A terminating schema for higher-order rewrite systems

Frédéric Blanqui[†], Jean-Pierre Jouannaud[†] and Mitsuhiro Okada[‡]
[†] LRI, CNRS and Université de Paris-Sud
Bât 405, 91405 Orsay Cedex, France
[‡] Department of Philosophy, Keio University,
108 Minatoku, Tokyo, JAPAN
Tel: +33-1-69156905 FAX: +33-1-69156586 Tel-FAX: +33-1-43212975
http://www.lri.fr/~blanqui/

February 1st, 1999

We propose to present a new technique to prove the termination of computations for languages combining terms of some typed λ-calculus [1], inductively defined base types [7], and function symbols defined by higher-order rewrite rules [11, 12].

In contrast with Combinatory Rewrite Systems [13] or, equivalently, Higher-order Rewrite Systems [14] which, in general, are not terminating systems since they are based on the untyped λ-calculus (their authors are essentially interested in the confluence/Church-Rosser property), in our higher-order rewrite rules, we assume that the lefthand sides are made only of type constructors and function symbols: they are algebraic. However, this is powerful enough to capture any algebraic rewrite systems [8], or higher-order rewrite rules defining recursors for higher-order inductive types (called "strictly positive" in [7]) like the ones for ordinals.

Since the works by Breazu-Tannen and Gallier [4, 5], and independently by Okada [15], on the termination of computations for the simply-typed λ-calculus combined with first-order (algebraic) rewriting [1], it is well known how to deal with first-order rewriting even with richer type systems: their combination is modular, that is the combined system is terminating if the first-order rewrite system and the type system are both terminating. Now, for further combining such a system with higher-order rewriting, it is necessary for the first-order rewrite system to be non-duplicating. Termination of higher-order rewriting with algebraic lefthand sides has been studied in [11, 12] where a rule schema was introduced that generalizes that of higher-order primitive recursion. It was capturing the rules for recursors of "basic" inductive types like naturals and lists, or the "map" function on lists, but it was unable to treat the case of the recursor for non-basic inductive types like the one of ordinals.

Our new proof technique allows us to deal with such rewrite rules. It relies on two important novelties. Firstly, we define the set of higher-order rewrite rules as an inductively defined set, that we call the "General Schema". This will allow us to inductively reason about these rules. Secondly, we use an interpretation for comparing the arguments of a function call defined by the lefthand side of a rule with the recursive calls occurring in the righthand side. This interpretation is based on the notion of "critical subterm" for a strictly positive inductive type. Let us consider the example of the recursor `rec` on the strictly positive inductive type `ord` of ordinals whose constructors are `0` of type `ord`, `S` of type `ord`$\rightarrow$`ord` and `lim` of type (`nat`$\rightarrow$`ord`)$\rightarrow$`ord`, where `nat` is the basic inductive type of naturals. `rec` is defined by the following rewrite rules:

$$\begin{array}{rcl}
\mathtt{rec}(\mathtt{0},u,v,w) & \longrightarrow & u\\
\mathtt{rec}(\mathtt{S}(x),u,v,w) & \longrightarrow & v\ x\ \mathtt{rec}(x,u,v,w)\\
\mathtt{rec}(\mathtt{lim}(f),u,v,w) & \longrightarrow & w\ f\ \lambda n{:}\mathtt{nat}.\mathtt{rec}(f\ n,u,v,w)
\end{array}$$

[1]Both works follow from the pioneer work by Breazu-Tannen on the confluence of the combination of the simply-typed λ-calculus with first-order (algebraic) rewriting [3].

In the third rule, we have to compare $\mathtt{lim}(f)$ to $(f\ n)$. The fact that, for a strictly positive inductive type σ, the type of an argument of a constructor is of the form $\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$ where σ does not occur in any σ_i, allows us to neglect the terms to which an argument of a constructor is applied, giving the "critical subterm". Within the example, the critical subterm of $(f\ n)$ is simply f which is a strict subterm of $\mathtt{lim}(f)$.

To illustrate our discourse, we will take the example of the simply-typed λ-calculus. We have submitted an article to RTA'99 [2] where we present this method in the case of the Calculus of Constructions [6], and that you can find on the web. Assuming that we are given some base types, upon which function types may be built using the arrow operator $\rightarrow$, the language we consider is defined as follows:

$$t := x \mid \lambda x{:}\sigma.t \mid (t_1\ t_2) \mid C(t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n)$$

where x designates a variable, $\lambda x{:}\sigma.t$ the function that associates t to each variable x of type σ, $(t_1\ t_2)$ the application of the function t_1 to the term t_2, $C(t_1, \ldots, t_n)$ the constructor term of some inductive type, and $f(t_1, \ldots, t_n)$ the application of some defined function symbol f to some terms $t_1, \ldots, t_n$. Naturally, assuming that each function symbol is defined by a set of rewrite rules $f(t_1, \ldots, t_n) \longrightarrow t$, the computations associated to this language will be the combination of these rewrite rules with the β-reduction, that is, the operation that substitutes the bound variable of an abstraction by the argument to which it is applied: $(\lambda x{:}\sigma.t_1\ t_2) \longrightarrow t_1\{x \mapsto t_2\}$. Since such a calculus is not terminating in general, we restrict our attention to the so-called "well-typed" terms, that is the terms to which we can give some type. For example, an abstraction $\lambda x{:}\sigma.t$ has type $\sigma \rightarrow \tau$ if x is a variable of type σ and t is a term of type τ. We refer the reader to [1] for an introduction to generalized type systems. We denote by $\vdash t : \sigma$ the fact that t has type σ.

The termination proof technique that we propose relies on the well known proof technique of "reducibility candidates" due to Tait and Girard [10, 9]. Since a direct proof by induction on the structure of the typed judgements does not go through, the idea is to use a strengthened induction hypothesis and to instead prove that, every term t of type σ is "reducible", that is, belongs to the interpretation of its type that we denote by $[\![\sigma]\!]$. Then, the termination follows from the fact that $[\![\sigma]\!]$ is chosen so as to be included into the set of terminating terms. In fact, to treat the abstraction case, we always reason modulo a "reducible" substitution, that is, a substitution that associates reducible terms to each variable of its domain. Such a proof can be further extended in a modular way to deal with the case of function symbols. Indeed, it suffices to prove that, for every well-typed terms $f(t_1, \ldots, t_n)$ and reducible substitution θ, $f(t_1\theta, \ldots, t_n\theta)$ is also reducible.

So, one can see how interesting it is to inductively define the set of higher-order rewrite rules for which we would be able to prove the previous result. Given a lefthand side $f(t_1, \ldots, t_n)$, it suffices to start with the "accessible" subterms, that is, the subterms that are reducible whenever $t_1, \ldots, t_n$ are so. Then, we can extend this base set with any operation that preserves the reducibility. This includes the abstraction, the application, the application of a constructor or of a defined function symbol and the reduction. We can also add recursive calls whose arguments are strict subterms of $t_1, \ldots, t_n$ modulo the critical subterm interpretation. But, in this case, since the critical subterm interpretation is not compatible with the reduction relation, and not stable by substitution either, in order to carry out the proof of reducibility of the higher-order function symbols, we need to define yet another interpretation which is compatible with the critical subterm interpretation and satisfies the previous properties.

Finally, we expect to extend our technique to generalized inductive types as defined in [7], that is, inductively defined polymorphic types, and inductively defined predicates (dependant types). Together with function symbol definitions at the type level, our technique would lead to a simpler proof of termination of the Calculus of Inductive Constructions with strong elimination [16] (but without taking into account the η-reduction into the type conversion rule).

We also think that our technique could be useful for resolving two problems left open. First, the extension of the General Schema so as to capture the recursor rules for non-strictly positive inductive types. One example is given by a coding of Montague' semantical objects as an inductive type $\mathtt{mon}$ with a constructor of type $((\mathtt{mon} \rightarrow \mathtt{bool}) \rightarrow \mathtt{mon}) \rightarrow \mathtt{mon})$, where $\mathtt{bool}$ is the type of booleans. Secondly, it would be nice to allow function symbols to be defined by pattern-matching on some higher-order function symbol like the addition operator on ordinals. Indeed, it is possible to do so with first-order operators like the addition operator on naturals or the append operator on lists, allowing one to refine the definition of the map function for example.

References

[1] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1992.

[2] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions, 1998. RTA'99 submission.

[3] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 82–90. IEEE Computer Society, 5–8 July 1988.

[4] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 137–150, Stresa, Italy, July 1989. European Association of Theoretical Computer Science, Springer-Verlag.

[5] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, June 1991.

[6] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[7] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, LNCS 417. Springer-Verlag, 1990.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–309. North-Holland, 1990.

[9] J. Gallier. On Girard's "Candidats de Réductibilité". In P.-G. Odifreddi, editor, *Logic and Computer Science*. North Holland, 1990.

[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[11] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 350–361, 1991.

[12] J.-P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, February 1997.

[13] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, December 1993.

[14] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 342–349, 1991.

[15] M. Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In G. H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 357–363. ACM Press, July 1989.

[16] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, 1994.