Reducibility and Strong Normalisation for the Computational Metalanguage

Ian Stark and Sam Lindley

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh

Tuesday 14 October 2003

Overview

We prove strong normalisation for λ_{ML}, a lambda-calculus with types that distinguish computations from values. This leads to a general method to lift notions defined on values up to computations.

Outline of talk:

- Background and motivation: λ_{ML}, computation types.
- Strong normalisation by translation and some combinatorics
- Strong normalisation by Girard-Tait reducibility.

The challenge for reducibility is to apply this semantic notion to terms of computation type *whether or not we know what counts as a "computation"*.

Background

Moggi's *computational metalanguage* λ_{ML} provides a way to explicitly describe computations with side-effects within a pure typed lambda-calculus. The central feature is a new type constructor:

> For any type A of values there is a type $\mathrm{T}A$ of computations that return an answer in A.

Examples of computational effects include non-termination, exceptions, I/O, state, nondeterminism and jumps.

Types and terms of λ_{ML}

$$
\begin{array}{lrcl}
\text{Types} & A, B, C & ::= & 0 \mid A \to B \mid TA \\
\text{Terms} & M, N, P & ::= & x{:}A \mid \lambda x{:}A.M \mid MN \\
 & & \mid & [M] \mid \text{let } x{:}A \Leftarrow M \text{ in } N
\end{array}
$$

$$
\frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : TA}
\qquad
\frac{\Gamma \vdash M : TA \quad \Gamma, x{:}A \vdash N : TB}{\Gamma \vdash \text{let } x{:}A \Leftarrow M \text{ in } N : TB}
$$

The type constructor T acts as a categorical *strong monad*.

Some applications of λ_{ML}

- Denotational semantics: adapt pure models (domains, categories) uniformly to handle computational effects.
- Haskell: monads for mixing functional and stateful code, programming interactions with the real world.
- Compilers: MLj and SML.NET use a monadic intermediate language to carry out type-preserving compilation.

Reduction in λ_{ML}

$$(\beta) \qquad (\lambda x.M)N \longrightarrow M[N/x]$$

$$(\eta) \qquad \lambda x.Mx \longrightarrow M$$

$$(\text{let } \beta) \qquad \text{let } x \Leftarrow [V] \text{ in } N \longrightarrow N[V/x]$$

$$(\text{let } \eta) \qquad \text{let } x \Leftarrow M \text{ in } [x] \longrightarrow M$$

$$(\text{let assoc}) \qquad \text{let } x \Leftarrow (\text{let } y \Leftarrow M \text{ in } N) \text{ in } P$$

$$\longrightarrow \quad \text{let } y \Leftarrow M \text{ in } (\text{let } x \Leftarrow N \text{ in } P) \qquad y \notin fn(P)$$

Theorem. λ_{ML} *is strongly normalising: no term* $M \in \lambda_{ML}$ *has an infinite reduction sequence* $M \to M_1 \to \cdots$

First proof — translation

$$\phi(0) = 0 \qquad \phi(x) = x$$

$$\phi(TA) = \phi(A) \qquad \phi(MN) = \phi(M)\phi(N)$$

$$\phi(A \to B) = \phi(A) \to \phi(B) \qquad \phi(\lambda x.M) = \lambda x.\phi(M)$$

$$\phi(\,[M]\,) = \phi(M)$$

$$\phi(\text{let } x \Leftarrow M \text{ in } N) = (\lambda x.\phi(N))\phi(M)$$

Interpret T as the identity type constructor, with no computational effects.

Reductions translated

Standard lambda-calculus reductions are unchanged: β to β, η to η.

$$\begin{array}{rll} \phi(\text{let } \beta) & (\lambda x.N)M \longrightarrow N[M/x] & \\ \phi(\text{let } \eta) & (\lambda x.x)M \longrightarrow M & \\ \phi(\text{let assoc}) & (\lambda x.P)((\lambda y.N)M) \longrightarrow (\lambda y.(\lambda x.P)N))M & y \notin fn(P) \end{array}$$

This last rule is a strict extension of $\lambda_{\beta\eta}$, although it is admissible and a known “administrative” reduction from continuation-passing work.

Strong normalisation for $\lambda_{\beta\eta\text{assoc}}$

The following asymmetric measure decreases under η and (assoc).

$$s(x) = 1 \qquad s(\lambda x.M) = s(M) \qquad s(MN) = s(M) + 2s(N)$$

It may increase under β, so in addition we define $b(M) = ($max # β-reductions of $M)$ and use $\langle b(M), s(M)\rangle$ ordered lexicographically.

Lemma. $b(\,(\lambda x.P)((\lambda y.N)M)\,) \geq b(\,(\lambda y.(\lambda x.P)N)M\,)$

Proof. Explicit matching of β-reduction sequences on the right with others on the left, with some careful carrying and borrowing. $\square$

Thus $\lambda_{\beta\eta\text{assoc}}$is strongly normalising, hence $\lambda_{ML}$is also.

Second proof — reducibility

Translation works, but only because we happen to have a result for $\lambda_{\beta\eta}$ to hand. What can we do working with λ_{ML} directly?

For example, Tait's method for $\lambda_{\beta\eta}$, as presented in [GLT89]:

- Define *reducibility* of terms, by induction on types.
- Show useful properties of reducibility by induction on types; in particular that all reducible terms are strongly normalising.
- Show that all terms are reducible, by induction on term structure.

Reducibility for $\lambda_{\beta\eta}$

The definition of reducibility is by induction on types:

- A ground term $M : 0$ is reducible iff M is strongly normalising.
- A product term $M : A \times B$ is reducible iff $\mathtt{fst}(M)$ and $\mathtt{snd}(M)$ are both reducible.
- A function term $M : A \to B$ is reducible iff for all reducible $N : A$ the application $MN : B$ is reducible.

Properties of reducibility

(CR1) If M is reducible then it is strongly normalising.

(CR2) If M is reducible and $M \to M'$ then M' is reducible.

(CR3) If M is *neutral* (a variable or an application), and for all $M \to M'$ we have M' reducible, then M is reducible too.

Theorem. *All terms are reducible.*

Corollary. *All terms are strongly normalising.*

Non-definitions of reducibility at computation types

(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.

Not inductive over types.

(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.

Inductive, but not strong enough.

Continuations

- A *term abstraction* $(x)N$ is a computation term N with a distinguished free variable x.
- A *continuation* is a list of term abstractions:

$$K ::= \mathrm{Id} \quad | \quad K \circ (x)N$$

- We apply continuations as nested let-sequence:

$$\mathrm{Id}@M = M$$
$$(K \circ (x)N)@M = K@(\text{let } x \Leftarrow M \text{ in } N)$$

- Continuations reduce: $K \to K'$ iff $\forall M.K@M \to K'@M$.

Reducibility at computation types

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

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

Moving from TA to A avoids circularity, and we have a definition inductive over types. The characterisation is strong enough to follow through the standard results on reducibility and strong normalisation.

General “leap-frog” technique

Given a property Q_A defined by induction on the structure of type A, define some further properties as follows:

$$K \top M \iff K@M \text{ is strongly normalising}$$

Values	$V \in Q_A$	
Continuations	$K \in Q_A^\top$	$\iff \forall V \in Q_A \,.\, K \top [V]$
Computations	$M \in Q_A^{\top\top}$	$\iff \forall K \in Q_A^\top \,.\, K \top M$

Take $Q_{TA} = Q_A^{\top\top}$

This jump over continuations pushes any concept on values A up to one on computations TA, whether or not we know the nature of T.

Summary of results

$\lambda_{\beta\eta\mathrm{assoc}}$ is strongly normalising, building on the fact that $\lambda_{\beta\eta}$ is.

$\lambda_{\mathcal{ML}}$ is strongly normalising, by translation to $\lambda_{\beta\eta\mathrm{assoc}}$.

$\lambda_{\mathcal{ML}}$ is strongly normalising, by reducibility.

"Leapfrog" allows us to define reducibility for computations without knowing any specific details of the type constructor T.

Some related work

Normalisation in the computational metalanguage:

- Benton, Bierman and de Paiva (1998) give a modal logic corresponding to λ_{ML}, with accompanying proof normalisation.
- Filinski (2001) performs normalisation by evaluation for λ_{C}, which is equivalent to a proper subsystem of λ_{ML}.

Extending reasoning methods from values to computations:

- Pitts and Stark (1998) leapfrog a relation for proving operational equivalences between functional programs with local state.
- Pitts (2000) leapfrogs over nontermination to define an operational form of relational parametricity for polymorphic PCF. Abadi (2000) links that to admissibility in denotational semantics.

Intermediate λ_{ML}

The MLj and SML.NET compilers use a monadic intermediate language (MIL) to manage the translation from a higher-order functional language (Standard ML) into an imperative object-oriented bytecode (JVM / .NET).

Typed SML source code
↓
Complex MIL
⇣
Simplified MIL
↓
Verifiable bytecode

MIL is λ_{ML} extended with datatypes, exceptions, effects, *etc.*

This is *type-preserving* compilation, carrying types right through compilation to guide optimisation and help generate verifiable code.