Termination of $\lambda\Pi$ modulo rewriting using the size-change principle (work in progress)

Frédéric Blanqui, Guillaume Genestier

Termination of $\lambda\Pi$ modulo rewriting using the size-change principle (work in progress)

Frédéric Blanqui[1,2] and Guillaume Genestier[1,2,3]

1 **LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay**
2 **Inria**
3 **MINES ParisTech, PSL University**

Abstract

The Size-Change Termination principle was first introduced to study the termination of first-order functional programs. In this work, we show that it can also be used to study the termination of higher-order rewriting in a system of dependent types extending LF.

Keywords and phrases Termination, Higher-Order Rewriting, Dependent Types, Lambda-Calculus.

1 Introduction

The Size-Change Termination principle (SCT) was first introduced by Lee, Jones and Ben Amram [8] to study the termination of first-order functional programs. It proved to be very effective, and a few extensions to typed λ-calculi and higher-order rewriting were proposed.

In his PhD thesis [13], Wahlstedt proposes one for proving the termination, in some presentation of Martin-Löf's type theory, of an higher-order rewrite system $\mathcal{R}$ together with the β-reduction of λ-calculus. He proceeds in two steps. First, he defines an order, the instantiated call relation, and proves that $\longrightarrow = \longrightarrow_{\mathcal{R}} \cup \longrightarrow_{\beta}$ terminates on well-typed terms whenever this order is well-founded. Then, he uses SCT to eventually show the latter.

However, Wahlstedt's work has some limitations. First, it only considers weak normalization, that is, the mere existence of a normal form. Second, it makes a strong distinction between "constructor" symbols, on which pattern matching is possible, and "defined" symbols, which are allowed to be defined by rewrite rules. Hence, it cannot handle all the systems that one can define in the $\lambda\Pi$-calculus modulo rewriting, the type system implemented in Dedukti [2].

Other works on higher-order rewriting do not have those restrictions, like [3] in which strong normalization (absence of infinite reductions) is proved in the calculus of constructions by requiring each right-hand side of rule to belong to the Computability Closure (CC) of its corresponding left-hand side.

In this paper, we present a combination and extension of both approaches.

2 The $\lambda\Pi$-calculus modulo rewriting

We consider the $\lambda\Pi$-calculus modulo rewriting [2]. This is an extension of Automath, Martin-Löf's type theory or LF, where functions and types can be defined by rewrite rules, and where types are identified modulo those rules and the β-reduction of λ-calculus.

Assuming a signature made of a set $\mathcal{C}_T$ of type-level constants, a set $\mathcal{F}_T$ of type-level definable function symbols, and a set $\mathcal{F}_o$ of object-level function symbols, terms are inductively defined into three categories as follows:

kind-level terms	$K ::= \text{Type} \mid (x : U) \to K$	
type-level terms	$T, U ::= \lambda x : U.\, T \mid (x : U) \to T \mid U\, t \mid D \mid F$	where $D \in \mathcal{C}_T$ and $F \in \mathcal{F}_T$
object-level terms	$t, u ::= x \mid \lambda x : U.\, t \mid t\, u \mid f$	where $f \in \mathcal{F}_o$

By $\bar{t}$, we denote a sequence of terms $t_1 \dots t_n$ of length $|\bar{t}| = n$.

Next, we assume given a function τ associating a kind to every symbol of $\mathcal{C}_T$ and $\mathcal{F}_T$, and a type to every symbol of $\mathcal{F}_o$. If $\tau(f) = (x_1 : T_1) \to \dots \to (x_n : T_n) \to U$ with U not an arrow, then f is said of arity $\mathrm{ar}(f) = n$.

An object-level function symbol f of type $(x_1 : T_1) \to \dots \to (x_n : T_n) \to D\, u_1 \dots u_{\mathrm{ar}(D)}$ with $D \in \mathcal{C}_T$ and every T_i of the form $E\, v_1 \dots v_{\mathrm{ar}(E)}$ with $E \in \mathcal{C}_T$ is called a *constructor*. Let $\mathcal{C}_o$ be the set of constructors.

Terms built from variables and constructor application only are called *patterns*: $p ::= x \mid c\, p_1 \dots p_{\mathrm{ar}(c)}$ where $c \in \mathcal{C}_o$.

Next, we assume given a set $\mathcal{R}$ of rewrite rules of the form $f\, p_1 \dots p_{\mathrm{ar}(f)} \longrightarrow r$, where f is in $\mathcal{F}_o$ or $\mathcal{F}_T$, the p_i's are patterns and r is β-normal. Then, let $\longrightarrow = \longrightarrow_\beta \cup \longrightarrow_\mathcal{R}$ where $\longrightarrow_\mathcal{R}$ is the smallest rewrite relation containing $\mathcal{R}$.

Note that rewriting at type level is allowed. For instance, we can define a function taking a natural number n and returning Nat → Nat → ⋯ → Nat with as many arrows as n. In Dedukti syntax, this gives:

```
def F : Nat -> Type.
[]  F 0     --> Nat.
[n] F (S n) --> Nat -> (F n).
```

Well-typed terms are defined as in LF, except that types are identified not only modulo β-equivalence but modulo $\mathcal{R}$-equivalence also, by adding the following type conversion rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B} \quad \text{if } A \longleftrightarrow^* B \text{ and } s \in \{\mathrm{Type}, \mathrm{Kind}\}$$

Convertibility of A and B, $A \longleftrightarrow^* B$, is undecidable in general. However, it is decidable if $\longrightarrow$ is confluent and terminating. So, a type-checker for the λΠ-calculus modulo $\longleftrightarrow^*$, like Dedukti, needs a criterion to decide termination of $\longrightarrow$. This is the reason of this work.

To this end, we assume that $\longrightarrow$ is confluent and preserves typing.

There exist tools to check confluence, even for higher-order rewrite systems, like CSIˆho or ACPH. The difficulty in presence of type-level rewrite rules, is that we cannot assume termination to show confluence since we need confluence to prove termination. Still, there is a simple criterion in this case: orthogonality [12].

Checking that $\longrightarrow$ preserves typing is undecidable too (for $\longrightarrow_\beta$ alone already), and often relies on confluence except when type-level rewrite rules are restricted in some way [3]. Saillard designed and implemented an heuristic in Dedukti [10].

Finally, note that constructors can themselves be defined by rewrite rules. This allows us to define, for instance, the type of integers with two constructors for the predecessor and successor, together with the rules stating that they are inverse of each other.

3 The Size-Change Termination principle

Introduced for first-order functional programming languages by Lee, Jones and Ben Amram [8], the SCT is a simple but powerful criterion to check termination. We recall hereafter the matrix-based presentation of SCT by Lepigre and Raffalli [9].

▸ **Definition 1** (Size-Change Termination principle)**.** The (strict) *constructor subterm relation* $\lhd$ is the smallest transitive relation such that $t_i \lhd c\, t_1 \dots t_n$ when $c \in \mathcal{C}_o$.

We define the *formal call relation* by $f\, \bar{p} >_{call} g\, \bar{t}$ if there is a rewrite rule $f\, \bar{p} \longrightarrow r \in \mathcal{R}$ such that $g \in \mathcal{F}_T \cup \mathcal{F}_o$ and $g\, \bar{t}$ is a subterm of r with $|\bar{t}| = \mathrm{ar}(g)$.

From this relation, we construct a *call graph* whose nodes are labeled with the defined symbols. For every call $f\,\bar{p} >_{call} g\,\bar{t}$, an edge labeled with the *call matrix* $(a_{i,j})_{i\leqslant \mathrm{ar}(f), j\leqslant \mathrm{ar}(g)}$ links the nodes f and g, where $a_{i,j} = -1$ if $t_j \lhd p_i$, $a_{i,j} = 0$ if $t_j = p_i$, and $a_{i,j} = \infty$ otherwise.

A set of rewrite rules $\mathcal{R}$ satisfies the *size-change termination principle* if the transitive closure of the call graph (using the max-plus semi-ring to multiply the matrices) is such that all arrows linking a node with itself are labeled with a matrix having at least one -1 on the diagonal.

The formal call relation is also called the dependency pair relation [1].

4 Wahlstedt's extension of SCT to Martin-Löf's Type Theory

The proof of weak normalization in Wahlstedt's thesis uses an extension to rewriting of Girard's notion of reducibility candidate [7], called computability predicate here. This technique requires to define an interpretation of every type T as a set of normalizing terms $[\![T]\!]$ called the set of *computable* terms of type T. Once this interpretation is defined, one shows that every well-typed term $t : T$ is *computable*, that is, belongs to the interpretation of its type: $t \in [\![T]\!]$, ending the normalization proof. To do so, Wahlstedt proceeds in two steps. First, he shows that every well-typed term is computable whenever all symbols are computable. Then, he introduces the following relation which, roughly speaking, corresponds to the notion of minimal chain in the DP framework [1]:

▸ **Definition 2** (Instantiated call relation)**.** Let $f\,\bar{t} \gtrsim g\,\bar{v}$ if there exist $\bar{p}$, $\bar{u}$ and a substitution γ such that $\bar{t}$ is normalizing, $\bar{t} \longrightarrow^* \bar{p}\gamma$, $f\,\bar{p} >_{call} g\,\bar{u}$ and $\bar{u}\gamma = \bar{v}$.

and proves that all symbols are computable if $\gtrsim$ is well-founded:

▸ **Lemma 3** ([13, Lemma 3.6.6, p. 82])**.** *If $\gtrsim$ is well-founded, then all symbols are computable.*

Finally, to prove that $\gtrsim$ is well-founded, he uses SCT:

▸ **Lemma 4** ([13, Theorem 4.2.1, p. 91])**.** *$\gtrsim$ is well-founded whenever the set of rewrite rules satisfies* SCT.

Indeed, if $\gtrsim$ were not well-founded, there would be an infinite sequence $f_1\,\bar{t}_1 \gtrsim f_2\,\bar{t}_2 \gtrsim \ldots$, leading to an infinite path in the call graph which would visit infinitely often at least one node, say f. But the matrices labelling the looping edges in the transitive closure all contain at least one -1 on the diagonal, meaning that there is an argument of f which strictly decreases in the constructor subterm order at each cycle. This would contradict the well-foundedness of the constructor subterm order.

However, Wahlstedt only considers weak normalization of orthogonal systems, in which constructors are not definable. There exist techniques which do not suffer those restrictions, like the Computability Closure.

5 Computability Closure

The Computability Closure (CC) is also based on an extension of Girard's computability predicates [4], but for strong normalization. The gist of CC is, for every left-hand side of a rule $f\,\bar{l}$, to inductively define a set $\mathrm{CC}_\sqsupset(f\,\bar{l})$ of terms that are computable whenever the l_i's so are. Function applications are handled through the following rule:

$$\frac{f\,\bar{l} \sqsupset g\,\bar{u} \qquad \bar{u} \in \mathrm{CC}_\sqsupset(f\,\bar{l})}{g\,\bar{u} \in \mathrm{CC}_\sqsupset(f\,\bar{l})}$$

where $\sqsupset = (>_{\mathcal{F}}, (\triangleright \cup \longrightarrow)_{\text{stat}})_{\text{lex}}$ is a well-founded order on terms $f\,\bar{t}$ such that $\bar{t}$ are computable, with $>_{\mathcal{F}}$ a precedence on function symbols and stat either the multiset or the lexicographic order extension, depending on f.

Then, to get strong normalization, it suffices to check that, for every rule $f\,\bar{l} \longrightarrow r$, we have $r \in \text{CC}_{\sqsupset}(f\,\bar{l})$. This is justified by Lemma 6.38 [3, p.85] stating that all symbols are computable whenever the rules satisfy CC, which looks like Lemma 3. It is proved by induction on $\sqsupset$. By definition, $f\,\bar{t}$ is computable if, for every u such that $f\,\bar{t} \longrightarrow u$, u is computable. There are two cases. If $u = f\,\bar{t}'$ and $\bar{t} \longrightarrow \bar{t}'$, then we conclude by the induction hypothesis. Otherwise, $u = r\,\gamma$ where r is the right-hand side of a rule whose left-hand side is of the form $f\,\bar{l}$. This case is handled by induction on the proof that $r \in \text{CC}_{\sqsupset}(f\,\bar{l})$.

So, except for the order, the structures of the proofs are very similar in both works. This is an induction on the order, a case distinction and, in the case of a recursive call, another induction on a refinement of the typing relation, restricted to β-normal terms in Wahlstedt's work and to the Computability Closure membership in the other one.

6 Applying ideas of Computability Closure in Wahlstedt's criterion

We have seen that each method has its own weaknesses: Wahlstedt's SCT deals with weak normalization only and does not allow pattern-matching on defined symbols, while CC enforces mutually defined functions to perform a strict decrease in each call.

We can subsume both approaches by combining them and replacing in the definition of CC the order $\sqsupset$ by the formal call relation:

$$\frac{f\,\bar{l} >_{call} g\,\bar{u} \qquad \bar{u} \in \text{CC}_{>_{call}}(f\,\bar{l})}{g\,\bar{u} \in \text{CC}_{>_{call}}(f\,\bar{l})}$$

We must note here that, even if $>_{call}$ is defined from the constructor subterm order, this new definition of CC does not enforce an argument to be strictly smaller at each recursive call, but only smaller or equal, with the additional constraint that any looping sequence of recursive calls contains a step with a strict decrease, which is enforced by SCT.

▸ **Proposition 5.** *Let $\mathcal{R}$ be a rewrite system such that $\longrightarrow = \longrightarrow_{\beta} \cup \longrightarrow_{\mathcal{R}}$ is confluent and preserves typing. If $\mathcal{R}$ satisfies $\text{CC}_{>_{call}}$ and* SCT*, then $\longrightarrow$ terminates on every term typable in the λΠ-calculus modulo $\longleftrightarrow^*$.*

Note that $\text{CC}_{>_{call}}$ essentially reduces to checking that the right-hand sides of rules are well-typed which is a condition that is generally satisfied.

The main difficulty is to define an interpretation for types and type symbols that can be defined by rewrite rules. It requires to use induction-recursion [6]. Note that the well-foundedness of the call relation $\gtrdot$ is used not only to prove reducibility of defined symbols, but also to ensure that the interpretation of types is well-defined.

If we consider the example of integers mentioned earlier and define the function erasing every constructor using an auxiliary function, we get a system rejected both by Wahlstedt's criterion since `S` and `P` are defined, and by the $CC_{\sqsupset}$ criterion since there is no strict decrease in the first rule. On the other hand, it is accepted by our combined criterion.

```
Int : Type.                          0 : Int.
def S : Int -> Int.                  def P : Int -> Int.

[x] S (P x) --> x.                   [x] P (S x) --> x.
[x] returnZero x --> aux x.          []  aux 0     --> 0.
[x] aux (S x) --> returnZero x.      [x] aux (P x) --> returnZero x.
```

7 Conclusion

We have shown that Wahlstedt's thesis [13] and the first author's work [3] have strong similarities. Based on this observation, we developed a combination of both techniques that strictly subsumes both approaches.

This criterion has been implemented in the type-checker Dedukti [2] and gives promising results, even if automatically proving termination of expressive logic encodings remains a challenge. The code is available at https://github.com/Deducteam/Dedukti/tree/sizechange.

Many opportunities exist to enrich our new criterion. For instance, the use of an order leaner than the strict constructor subterm for SCT, like the one defined by Coquand [5] for handling data types with constructors taking functions as arguments. This question is studied in the first-order case by Thiemann and Giesl [11].

Finally, it is important to note the modularity of Wahlstedt's approach. Termination is obtained by proving 1) that all terms terminate whenever the instantiated call relation is well-founded, and 2) that the instantiated call relation is indeed well-founded. Wahlstedt and we use SCT to prove 2) but it should be noted that other techniques could be used as well. This opens the possibility of applying to type systems like the ones implemented in Dedukti, Coq or Agda, techniques and tools developed for proving the termination of DP problems.

Acknowledgments. The authors thank Olivier Hermant for his comments, as well as the anonymous referees.

References

1 T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. *TCS* 236, 2000.
2 A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory, 2016. Draft.
3 F. Blanqui. Definitions by rewriting in the calculus of constructions. *MSCS* 15(1), 2005.
4 F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *TCS*, 611:50–86, 2016.
5 T. Coquand. Pattern matching with dependent types. In *Proc. of TYPES'92*.
6 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. of Symbolic Logic*, 65(2):525–549, 2000.
7 J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and types*. Cambridge University Press, 1988.
8 C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. of POPL'01*.
9 R. Lepigre and C. Raffalli. Practical Subtyping for System F with Sized (Co-)Induction. https://arxiv.org/abs/1604.01990, 2017.
10 R. Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, Mines ParisTech, France, 2015.
11 R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *AAECC*, 16(4):229–270, 2005.
12 V. van Oostrom and F. van Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In *Proc. of LFCS'94*, LNCS 813.
13 D. Wahlstedt. *Dependent type theory with first-order parameterized data types and well-founded recursion*. PhD thesis, Chalmers University of Technology, Sweden, 2007.