Appears in *Proceedings of TLCA 2005, Nara, Japan*,
LNCS 3461, pp.262–277. Springer-Verlag, 2005

Reducibility and $\top\top$-lifting for Computation Types

Sam Lindley and Ian Stark*

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
{Ian.Stark,Sam.Lindley}@ed.ac.uk

Abstract. We propose $\top\top$-*lifting* as a technique for extending operational predicates to Moggi's monadic computation types, independent of the choice of monad. We demonstrate the method with an application to Girard-Tait reducibility, using this to prove strong normalisation for the computational metalanguage λ_{ml}. The particular challenge with reducibility is to apply this semantic notion at computation types when the exact meaning of "computation" (stateful, side-effecting, nondeterministic, etc.) is left unspecified. Our solution is to define reducibility for *continuations* and use that to support the jump from value types to computation types. The method appears robust: we apply it to show strong normalisation for the computational metalanguage extended with sums, and with exceptions. Based on these results, as well as previous work with local state, we suggest that this "leap-frog" approach offers a general method for raising concepts defined at value types up to observable properties of computations.

1 Introduction

Moggi's computational metalanguage λ_{ml} is a typed calculus for describing programming languages with real-world features like exceptions, nondeterminism and side-effects. It refines the pure simply-typed lambda-calculus by explicitly distinguishing *values* from *computations* in the type system: for each type A of values, there is a type TA of programs that compute a value of type A. The calculus specifies that the type constructor T be a *strong monad*, which is enough to support a wide range of notions of computation [5, 21, 22, 33].

In this paper we present $\top\top$-*lifting*: a method for reasoning about properties of computations in λ_{ml}, independent of the underlying monad, by raising up concepts defined explicitly on values.

We demonstrate the technique with a type-directed proof of strong normalisation for λ_{ml}, extending Girard-Tait reducibility to handle computation types. We also apply it to some extensions of λ_{ml}, and observe that $\top\top$-lifting gives a smooth treatment of reducibility for commuting conversions.

Section 2 provides a brief review of the computational metalanguage and related systems. Reduction in λ_{ml} properly extends that in the simply-typed lambda-calculus, with three reductions specific to computations. One of these, $T.assoc$, is a commuting

* Supported by an EPSRC Advanced Research Fellowship

conversion; another, $T.\beta$, involves substituting one term within another; which may make a term grow larger, and create subterms not present before. As usual with these kinds of reduction, the consequence is that straightforward induction over the structure of terms or types is not enough to prove termination of λ_{ml} reduction.

In earlier work, Benton et al. proved strong normalisation for λ_{ml} terms by translating them into a lambda-calculus with sums, and then invoking Prawitz's result for that system [4]. Our alternative is to use $\top\top$-lifting to give a standalone proof of strong normalisation, inductively on the structure of λ_{ml} types.

Section 3 sets out the details. We define an auxiliary notion of *reducibility* at every type, that is linked to strong normalisation but amenable to induction over the structure of types. This is a standard technique from the lambda-calculus: roughly, reducibility is the logical predicate induced by strong normalisation at ground types. We show that all reducible terms are strongly normalising, and go on to prove the fundamental theorem of logical relations, that in fact all definable terms are reducible.

The challenge, and the chief technical contribution of this paper, is to find a suitable definition for reducibility at computation types. Some such definition is essential, as the type constructor T is intentionally left unspecified. A first informal attempt might be to echo the definition for functions, and look at the immediate application of a computation:

(Bad 1) Term M of type TA is reducible if for all reducible N of type TB, the term $let\, x \Leftarrow M\ in\ N$ is reducible.

This is not inductive over types, as the definition of reducibility at type TA depends on reducibility at type TB, which may be more complex. We can try to patch this:

(Bad 2) Term M of type TA is reducible if for all strongly normalising N of type TB, the term $let\, x \Leftarrow M\ in\ N$ is strongly normalising.

However, this turns out to be too weak to prove properties of M in richer contexts like $let\, y \Leftarrow (let\, x \Leftarrow (-)\ in\ N)\ in\ P$. Examining the structure of these, we define a *continuation* K as a nested sequence of $let\, x_i \Leftarrow (-)\ in\ N_i$, and use these for our definition of reducibility:

(Good 1) Term M of type TA is reducible if for all reducible continuations K, the application $K \mathbin{@} M$ is strongly normalising.

Here application means pasting term M into the hole $(-)$ within K. Of course, we now have to define reducibility for continuations:

(Good 2) Continuation K accepting terms of type TA is reducible if for all reducible V of type A, the application $K \mathbin{@} [V]$ is strongly normalising.

The term $[V]$ is the trivial computation returning value V. By moving to the simpler value type A we avoid a potential circularity, and so get a notion of reducibility defined by induction on types. What is more, the characterisation by continuations is strong enough to treat both the commuting conversion $T.assoc$ and substitution in $T.\beta$, and the strong normalisation proof goes through without undue difficulty.

Looking beyond reducibility, this jump over continuations offers a quite general method to raise concepts from value type A up to computation type TA, whether or not we know the nature of T. Suppose that we write $K \top M$ when K applied to M is strongly normalising, and for any predicate $\phi \subseteq A$ define in turn:

$$\phi^{\top} = \{\, K \mid K \top [V] \text{ for all } V \in \phi \,\}$$
$$\phi^{\top\top} = \{\, M \mid K \top M \text{ for all } K \in \phi^{\top} \,\} \subseteq TA\ .$$

This is our operation of $\top\top$-lifting: to take a predicate ϕ on value type A and return another $\phi^{\top\top}$ on the computation type TA, by a "leap-frog" over $\phi^{\top}$ on continuations. One informal view of this is that continuations K represent possible observations on terms, and $\phi^{\top\top}$ lifts ϕ to computations based on their observable behaviour.

We believe that the use of $\top\top$-lifting in the metalanguage λ_{ml} is original. It was inspired by similar constructions applied to specific notions of computation; it is also related to Pitts's $\top\top$-closure, and that in turn has analogues in earlier work on reducibility. Section 5.1 discusses this further.

In Sect. 4 we demonstrate $\top\top$-lifting for reducibility in some variations of λ_{ml}; treating sums, exceptions, and Moggi's λ_c. For each case we vary our notion of continuation, but leave the definition of $(-)^{\top\top}$ unchanged. Notably, this includes the commuting conversions introduced by sums. Section 5 discusses related work, and concludes with some possible future directions.

2 The Computational Metalanguage

We start with a standard simply-typed lambda-calculus with ground type 0, product $A \times B$ and function space $A \to B$ for all types A and B. The *computational metalanguage* extends this with a type constructor T and two term constructions:

- For each A there is a type TA, of computations that return an answer in A.
- The *lifted* term $[M]$ is the computation which simply returns the answer M.
- The *composition* term $let\ x \Leftarrow M\ in\ N$ denotes computing M, binding the answer to x and then computing N.

Fig. 1 presents typing[1] and reduction rules for this language λ_{ml}. It corresponds to Moggi's λML_T [21]. In categorical terms, this is the internal language for a cartesian closed category with a strong monad T. More concretely, it is also what lies behind the use of monads in the Haskell programming language, where T is any type constructor in the `Monad` class and the term formers are `return M` for lifting and `do {x<-M; N}` for composition [24].

Often we do not require the full power of λ_{ml}, and there are two common simplifications: first, that all functions must return computations, thus having type $A \to TB$; and second, that this is the only place where T can occur. These constrain the

[1] Our presentation of typing follows Girard et al. [14], in that we assume a global assignment of types to variables. This is in contrast to typing "à la Curry" and typing "à la Church" [2], which use local *typing contexts*.

calculus to represent computations and only computations, disallowing *pure* functions of type $A \to B$ as well as *metacomputations* like those with type $TA \to TB$ and $T(TA)$.

With both of these restrictions in place we obtain the sub-calculus λ_{ml*}. This contains the call-by-value embedding [16] of the simply-typed lambda-calculus into the computational metalanguage; with the attention on functions of type $A \to TB$ embodying call-by-value semantics.

It turns out that the terms of λ_{ml*} are so constrained that we can dispense with explicit lifting and computation types, replacing them by a simple syntactic separation of values V from non-values M. This leaves only the let-construction, and we have a subset λ_{c*} of Moggi's computational lambda-calculus λ_c [20]. Sabry and Wadler discuss in detail the correspondences between λ_{ml}, λ_{ml*}, λ_{c*} and λ_c [28]. Our results on λ_{ml} apply directly to its restriction λ_{ml*}; however, λ_c has extra reduction rules, and in Sect. 4.3 we give a $\top\top$-lifting approach to cover these too.

The reductions for λ_{ml} appear in the last part of Fig. 1. These extend those for the simply-typed lambda-calculus with three reductions that act only on terms of computation type: $T.\beta$, $T.\eta$ and $T.assoc$. Before looking more closely at these three, we review some relevant properties of typed reduction, and the notion of strong normalisation.

Proposition 1. *Reduction in the computational metalanguage preserves types and is itself preserved under substitution.*

(i) *If* $M : A$ *and* $M \to M'$ *then* $M' : A$.
(ii) *If* $M \to M'$ *then* $M[x := N] \to M'[x := N]$.

Proof. Induction on the derivation of $M : A$ and the structure of M respectively. □

Definition 2. A term M in some calculus is *strongly normalising* (it is *SN*) if there is no infinite reduction sequence $M \to M_1 \to \cdots$. In this case we write $max(M)$ for the length of the longest reduction sequence starting from M. A calculus itself is strongly normalising if every term in it is strongly normalising.

We use the results of Prop. 1 repeatedly in the proofs for Sect. 3, and also the following:

Corollary 3. *If the* λ_{ml} *term* $M[x := N]$ *is strongly normalising, then so is* M.

Proof. By contradiction, from Prop. 1(ii): suppose M has some infinite reduction sequence $M \to M_1 \to \cdots$; then so does $M[x := N] \to M_1[x := N] \to \cdots$. If $M[x := N]$ has no such sequence, then neither does M and both are SN. □

It is standard that under β-reduction the untyped lambda-calculus is not strongly normalising. For example, the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ β-reduces to itself, leading to the infinite reduction sequence $\Omega \to_\beta \Omega \to_\beta \ldots$. On the other hand, the simply-typed lambda-calculus is strongly normalising with respect to β-reduction [14]: in particular, Ω has no simple type.

We shall be investigating strong normalisation with the additional terms and reductions of λ_{ml} from Fig. 1. The reductions to watch are $T.\beta$ and $T.assoc$: like $\to.\beta$, a

Syntax

$$\begin{array}{lrcl}
\text{Types} & A, B & ::= & 0 \mid A \to B \mid A \times B \mid TA \\
\text{Terms} & L, M, N, P & ::= & x^A \mid \lambda x^A.M \mid MN \mid \langle M, N \rangle \mid \pi_1(M) \mid \pi_2(M) \\
& & & \mid [M] \mid \mathit{let}\, x^A \Leftarrow M\, \mathit{in}\, N
\end{array}$$

Typing

$$\frac{}{x^A : A} \qquad \frac{M : B}{\lambda x^A.M : A \to B} \qquad \frac{M : A \to B \quad N : A}{MN : B} \qquad \frac{M : A \quad N : B}{\langle M, N \rangle : A \times B}$$

$$\frac{M : A}{[M] : TA} \qquad \frac{M : TA \quad N : TB}{\mathit{let}\, x^A \Leftarrow M\, \mathit{in}\, N : TB} \qquad \frac{M : A_1 \times A_2}{\pi_i(M) : A_i}\ i = 1, 2$$

Reductions

$$\begin{array}{lll}
\to.\beta & (\lambda x.M)N \longrightarrow M[x := N] & \\
\to.\eta & \lambda x.Mx \longrightarrow M & \text{if } x \notin fv(M) \\
\times.\beta i & \pi_i(\langle M_1, M_2 \rangle) \longrightarrow M_i & i = 1, 2 \\
\times.\eta & \langle \pi_1(M), \pi_2(M) \rangle \longrightarrow M & \\
T.\beta & \mathit{let}\, x \Leftarrow [N]\, \mathit{in}\, M \longrightarrow M[x := N] & \\
T.\eta & \mathit{let}\, x \Leftarrow M\, \mathit{in}\, [x] \longrightarrow M & \\
T.\mathit{assoc} & \mathit{let}\, y \Leftarrow (\mathit{let}\, x \Leftarrow L\, \mathit{in}\, M)\, \mathit{in}\, N \longrightarrow \mathit{let}\, x \Leftarrow L\, \mathit{in}\, (\mathit{let}\, y \Leftarrow M\, \mathit{in}\, N) & \text{if } x \notin fv(N)
\end{array}$$

Fig. 1. The computational metalanguage λ_{ml}.

$T.\beta$ step performs substitution, and so may enlarge the term at hand; while $T.\mathit{assoc}$ is a *commuting conversion*, also termed a *permutation* or *permutative conversion*. Commuting conversions are so named for their transforming action, via the Curry-Howard isomorphism, on derivation trees in natural deduction (indeed, the counterpart in logic of $T.\mathit{assoc}$ is described in [4]). They also arise when the lambda-calculus is extended with sums, and are known for the issues they can cause in proofs over reduction systems. Prawitz originally addressed this in [27]; see [17] for a discussion and further references. As we shall see below, $\top\top$-lifting uses structured continuations to perform proof over commuting conversions.

3 Reducibility

We present $\top\top$-lifting with the concrete example of a proof of strong normalisation in λ_{ml}, by extending the type-directed reducibility approach originally due to Tait [29]. We follow closely the style of Girard et al. [14, Chap. 6]; although in this short

presentation we focus on the proof parts specific to λ_{ml}, with full details appearing elsewhere [19]. As explained earlier, the key step is to find an appropriate definition of reducibility for computation types, which we do by introducing a mechanism for managing continuations.

3.1 Continuations

Informally, a continuation should capture how the result of a computation might be used in a larger program. Our formal definition is structured to support inductive proof about these uses.

- A *term abstraction* $(x)N$ of type $TA \multimap TB$ is a computation term N of type TB with a distinguished free variable x of type A.
- A *continuation* K is a finite list of term abstractions, with length $|K|$.

$$K ::= Id \mid K \circ (x)N \qquad \begin{aligned} |Id| &= 0 \\ |K \circ (x)N| &= |K| + 1 \end{aligned}$$

- Continuations have types assigned using the following rules:

$$Id : TA \multimap TA \qquad \frac{(x)N : TA \multimap TB \qquad K : TB \multimap TC}{K \circ (x)N : TA \multimap TC} \ .$$

- We apply a continuation of type $TA \multimap TB$ to a computation term M of type TA by wrapping M in let-statements that use it:

$$\begin{aligned} Id \mathbin{@} M &= M \\ (K \circ (x)N) \mathbin{@} M &= K \mathbin{@} (let\, x \Leftarrow M\, in\, N) \end{aligned}$$

  Notice that when $|K| > 1$ this is a nested stack of computations, not simple sequencing: i.e.

$$let\, x_1 \Leftarrow (let\, x_2 \Leftarrow (\ldots (let\, x_n \Leftarrow M\, in\, N_n)) \ldots in\, N_2)\, in\, N_1$$

  rather than

$$let\, x_1 \Leftarrow M_1\, in\, let\, x_2 \Leftarrow M_2\, in \ldots in\, let\, x_n \Leftarrow M_n\, in\, N \ .$$

  Although these two are interconvertible by a sequence of $T.assoc$ rewrites, we cannot identify them while we are looking to confirm strong normalisation in the presence of substituting rewrites like $\rightarrow.\beta$ and $T.\beta$.
  In fact, it is exactly this nesting structure that we use to tackle $T.assoc$ in our key Lemma 7; essentially, the stack depth of a continuation tracks the action of the commuting conversion.
- We define a notion of reduction on continuations:

$$\begin{aligned} K \rightarrow K' \quad &\stackrel{def}{\iff} \quad \forall M \,.\, K \mathbin{@} M \rightarrow K' \mathbin{@} M \\ &\iff \quad K \mathbin{@} x \rightarrow K' \mathbin{@} x \end{aligned}$$

  where the second equivalence follows from Prop. 1(ii). A continuation K is *strongly normalising* if all reduction sequences starting from K are finite; and in this case we write $max(K)$ for the length of the longest.

Lemma 4. *If $K \to K'$, for continuations K and K', then $|K'| \leq |K|$.*

Proof. Suppose $K = Id \circ (x_1)N_n \circ \cdots \circ (x_n)N_n$. Then its application $K \mathbin{@} x = let\, x_1 \Leftarrow (\ldots (let\, x_n \Leftarrow x\, in\, N_n) \ldots)\, in\, N_1$ and there are only two reductions that might change the length of K.

- $T.\eta$ where $N_i = [x_i]$ for some i. Then $K \to K'$ where $K' = Id \circ (x_1)N_1 \circ \cdots \circ (x_{i-1})N_{i-1} \circ (x_{i+1})N_{i+1} \circ \cdots \circ (x_n)N_n$ and $|K'| = |K| - 1$.
- $T.assoc$ may occur at position i for $1 \leq i < n$ to give $K' = (x_1)N_1 \circ \cdots \circ (x_{i-1})N_i \circ (x_{i+1})(let\, x_i \Leftarrow N_{i+1}\, in\, N_i) \circ (x_{i+2})N_{i+2} \circ \cdots \circ (x_n)N_n$. Again $|K'| = |K| - 1$.

Hence $|K'| \leq |K|$ as required. □

3.2 Reducibility and Neutrality

Figure 2 defines two sets by induction on the structure of types: reducible terms red_A of type A, and reducible continuations $\mathsf{red}_A^\top$ of type $TA \multimap TB$ for some B. As described in the introduction, for computations we use $\mathsf{red}_{TA} = \mathsf{red}_A^{\top\top}$.

We also need to classify some terms as *neutral*; we do this by decomposing every reduction into a rewrite context with a hole that must be plugged with a term of a particular form (see Fig. 2 again). From this we define:

- Term M is *active* if $R[M]$ is a redex for at least one of the rewrite contexts.
- Term M is *neutral* if $R[M]$ is not a redex for any of the rewrite contexts.

The neutral terms are those of the form x, MN, $\pi_1(M)$ and $\pi_2(M)$; i.e. computation types add no new neutral terms. The basic properties of reducibility now follow (**CR 1**)–(**CR 4**) of [14].

Theorem 5. *For every term M of type A, the following hold.*

(i) *If $M \in \mathsf{red}_A$, then M is strongly normalising.*
(ii) *If $M \in \mathsf{red}_A$ and $M \to M'$, then $M' \in \mathsf{red}_A$.*
(iii) *If M is neutral, and whenever $M \to M'$ then $M' \in \mathsf{red}_A$, then $M \in \mathsf{red}_A$.*
(iv) *If M is neutral and normal (has no reductions) then $M \in \mathsf{red}_A$.*

Proof. Part (iv) is a trivial consequence of (iii), so we only need to prove (i)–(iii), which we do by induction over types. The proof for ground, function and product types proceeds as usual [14]. Here we expand the details for computation types:

(i) Say $M \in \mathsf{red}_{TA}$. By the induction hypothesis (i), for every $N \in \mathsf{red}_A$ we have that N is SN, and so $[N]$ is too. This is enough to show that $Id : TA \multimap TA$ is in $\mathsf{red}_A^\top$, and so $Id \mathbin{@} M = M$ is SN as required.

(ii) Suppose $M \in \mathsf{red}_{TA}$ and $M \to M'$. For all $K \in \mathsf{red}_A^\top$, application $K \mathbin{@} M$ is SN, and $K \mathbin{@} M \to K \mathbin{@} M'$; thus $K \mathbin{@} M'$ is SN and $M' \in \mathsf{red}_{TA}$ as required.

(iii) Take $M : TA$ neutral with $M' \in \mathsf{red}_{TA}$ whenever $M \to M'$. We have to show that $K \mathbin{@} M$ is SN for each $K \in \mathsf{red}_A^\top$. First, we have that $K \mathbin{@} [x]$ is SN, as $x \in \mathsf{red}_A$ by the induction hypothesis (iv). Hence K itself is SN, and we can work by induction on $max(K)$.

Reducibility for terms and continuations

$M \in \mathsf{red}_0$	if the ground term M is strongly normalising
$F \in \mathsf{red}_{A \to B}$	if $FM \in \mathsf{red}_B$ for all $M \in \mathsf{red}_A$
$P \in \mathsf{red}_{A \times B}$	if $\pi_1(P) \in \mathsf{red}_A$ and $\pi_2(P) \in \mathsf{red}_B$
$M \in \mathsf{red}_{TA}$	if $K \mathbin{@} M$ is strongly normalising for all $K \in \mathsf{red}_A^\top$
$K \in \mathsf{red}_A^\top$	if $K \mathbin{@} [N]$ is strongly normalising for all terms $N \in \mathsf{red}_A$.

Reduction	**Rewrite context**	**Active term**
$\to.\beta$	$-N$	$\lambda x.M$
$\to.\eta$	$-$	$\lambda x.Mx$
$\times.\beta i$	$\pi_i(-)$	$\langle M, N \rangle$
$\times.\eta$	$-$	$\langle \pi_1(M), \pi_2(M) \rangle$
$T.\beta$	$let\ x \Leftarrow -\ in\ M$	$[N]$
$T.\eta$	$let\ x \Leftarrow M\ in\ -$	$[x]$
$T.assoc$	$let\ y \Leftarrow -\ in\ N$	$let\ x \Leftarrow L\ in\ M$

Fig. 2. Reducibility and neutrality for λ_{ml}

Application $K \mathbin{@} M$ may reduce as follows:

- $K \mathbin{@} M'$, where $M \to M'$, which is SN as $K \in \mathsf{red}_A^\top$ and $M' \in \mathsf{red}_{TA}$.
- $K' \mathbin{@} M$, where $K \to K'$. For any $N \in \mathsf{red}_A$, $K \mathbin{@} [N]$ is SN as $K \in \mathsf{red}_A^\top$; and $K \mathbin{@} [N] \to K' \mathbin{@} [N]$, so $K' \mathbin{@} [N]$ is also SN. From this we have $K' \in \mathsf{red}_A^\top$ with $max(K') < max(K)$, so by the induction hypothesis $K' \mathbin{@} M$ is SN.

There are no other possibilities as M is neutral. Hence $K \mathbin{@} M$ is strongly normalising for every $K \in \mathsf{red}_A^\top$, and so $M \in \mathsf{red}_{TA}$ as required. □

3.3 Reducibility Theorem

We show that all terms are reducible, and hence strongly normalising, by induction on their syntactic structure. This requires an appropriate lemma for each term constructor. Here we set out proofs for the new constructors associated with computation: lifting $[-]$ and let. The other cases follow as usual from the properties of Thm. 5, and are set out in [19].

Lemma 6. *Lifting preserves reducibility: if term $P \in \mathsf{red}_A$ then $[P] \in \mathsf{red}_{TA}$.*

Proof. For any continuation $K \in \mathsf{red}_A^\top$, the application $K \mathbin{@} [P]$ is SN, as $P \in \mathsf{red}_A$; and so $[P] \in \mathsf{red}_{TA}$. □

We next wish to show that formation of let-terms preserves reducibility. That will be Lemma 8, but we first need a result on the strong normalisation of let-terms in context.

This is the key component of our overall proof, and is where our attention to the stack-like structure of continuations pays off: the challenging case is the commuting conversion $T.assoc$, which does not change its component terms; but it does alter the continuation stack length, and this gives enough traction to maintain the induction proof.

Lemma 7. *Let $P : A$ be a term, $(x)N : TA \multimap TB$ a term abstraction, and $K : TB \multimap TC$ a continuation, such that both P and $K \mathbin{@} (N[x := P])$ are strongly normalising. Then $K \mathbin{@} (let\ x \Leftarrow [P]\ in\ N)$ is strongly normalising.*

Proof. We show by induction on $|K| + max(K \mathbin{@} N) + max(P)$ that the reducts of $K \mathbin{@} (let\ x \Leftarrow [P]\ in\ N)$ are all SN. The interesting reductions are as follows:

- $T.\beta$ giving $K \mathbin{@} (N[x := P])$, which is SN by hypothesis.
- $T.\eta$ when $N = [x]$, giving $K \mathbin{@} [P]$. But $K \mathbin{@} [P] = K \mathbin{@} (N[x := P])$, which is again SN by hypothesis.
- $T.assoc$ in the case where $K = K' \circ (y)M$ with $x \notin fv(M)$; giving the reduct $K' \mathbin{@} (let\ x \Leftarrow [P]\ in\ (let\ y \Leftarrow N\ in\ M))$. We aim to apply the induction hypothesis with K' and $(let\ y \Leftarrow N\ in\ M)$ for K and N, respectively. Now

$$\begin{aligned} K' \mathbin{@} ((let\ y \Leftarrow N\ in\ M)[x := P]) &= K' \mathbin{@} (let\ y \Leftarrow N[x := P]\ in\ M) \\ &= K \mathbin{@} (N[x := P]) \end{aligned}$$

  which is SN by hypothesis. Also

$$|K'| + max(K' \mathbin{@} (let\ y \Leftarrow N\ in\ M)) + max(P) < |K| + max(K \mathbin{@} N) + max(P)$$

  as $|K'| < |K|$ and $(K' \mathbin{@} (let\ y \Leftarrow N\ in\ M)) = (K \mathbin{@} N)$. This last equality explains our use of $max(K \mathbin{@} N)$; it remains fixed under $T.assoc$, unlike $max(K)$ and $max(N)$. Applying the induction hypothesis gives that $K' \mathbin{@} (let\ x \Leftarrow [P]\ in\ (let\ y \Leftarrow N\ in\ M))$ is SN as required.

Other reductions are confined to K, N or M, and can be treated by the induction hypothesis, decreasing either $max(K \mathbin{@} N)$ or $max(M)$. □

We are now in a position to prove that composing computations in let-terms preserves reducibility.

Lemma 8. *If $M \in \mathsf{red}_{TA}$ and $(x)N : TA \multimap TB$ has $N[x := P] \in \mathsf{red}_{TB}$ for all $P \in \mathsf{red}_A$, then $(let\ x \Leftarrow M\ in\ N) \in \mathsf{red}_{TB}$.*

Proof. Given a continuation $K \in \mathsf{red}_B^\top$, we must show that $K \mathbin{@} (let\ x \Leftarrow M\ in\ N)$ is SN. Now for any $P \in \mathsf{red}_A$, application $K \mathbin{@} (N[x := P])$ is SN, as $K \in \mathsf{red}_B^\top$ and $N[x := P] \in \mathsf{red}_{TB}$ by hypothesis. But P is also SN, by Thm. 5(i), and so Lemma 7 shows that $K \mathbin{@} (let\ x \Leftarrow [P]\ in\ N)$ is SN too. This proves that $(K \circ (x)N) \in \mathsf{red}_A^\top$, so applying it to $M \in \mathsf{red}_{TA}$ gives that $K \mathbin{@} (let\ x \Leftarrow M\ in\ N)$ is SN as required. □

We finally reach the desired theorem via a stronger result on substitutions into open terms.

Theorem 9. *Let $M : B$ be some term with free variables $x_1 : A_1, \ldots, x_k : A_k$. Then for any $N_1 \in \mathsf{red}_{A_1}, \ldots, N_k \in \mathsf{red}_{A_k}$ we have $M[x_1 := N_1, \ldots, x_k := N_k] \in \mathsf{red}_B$.*

Proof. By induction on the structure of the main term. For computation terms we have:

- $[P]$, where $P : A$. By the induction hypothesis $P[\vec{x} := \vec{N}] \in \mathsf{red}_A$, and by Lemma 6 we get $[P][\vec{x} := \vec{N}] = [P[\vec{x} := \vec{N}]] \in \mathsf{red}_{TA}$ as required.
- $\mathit{let}\, x \Leftarrow L \,\mathit{in}\, M$, where $L : TC$ and $M : TB$. The induction hypothesis is that $L[\vec{x} := \vec{N}] \in \mathsf{red}_{TC}$, and $M[\vec{x} := \vec{N}, x := P] \in \mathsf{red}_{TA}$ for all $P \in \mathsf{red}_C$. Lemma 8 gives $(\mathit{let}\, x \Leftarrow L \,\mathit{in}\, M)[\vec{x} := \vec{N}] = \mathit{let}\, x \Leftarrow L[\vec{x} := \vec{N}] \,\mathit{in}\, M[\vec{x} := \vec{N}] \in \mathsf{red}_{TA}$. □

Theorem 10. *Each λ_{ml} term $M : A$ is in red_A, and hence strongly normalising.*

Proof. Apply Thm. 9 with $N_i = x_i$, where $x_i \in \mathsf{red}_{A_i}$ by Thm. 5(iv). This tells us that $M \in \mathsf{red}_A$, and by Thm. 5(i) also strongly normalising. □

4 Extensions

In this section we apply ⊤⊤-lifting to reducibility in some extensions of λ_{ml}: with sum types, with exceptions; and in the computational lambda-calculus λ_c. Both sums and exceptions have existing normalisation results in the standard lambda-calculus (for example, [11] and [18, Thm. 6.1]); we know of no prior proofs for them in λ_{ml}. More important, though, is to see how ⊤⊤-lifting adapts to these features. The key step is to extend our formalized continuations with new kinds of observation. Once this is done, we can use these to lift predicates to computation types. The case of reducibility, and hence a proof of strong normalisation, then goes through as usual. Here we can only summarize, and full details appear in [19].

4.1 Reducibility for Sums

Prawitz first showed how to extend reducibility to sums [27]. His method is quite intricate: for a term M of sum type to be reducible, not only must the immediate subterms of M be reducible, but also a certain class of subterms of M' must be reducible whenever M reduces to M'. We avoid this complexity by defining reducibility for sums as we do for computations, by a leap-frog over continuations.

We begin by extending λ_{ml} with sum types and a *case* construct where each branch must be a computation (we later lift this constraint):

$$\frac{M : A}{\iota_1(M) : A + B} \qquad \frac{M : B}{\iota_2(M) : A + B}$$

$$\frac{M : A + B \quad N_1 : TC \quad N_2 : TC}{\mathit{case}\, M \,\mathit{of}\, \iota_1({x_1}^A) \Rightarrow N_1 \mid \iota_2({x_2}^B) \Rightarrow N_2 : TC}$$

To record possible observations of sum terms, we introduce *sum continuations*:

$$S ::= K \circ \langle (x_1)N_1, (x_2)N_2 \rangle$$
$$(K \circ \langle (x_1)N_1, (x_2)N_2 \rangle) \mathbin{@} M \;=\; K \mathbin{@} (\mathit{case}\, M \,\mathit{of}\, \iota_1(x_1) \Rightarrow N_1 \mid \iota_2(x_2) \Rightarrow N_2).$$

We can now define reducibility for sum continuations, and thence for sums.

– Sum continuation $S : A + B \multimap TC$ is in $\mathsf{red}^{\top}_{A+B}$ if:
  - $S \mathbin{@} (\iota_1(M))$ is strongly normalising for all $M \in \mathsf{red}_A$ and
  - $S \mathbin{@} (\iota_2(N))$ is strongly normalising for all $N \in \mathsf{red}_B$.
– Sum term $P : A + B$ is in red_{A+B} if $S \mathbin{@} P$ is strongly normalising for all $S \in \mathsf{red}^{\top}_{A+B}$.

This is then sufficient to prove strong normalisation for λ_{ml} with sums in the manner of Sect. 3.3.

To apply this to a more general *case* construction, we can move to *frame stacks*: nested collections of elimination contexts for any type constructor [26]. Frame stacks generalise continuations, and we have been able to use them to give a leap-frog definition of reducibility not just for computations, but also for sums, products and function types. This in turn gives a proof of strong normalisation for λ_{ml} with full sums, as well as the simply-typed lambda-calculus with sums [19, §3.5].

One special case of this brings us full circle: λ_{ml} trivially embeds into the simply-typed lambda-calculus with *unary* sums.

$$[M] \longmapsto \iota(M) \qquad \mathit{let}\, x \Leftarrow M\, \mathit{in}\, N \longmapsto \mathit{case}\, M\, \mathit{of}\, \iota(x) \Rightarrow N$$

The two languages are essentially the same, except that λ_{ml} has tighter typing rules and admits fewer reductions. Frame stacks and $\top\top$-reducibility then provide strong normalisation for both calculi.

4.2 Reducibility for Exceptions

Benton and Kennedy propose a novel syntax for incorporating exceptions into λ_{ml}, which they use within the SML.NET compiler [9]. They combine exceptions and let into the single construction $\mathit{try}\, x^A \Leftarrow M\, \mathit{in}\, N\, \mathit{unless}\, H$. This first evaluates M, then binds the result to x and evaluates N; unless an exception was raised in M, in which case it evaluates the *handler* H instead. The control flow of try-in-unless strictly extends the classic try-catch metaphor: for more on this see [9]; and also the rationale [10] for a similar recent extension of exception handling in the Erlang programming language.

Here we take exceptions E ranging over some fixed (possibly infinite) set; this is necessary to ensure termination [18]. A handler $H : TB$ is then a list of pairs (E, P) of exceptions and computations of type TB: evaluation picks the first pair that matches the exception to be handled; unmatched exceptions are re-raised. Typing rules are:

$$\frac{}{\mathit{raise}(E) : TA} \qquad \frac{M : TA \quad N : TB \quad H : TB}{\mathit{try}\, x^A \Leftarrow M\, \mathit{in}\, N\, \mathit{unless}\, H : TB}\,.$$

The original let is now a special case of try, with empty handler: $\mathit{let}\, x \Leftarrow M\, \mathit{in}\, N = \mathit{try}\, x \Leftarrow M\, \mathit{in}\, N\, \mathit{unless}\, \{\}$. Notice that we are not fixing our choice of monad T; it must support exceptions, but it may incorporate other effects too.

For $\top\top$-lifting in this calculus, we generalise continuations to cover the new observable behaviour of exception raising, by associating a handler to every step of the continuation.

$$K ::= \mathit{Id} \mid K \circ \langle (x)N, H \rangle$$
$$(K \circ \langle (x)N, H \rangle) \mathbin{@} M = K \mathbin{@} (\mathit{try}\, x \Leftarrow M\, \mathit{in}\, N\, \mathit{unless}\, H)$$

We now say that continuation K is in $\mathsf{red}_A^\top$ if:

- $K \mathbin{@} [N]$ is strongly normalising for all $N \in \mathsf{red}_A$; and in addition
- $K \mathbin{@} (raise(E))$ is strongly normalising for all exceptions E.

Building $\top\top$-reducibility on this is enough to give strong normalisation for λ_{ml} with exceptions, with a proof in the style of Sect. 3.3.

4.3 Reducibility for the Computational Lambda-Calculus

Strong normalisation for λ_{ml} immediately gives strong normalisation for the subcalculus λ_{ml*} described in Sect. 2. However, despite the close correspondence between λ_{ml*} and λ_c, explored in [28], we do not immediately get strong normalisation for λ_c. This is because of two additional reduction rules in λ_c:

$$\begin{array}{lrcll} let.1 & PM & \longrightarrow & let\, x \Leftarrow P\, in\, xM & \text{if } x \notin fv(M) \\ let.2 & VQ & \longrightarrow & let\, y \Leftarrow Q\, in\, Vy & \text{if } y \notin fv(V) \end{array}$$

where P, Q range over non-values, and V ranges over values.

We can adapt our proof, again using continuations in a leap-frog definition of reducibility:

Ground value	$V \in \mathsf{red}_0$	if V is strongly normalising
Function value	$V \in \mathsf{red}_{A \to B}$	if, for all $M \in \mathsf{red}_A \cup \mathsf{red}_A^{\top\top}$, $VM \in \mathsf{red}_B^{\top\top}$
Continuation	$K \in \mathsf{red}_A^\top$	if, for all $V \in \mathsf{red}_A$, $K \mathbin{@} V$ is strongly normalising
Non-value	$P \in \mathsf{red}_A^{\top\top}$	if, for all $K \in \mathsf{red}_A^\top$, $K \mathbin{@} P$ is strongly normalising

The distinction between values and non-values is crucial. There is no explicit computation type constructor in λ_c, but non-values are always computations. Thus red_A is reducible values of type A, and $\mathsf{red}_A^{\top\top}$ is reducible non-values of type A, playing the role of red_{TA}. This $\top\top$-reducibility leads as before to a proof of strong normalisation for λ_c, accounting for both additional reductions.

5 Conclusion

We have presented the leap-frog method of $\top\top$-lifting as a technique for raising operational predicates from type A to type TA, based on the observable behaviour of terms. This is independent of the nature of computations T, and introduces the opportunity of proof by induction on the structure of continuations.

As a concrete example, we demonstrated $\top\top$-lifting in a definition of reducibility for λ_{ml}, and thence a type-directed proof of strong normalisation. We have also applied this to some extensions of λ_{ml}, addressing in particular the robustness of the method in treating systems with commuting conversions.

In this final section we expand on the relation to other work on this topic, and comment on some possibilities for future research.

5.1 Related Work

We believe that our use of $\top\top$-lifting for computation types in λ_{ml} is new. It is, however, inspired by similar constructions applied to specific notions of computation. Pitts and Stark [25] apply the method to give a structurally inductive characterisation of observational equivalence for a functional language with local state. They then use this to validate certain proof techniques for reasoning about dynamically-allocated reference cells. Direct validation of these techniques had proved fruitless, because even though the precise form of computational effects was known — non-termination, state, and dynamic allocation — the interaction between them was intractable.

In [26], Pitts employs $\top\top$-*closure* to define an operational form of relational parametricity for a polymorphic PCF. Here the computational effect is nontermination, and $(-)^{\top\top}$ leads to an operational analogue of the semantic concept of "admissible" relations. Abadi in [1] investigates further the connection between $\top\top$-closure and admissibility.

The notion of $\top\top$-closed is different from our lifting: it expresses a property of a set of terms at a single type, whereas we lift a predicate ϕ on terms of type A to $\phi^{\top\top}$ on terms of a different type TA. However, the concept is clearly related, and the closure operation makes some appearance in the literature on reducibility, in connection with *saturation* and *saturated* sets of terms. Loosely, saturation is the property one wishes candidates for reducibility to satisfy; and this can sometimes be expressed as $\top\top$-closure. Examples include Girard's reducibility candidates for linear logic [13, pp. 72–73] and Parigot's work on $\lambda\mu$ and classical natural deduction [23, pp. 1469–1471]. For Girard the relevant continuations are the linear duals $A^{\perp}$, while for Parigot they are applicative contexts, lists of arguments in normal form $\mathcal{N}^{<\omega}$. We conjecture that in their style our $\top\top$-lifting could be presented as an insertion $\{\, [V] \mid V : \mathsf{red}_A \,\}$ followed by saturation (although we then lose the notion of reducible continuations).

Melliès and Vouillon use *biorthogonality* in their work on ideal models for types; this is a closure operation based on an orthogonality relation matching our $K \perp M$ [31, 32]. They make a case for the importance of orthogonality, highlighting the connection to reducibility. They also deconstruct contexts into frame stacks for finer analysis: elsewhere, Vouillon notes the correspondence between different forms of continuation and possible observations [30].

There are evident echoes of continuation-passing style in the leap-frog character of $\top\top$-lifting; and its independence from the choice of monad recalls Filinski's result that composable continuations can simulate all definable monads [12]. The apparent connection here is appealing, but we have not been able to make any formal link.

Goubault-Larrecq et al. investigate logical relations for computation types, proposing a distributivity law that these should satisfy [15]. They give a number of examples of logical relations lifted to specific monads; and, again, their chosen relation for the continuations monad has a similar structure to our $\top\top$-lifting.

As mentioned in the introduction, existing proofs of strong normalisation for λ_{ml} are based on translations into other calculi that are already known to be strongly normalising. We have said how Benton et al., working from a logical perspective, used a translation into a lambda-calculus with sums [4]. In a report on *monadic type systems* — a generalisation of pure type systems and the computational metalanguage

— Barthe et al. [3] prove strong normalisation by translation into a lambda-calculus with an extra reduction β'. Finally, Hatcliff and Danvy [16] state that T-reductions are strongly normalising, although they do not indicate a specific proof method.

5.2 Further Work

Subsequent to the work described here, we have developed a *normalisation by evaluation* algorithm for λ_{ml}, which we prove correct using the strong normalisation result. Normalisation by evaluation (NBE) then leads to further results on the theory of λ_{ml}: namely, that convertibility of terms is decidable, and reduction is confluent. This is described in detail in the first author's PhD thesis [19], which implements NBE for the version of λ_{ml} used as an intermediate language in the SML.NET compiler [7, 8], and evaluates its performance compared to conventional rewriting.

There is an extensive and growing body of work on the problem of normalisation for many varieties of typed lambda-calculi, with reducibility as just one approach. Joachimski and Matthes have proposed an alternative induction method, that characterises the strongly normalisable terms in a calculus [17]. This is proof-theoretically simpler, and it would be interesting to see how this applies to computation types in λ_{ml}. Their method covers sum types, commuting conversions and, most interestingly for us, *generalized applications* of the form $s(t, y.r)$. These have some resemblance to our decomposition of continuations: here $y.r$ is a term abstraction, to which will be passed the result of applying function s to argument t.

The broader test for $\top\top$-lifting is to investigate its application to other predicates or relations on λ_{ml} terms. Ultimately we want to make precise, and confirm, the informal conjecture of Kennedy and Benton that $(-)^{\top\top}$ captures "observation": if ϕ is some predicate on values, then $\phi^{\top\top}$ is a "best observable approximation" to it on computations [6].

References

[1] M. Abadi. $\top\top$-closed relations and admissibility. *Math. Struct. Comp. Sci.*, 10(3):313–320, 2000.

[2] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, vol. II, pp. 118–309. OUP, 1992.

[3] G. Barthe, J. Hatcliff, and P. Thiemann. Monadic type systems: Pure type systems for impure settings. In *Proc. HOOTS II*, ENTCS 10. Elsevier, 1997.

[4] P. N. Benton, G. Bierman, and V. de Paiva. Computational types from a logical perspective. *J. Funct. Prog.*, 8(2):177–193, 1998.

[5] P. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *Applied Semantics; Advanced Lectures*, LNCS 2395, pp. 42–122. Springer-Verlag, 2002.

[6] P. N. Benton and A. Kennedy. Personal communication, December 1998.

[7] P. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proc. ICFP '98*. ACM Press, 1998.

[8] P. N. Benton, A. Kennedy, C. Russo, and G. Russell. The SML.NET compiler. Available at `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

[9] P. N. Benton and A. J. Kennedy. Exceptional syntax. *J. Funct. Prog.*, 11(4):395–410, 2001.

[10] R. Carlsson, B. Gustavsson, and P. Nyblom. Erlang's exception handling revisited. In *Proc. ERLANG '04*, pp. 16–26. ACM Press, 2004.
[11] P. de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Inf. & Comp.*, 178(2):441–464, 2002.
[12] A. Filinski. Representing monads. In *Conf. Record POPL '94*, pp. 446–457. ACM Press, 1994.
[13] J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50(1):1–102, 1987.
[14] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. CUP, 1989.
[15] J. Goubault-Larrecq, S. Lasota, and D. Nowak. Logical relations for monadic types. In *Proc. CSL '02*, pp. 553–568, 2002.
[16] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Conf. Record POPL '94*, pp. 458–471. ACM Press, 1994.
[17] F. Joachimski and R. Matthes. Short proofs of normalization. *Arch. Math. Log.*, 42(1):58–87, 2003.
[18] M. Lillibridge. Unchecked exceptions can be strictly more powerful than call/cc. *Higher-Order & Symb. Comp.*, 12(1):75–104, 1999.
[19] S. Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, U. Edinburgh, 2005.
[20] E. Moggi. Computational lambda-calculus and monads. In *Proc. LICS '89*, pp. 14–23. IEEE Comp. Soc. Press, 1989.
[21] E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
[22] J. Newburn. All about monads, v1.1.0. `http://www.nomaware.com/monads`.
[23] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.
[24] S. Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. CUP, 2003.
[25] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pp. 227–273. CUP, 1998.
[26] A. M. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. Comp. Sci.*, 10:321–359, 2000.
[27] D. Prawitz. Ideas and results in proof theory. In *Proc. 2nd Scand. Log. Symp.*, Stud. Log. Found. Math. 63, pp. 235–307. North Holland, 1971.
[28] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Prog. Lang. Syst.*, 19(6):916–941, 1997.
[29] W. W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.
[30] J. Vouillon. Subtyping union types. In *Proc. CSL '04*, LNCS 3210, pp. 415–429. Springer-Verlag, 2004.
[31] J. Vouillon and P.-A. Melliès. Recursive polymorphic types and parametricity in an operational framework. Submitted for publication, 2004.
[32] J. Vouillon and P.-A. Melliès. Semantic types: a fresh look at the ideal model for types. In *Conf. Record POPL '04*, pp. 52–63. ACM Press, 2004.
[33] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, LNCS 925, pp. 24–52. Springer-Verlag, 1995.