JITty: A Rewriter with Strategy Annotations

Jaco van de Pol

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

1 Introduction

We demonstrate JITty, a simple rewrite implementation with strategy annotations, along the lines of the Just-In-Time rewrite strategy, explained and justified in [4]. Our tool has the following distinguishing features:

- It provides the flexibility of user defined strategy annotations, which specify the order of normalizing arguments and applying rewrite rules.
- 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 shortly review strategy annotations in Section 2. JITty is available via `http://www.cwi.nl/~vdpol/jitty/` together with a small demonstrator (Section 3) that can be used to experiment with strategy annotations. The rewrite engine has also been integrated in the μCRL tool set [1]. Although performing a rewrite step takes more time in JITty than in the standard compiling rewriter of the μCRL toolset, the former is often preferred, owing to avoidance of compilation time, and better normalization properties of the just-in-time strategy. In Section 4 we emphasize certain requirements on the rewriter imposed by the μCRL toolset. This leads to an unconventional application programmer's interface, which is described in Section 5.

2 User Defined and Predefined Strategy Annotations

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 $(1 \leq i \leq \mathit{arity}(f))$ 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.

```
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 of a demonstrator file.

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*. Therefore JITty checks these criteria. The following strategies are predefined: *leftmost innermost*, which first normalizes all arguments, and subsequently tries all rewrite rules; and *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. See [4] for other rule based systems with user-controlled strategies (ELAN, Maude, OBJ-family, Stratego).

Although, strictly speaking, evaluation can be done when strategy annotations are *not* in-time or full, an interesting optimization can be applied if they are. In particular, for in-time annotations we have that all subterms of a normal form are in normal form. Hence, in $\alpha : f(g(x)) \to h(x)$, with $f : [1, \alpha]$, it is guaranteed that the argument of h will be normal, so it will not be traversed. Therefore, JITty currently only supports "correct" annotations.

3 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 strategy annotations, 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 three examples in Figure 1 are carefully chosen to terminate. Other examples may loop for ever. After replacing *justintime* by *innermost*, the first term will terminate, but the last two will not. The website shows more examples, such as the following rule for division, which terminates for closed x and y by virtue of the just-in-time strategy: `div(x,y)` $\rightarrow$ `if(lt(x,y),0,S(div(minus(x,y),y)))`.

4 Embedding in the μCRL Toolset

A μCRL specification consists of an equational data theory and a process part. The μCRL toolset [1] contains a.o. an automated theorem prover for the equational theory, and a simulator for the process part which serves as the front end of visualization and model checking tools. Both tools depend heavily on term rewriting in order to decide the equational theory. Therefore we need that the strategy annotations always yield normal forms of the TRS.

The simulator has to normalize the guards of transition rules (i.e., terms with state variables). The same guard is rewritten in many states. For efficiency reasons, JITty maintains a current environment, which is a normalized substitution. Given global environment σ, rewriting a term t now means to get the normal form of t^σ, assuming that the substitution σ is normalized. This can be exploited in the implementation: t has to be traversed only once, and x^σ (for variables x in t) is not traversed at all, because it is supposed to be in normal form. The user can modify the current environment by assigning a term to a variable, provided this term is in normal form. 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 theorem prover is based on binary decision diagrams (BDD) with equations instead of proposition symbols. BDDs are nothing but highly shared if-then-else trees. An important optimization, crucial when rewriting BDDs, 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 sub-computation is performed only once.

5 Application Programmer's Interface

JITty is implemented in the programming language C, and relies on the ATerm library [2]. This library is supposed to have an efficient term implementation. It guarantees that terms are always stored in maximally shared form. Moreover, the μCRL toolset uses ATerms as well, so at term level there is no translation between μCRL and JITty. Therefore, ATerms also show up in the Application Programmer's Interface of JITty, shown 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 strat annotations, an indication of the default strategy (currently one of the co INNERMOST or JUSTINTIME) and an indication whether hash tables shoulc

```
#include "aterm2.h"
#define INNERMOST 1
#define JUSTINTIME 2
void JIT_init(ATermList funs, ATermList rules, ATermList strategy,
         int default_strat, char withhash);
ATerm JIT_normalize(ATerm t);
void JIT_flush(void);
void JIT_assign(Symbol v, ATerm t);
void JIT_enter(void);
void JIT_leave(void);
void JIT_clear(void);
int  JIT_level(void);
```

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    */
         0);                                     /* without hashing     */
 ATprintf("%t\n",JIT_normalize(ATparse("f(b)"))); }
```

Fig. 3. Example program of using JITty

(0 = no, 1 = yes). JIT_normalize(t) returns the normal form of t in the current environment. JIT_flush() is used to clear the hash table, in case it becomes too memory consuming. The other functions are used to manipulate the global environment, as explained in Section 4. JIT_enter() and JIT_leave() can be used to enter or leave a new block. JIT_clear() undoes all 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.

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.
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).