ROMA TRE DIA
Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

ElfRW: A Tool for Higher-Order Dependently Typed Rewriting (System Description)

Wolfgang Gehrke

RT-INF-20-97 **1997**

Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale 84
I - 00146 Roma, Italy
wgehrke@inf.uniroma3.it

supported by the Italian project MURST ex 40% "Rappresentazione della conoscenza e meccanismi di ragionamento"

ABSTRACT

We report on an extension of the SML implementation of the logic programming language Elf [Pfe94] to support the check of convergence for higher-order critical pairs. Since Elf is based on the Edinburgh Logical Framework [HHP93] it utilizes dependent types. Therefore in the implementation a generalization of the critical pair lemma to this case as done in [Vir96] had to be employed.

1 Motivation

Higher-order rewrite systems (HRS) as an extension of term rewriting to simply typed λ-terms were introduced by Nipkow [Nip91]. Several confluence and local confluence results could be generalized to the higher-order case [MN94]. In particular the convergence of all critical pairs implies local confluence and (weak) orthogonality implies confluence.

Since the logic programming language Elf [Pfe94] provides a higher-order setting it can be used in a similar way as λ-Prolog [Mil91] as done by Felty in [Fel92] to implement higher-order rewriting. Nevertheless Elf and λ-Prolog differ in several aspects. Additionally it is desirable to provide the user with means e.g. to check (local) confluence by critical pair criteria.

Elf is based on the the Edinburgh Logical Framework [HHP93]. Therefore unlike the setting of Nipkow [Nip91] Elf additionally supports dependent types. This is also reflected by the implementation.

Here we report by example on the following work done:

- the extension of the SML implementation of Elf
- allowing the automatic check of higher-order critical pairs
- taking dependent types into account.

The used generalization of the critical pair criterion to dependent types follows [Vir96]. At the end we conclude and suggest future work.

Acknowledgment: We are greatly indebted to Frank Pfenning for detailed discussions on the topic and intensive support regarding the implementation issues of Elf and this extension.

2 Commented Example

As example to demonstrate the use of the extended version of Elf we choose the standard example for HRS: the λ-calculus itself. But in contrast to the definition of HRS we will not just treat the untyped version but even the typed version with the help of dependent types. We will consider the $\lambda^{\rightarrow}_{\beta\eta}$-calculus, i.e. with the β and η rules.

For the typed calculus we have $\mathcal{B} := \{tp, term\}$ as the types and $\mathcal{C} := \{abs, app\}$ as constants where $abs : (term_\sigma \to term_\tau) \to term_{\sigma\to\tau}$ and $app : term_{\sigma\to\tau} \to term_\sigma \to term_\tau$ with the rules:

$$\textbf{(beta)}\quad app(abs(t_{\sigma\to\tau}), s_\sigma) \to (t\ s) \qquad \textbf{(eta)}\quad abs(\lambda v.app(t_{\sigma\to\tau}, v_\sigma)) \to t$$

abs and *app* are here notated as operators with arity 1 and 2, respectively (in an uncurried form).

An Elf program is split into a static and a dynamic part. The static part is exclusively used for type checking information. The dynamic part is used for proof search in the style of logic programming:

```
%%% static part
tp   : type.
term : tp -> type.
=>   : tp -> tp -> tp.
%infix right 10 =>
abs  : (term A -> term B)
       -> term (A => B).
app  : term (A => B)
       -> term A
       -> term B.
```

```
%%% dynamic part
%%  rules
|>   : term T -> term T -> type.
%infix none 1 |>
beta : (app (abs M) N) |> (M N).
eta  : (abs ([x] app M x)) |> M.
%%  rewriting and one step reduction
*>   : term T -> term T -> type.
%infix none 1 *>
+>   : term T -> term T -> type.
%infix none 1 +>
step0: M +> M' <- M |> M'.
step1: (app M N) +> (app M' N) <- M +> M'.
step2: (app M N) +> (app M N') <- N +> N'.
step3: (abs F) +> (abs F') <- {x}(F x) +> (F' x).
try  : F *> F' <- F +> F'' <- F'' *> F'.
last : F *> F.
==   : term T -> term T -> type.
%infix none 1 ==
trmeq: M == N <- M *> O <- N *> O.
```

Above both parts of the implementation are shown. The predicate `|>` written in infix notation codes the two rules. The predicate `*>` attempts continuously rewrite steps or returns the term if no rewrite step is possible whereas the predicate `+>` does at least one rewrite step starting from the given term. A rewrite step consists of either applying one of the rules at the top position or doing at least one rewrite step inside

the subterms of the given term (both infix, too). Since the Elf program is executed in a similar fashion as Prolog with this order of the rules rewriting is done from outside to inside and from left to right. Nevertheless this strategy can be changed by rearranging the clauses for `+>`.

In the clause `step3` it should be noted that Elf can introduce newly bound variables. This is necessary for providing an argument for M such that rewriting can take place only on the base type level. Nevertheless this is here the dependent type `term`.

Finally the infix equality `==` is defined by trying to normalize both terms to an equal reduct. This predicate will return a term representing the proof of equality or fail if there is no common reduct. In the latter case for bigger terms the computation will take quite long since Elf does not provide an equivalent to the cut ("!") in Prolog thus trying all possible reducts for a given term.

Assuming the file containing the static part is called "static.elf" and the file containing the dynamic part is called "dynamic.elf" a possible interaction with Elf follows. Note that Elf is a saved image of the SML compiler providing special built-in functions to invoke the top level of Elf:

```
- initload ["static.elf"] ["dynamic.elf"];
...
- top();
Using: dynamic1.elf
Solving for: |> *> +> ==
?- {f} (app (app (abs [x] x)(abs [y] y))(abs [z] (app f z))) *> (X f).
Solving...
X = [f:term (X1 => X2)] f.
yes
?- F : (abs [x] x) == (abs [x] (abs [y] (app x y))).
Solving...
F = trmeq (try last (step3 [x:term (X => X1)] step0 eta)) last.
yes
?- (abs [x] (abs [y] x)) == (abs [x] (abs [y] y)).
Solving...
no
?- val it = () : unit
```

The first query demonstrates a normalization of a term having one free variable f which is all-quantified for the use by Elf. The other queries demonstrate the test of equality where the successful query also returns a proof term. This shows the style of using Elf for higher-order rewriting.

So far the normalization with `*>` has been done without knowing anything about confluence or termination. Since the test of (higher-order) critical pairs is a pure syntactic test the implementation of Elf was extended by the following function:

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: dynamic2.elf
Solving for: |> *> +> ==
Checking 2 rules for critical pairs:
 beta eta
All critical overlaps for beta < beta joinable
Critical pair for beta < eta
Term 1: app M N
Term 2: app M N
Joined trivially.
All critical overlaps for beta < eta joinable
Critical pair for eta < beta
Term 1: abs M1
Term 2: abs [x:term X] M1 x
Joined trivially.
All critical overlaps for eta < beta joinable
All critical overlaps for eta < eta joinable
Checking subsumption
Checked 2 rules for critical pairs:
 beta eta
Remaining Constraint Errors: 0
Critical Pairs: 2
Trivially Joined: 2
Equal Normal Forms: 0
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

The function `crit_pairs` takes two arguments: firstly the name of the predicate containing the rules and secondly the name of the predicate used for normalization. Concerning the normalizing predicate the user is responsible for its termination. At the end a summary of the critical pair check is printed.

Note the following:

1. The Elf implementation differs from the definition of HRS in that it does not use the $\beta\eta$-long form of terms cf. [Nip91]. Therefore in certain circumstances the confluence of a critical pair might not be detected.
2. The notion of critical pairs is already representable on the Elf level with the help of unification. The actual problem lies in the normalization phase where free variables had to be temporarily bound which required to work with the SML level of the implementation.

3. Rules are represented by a single predicate in Elf which has to pass the type check. This way of coding immediately enforces the validity of the dependence relation from [Vir96].

3 Conclusions and Future Work

The extended version of Elf as reported in this paper is available under the following URL-address "http://www.inf.uniroma3.it/people/wgehrke/ElfRW/". The source is written in SML/NJ version 109 and makes use of the compiler manager CM [1]. This page contains also other applications.

The commented example demonstrates how to make use of Elf as a higher-order rewriting tool. Additionally also dependent types are treated. The theoretical ground for this extension is laid in [Vir96].

As advantages of using Elf as a rewriting tool we consider:
1. the possibility to implement different rewriting strategies,
2. the availability of proof terms to inspect a normalization,
3. the extension of higher-order rewriting to dependent types.

As future work we plan to investigate the possibility of incorporating means to support semi-automatic methods to check termination of a higher-order rewrite system. This could be done by combining our implementation with current work by Rohwedder and Pfenning [RP96]. Furthermore a method by Kahrs [Kah95] might be promising as a semi-automatic procedure with a required build-in knowledge about ordinal arithmetic.

References

[Fel92] A. Felty. A logic programming approach to implementing higher-order term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, volume 596 of *Lecture Notes in Computer Science*, pages 135–161. Springer-Verlag, 1992.

[HHP93] R. Harper, F. Honsel, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Kah95] S. Kahrs. Towards a Domain Theory for Termination Proofs. In J. Hsiang, editor, *Proceedings of the 6th International Conference, Rewriting Techniques and Applications, RTA-95*, number 914 in Lecture Notes in Computer Science, pages 241–255. Springer-Verlag, 1995.

[Mil91] D. Miller. A logic programming language with Lambda-Abstraction, Function variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MN94] R. Mayr and T. Nipkow. Higher-Order Rewrite Systems and their Confluence. Technical Report TUM-I9433, Technical University, Munich, August 1994.

[Nip91] T. Nipkow. Higher-Order Critical Pairs. In *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[Pfe94] F. Pfenning. Elf: A Meta-Language for Deductive Systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 811–815, 1994.

[RP96] E. Rohwedder and F. Pfenning. Mode and Termination Checking for Higher-Order Logic Programs. In H.R. Nielson, editor, *Programming Languages and Systems ESOP'96*, number 1058 in Lecture Notes in Computer Science, pages 296–310. Springer-Verlag, 1996.

[Vir96] R. Virga. Higher-Order Superposition for Dependent Types. In H. Ganzinger, editor, *Proceedings of the 7th International Conference, Rewriting Techniques and Applications, RTA-96*, number 1103 in Lecture Notes in Computer Science, pages 123–137. Springer-Verlag, 1996.

This article was processed using the LaTeX macro package with LLNCS style