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

$$
\begin{array}{rclll}
\langle expr \rangle & ::= & a & & a \in \langle atom \rangle \\
& | & e_1 \mid e_2 & & \\
& | & e_1\, e_2 & & \\
& | & e^* & & \\
& | & e? & \mathbf{1} \mid e & \\
& | & e^+ & ee^* & \\
& | & e^n & e^n & 0 \leq n \\
& | & e^{n,m} & e^n \mid e^{n+1} \mid \cdots \mid e^m & 0 \leq n \leq m \\
\\
\langle atom \rangle & ::= & \mathbf{0} & & \\
& | & \mathbf{1} & & \\
& | & \ldots & &
\end{array}
$$

Examples

Plain regular expressions:

$$ab^* \Rightarrow \mathit{char\_is\_a}\ (\mathit{next\_char}\ \ \mathit{char\_is\_b})^*$$

XPath axes:

`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`	$\Rightarrow$	$\mathit{parent}^*\ \mathit{next\_sibling}^+(\mathit{first\_child}\ \mathit{next\_sibling}^*)^*$

Generic regular paths: semantics

$$
\begin{array}{rcl}
\mathcal{S}[\![e]\!] & : & \mathit{Object} \to \mathit{Object}\ \texttt{set} \\[1em]
\mathcal{S}[\![\mathbf{0}]\!]x & = & \emptyset \\
\mathcal{S}[\![\mathbf{1}]\!]x & = & \{x\} \\
\mathcal{S}[\![a]\!]x & = & a(x) \\[1em]
\mathcal{S}[\![e_1 \mid e_2]\!]x & = & \mathcal{S}[\![e_1]\!]x \cup \mathcal{S}[\![e_2]\!]x \\
\mathcal{S}[\![e_1\, e_2]\!]x & = & \mathcal{S}[\![e_2]\!]\mathcal{S}[\![e_1]\!]x = \bigcup_{y \in \mathcal{S}[\![e_1]\!]x} \mathcal{S}[\![e_2]\!]y \\
\mathcal{S}[\![e^*]\!]x & = & \bigcup_{i=0}^{\infty} \mathcal{S}[\![e^i]\!]x
\end{array}
$$

Towards a functional semantics

$$\mathcal{F}_0[\![e]\!] : \mathit{Object} \to \mathit{Object}\ \texttt{set}$$

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

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

$$
\begin{array}{rcl}
\mathcal{F}[\![e]\!] & : & (\mathit{Object} \to \mathit{Object}\ \texttt{option}) \to \mathit{Object} \to \mathit{Object}\ \texttt{option} \\
\mathcal{F}[\![e]\!]\ k\ x & = & \text{“if } x \xrightarrow{e} \texttt{Some}\ y \text{ then } (k\ y) \text{ else } \texttt{None}\text{”}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{null} & \equiv & \lambda x.\texttt{None} \\
\mathsf{id} & \equiv & \lambda x.\texttt{Some}\ x \\
\mathsf{fork} & \equiv & \lambda k_1.\lambda k_2.\lambda k_3.\lambda x.\texttt{match}\ (k_1\ x)\ \texttt{with} \\
& & \qquad\qquad \begin{array}{rcl} \texttt{Some}\ y & \to & (k_2\ y) \\ \texttt{None} & \to & (k_3\ x) \end{array}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathbf{0}]\!] & = & \mathsf{null} \\
\mathcal{A}[\![\mathbf{1}]\!] & = & \mathsf{id} \\
\mathcal{A}[\![a]\!] & = & \lambda x.a(x)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}[\![a]\!] & = & \lambda k.(\mathsf{fork}\ \ \mathcal{A}[\![a]\!]\ \ k\ \ \mathsf{null}) \\
\mathcal{F}[\![e_1 \mid e_2]\!] & = & \lambda k.(\mathsf{fork}\ \ (\mathcal{F}[\![e_1]\!]\ k)\ \ \mathsf{id}\ \ (\mathcal{F}[\![e_2]\!]\ k)) \\
\mathcal{F}[\![e_1\ e_2]\!] & = & \lambda k.(\mathcal{F}[\![e_1]\!]\ (\mathcal{F}[\![e_2]\!]\ k)) \\
\mathcal{F}[\![e^*]\!] & = & \lambda k.(\mathsf{fix}\ \lambda f.(\mathsf{fork}\ \ k\ \ \mathsf{id}\ \ (\mathcal{F}[\![e]\!]\ f)))\quad \text{non-greedy!}
\end{array}
$$

Implementation

Straightforward if the host language is functional

Key observation: continuations are statically applied

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

The plan

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

Implementation of basic terms

$$
\begin{array}{rcl}
\mathsf{null} & \equiv & \lambda x.\texttt{None} \\
\mathsf{id} & \equiv & \lambda x.\texttt{Some}\ x \\
\mathsf{fork} & \equiv & \lambda k_1.\lambda k_2.\lambda k_3.\lambda x.\texttt{match}\ (k_1\ x)\ \texttt{with} \\
 & & \qquad \begin{array}{rcl} \texttt{Some}\ y & \rightarrow & (k_2\ y) \\ \texttt{None} & \rightarrow & (k_3\ x) \end{array}
\end{array}
$$

```
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

$$
\begin{array}{rcl}
\mathsf{null} & \equiv & \lambda x.\mathtt{None} \\
\mathsf{id} & \equiv & \lambda x.\mathtt{Some}\ x \\
\mathsf{fork} & \equiv & \lambda k_1.\lambda k_2.\lambda k_3.\lambda x.\mathtt{match}\ (k_1\ x)\ \mathtt{with} \\
 & & \qquad \begin{array}{rcl} \mathtt{Some}\ y & \to & (k_2\ y) \\ \mathtt{None} & \to & (k_3\ x) \end{array}
\end{array}
$$

```
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

$$
\begin{array}{rcl}
\mathsf{null} & \equiv & \lambda x.\mathtt{None} \\
\mathsf{id} & \equiv & \lambda x.\mathtt{Some}\ x \\
\mathsf{fork} & \equiv & \lambda k_1.\lambda k_2.\lambda k_3.\lambda x.\mathtt{match}\ (k_1\ x)\ \mathtt{with} \\
 & & \qquad \begin{array}{rcl} \mathtt{Some}\ y & \rightarrow & (k_2\ y) \\ \mathtt{None} & \rightarrow & (k_3\ x) \end{array}
\end{array}
$$

```
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; }
};
```

$$parent\,(parent \mid \mathbf{1})$$

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

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

Example

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

$$parent\,(parent \mid \mathbf{1})$$

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

- unconvenient ☹
- what about the fix term?
- let’s have the compiler synthesize these types!

Example

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

$$parent\,(parent \mid \mathbf{1})$$

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

- unconvenient ☹
- 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; }
};
```

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

```
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 $(\mathsf{fork}\ \ \mathcal{A}[\![a]\!]\ \ k\ \ \mathsf{null})$

Compiling choice paths

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

```
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 $(\mathsf{fork}\ \ (\mathcal{F}[\![e_1]\!]\ k)\ \ \mathsf{id}\ \ (\mathcal{F}[\![e_2]\!]\ k))$

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

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

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

$$\begin{array}{rcl} \mathcal{F}[\![e^*]\!] & = & \lambda k.(\mathsf{fix}\ F) \\ F & \equiv & \lambda f.(\mathsf{fork}\ \ k\ \ \mathsf{id}\ \ (\mathcal{F}[\![e]\!]\ f)) \end{array}$$

```
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

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

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

$$\begin{aligned} \mathcal{F}[\![e^*]\!] &= \lambda k.(\mathsf{fix}\ F) \\ F &\equiv \lambda f.(\mathsf{fork}\ \ k\ \ \mathsf{id}\ \ (\mathcal{F}[\![e]\!]\ f)) \end{aligned}$$

```
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

$$\mathit{parent}\,(\mathit{parent} \mid \mathbf{1})$$

- 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

$$\mathit{parent}\,(\mathit{parent} \mid \mathbf{1})$$

- 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.i1 = seteq %List* %tmp.2.i, null
        br bool %tmp.2.i1, 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