

jInfer Architecture

Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, Matej Vitásek
Advisors: RNDr. Irena Mlýnková, Ph.D., Martin Nečaský, Ph.D.

Praha, 2011

Target audience: developers willing to extend jInfer.

*Note: we use the term **inference** for the act of creation of schema throughout this and other jInfer documents.*

The description of jInfer architecture will commence by describing the data structures, namely representations of regular expressions and XML elements, attributes and simple data.

Afterwards the interfaces of basic inference modules - Initial Grammar Generator, Simplifier and Schema Generator - will be explained.

Finally, the process of inference will be described.

1 Package naming conventions

All packages start with `cz.cuni.mff.ksi.jinfer`. Afterwards is the short, normalized name of the module (e.g. `base`) and finally the package structure in this module (e.g. `objects.utils`). All in all, a package in the Base module could look like `cz.cuni.mff.ksi.jinfer.base.objects.utils`

2 Data structures

2.1 Regular expressions

For general information on regular expressions, please refer to [wik], [HMU01]. All classes pertaining to regular expressions can be found in the package `cz.cuni.mff.ksi.jinfer.base.regexp`. In jInfer, we use extended regular expressions as they give us nicer syntax (and easier programming).

Regular expression is implemented as class `Regexp` with supplementing classes `RegexpInterval` and `RegexpType`. Each `Regexp` instance has one of the enum `RegexpType` type:

- Lambda - empty string (also called ϵ in literature),
- Token - a letter of the alphabet,
- Concatenation - one or more regular expression in an ordered sequence. Eg. (a, b, c, d) ,
- Alternation - a choice between one or more regular expressions. Eg. $(a|b|c|d)$,
- Permutation - shortcut for all possible permutations of regular expressions. Our syntax to write down permutation is $(a\&b\&c\&d)$.

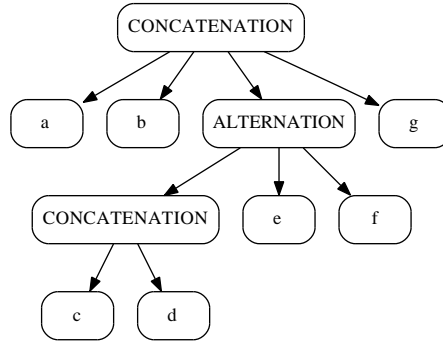
Type of `regexp` is held in `type` member in class `Regexp` and can be tested by calling methods `isLambda()`, `isToken()` etc.

Each `Regexp` instance has one instance of `RegexpInterval` as member. Class `RegexpInterval` represents POSIX-like intervals for expression:

- $a\{m, n\}$ means a at least m -times, at most n -times,
- $a\{m, \}$ means at least m -times (unbounded).

Interval is either bounded (you have to set both lower and upper bound integers), or unbounded (you have to set only lower bound). Testing interval commonly follows routine:

Figure 1: Example tree for regular expression $(a, b, ((c|d), e), f)$



```

RegexpInterval i = r.getInterval();
if (i.isUnbounded()) {
    print(i.getMin());
} else {
    print(i.getMin(), i.getMax());
}

```

That is, first check interval for being unbounded, only if it is bounded, you can ask for maximum.

Class `Regexp` can represent regular expression over any alphabet. This is done by using java generics, `Regexp` is implemented as `Regexp<T>`. Only token regexps hold instance of type `T` in member `content`.

Regular expression is in fact n -ary tree, for example expression $(a, b, ((c|d), e), f)$ can be viewed as in figure 1. We implement this tree by member of `Regexp` class called `children`, which is of type `List<Regexp<T>`. List contains children of regexp in means of regexp tree.

`Regexp` has to obey constraints:

- type, children and interval have to be non-null references,
- when type is lambda, content and interval has to be null,
- when type is token, content has to be non-null,
- when type concatenation, alternation or permutation, content has to be null.

These constraints are checked by constructors, so the best way to construct new regexps is by using methods `getToken()`, `getConcatenation()` etc.

`Regexp` instance is by default created as immutable, that is, once instantiated, you cannot add more children to list of children, cannot change type, content etc. It is to prevent misuse. In special circumstances, one does not know future children of regexp in time of creation. This occurs mainly in input modules, where by parsing XML data sequentially, one does not know contents of element in time of handling start element event. For these cases, special `getMutable()` method is implemented to obtain regexp with none of members set. One has to fill in all properties carefully and call `setImmutable()` afterwards. Proper usage should be one of following:

```

Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.LAMBDA);
r.setImmutable();

```

```

Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.TOKEN);
r.setContent(...)
r.setImmutable();

```

```

Regexp<T> r = Regexp.<T>getMutable();
r.setInterval(...);
r.setType(RegexpType.CONCATENATION);
r.addChild(...);
r.addChild(...);
r.addChild(...);
r.setImmutable();

```

Finally, regexp contain one useful method for obtaining all leaves in the regexp tree, it is called `getTokens()` and it recursively traverses tree returning list of leaves (token type regexps).

2.2 XML representation

References

- [Aho96] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.
- [HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2001.
- [HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
- [VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASFAA '08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.
- [wik] Regular expression. http://en.wikipedia.org/wiki/Regular_expression.