Strong Normalization for the λ-calculus with Computational Monads

Ian Stark and Sam Lindley

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

Friday 15 November 2002

Overview

We are interested in general methods for reasoning about λ_{ML}, a lambda-calculus with types that distinguish computations from values. As an example, we prove strong normalization in two different ways.

Outline of talk:

- Background and motivation: λ_{ML}, computation types, MLj.
- Strong normalization by translation
- Strong normalization by reducibility

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}

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

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

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

Motivation

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

$\downarrow$

Complex MIL

$\downarrow$

Simplified MIL

$\downarrow$

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.

Reduction in λ_{ML}

$$
\begin{array}{lrcl}
(\beta) & (\lambda x.M)N & \longrightarrow & M[N/x] \\
(\eta) & \lambda x.Mx & \longrightarrow & M \\
(\text{let }\beta) & \text{let } x \Leftarrow [V] \text{ in } N & \longrightarrow & N[V/x] \\
(\text{let }\eta) & \text{let } x \Leftarrow M \text{ in } [x] & \longrightarrow & M
\end{array}
$$

$$
\begin{array}{ll}
(\text{let assoc}) & \text{let } x \Leftarrow (\text{let } y \Leftarrow M \text{ in } N) \text{ in } P \\
& \qquad \longrightarrow \quad \text{let } y \Leftarrow M \text{ in } (\text{let } x \Leftarrow N \text{ in } P) \qquad y \notin fn(P)
\end{array}
$$

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

First proof — translation

$$[\![O]\!] = O \qquad [\![x]\!] = x \qquad [\![\,[M]\,]\!] = [\![M]\!]$$

$$[\![TA]\!] = [\![A]\!] \qquad [\![MN]\!] = [\![M]\!][\![N]\!] \qquad [\![\text{let } x \Leftarrow M \text{ in } N]\!] = (\lambda x.[\![N]\!])[\![M]\!]$$

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad [\![\lambda x.M]\!] = \lambda x.[\![M]\!]$$

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.

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

This last rule is a strict extension of $\lambda_{\beta\eta}$, although it is known in work on continuation-passing.

Strong normalization for $\lambda_{\beta\eta\mathbf{assoc}}$

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

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

It may increase under β, so in addition we take
$b(M) =$ 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-reductions on the right with others on the left, with some careful carrying and borrowing. $\square$

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

Second proof — reducibility

By translating to $\lambda_{\beta\eta\mathrm{assoc}}$, we are reusing strong normalization for β-reduction. Can we instead show this for 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 (CR 1–3) by induction on types.
- 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 $M : O$ is reducible iff M is strongly normalizing.
- A function term $M : A \rightarrow B$ 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 normalizing.

(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 normalizing.*

Defining reducibility at computation types

- A *continuation* $(x)K : A \multimap TB$ is a computation term with a distinguished free variable x of type A.
- A continuation K is defined as *let-reducible* if $(\text{let } x \Leftarrow [V] \text{ in } K)$ is strongly normalizing for all reducible values V.
- Define a computation $M : TA$ to be reducible if $(\text{let } x \Leftarrow M \text{ in } K)$ is strongly normalizing for all let-reducible continuations K.

Now follow your nose to prove properties (CR1–3) and hence strong normalization for all of λ_{ML}.

General technique

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

$$M \perp K \iff (\text{let } x \Leftarrow M \text{ in } K) \text{ is strongly normalizing}$$

$$\text{Value } V \in Q_A$$

$$\text{Continuation } K \in Q_A^{\perp} \iff \forall V \in Q_A \,.\, [V] \perp K$$

$$\text{Computation } M \in Q_A^{\perp\perp} \iff \forall K \in Q_A^{\perp} \,.\, M \perp K$$

$$\text{Take } Q_{TA} = Q_A^{\perp\perp}$$

In situations without explicit computation types, this game of "leapfrog" can create a notion of property Q on expressions from one on values only.

Summary of results

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

λ_{ML} is strongly normalizing, by translation to $\lambda_{\beta\eta\text{assoc}}$.

λ_{ML} is strongly normalizing, by reducibility.

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

Some related work

Normalization in the computational metalanguage:

- Benton, Bierman and de Paiva (1998) give a modal logic corresponding to λ_{ML}, with accompanying proof normalization.
- Filinski (2001) performs normalization 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 (1997) leapfrog a relation for proving operational equivalences between functional programs with local state.
- Pitts (1998) uses leapfrog in operational reasoning about parametric polymorphism, where the relevant computational effect is nontermination.