A Termination Proof of Reduction in a Simply Typed Calculus with Constructors

Manfred Schmidt-Schauß and David Sabel

Institut für Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
{schauss,sabel}@ki.informatik.uni-frankfurt.de

Technical Report Frank-42

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

October 11, 2010

Abstract. The well-known proof of termination of reduction in simply typed calculi is adapted to a monomorphically typed lambda-calculus with case and constructors and recursive data types. The proof differs at several places from the standard proof. Perhaps it is useful and can be extended also to more complex calculi

1 Introduction

It is well-known that beta-reduction in the simply typed lambda calculus terminates. The goal is to provide a simple proof that this can be extended to lambda calculi with case and constructors. The original proof is by Tait [Tai71], see also [Ste72]. There are proofs of strong termination also for different extensions of the simply typed lambda calculus. Nevertheless, we think it is worthwhile to have a proof pattern for the case-constructor-extension, since we are not aware of an easily accessible strong normalization proof.

2 The Calculus

We define the syntax and reduction of a simply-typed lambda calculus extended with case, constructors, and recursive data types and its call-by-name reduction rules. We also use structured types to support inductive arguments.

Let $\mathcal{K}$ be a finite set of *type constructors*, where every type constructor K comes with an arity $ar(K)$.
Types T are defined by the grammar $T ::= (T_1 \to T_2) \mid K(T_1, \ldots, T_{ar(K)})$, where T, T_i stand for types, and $K \in \mathcal{K}$ is a type constructor. As usual we assume function types to be right-associative, i.e. $T_1 \to T_2 \to T_3$ means $T_1 \to (T_2 \to T_3)$. Types of the form $T_1 \to T_2$ are called *function types* and types of the form $K(T_1, \ldots, T_{ar(K)})$ are called *constructed types*.
Let $\mathcal{D}$ be a finite set of *data constructors*. For every $K \in \mathcal{K}$ there is a finite set $\emptyset \neq D_K \subseteq \mathcal{D}$ of data constructors $c_{K,i}$ where $c_{K,i} \in D_K$ comes with a fixed arity $ar(c_{K,i})$. For different $K_1, K_2 \in \mathcal{K}$ it holds $D_{K_1} \cap D_{K_2} = \emptyset$ and $\mathcal{D} = \bigcup_{K \in \mathcal{K}} D_K$. We assume that there is a strict and total partial order $<$ on $\mathcal{K}$.

Definition 2.1. *The calculus is called* well-structured*, iff the following restrictions hold:*
The polymorphic type of a data constructor $c \in D_K$ *are of the form* $T_1 \to \ldots \to T_{ar(c)} \to K(X_1, \ldots, X_m)$ *where* T_i *may be of one of the following forms:*

- X_i
- $S_1 \to \ldots \to S_k \to S_{k+1}$*, where* S_i *is either a type variable or a 0-ary type constructor* K' *with* $K' < K$*.*
- $K(X_1, \ldots, X_m)$*.*

Note that generalizations are possible, but we use a simplified version that applies to the usual data structures like lists, Booleans and Peano-numbers.

2.1 Syntax of Expressions

The (type-free) syntax of expressions *Expr* is as follows, where c, c_i are data constructors, where every data constructor c has a fixed arity $ar(c)$, x, x_i are variables of some infinite set of variables, and *Alt* is a `case`-alternative:

$$\begin{aligned} s, s_i, t \in \mathit{Expr} &::= x \mid (s\ t) \mid \lambda x.s \mid (c_i\ s_1 \ldots s_{ar(c_i)}) \\ &\quad \mid (\texttt{case}\ s\ \mathit{Alt}_1 \ldots \mathit{Alt}_n) \\ \mathit{Alt}_i &::= ((c_i\ x_1 \ldots x_{ar(c_i)})\ \texttt{->}\ s_i) \end{aligned}$$

Note that data constructors can only be used with all their arguments present. We assume the variables in a pattern have to be distinct. The scoping rules in expressions are as usual. We assume that expressions satisfy the distinct variable convention before reduction is applied, which can be achieved by a renaming of bound variables.
For an expression t the set of free variables of t is denoted as $FV(t)$. An expression t is called *closed* iff $FV(t) = \emptyset$, and otherwise called *open*.

2.2 Typing of Expressions

Expressions are monomorphically typed, i.e., the types have no occurrences of type variables. It is no restriction to assume that every variable is labeled with its

(beta) $((\lambda x.s)\ t) \to s[t/x]$
(case) $(\texttt{case}\ (c\ s_1 \dots s_n)\ \dots ((c\ y_1 \dots y_n)\,\texttt{->}\,s) \dots)$
$\to s[s_1/y_1, \dots, s_n/y_n]$

Fig. 1. Call-by-name reduction rules

type. Every subexpression is annotated with a type, and every subexpression is monomorphically (i.e. simply) typed. The difference w.r.t. a simply typed lambda calculus are as follows. Constructor expressions are typed like an application. Case-expressions $(\texttt{case}\ s\ (c_1\ x_{1,1} \dots x_{1,n_1})\,\texttt{->}\,r_1, \dots, (c_k\ x_{k,1} \dots x_{k,n_1})\,\texttt{->}\,r_k)$ are typed, such that the types of s and the patterns $(c_1\ x_{i,1} \dots x_{i,n_i})$ must be the same. Also the types of the following expressions are equal: r_i, s and the case-expression.

2.3 Reduction

Reduction of expressions is by an application of one of the two rules (beta) and (case) in Fig. 1, where reduction is allowed in any context, i.e., there is no strategy. When we speak of reductions in the following, we mean reduction sequences of case- and beta-reduction in any context.
Note that reductions do not change the types of expressions.
Note that this reduction model also allows stuck closed expressions like $\texttt{case}\ c\ (d\,\texttt{->}\,d)$, since this cannot be further reduced.
For an expression t let the set $\mathrm{MC}(t)$ of maximal critical abstractions be recursively defined as:

- If $t = (c\ t_1 \dots t_n)$, then $\mathrm{MC}(t) := \bigcup_{i=1,\dots,n} \mathrm{MC}(t_i)$.
- If t is an abstraction, then $\mathrm{MC}(t) := \{t\}$.
- Otherwise $\mathrm{MC}(t) := \emptyset$.

Lemma 2.2. *Let t be an expression of type $T = K(T_1, \dots, T_n)$. Let $s \in \mathrm{MC}(t)$ be of type $S = S_1 \to \dots \to S_m \to S_{m+1}$, where S_{m+1} is not a function type. Then for all i: $|S_i| < |T|$ or S_i is a type constructor with $S_i < K$.*

Proof. By induction on the size of types and then on expression size.
If $t = c\ t_1 \dots t_{n'}$ then the type of t_j may be in $\{T_1, \dots, T_n\}$, which is strictly smaller than T; The type of t_j may be $K(T_1, \dots, T_n)$ and we can use induction on the term structure; the type may be $S_1 \to \dots \to S_m \to S_{m+1}$, where all S_i are strictly smaller in size than $K(T_1, \dots, T_n)$, or $S_i < K$.

Before we start with the termination proof, we present a counter example to strong termination of reduction if the conditions are not satisfied.

Example 2.3. Let the type and function definitions be

```
data T = T
data U = Fold (U -> T)
```

```
unfold :: U -> U -> T
unfold = \x -> case x of  (Fold y) -> y
yy =  ff        (Fold ff)
ff = (\x ->  (unfold x x))
```

The example is monomorphically typed. Reducing `yy` results in
`yy` → `(ff (Fold ff))` → `(unfold (Fold ff) (Fold ff))`
→ `(ff (Fold ff))` → ...
which is the start of a non-terminating reduction. This is even a non-terminating normal-order reduction.
Note that this example does not satisfy the well-structured condition.

3 Termination of Reduction in Well-Structured Monomorphic Lambda-Calculi with Case and Constructors

In this section we look for the termination of the monomorphic calculus with beta- and case-reductions if the calculus is well-structured.
The proof is an adaptation of well-known termination proofs of reduction of the simply-typed lambda-calculus, but adapted to the extended syntax and the extended set of rules. There are two differences: Our types have type constructors other than function types, and there are constructors and a case-construct.
The idea is to define a particular set of strongly computable (SC) expressions and analyzing their properties. First it is shown that SC expressions are strongly normalizable (SN), and then it is shown in a series of lemmas that all expressions are SC, which finally implies that all expressions are SN.

Definition 3.1. *An expression t is called* strongly normalizing (SN) *iff every reduction sequence of t terminates.*

Definition 3.2. *An expression t is called* strongly computable (SC) *iff the following holds (inductively):*

- *if t is of base type, then t is SN and when $t \xrightarrow{*} t'$, then every expression in* $\mathrm{MC}(t')$ *is SC.*
- *If t is of function type, then for all appropriately typed SC-expressions s_i: if $t\ s_1 \dots s_n$ is of constructed type, then it is SN and for $t\ s_1 \dots s_n \xrightarrow{*} t'$, also every expressions in* $\mathrm{MC}(t')$ *is SC.*

This inductive definition is based on a well-founded measure due to Lemma 2.2, which is only valid under the well-structured assumption.
Obviously the following holds:

Lemma 3.3. *Let t be SN. Then every subexpression of t is SN.*

Lemma 3.4. *If s, t are SC of appropriate type, then $(s\ t)$ is SC.*

Proof. Let $s_1, \dots, s_n$ be SC-expressions such that $s\ t\ s_1 \dots s_n$ is of base type. Since s, t are SC-expressions, the expression $s\ t\ s_1 \dots s_n$ is SN, by definition of SC. Since s is SC, by Definition 3.2, whenever $s\ t\ s_1 \dots s_n \xrightarrow{*} t'$ then also the expressions in $\mathrm{MC}(t')$ are SC. Hence $(s\ t)$ is also SC.

Lemma 3.5. *Every reduct of an SC-expression t is also SC.*

Proof. Let $t \to t'$. If t is of base type, then also t' is SN. If $t' \xrightarrow{*} t''$, then also $t \xrightarrow{*} t''$, hence the SC-condition holds. If t is of functional type, and s_i are SC, then $t\ s_1 \dots s_n \to t'\ s_1 \dots s_n$, and $t'\ s_1 \dots s_n$ is SN and also if $t'\ s_1 \dots s_n \xrightarrow{*} t''$, then also $t'\ s_1 \dots s_n \xrightarrow{*} t''$, and the SC-condition holds.

Lemma 3.6.

1. *All variables are SC.*
2. *All SC expressions are SN.*

Proof. Obvious, since $x\ s_1 \dots s_n$ has no top level reduction.

Lemma 3.7. *Let s_i be expressions and c be a constructor such that $(c\ s_1 \dots s_n)$ is typed. Then all s_i are SC iff $(c\ s_1 \dots s_n)$ is SC.*

Proof. Let s_i be SC. The expression is of constructed type, hence we have to prove that it is SN. Since reductions may only be in the subexpressions s_i, this follows from Lemma 3.6. Since every reduct of s_i is also SC by Lemma 3.5, the SC-condition holds.
Now assume that $(c\ s_1 \dots s_n)$ is SC. Obviously, s_i are SN. The fact $\mathrm{MC}(s_i) \subseteq \mathrm{MC}(c\ s_1 \dots s_n)$ shows that s_i are SC.

Lemma 3.8. *If t is SC and for all SC-expressions s, $(t[s/x])$ is SC, then $(\lambda x.t)$ is SC.*

Proof. Let s, s_i be SC-expressions such that $(t[s/x]\ s_1\ \dots\ s_n)$ is of constructed type. From the definition of SC and Lemma 3.6, we see that $(t[s/x]\ s_1\ \dots\ s_n)$ is SC, hence also SN. Let us show that $(\lambda x.t)$ is SC. Therefore again let s, s_i be any SC-expressions such that $(((\lambda x.t)\ s)\ s_1 \dots s_n)$ is of constructed type.
We have to show that this expression is SN. Consider an infinite reduction sequence of $(((\lambda x.t)\ s)\ s_1 \dots s_n)$. We know that t, s, s_i are all SN. Hence there is also an infinite reduction sequence of $(t[s/x]\ s_1\ \dots\ s_n)$, which is impossible by assumption and Lemma 3.6.
We also have to show that $((\lambda x.t)\ s)\ s_1 \dots s_n \xrightarrow{*} t'$ implies that $\mathrm{MC}(t')$ only contains SC-expressions. It is easy to see that for any reduction sequence $((\lambda x.t)\ s)\ s_1 \dots s_n \xrightarrow{*} t'$, there is also a reduction $(t[s/x])\ s_1\ \dots\ s_n \xrightarrow{*} t'$, by rearranging the reduction. Since $(t[s/x])$ is SC, all the expressions in $\mathrm{MC}(t')$ are SC.

Lemma 3.9. *For $l = 1, \dots, k$ let $Alt_l = (c_l\, x_{l,1} \dots x_{l,ar(c_l)}) \rightarrow r_l$. If $s_1, \dots, s_{ar(c_i)}, r_1, \dots, r_k$ and $(r_i[s_1/x_{i,1}, \dots, s_n/x_{i,ar(c_i)}])$ are SC, then $(\texttt{case}\ (c_i\ s_1 \dots s_{ar(c_i)})\ Alt_1\ \dots\ Alt_k)$ is SC.*

Proof. Let $a_i, i = 1, \ldots, m$ be arbitrary SC-expressions such that $((r_i[s_1/x_{i,1}, \ldots, s_n/x_{i,ar(c_i)}]\ a_1) \ldots a_m)$ is of constructed type. Since $r_i[s_1/x_{i,1}, \ldots, s_{ar(c_i)}/x_{i,ar(c_i)}]$ is SC it is also SN by Lemma 3.6.
We show that $(\texttt{case}\ (c_i\ s_1 \ldots s_{ar(c_i)})\ Alt_1 \ldots Alt_k)\ a_1 \ldots a_m$ is SN: Any infinite reduction will first reduce s_i, r_l, a_j to s'_i, r'_l, a'_j and since these are all SN, a case-reduction must follow with result $(r'_i[s'_1/x_{i,1}, \ldots, s'_{ar(c_i)}/x_{i,ar(c_i)}])\ a'_1 \ldots a'_m$, and then perhaps there may be other reductions. It is easy to see, that the expression $(r'_i[s'_1/x_{i,1}, \ldots, s'_{ar(c_i)}/x_{i,ar(c_i)}])\ a'_1 \ldots a'_m$ could be obtained by first performing the case-reduction with result $(r_i[s_1/x_{i,1}, \ldots, s_{ar(c_i)}/x_{i,ar(c_i)}])\ a_1 \ldots a_m$, and then reducing s_i, r_i, a_j to s'_i, r'_i, a'_j, where the reduction sequences may be necessary multiple times for the different copies of s_i and a_j and reductions for r_l with $l \neq i$ are omitted. Since $(r_i[s_1/x_{i,1}, \ldots, s_{ar(c_i)}/x_{i,ar(c_i)}])$ is SC by assumption, this contradicts Lemma 3.6, hence $(\texttt{case}\ (c_i\ s_1 \ldots s_{ar(c_i)})\ Alt_1 \ldots Alt_l)$ is SN.
The second part is to show that $(\texttt{case}\ (c_i\ s_1 \ldots s_{ar(c_i)})\ Alt_1 \ldots Alt_k)\ a_1 \ldots a_m \xrightarrow{*} t'$ implies that all expressions in MC(t') are SC. It is easy to see that also $(r_i[s_1/x_{i,1}, \ldots, s_{ar(c_i)}/x_{i,ar(c_i)}])\ a_1 \ldots a_m \xrightarrow{*} t'$, since the only potential reduction that does not only reduce within the expressions s_i, r_l, a_i is the case-reduction. Since $r_i[s_1/x_{i,1}, \ldots, s_{ar(c_i)}/x_{i,ar(c_i)}])$ is SC, it follows that all expressions in MC(t') are SC.

Lemma 3.10. *Let t be an expression all of whose free variables are in the set $\{x_1, \ldots, x_n\}$. Let s_i be expressions of the same type as x_i for $i = 1, \ldots, n$. If all s_i are SC, then with $\sigma := [s_1/x_1, \ldots, s_n/x_n]$, the expression $\sigma(t)$ is also SC.*

Proof. This proof is by induction on the expression structure:

- If t is one of the variables x_i, then $\sigma(t) = s_i$ which is SC by assumption.
- If t is a variable y not in $\{x_1, \ldots, x_n\}$, then y is SC by Lemma 3.6.
- If t is of the form $(c\ t_1 \ldots t_m)$, then every $\sigma(t_i)$ is SC by induction hypothesis. The expression $(c\ \sigma(t_1) \ldots \sigma(t_m))$ is SC by Lemma 3.7.
- If $t = t_1\ t_2$, then $\sigma(t) = \sigma(t_1)\ \sigma(t_2)$, and by induction the expressions $\sigma(t_i)$ are SC, hence by Lemma 3.4 $\sigma(t) = \sigma(t_1)\ \sigma(t_2)$ is SC.
- If $t = \lambda x.t_1$, then $\sigma(t)$ is $\lambda x.\sigma(t_1)$. Let $t_2 = \sigma'(t_1)$ where $\sigma' := [r/x, s_1/x_1, \ldots, s_n/x_n]$ and where r is any SC-expression. The expression t_2 is SC by the induction hypothesis of our proof, since t_1 is strictly smaller than t. Hence by Lemma 3.8, we obtain that $\sigma(\lambda x.t_1)$ is SC.
- If t is of the form $\texttt{case}\ t_1\ (c_1\ y_1 \ldots y_{m_1})\ \texttt{->}\ r_1; alts; (c_h\ y_1 \ldots y_{m_h})\ \texttt{->}\ r_h$, then t_1, $\sigma(t_1)$, r_i, and $\sigma(r_i)$ are SC by induction hypothesis. Let $a_1 \ldots a_p$ be SC-expressions such that $(\texttt{case}\ \sigma(t_1)\ (c_1\ y_1 \ldots y_{m_1})\ \texttt{->}\ \sigma(r_1); alts)\ a_1 \ldots a_p$ is of constructed type. If the reduction is only within $\sigma(t_1), \sigma(r_j), \sigma(a_i)$, then there can be no infinite reduction and also no reduction to a constructor expression. The other case is that there is a reduction $\sigma(t_1) \xrightarrow{*} (c_j\ d_1 \ldots d_k)$, and then a case-reduction. Lemma 3.5 shows that $(c_j\ d_1 \ldots d_k)$ is SC, and hence by Lemma 3.7, the d_i are SC. Let $\sigma' := \sigma \cup \{y_1 \mapsto d_1, \ldots, y_m \mapsto d_m\}$. Then $\sigma'(r_j)$ is SC by induction. Using Lemma 3.9, we see that also $(\texttt{case}\ (c_j\ d_1 \ldots d_k)\ (c_1\ y_1 \ldots y_{m_1})\ \texttt{->}\ \sigma(r_1); alts)\ a_1 \ldots a_p$ is SC. Whenever

there is a reduction (case $\sigma(t_1)$ $(c_1\ y_1 \ldots y_{m_1})$ -> $\sigma(r_1)$; *alts*) $a_1 \ldots a_p$ to an expression $(c' e_1 \ldots)$, there is also a reduction via an expression of the form (case $(c_j\ d_1 \ldots d_k)$ $(c_1\ y_1 \ldots y_{m_1})$ -> $\sigma(r_1)$; *alts*) $a_1 \ldots a_p$, which is SC, hence e_i are SC, and the proof is finished.

Theorem 3.11. *Every expression is SC, and hence SN.*

Proof. Simply use $t[x_1/x_1, \ldots x_n/x_n]$ where x_i are the free variables of t and then apply Lemma 3.10 and Lemma 3.6.

Corollary 3.12. *If t is a Haskell-expression with case and constructors and abstractions but* seq *is disallowed, and the well-structured condition holds for the data types, and typing uses only polymorphic typing without type classes, and supercombinator-reduction is not used, then monomorphically typed expressions have a terminating reduction.*

Corollary 3.13. *If t is an expression in a functional language with case and constructors and abstractions and there are polymorphic lists with types of constructors* Nil $::$ List(a), Cons $:: a \to$ List$(a) \to$ List(a), *Booleans, and Peano-numbers, and beta and case are used as reduction rules, then monomorphically typed expressions have a terminating reduction.*

Remark 3.14. As a first application of the claim that the proof can be extended: The strong termination claim can be extended if the expression syntax allows a seq in expressions (seq s t). Assume the seq-reduction is as follows:
seq $s\ t \to t$ if s is a constructor application or an abstraction.
Then the following has to be added as a lemma:
If s, t are SC, then (seq s t) is also SC. But this is easy, since (seq s t) $s_1 \ldots s_n$ for SC-expressions s_i is SN, provided all s_i, s, t are SN. If (seq s t) $s_1 \ldots s_n \xrightarrow{*} c\ r_1 \ldots r_m)$, then also $t\ s_1 \ldots s_n \xrightarrow{*} (c\ r_1 \ldots r_m)$, and all r_i are SC.

References

Ste72. Sören Stenlund. *Combinators, Lambda Terms, and Proof Theory.* D. Reidel, 1972.

Tai71. W.W. Tait. Normal form theorem for barrecursive functions of finite type. In J.E. Fenstad, editor, *Second Scandinavian Logic Symposium*, page 353367. NorthHolland, 1971.