text{fork } \mathcal{A}[a] \ k \ \text{null})$$

$$\mathcal{F}[e_1 \mid e_2] \quad = \quad \lambda k. (\text{fork } (\mathcal{F}[e_1] \ k) \ \text{id} \ (\mathcal{F}[e_2] \ k))$$

$$\mathcal{F}[e_1 e_2] \quad = \quad \lambda k. (\mathcal{F}[e_1] \ (\mathcal{F}[e_2] \ k))$$

$$\mathcal{F}[e^*] \quad = \quad \lambda k. (\text{fix } \lambda f. (\text{fork } k \ \text{id} \ (\mathcal{F}[e] \ f))) \quad \text{non-greedy!}$$

Implementation

Straightforward if the host language is functional

Key observation: continuations are statically applied

$$\begin{aligned}\mathcal{F}[e_1 \mid e_2] &= \lambda k.(\text{fork } (\mathcal{F}[e_1] k) \text{ id } (\mathcal{F}[e_2] k)) \\ \mathcal{F}[e_1 e_2] &= \lambda k.(\mathcal{F}[e_1] (\mathcal{F}[e_2] k)) \\ \mathcal{F}[e^*] &= \lambda k.(\text{fix } \lambda f.(\text{fork } k \text{ id } (\mathcal{F}[e] f)))\end{aligned}$$

The plan

- ⊗ represent terms as C++ types
- ⊗ use templates for continuation abstraction
- ⊗ use (static) class methods for *Object* abstraction

Implementation of basic terms

```

null  ≡  λx.None
id    ≡  λx.Some x
fork  ≡  λk1.λk2.λk3.λx.match (k1 x) with
                                     Some y  →  (k2 y)
                                     None    →  (k3 x)

```

```

struct NullTerm
{ static Object* walk(Object*) { return 0; } };

struct IdTerm
{ static Object* walk(Object* x) { return x; } };

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

```

Implementation of basic terms

```

null  ≡  λx.None
id    ≡  λx.Some x
fork  ≡  λk1.λk2.λk3.λx.match (k1 x) with
                                     Some y  →  (k2 y)
                                     None    →  (k3 x)

```

```

struct NullTerm
{ static Object* walk(Object*) { return 0; } };

```

```

struct IdTerm
{ static Object* walk(Object* x) { return x; } };

```

```

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

```

Implementation of basic terms

```

null  ≡  λx.None
id    ≡  λx.Some x
fork  ≡  λk1.λk2.λk3.λx.match (k1 x) with
                                     Some y  →  (k2 y)
                                     None    →  (k3 x)

```

```

struct NullTerm
{ static Object* walk(Object*) { return 0; } };

```

```

struct IdTerm
{ static Object* walk(Object* x) { return x; } };

```

```

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

```

Example

```
struct Parent {
    static Object* walk(Object* x) { return x->parent; }
};
```

parent (parent | 1)

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

- ⊗ inconvenient ☹
- ⊗ what about the fix term?
- ⊗ let's have the compiler synthesize these types!

Example

```
struct Parent {
    static Object* walk(Object* x) { return x->parent; }
};
```

parent (parent | 1)

```
ForkTerm<Parent,
    ForkTerm<ForkTerm<Parent, k, NullTerm>,
        IdTerm,
        ForkTerm<IdTerm, k, NullTerm> >,
    NullTerm>::walk(x)
```

- ⊗ inconvenient ☹
- ⊗ what about the fix term?
- ⊗ let's have the compiler synthesize these types!

Example

```
struct Parent {
    static Object* walk(Object* x) { return x->parent; }
};
```

parent (parent | 1)

```
ForkTerm<Parent,
    ForkTerm<ForkTerm<Parent, k, NullTerm>,
        IdTerm,
        ForkTerm<IdTerm, k, NullTerm> >,
    NullTerm>::walk(x)
```

- ⊗ inconvenient ☹
- ⊗ what about the *fix* term?
- ⊗ let's have the compiler synthesize these types!

Compiling atomic paths

```
struct Parent {
    static Object* walk(Object* x) { return x->parent; }
};
```

$$\mathcal{F}[a] = \lambda k.(\text{fork } \mathcal{A}[a] \ k \ \text{null})$$

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

⊗ AtomPath< a > represents a

⊗ AtomPath< a >::Compile< k >::RES represents (fork $\mathcal{A}[a]$ k null)

Compiling choice paths

$$\mathcal{F}[e_1 \mid e_2] = \lambda k.(\text{fork } (\mathcal{F}[e_1] k) \text{ id } (\mathcal{F}[e_2] k))$$

```
template <typename E1, typename E2>
struct ChoicePath {
  template <typename K>
  struct Compile {
    typedef typename E1::template Compile<K>::RES T1;
    typedef typename E2::template Compile<K>::RES T2;
    typedef ForkTerm<T1, IdTerm, T2> RES;
  };
};
```

- ⊗ ChoicePath< e_1, e_2 > represents $e_1 \mid e_2$
- ⊗ ChoicePath< e_1, e_2 >::Compile< k >::RES represents
(fork ($\mathcal{F}[e_1] k$) id ($\mathcal{F}[e_2] k$))

Compiling sequential paths

$$\mathcal{F}[[e_1 e_2]] = \lambda k. (\mathcal{F}[[e_1]] (\mathcal{F}[[e_2]] k))$$

```
template <typename E1, typename E2>
struct SeqPath {
    template <typename K>
    struct Compile {
        typedef typename E2::template Compile<K>::RES T1;
        typedef typename E1::template Compile<T1>::RES RES;
    };
};
```

⊗ SeqPath< e_1, e_2 > represents $e_1 e_2$

⊗ SeqPath< e_1, e_2 >::Compile< k >::RES represents $(\mathcal{F}[[e_1]] (\mathcal{F}[[e_2]] k))$

Compiling repeated paths

$$(\text{fix } F) = (F (\text{fix } F))$$

```
template <template <typename> class F>
struct FixTerm : public F<FixTerm<F> > { }; /* CRTP! */
```

$$\mathcal{F}[[e^*]] = \lambda k.(\text{fix } F)$$

$$F \equiv \lambda f.(\text{fork } k \text{ id } (\mathcal{F}[[e]] f))$$

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

Compiling repeated paths

$$(\text{fix } F) = (F (\text{fix } F))$$

```
template <template <typename> class F>
struct FixTerm : public F<FixTerm<F> > { }; /* CRTP! */
```

$$\mathcal{F}[[e^*]] = \lambda k.(\text{fix } F)$$

$$F \equiv \lambda f.(\text{fork } k \text{ id } (\mathcal{F}[[e]] f))$$

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

Example

The regular path expression

parent (parent | 1)

- ⊗ as a type representing its structure

```
SeqPath<AtomPath<Parent>,
        ChoicePath<AtomPath<Parent>,
                  AtomPath<IdTerm>
        >
>::Compile<k>::RES
```

- ⊗ as a type representing its semantics

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

Example

The regular path expression

parent (parent | 1)

- ⊗ as a type representing its **structure**

```
SeqPath<AtomPath<Parent>,
      ChoicePath<AtomPath<Parent>,
                AtomPath<IdTerm>
      >
>::Compile<k>::RES
```

- ⊗ as a type representing its **semantics**

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

Atoms with state

Since atoms are implemented as **static methods in template classes** they can only access global variables or template parameters

⇒ poor support for localized state management

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

Atoms with state: implementation

```
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)
        { }

    static 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;
};
```

Atoms with state: issues

Making atoms real objects instead of types is possible but technically involved because of **repeated paths**:

- ⊗ avoiding paradoxical self-containment (instances versus references)
- ⊗ preserving cyclic instances with copy constructors (sharing)
- ⊗ memory allocation (reference counting)
- ⊗ . . .

Example: regular expression templates

The regular path expression

$$parent^* next_sibling^+ (first_child next_sibling^*)^*$$

⊗ as a type representing its structure

```
Seq<Star<Atom<Parent>>,
    Seq<Plus<Atom<NextSibling>>,
        Star<Seq<Atom<FirstChild()>,
            Star<Atom<NextSibling>>>>
    >
>
```

⊗ as an object

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

Example: regular expression templates

The regular path expression

$$parent^* next_sibling^+ (first_child next_sibling^*)^*$$

⊗ as a type representing its structure

```
Seq<Star<Atom<Parent>>,
    Seq<Plus<Atom<NextSibling>>,
        Star<Seq<Atom<FirstChild()>,
            Star<Atom<NextSibling>>>>
    >
>
```

⊗ as an **object**

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

Usage patterns

✧ Pattern matching

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

✧ Traversal

```
if (Object* y = p(x)) { /* there is a path  $x \xrightarrow{p} y$  */ }
```

✧ Selection

```
Sink sink;  
(p >> atom(sink) >> empty())(x);
```

✧ Visit

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

✧ Unique visit

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

Generated code (LLVM)

next (value_is 7) next (value_is 4)*

```

linkonce %search74(%List* %x)
entry:
    br label %tailrecurse
tailrecurse:
    %x.tr = phi %List* [ %x, %entry ], [ %tmp.2.i, %else ]
    %tmp.1.i.i = getelementptr %List* %x.tr, int 0, uint 0
    %tmp.2.i.i = load int* %tmp.1.i.i
    %tmp.3.i.i = seteq int %tmp.2.i.i, 7
    %tmp.5.i.i = select bool %tmp.3.i.i, %List* %x.tr, %List* null
    %tmp.2.i.i = seteq %List* %tmp.5.i.i, null
    br bool %tmp.2.i.i, label %else, label %then.i
then.i:
    %tmp.1.i.i.i = getelementptr %List* %tmp.5.i.i, int 0, uint 0
    %tmp.2.i.i.i = load int* %tmp.1.i.i.i
    %tmp.3.i.i.i = seteq int %tmp.2.i.i.i, 4
    %tmp.5.i.i.i = select bool %tmp.3.i.i.i, %List* %tmp.5.i.i, %List* null
    %bothcond = seteq %List* %tmp.5.i.i.i, null
    br bool %bothcond, label %else, label %UnifiedReturnBlock
else:
    %tmp.1.i = getelementptr %List* %x.tr, int 0, uint 1
    %tmp.2.i = load %List** %tmp.1.i
    %tmp.2.i.i = seteq %List* %tmp.2.i, null
    br bool %tmp.2.i.i, label %else.i3, label %tailrecurse
else.i3:
    ret %List* null
UnifiedReturnBlock:
    ret %List* %tmp.5.i.i.i

```

Generated code (LLVM)

(left|right) visitor*

```

linkonce %search_leaves(%Node* %x)
entry:
    br label %tailrecurse

tailrecurse:
    %x.tr = phi %Node* [ %x, %entry ], [ %tmp.2.i.i, %else ]
    %tmp.1.i = getelementptr %Node* %x.tr, int 0, uint 1
    %tmp.2.i = load %Node** %tmp.1.i
    %tmp.2.i5 = seteq %Node* %tmp.2.i, null
    br bool %tmp.2.i5, label %else, label %then.i

then.i:
    %tmp.8.i1 = call %search_leaves( %Node* %tmp.2.i )
    %tmp.2.i811 = seteq %Node* %tmp.8.i1, null
    br bool %tmp.2.i811, label %else, label %UnifiedReturnBlock

else:
    %tmp.1.i.i = getelementptr %Node* %x.tr, int 0, uint 2
    %tmp.2.i.i = load %Node** %tmp.1.i.i
    %tmp.2.i4 = seteq %Node* %tmp.2.i.i, null
    br bool %tmp.2.i4, label %else.i5, label %tailrecurse

else.i5:
    ret %Node* null

UnifiedReturnBlock:
    ret %Node* %tmp.8.i1

```

Comparison (absolute)

matching time (milliseconds)

XPath expression	Nodes	PET	Xalan	libxml2	Fxgrep
//node()	33806	238	100	18368	4102
//mrow[@xref]	750	158	120	3807	4007
//mrow[@xref]/text()	3162	161	190	5435	3942
//text()[../mrow[@xref]]	3162	202	930	8298	-
//*[@xref][text()]	2486	147	510	5634	3603
//text()/../*[@xref]	2486	175	1220	14729	-

Concluding remarks

On the library

- ✿ continuations, templates, partial evaluation
- ✿ CRTP for creating implicitly recursive C++ functions
- ✿ good runtime performances

On the compilers

- ✿ long compilation time
- ✿ good to have support for code generation (more control desirable)
- ✿ good generated code (LLVM)

To do

- ✿ greedyness
- ✿ mixing stateless and stateful expressions