Variations on Normalisation by Evaluation for the Lambda Calculus

Lucas O'Dowd-Jones

1 Introduction

1.1 What is NbE?

Indentify common features - eval and reify, interpret by semantic set in "host" language

Difference to reduction-based

1.2 Motivation for NbE

More efficient?

Avoids infinite reduction (terms not strongly normalisable - eg K x omega)

1.3 Why NbE in Haskell?

Modern technqiues for elegant implementation (GADTs)

2 NbE for the Untyped Lambda Calculus

2.1 Syntax

$$N_f := N_e \mid \lambda.N_f \tag{1}$$
$$N_e := N_e N_f \mid n \tag{2}$$

[6]

2.2 Semantics

Interpreted by the set V [6]

$$V := N_e \mid V \to V$$

Problem: fresh variables

2.3 Fresh variables solution 1 - Gensym

approach based on [6]

Implemented with State monad

Issue with solution 1: Have to add monad everywhere (inescapable) - all functions dependent on state

"Less functional" - carry around state (may as well use imperative)

2.4 Fresh variables solution 2 - Locally nameless terms

approach based on [1]

Uses de Bruijn Indicies for syntax and deBruijn levels for semantics

index n references nth abstraction, if m abstractions: if n ¡ m bound variables, otherwise free variable

Shifting for abstractions

3 Implementing the Typed Lambda Calculus

TypeOperators allows us to define types using infix operator :-¿. 'infixr :-¿ 9' sets strongest right assoicativity (matches usual convention that a -¿ b -¿ c a -¿ (b -¿ c))

Problem: Only want to normalise well typed terms. Criteria: (1) Exclude $\lambda x.y$ - y free variable so not in typing context (2) Exclude $\lambda x.xx$ - untypable (types are finite)

Problem with standard terms - can't carry type information

3.1 Typed Term solution - GADTs

[3] Explicit type constructor arguments Language extensions: DataKinds, TypeOperators, PolyKinds, GADTs.

Example 1: Vec - indexed type advantage since total function

Fails for replicate without work [7]

Advantage over ADTs: type refinement by constructor

Justification of base type *extension work??: string/integer type* Example 2: IsElem - value as proof Example 3: Expr - all values well typed by construction

Difference between [] and '[] (kinds vs types) in terms of levels

Big-step evalulator?

Input/Ouput type as pre-condition/post-condition (eg empty context)

extension work: Introduce context at value level (parameter to function)

Restricitions on Haskell Not full dependent types - need singleton pattern

4 NbE for the Typed Lambda Calculus

Investigation: Are GADTs in Haskell powerful enough? Types are erased at runtime so true dependent typing not part of Haskell (programs at type level)

Poissible to erase all type information, NbE on Untyped Issue: No proof that type preserved Solution: Track types as do evaluation - nbe program itself proof that types preserved (subject reduction parallel?)

Started by implementing same as untyped

Main difference in semantics (V := a -¿ b — Neutral) [6]

problem: Need to strengthen context evaluating body (eval Lam case)

4.1 Solution: Order Preserving Embeddings (OPEs)

Following implementation in Agda [4], agda has full dependent types (type system more powerful) - adapt for haskell, how nicely?

if a term well typed for one context, also well typed for any longer one

A value of type 'OPE strong weak' can derive weak from strong by dropping elements from context

OPE is a relation on typing contexts

4.2 Semantic set

Defintion of V using OPEs - Haskell vs agda

Need to quantify over 'strong' in function - OPE strong weak is guarentee that strong is a stronger context than weak (if quantified at start end up with values where weak stronger than strong) - need rank2 types extension for nested quantification

Helper functions (composition, strengthing relative to OPE) - explain derivations

4.3 implementing Eval

Defintion of environment (maps expressions in syntax context ctx to values in semantics with context ctxV)

problem: in app case how to we get identity OPE for semantic context?

But types erased at compile time to make Haskell efficient

How to generate a value at runtime dependent on type erased at compile time

dependent pattern match [5]

4.4 Solution: Singleton pattern

Method of Type to value known as reflection [5]

Idea: Create value-level tags for types - singleton types correspond type we're interested in, inhabited by only one value for each case

Examples: Reify case analysis, Ty reflection, Context reflection

Explicitly passing as value to pattern match on

Generate implictly using typeclass, use class constraint to imiplictly pass down ability to use contex methods through function calls. Is it a good idea to have class constraints in the GADTs/Syntax definitions?

Implementation in class vs full reflection - test this for speed?

problem : Inferring Any for ctxV (why?)

solution: scoped type variables - universally quantified variables used in type expressions bind over 'where' clause

(More usefully) can 'unpack' refined GADT types so that can create type definitions using refined types.

Analysis:

Have to specify type when normaling for correct eta-expansion (eta-long form)

Qs: How does locally nameless work in sematics? How does ctxV work in Env?

[2]

References

[1] Andreas Abel. "Normalization by Evaluation Dependent Types and Impredicativity". In: (). URL: http://www2.tcs.ifi.lmu.de/~abel/habil.pdf.

[2] Rowan Davies and Frank Pfenning. "A Modal Analysis of Staged Computation". In: (). URL: https://www.cs.cmu.edu/~fp/papers/jacm00.pdf.

[3] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". In: (). URL: https://www.cis.upenn.edu/~sweirich/papers/eisenberg-thesis.pdf.

[4] András Kovács. "A Machine-Checked Correctness Proof of Normalization by Evaluation for Simply Typed Lambda Calculus". URL: https://github.com/AndrasKovacs/stlc-nbe/blob/separate-PSh/thesis.pdf.

[5] Justin Le. *Introduction to Singletons*. URL: https://blog.jle.im/entry/introduction-to-singletons-1.html.

[6] Sam Lindley. *Normalisation by evaluation*. URL: http://homepages.inf.ed.ac.uk/slindley/nbe/nbe-cambridge2016.pdf.

[7] Niki Vazou. *CMSC498V*. URL: https://nikivazou.github.io/CMSC498V/lectures/DependentHaskell.html.