JITty: a rewriter with strategy annotations

http://www.cwi.nl/~vdpol/jitty/

Jaco van de Pol
vdpol@cwi.nl

Centrum voor Wiskunde en Informatica
P.O.-box 90.079, 1090 GB Amsterdam, The Netherlands

1 Introduction

We demonstrate `JITty`, a simple rewrite implementation along the lines of the Just-In-Time rewrite strategy, explained and justified at WRS 2001 (see [4]). The core rewrite engine normalizes terms w.r.t. a given term rewrite system (TRS) according to a given strategy annotation. We shortly review strategy annotations in Section 2. Our tool has the following distinguishing features:

- It provides the flexibility of user defined strategy annotations.
- Strategy annotations are checked for correctness, and it is guaranteed that all produced results are normal forms w.r.t. the underlying TRS.
- The tool is "light-weight" with compact but fast code.
- A TRS is interpreted, rather than compiled, so the tool has a short start-up time and is portable to many platforms.

We wrote a simple demonstrator (Section 4), which can be used to experiment with strategy annotations. The rewrite engine has also been integrated in the μCRL tool set [1], and can be used as a light-weight replacement of the standard compiling rewriter of this tool set. Although performing a rewrite step takes more time in the interpreter than in the compiler, the former is often preferred, owing to avoidance of compilation time, and the flexibility of having various strategies.

We will describe the interface to the system (Section 5), which is unconventional due to the requirements of the μCRL tool set. In particular a stack of global environments is maintained, and a term is always rewritten in the current environment (Section 3).

2 User defined and predefined strategies

A strategy annotation for a function symbol f is a list of integers and rule labels, where the integers refer to the arguments of f $(1 \leq i \leq arity(f))$ and the rule labels to rewrite rules for f, i.e. rules whose left hand side have top symbol f. For instance, the annotation $f : [1, \alpha, \beta, 2]$ means that a term with top symbol f should be normalized by first normalizing its first argument, then trying rule α and β; if both fail the second argument is normalized.

To normalize correctly, a strategy annotation must be *full* and *in-time*. It is full if all arguments and rules for f are mentioned. It is *in-time* if arguments are mentioned before rules that need them. A rule needs an argument if either the argument starts with a function symbol, or the argument is a non-linear variable. It has been proved in [4] that if a normal form is computed under a strategy annotation satisfying the above restrictions, then the result is a normal form of the original TRS *without strategies.* `JITty` checks these criteria. Moreover, `JITty` predefines the following strategies:

– left-most innermost, which first normalizes all arguments, and subsequently tries all rewrite rules.
– just-in-time, which also normalizes its arguments from left to right, but tries to apply rewrite rules as soon as their needed arguments have been evaluated.

`JITty`'s strategy annotations are similar to OBJ's annotations (e.g. [3]). The annotations of `JITty` are more refined, because rules can be mentioned individually, but less sophisticated, because laziness annotations are not supported.

3 Rewriting with global environment

In the μCRL toolset, certain open terms (i.e. guards of program lines) are rewritten under many substitutions (i.e. program states). To support this, `JITty` maintains a current environment. The user can modify the current environment by assigning a term to a variable, provided this term is normal. To resolve name conflicts, the user can enter and leave blocks. Entering a block doesn't change the global environment, but leaving a block restores the previous environment.

The global environment changes the semantics of rewriting as follows. Given global environment σ, rewriting a term t now means to get the normal form of t^σ, thereby assuming that the substitution σ is normal. This can be exploited in the implementation: t need not be traversed twice (for substitution and rewriting). Secondly, x^σ (for variables x in t) need not be traversed to find redexes, because it is supposed to be normal.

4 Simple demonstrator

We provide a simple demonstrator, which reads a file containing a signature, a number of rules, a default strategy, a number of user-defined strategies, and a number of commands. It has a fixed structure, as shown in Figure 1. The signature consists of a number of function symbols with their arity. A rule consists of a label, a list of variables, a left hand side and a right hand side. The default strategy should be either *innermost* or *justintime.* The default strategy can be overwritten for each function symbol, by an annotation, being a mixed list of integers and rule labels.

The commands are of the form `rewrite(term)` (to start rewriting). The environment is manipulated by the commands `assign(var,term)`, `enter`, `leave` and `clear`. The three examples in Figure 1 are carefully chosen to terminate. Other examples may loop for ever. After replacing *justintime* by *innermost*, only the first term will terminate. The website shows more examples that terminate by virtue of the just-in-time strategy, e.g. the following rewrite rule for division terminates for closed x and y:
`div(x,y)` $\rightarrow$ `if(lt(x,y),0,S(div(minus(x,y),y)))`.

```
signature
 T(0)      or(2)      loop(0)
 F(0)      and(2)
rules
 a1([x], and(x,T), x)     o1([x], or(T,x), T)     l([], loop, loop)
 a2([x], and(x,F), F)     o2([x], or(F,x), x)
default justintime
strategies
 and([2,a1,a2,1])
end
  rewrite( and(loop,F) )
  rewrite( or(T,loop) )
  rewrite( or(and(loop,F),or(T,loop)) )
stop
```

Fig. 1. Boolean example.

5 Application Programmer's Interface

JITty is written in the programming language C and makes use of the ATerm library [2] for implementing terms. Also many data structures in JITty make use of ATerms. Therefore, ATerms also show up in the API of JITty. The functionality of JITty is available via the API in Figure 2. A complete C program using the basic functionality is shown in Figure 3.

JIT_init is used to initialize (or reset) the rewriter. At initialization, the following information is needed: lists of function symbols, rewrite rules and strategy annotations, and an integer indicating the default strategy. The syntax of the various lists is similar to the demonstrator example (Figure 1). JIT_normalize(t) returns the normal form of t in the current environment.

The other functions are used to manipulate the global environment. JIT_enter() and JIT_leave() can be used to enter or leave a new block. JIT_clear() undoes all

```
#include "aterm2.h"
#define INNERMOST 1
#define JUSTINTIME 2

void JIT_init(ATermList functions, ATermList rules,
              ATermList strategy,int default_strat);
ATerm JIT_normalize(ATerm t);
void JIT_assign(Symbol v, ATerm t);
void JIT_enter();
void JIT_leave();
void JIT_clear();
int  JIT_level();
```

Fig. 2. JITty Application Programmer's Interface

```
#include "jitty.h"
int main(int argc, char* argv[]) {
 ATinitialize(argc,argv);                        /* initialize ATerms   */
 JIT_init(ATparse("[f(1),g(2),a(0),b(0)]"),     /* signature           */
          ATparse("[frule([x],f(x),g(x,b)),"    /* rule for f          */
                  " grule([x],g(x,x),f(a))]"), /* rule for g          */
          ATparse("[f([frule,1])]"),            /* strategy annotation */
          INNERMOST);                           /* default strategy    */
 ATprintf("%t\n",JIT_normalize(ATparse("f(b)")));
}
```

Fig. 3. Example program of using JITty

bindings in the current block. `JIT_assign(v,t)` assigns term t to variable v (represented as Symbol from the `ATerm` library). Finally, `JIT_level()` returns the current level.

6 Implementation Issues – Efficiency

`JITty` is implemented in `C`, using the `ATerm` library to represent terms. This library is supposed to have an efficient implementation, and it guarantees that terms are always stored in maximally shared form. Moreover, μCRL relies on `ATerms` as well, so at term level there is no translation between μCRL and `JITty`.

The actual rewrite code is very compact. Two optimizations are implemented. First, a special function symbol "normal" is put above subterms that are known to be in normal form. E.g. in $\alpha : f(g(x)) \to h(x)$, with $f : [1, \alpha]$, it is sure that the argument of h is normal. Here we use the assumption that the strategy annotation is *in-time*.

The second optimization is that the rewriter can be put in "hash mode". In this case, each computed result is stored in a look-up table. So each computation is performed only once. In hashing mode, double traversal of the term cannot be avoided, because we have to substitute the global environment before rewriting. However, putting "normal" symbols above instantiated variables avoids needless traversal of these subterms.

References

1. S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings of CAV 2001*, LNCS 2102, pages 250–254, 2001. See also `http://www.cwi.nl/~mcrl/`.
2. M. van den Brand, H. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000. See also `http://www.cwi.nl/projects/MetaEnv/aterm/`.
3. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.
4. J. van de Pol. Just-in-time: On strategy annotations. In B. Gramlich and S. Lucas, editors, *Electronic Notes in TCS*, volume 57, 2001. (Proc. of WRS 2001, Utrecht).