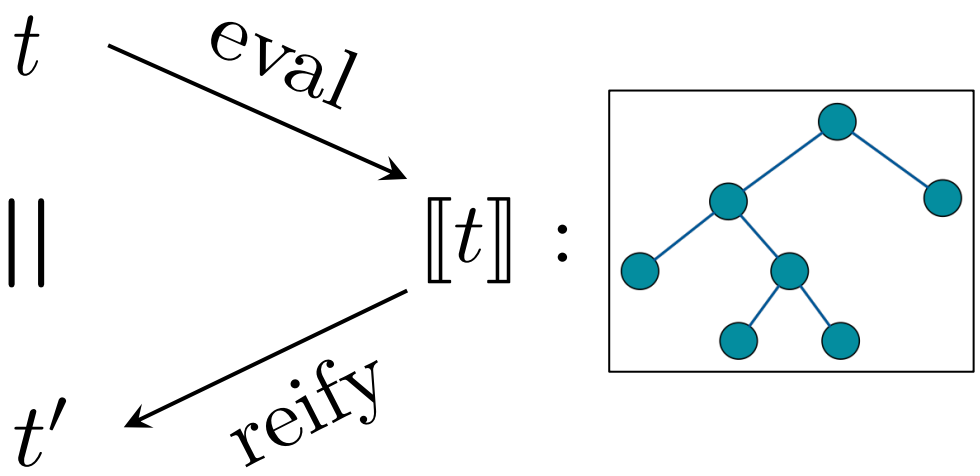
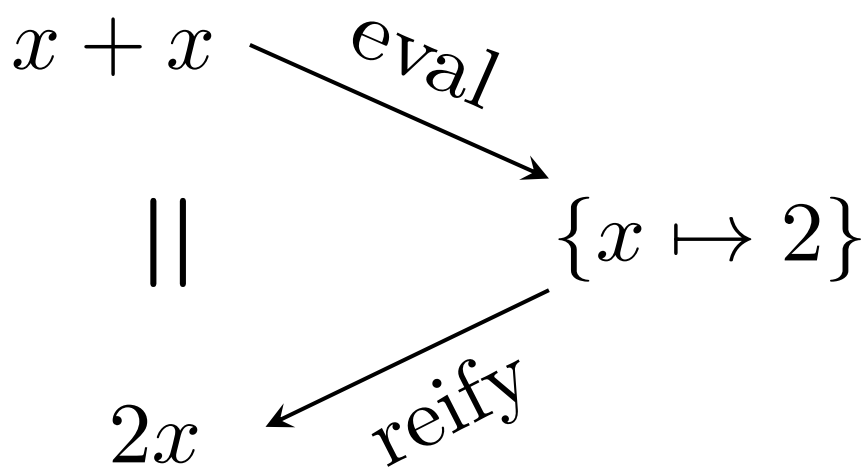
Residualizing Functors using Neighborhood Semantics

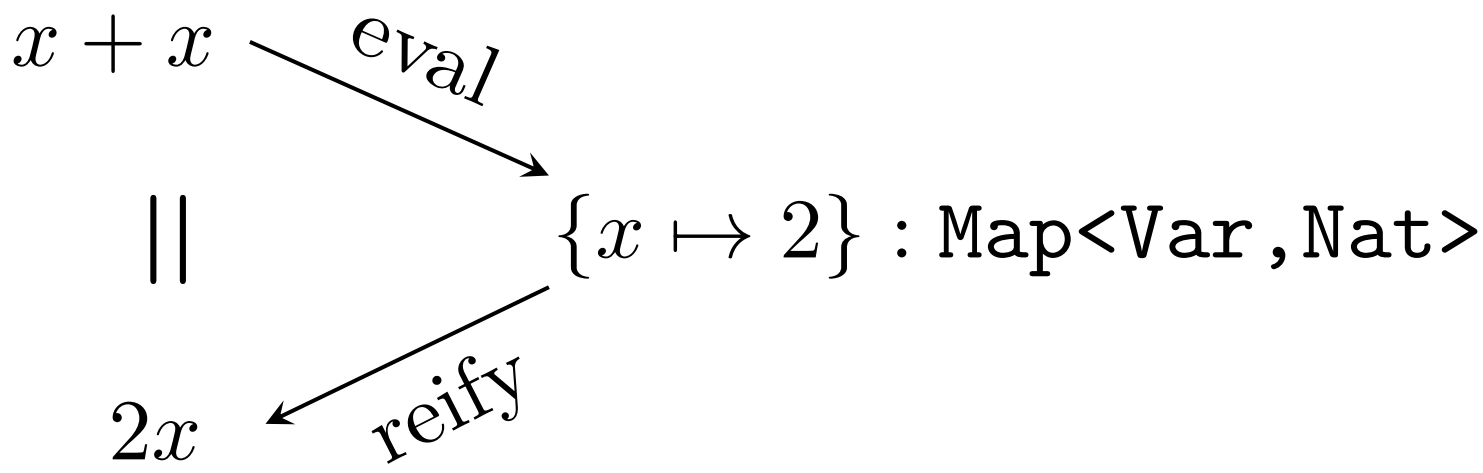
Nachi Valliappan

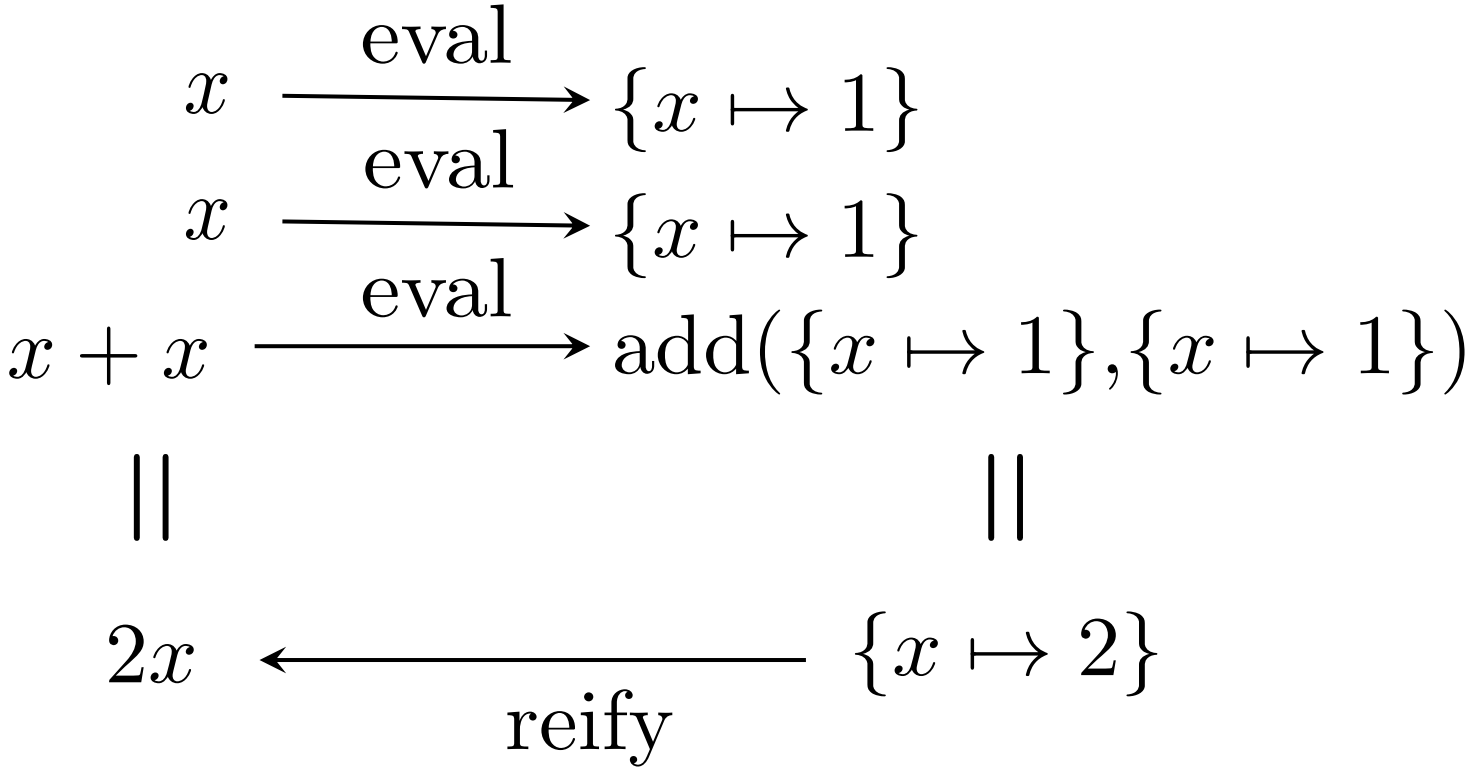
University of Edinburgh

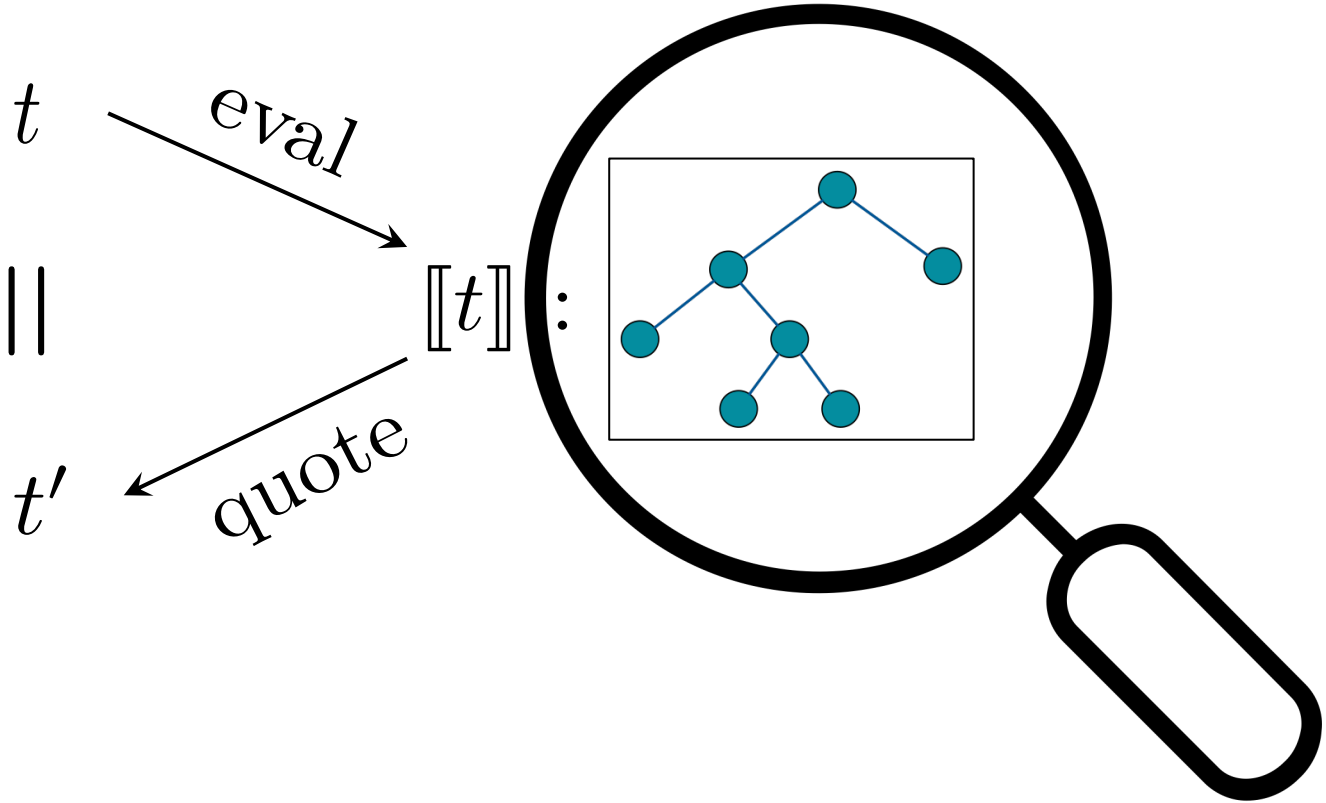
SPLS, University of Strathclyde, 03 December ‘25




$x + x$
eval
||
$\{x \mapsto 2\}$
reify
$2x$


$x + x$
eval
||
$\{x \mapsto 2\}$: Map<Var,Nat>
$2x$
reify


x
eval
{x ↦ 1}
x
eval
{x ↦ 1}
x + x
eval
add({x ↦ 1},{x ↦ 1})
||
||
2x
reify
{x ↦ 2}



$$\Gamma \vdash t : A$$

$$\Gamma \vdash t : FA$$

2. The Residualization Problem

Residualizing Semantics (conflated)

```
-- conflating object and host language types
class Res a where
      reify   :: a -> Exp a
      reflect :: Exp a -> a

eval :: Res a => Exp a -> a

norm :: Res a => Exp a -> Exp a
norm = reify ∘ eval
```

Residualizing Semantics (conflated)

```
-- conflating object and host language types
class Res a where
      reify   :: a -> Nf a
      reflect :: Ne a -> a

eval :: Res a => Exp a -> a

norm :: Res a => Exp a -> Nf a
norm = reify ∘ eval
```

Residualizing Monads (conflated)

```
class Monad m => ResMonad m where
      collect  :: m (Nf a) -> Nf (m a)
      register :: Ne (m a) -> m (Ne a)
```

Residualizing Monads (conflated)

```
instance (ResMonad m, Res a) => Res (m a) where
      reify :: m a -> Nf (m a)
      reify = collect ◦ fmap (reify @a)

      reflect :: Ne (m a) -> m a
      reflect = fmap (reflect @a) ◦ register
```

Residualizing Semantics

```
class ResFor a where
      type ⟦ a ⟧ :: *
      reify   :: ⟦ a ⟧ -> Nf a
      reflect :: Ne a -> ⟦ a ⟧

eval :: ResFor a => Exp a -> ⟦ a ⟧

norm :: ResFor a => Exp a -> Nf a
norm = reify ∘ eval
```

Residualizing Monads

```
class ResForMonad (m :: * -> *) where
      type ⟦ m ⟧ :: * -> *
      isMonad   :: Dict (Monad ⟦ m ⟧)
      collect   :: ⟦ m ⟧ (Nf a) -> Nf (m a)
      register  :: Ne (m a) -> ⟦ m ⟧ (Ne a)

instance ResForMonad m => Monad ⟦ m ⟧ where …
```

Residualizing Monads

```
instance (ResForMonad m, ResFor a) => ResFor (m a) where
      type ⟦ m a ⟧ = ⟦ m ⟧ ⟦ a ⟧

      reify :: ⟦ m a ⟧ -> Nf (m a)
      reify = collect ∘ fmap (reify @a)

      reflect :: Ne (m a) -> ⟦ m a ⟧
      reflect = fmap (reflect @a) ∘ register
```

Obligations for residualizing a monad

```
instance ResForMonad m where
  -- Identify an operator ⟦ m ⟧ s.t.
     -- ⟦ m ⟧ is a (strong) functor
     -- ⟦ m ⟧ is a monad
     -- ⟦ m ⟧ supports collect and register
```

2. The Residualization Trick

Recipe for residualizing monads

```
Inductively define a kind of free monad ⟦ m ⟧ and
  show ⟦ m ⟧ is a (strong) functor by induction
  show ⟦ m ⟧ is a monad by induction
  show ⟦ m ⟧ supports collect and register by induction
```

2001

Normalization by Evaluation for the Computational Lambda-Calculus

Andrzej Filinski

BRICS*, Department of Computer Science, University of Aarhus
andrzej@brics.dk

Abstract. We show how a simple semantic characterization of normalization by evaluation for the $\lambda_{\beta\eta}$-calculus can be extended to a similar construction for normalization of terms in the computational λ-calculus. Specifically, we show that a suitable *residualizing* interpretation of base types, constants, and computational effects allows us to extract a syntactic normal form from a term's denotation. The required interpretation can itself be constructed as the meaning of a suitable functional program in an ML-like language, leading directly to a practical normalization algorithm. The results extend easily to product and sum types, and can be seen as a formal basis for call-by-value type-directed partial evaluation.

2001

The accumulation monad over the monoid of $(\mathbf{V}\times\mathbf{E})$*-lists.* Writing $[]$ for the empty list, $[-]$ for a singleton list, and @ for list concatenation, we take:

$$
\begin{aligned}
T^{\mathrm{r}} A &= T^{\mathrm{g}}(A \times (\mathbf{V} \times \mathbf{E})^*) & \qquad \gamma^{\mathrm{g,r}}\, t &= t \star^{\mathrm{g}} \lambda a.\, \eta^{\mathrm{g}}\,(a, []) \\
\eta^{\mathrm{r}}\, a &= \eta^{\mathrm{g}}\,(a, []) & \qquad \mathit{bind}_\tau\, e &= \mathit{new}_\tau \star^{\mathrm{g}} \lambda v.\, \eta^{\mathrm{g}}\,(v, [(v, e)]) \\
t \star^{\mathrm{r}} f &= t \star^{\mathrm{g}} \lambda(a, l).\, f\, a \star^{\mathrm{g}} \lambda(b, l').\, \eta^{\mathrm{g}}\,(b, l \mathbin{@} l') & \qquad \mathit{collect}\, t &= t \star^{\mathrm{g}} \lambda(e, l).\, \eta^{\mathrm{g}}\,(\mathit{wrap}\, l\, e)
\end{aligned}
$$

with the auxiliary function $\mathit{wrap} : (\mathbf{V} \times \mathbf{E})^* \to \mathbf{E} \to \mathbf{E}$ defined inductively as

$$\mathit{wrap}\,[]\, e = e \qquad \mathit{wrap}\,([(v, e')] \mathbin{@} l)\, e = \mathit{LET}\,(v, e', \mathit{wrap}\, l\, e)$$

2009

Accumulating bindings

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

Abstract

We give a Haskell implementation of Filinski's normalisation by evaluation algorithm for the computational lambda-calculus with sums. Taking advantage of extensions to the GHC compiler, our implementation represents object language types as Haskell types and ensures that type errors are detected statically.

Following Filinski, the implementation is parameterised over a residualising monad. The standard residualising monad for sums is a continuation monad. Defunctionalising the uses of the continuation monad we present the binding tree monad as an alternative.

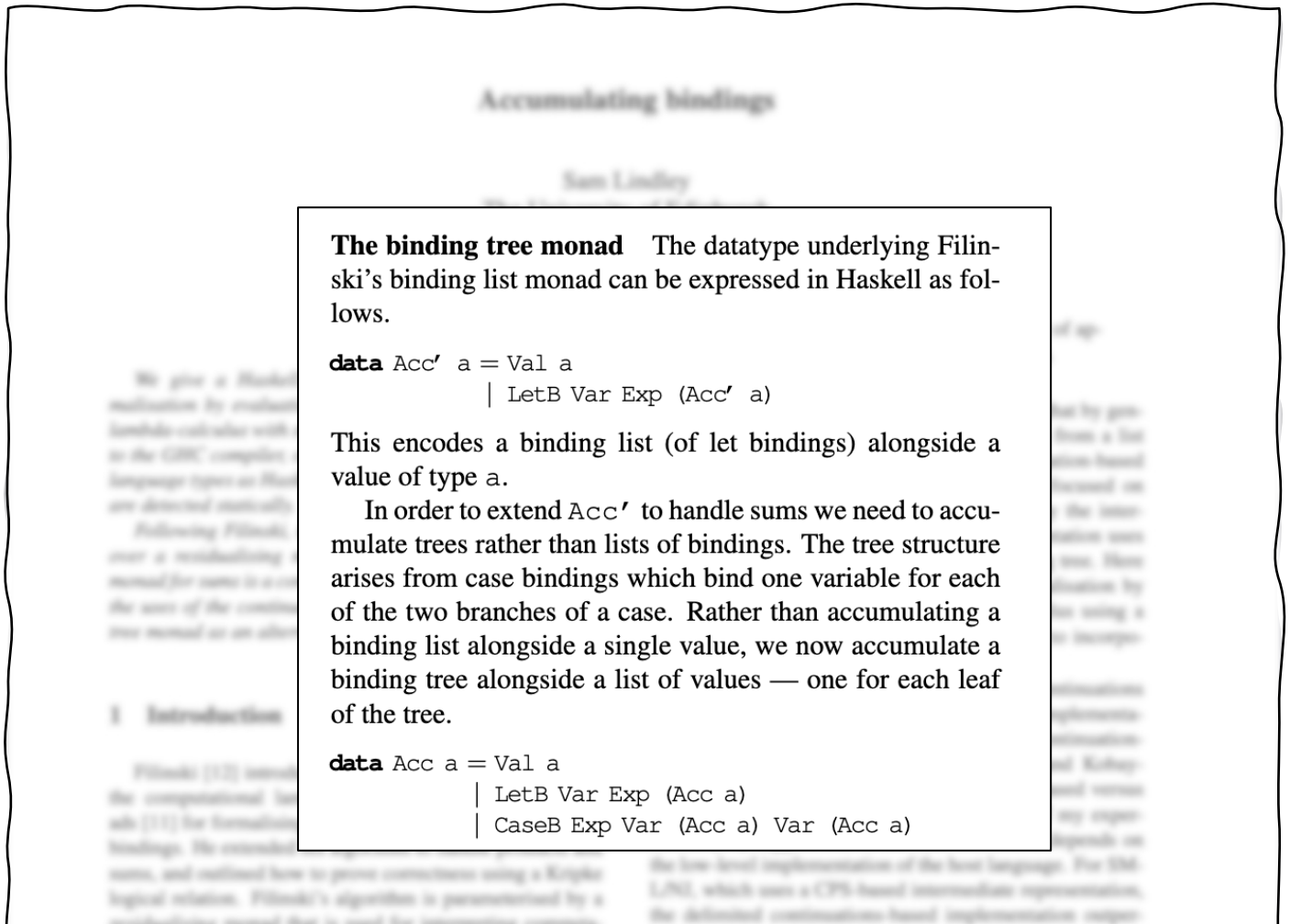
1 Introduction

Filinski [12] introduced normalisation by evaluation for the computational lambda calculus, using layered monads [11] for formalising name generation and for collecting bindings. He extended his algorithm to handle products and sums, and outlined how to prove correctness using a Kripke logical relation. Filinski's algorithm is parameterised by a *residualising* monad that is used for interpreting computations, but sums apparently require the full power of applying a single continuation multiple times.

In my PhD thesis I observed [14, Chapter 4] that by generalising the accumulation-based interpretation from a list to a tree that it is possible to use an accumulation-based interpretation for normalising sums. There I focused on an implementation using the state supported by the internal monad of the metalanguage. The implementation uses Huet's zipper [13] to navigate a mutable binding tree. Here we present a Haskell implementation of normalisation by evaluation for the computational lambda calculus using a generalisation of Filinski's binding list monad to incorporate a tree rather than a list of bindings.

(One motivation for using state instead of continuations is performance. We might expect a state-based implementation to be faster than an alternative delimited continuation-based implementation. For instance, Sumii and Kobayishi [17] claim a 3-4 times speed-up for state-based versus continuation-based let-insertion. The results of my experiments [14] suggest that in the case of sums it depends on the low-level implementation of the host language. For SML/NJ, which uses a CPS-based intermediate representation, the delimited continuations-based implementation outper-

2009



Accumulating bindings

Sam Lindley

The binding tree monad The datatype underlying Filinski's binding list monad can be expressed in Haskell as follows.

```
data Acc′ a = Val a
            | LetB Var Exp (Acc′ a)
```

This encodes a binding list (of let bindings) alongside a value of type `a`.

In order to extend `Acc′` to handle sums we need to accumulate trees rather than lists of bindings. The tree structure arises from case bindings which bind one variable for each of the two branches of a case. Rather than accumulating a binding list alongside a single value, we now accumulate a binding tree alongside a list of values — one for each leaf of the tree.

```
data Acc a = Val a
           | LetB Var Exp (Acc a)
           | CaseB Exp Var (Acc a) Var (Acc a)
```

2013

Normalization by evaluation and algebraic effects

Danel Ahman[1]

Laboratory for Foundations of Computer Science
University of Edinburgh

Sam Staton[2]

Computer Laboratory
University of Cambridge

Abstract

We examine the interplay between computational effects and higher types. We do this by presenting a normalization by evaluation algorithm for a language with function types as well as computational effects. We use algebraic theories to treat the computational effects in the normalization algorithm in a modular way. Our algorithm is presented in terms of an interpretation in a category of presheaves equipped with partial equivalence relations. The normalization algorithm and its correctness proofs are formalized in dependent type theory (Agda).

Keywords: Algebraic effects, Type theory, Normalization by evaluation, Presheaves, Monads

2013

Normalization by evaluation and algebraic effects

The monad T has the following concrete inductive description. Let $F : \mathsf{Ren} \to \mathsf{Set}$ be a presheaf. We define a new presheaf $TF : \mathsf{Ren} \to \mathsf{Set}$ so that the sets $TF(\Gamma)$ are the least satisfying the following rules:

$$\frac{d \in F(\Gamma)}{(\mathsf{T\text{-}return}\, d) \in TF(\Gamma)} \qquad \frac{\Gamma \Vdash^{\mathsf{a}}_{\mathsf{p}} M : \sigma \quad d \in TF(\Gamma, x{:}\sigma)}{(M\ \mathsf{T\text{-}to}\, x.\, d)\ \in TF(\Gamma)} \qquad \frac{d_1 \in TF(\Gamma)\ \ldots\ d_n \in TF(\Gamma)}{\mathsf{T\text{-}op}(d_1, \ldots, d_n)\ \in TF(\Gamma)}$$

2019

Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda-Calculus

Andreas Abel
Department of Computer Science and Engineering, Chalmers and Gothenburg University, Sweden
www.cse.chalmers.se/~abela
andreas.abel@gu.se

Christian Sattler
Department of Computer Science and Engineering, Chalmers and Gothenburg University, Sweden

Abstract

We observe that normalization by evaluation for simply-typed lambda-calculus with weak coproducts can be carried out in a weak bi-cartesian closed category of presheaves equipped with a monad that allows us to perform case distinction on neutral terms of sum type. The placement of the monad influences the normal forms we obtain: for instance, placing the monad on coproducts gives us eta-long beta-pi normal forms where pi refers to permutation of case distinctions out of elimination positions. We further observe that placing the monad on every coproduct is rather wasteful, and an optimal placement of the monad can be determined by considering polarized simple types inspired by focalization. Polarization classifies types into positive and negative, and it is sufficient to place the monad at the embedding of positive types into negative ones. We consider two calculi based

2019

Normalization by Evaluation for Call-by-Push-Value and Polarized

We are looking for a cover monad $\mathcal{C}$ that offers us these services:

$\mathsf{abort}^{\mathcal{C}}$	:	$\mathsf{Ne}\,0 \dot{\to} \mathcal{C}\,\mathcal{B}$	case on absurd neutral
$\mathsf{case}^{\mathcal{C}}_{\Gamma}$	:	$\mathsf{Ne}\,(A_1 + A_2)\,\Gamma \to \mathcal{C}\,\mathcal{B}\,(\Gamma.A_1) \to \mathcal{C}\,\mathcal{B}\,(\Gamma.A_2) \to \mathcal{C}\,\mathcal{B}\,\Gamma$	case on neutral
$\mathsf{runNf}^{\mathcal{C}}$	:	$\mathcal{C}\,(\mathsf{Nf}A) \dot{\to} \mathsf{Nf}A$	run the monad (Nf only)

To make things concrete, we shall immediately construct an instance of such a cover monad: the free cover monad Cov defined as an inductive family with constructors $\mathsf{return}^{\mathsf{Cov}}$, $\mathsf{abort}^{\mathsf{Cov}}$, and $\mathsf{case}^{\mathsf{Cov}}$. One can visualize an element $c : \mathsf{Cov}\,\mathcal{A}\,\Gamma$ as binary case tree whose inner nodes

We observe that normalization by evaluation for simply-typed lambda-calculus with weak coproducts can be carried out in a weak bi-cartesian closed category of presheaves equipped with a monad that allows us to perform case distinction on neutral terms of sum type. The placement of the monad influences the normal forms we obtain: for instance, placing the monad on coproducts gives us eta-long beta-pi normal forms where pi refers to permutation of case distinctions out of elimination positions. We further observe that placing the monad on every coproduct is rather wasteful, and an optimal placement of the monad can be determined by considering polarized simple types inspired by focalization. Polarization classifies types into positive and negative, and it is sufficient to place the monad at the embedding of positive types into negative ones. We consider two calculi based

2023

Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively

Matthew L. Daggitt
matthewdaggitt@gmail.com
Heriot-Watt University
Edinburgh, UK

Robert Atkey
robert.atkey@strath.ac.uk
University of Strathclyde
Glasgow, UK

Wen Kokke
University of Strathclyde
Glasgow, UK

Ekaterina Komendantskaya
Heriot-Watt University
Edinburgh, UK

Luca Arnaboldi
University of Edinburgh
Edinburgh, UK

Abstract

Modern verification tools frequently rely on compiling high-level specifications to SMT queries. However, the high-level specification language is usually more expressive than the available solvers and therefore some syntactically valid specifications must be rejected by the tool. In such cases, the challenge is to provide a comprehensible error message to the user that relates the original syntactic form of the specification to the semantic reason it has been rejected.

In this paper we demonstrate how this analysis may be performed by combining a standard unification-based type-checker with type classes and automatic generalisation. Concretely, type-checking is used as a constructive procedure for under-approximating whether a given specification lies

ACM Reference Format:
Matthew L. Daggitt, Robert Atkey, Wen Kokke, Ekaterina Komendantskaya, and Luca Arnaboldi. 2023. Compiling Higher-Order Specifications to SMT Solvers: How to Deal with Rejection Constructively. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3573105.3575674

1 Introduction

As the performance of SMT solvers and other automatic theorem provers has improved, they have been applied to

2023

The carrier of the Lift monad is defined as an indexed inductive type. Each constructor implicitly quantifies over all linear variable contexts Δ.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Lift}\ (A : \mathrm{Set}^{LinCtxt}) : \mathrm{Set}^{LinCtxt}\ \textbf{where}\\
&\quad \mathsf{return} \quad : A\,\Delta \to \mathsf{Lift}\,A\,\Delta\\
&\quad \mathsf{if} \quad : \mathsf{Constraint}\,\Delta \to \mathsf{Lift}\,A\,\Delta \to \mathsf{Lift}\,A\,\Delta \to \mathsf{Lift}\,A\,\Delta\\
&\quad \mathsf{letLinexp} \quad : \mathsf{LinExp}\,\Delta \to \mathsf{Lift}\,A\,(\Delta, \mathbb{Q}) \to \mathsf{Lift}\,A\,\Delta\\
&\quad \mathsf{letFunexp} : \mathsf{Var}\,\Delta \to \mathsf{Lift}\,A\,(\Delta, \mathbb{Q}) \to \mathsf{Lift}\,A\,\Delta
\end{aligned}
$$

$[\![$ `a` $]\!]$ `::` `*`

$[\![$ `m` $]\!]$ `::` `* -> *`

$[\![A]\!] : W \to \mathrm{Set}$

$[\![M]\!] : (W \to \mathrm{Set}) \to (W \to \mathrm{Set})$

The Maybe Monad

$$\Gamma \vdash \mathbf{nothing} : \mathbb{M}A \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{just}\ t : \mathbb{M}A}$$

$$\frac{\Gamma \vdash t : \mathbb{M}A \qquad \Gamma, x : A \vdash u : \mathbb{M}B}{\Gamma \vdash \mathbf{let}\ x = t\ \mathrm{in}\ u : \mathbb{M}B}$$

Normal Forms for the Maybe Monad

$$\Gamma \vdash_{\mathrm{NF}} \mathbf{nothing} : \mathbb{M}A \qquad \frac{\Gamma \vdash_{\mathrm{NF}} t : A}{\Gamma \vdash_{\mathrm{NF}} \mathbf{just}\ t : \mathbb{M}A}$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} t : \mathbb{M}A \qquad \Gamma, x : A \vdash_{\mathrm{NF}} u : \mathbb{M}B}{\Gamma \vdash_{\mathrm{NF}} \mathbf{let}\ x = t\ \mathrm{in}\ u : \mathbb{M}B}$$

Define Residualization Operator for Maybe

$$\mathbf{nothing} : [\![\mathbb{M}]\!]_\Gamma\, X \qquad \frac{x : X_\Gamma}{\mathbf{just}\ x : [\![\mathbb{M}]\!]_\Gamma\, X}$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} t : \mathbb{M}A \qquad m : [\![\mathbb{M}]\!]_{\Gamma,A}\, X}{\mathbf{let}\ t\ m : [\![\mathbb{M}]\!]_\Gamma\, X}$$

$$X : \mathrm{Ctx} \to \mathrm{Set}$$

Show the Residualization Operator is a (Maybe) Monad

$$(X \dot{\to} Y) \to (\llbracket \mathbb{M} \rrbracket X \dot{\to} \llbracket \mathbb{M} \rrbracket Y)$$

$$X \times \llbracket \mathbb{M} \rrbracket Y \dot{\to} \llbracket \mathbb{M} \rrbracket (X \times Y)$$

$$1 \dot{\to} \llbracket \mathbb{M} \rrbracket X$$

$$X \dot{\to} \llbracket \mathbb{M} \rrbracket X$$

$$\llbracket \mathbb{M} \rrbracket (\llbracket \mathbb{M} \rrbracket X) \dot{\to} \llbracket \mathbb{M} \rrbracket X$$

$$X \dot{\to} Y \triangleq \forall w.\, X_w \to Y_w$$

The Residualization Trick lacks reusability

… is laborious to prove correct

… requires intervention when features are added

… has nothing to do with monads

New Equations for Neutral Terms

A Sound and Complete Decision Procedure, Formalized

Guillaume Allais Conor McBride
University of Strathclyde
{guillaume.allais, conor.mcbride}@strath.ac.uk

Pierre Boutillier
PPS - Paris Diderot
pierre.boutillier@pps.univ-paris-diderot.fr

… is learnt by experience

… is understood by a select few

Residualization for all!

2. Residualizing Functors, Systematically

Obligations for residualizing functors with some structure

```
-- Identify an operator ⟦ f ⟧ s.t.
  -- ⟦ f ⟧ is a functor
  -- ⟦ f ⟧ is a strong functor
  -- ⟦ f ⟧ is a monad
  -- ⟦ f ⟧ is a comonad
  -- ⟦ f ⟧ is a product-preserving functor
  -- ⟦ f ⟧ is a …
  -- ⟦ f ⟧ supports collect and register
```

Scope for today

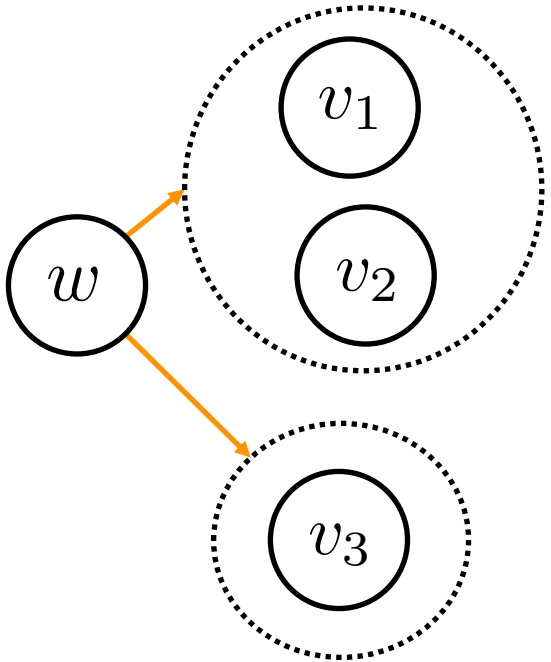
preorder

$(W, \leq, \mathcal{N})$ } configuration

$\mathcal{N} : W \to \mathcal{P}(\mathcal{P}(W))$

+ compatibility conditions

$$\mathcal{N}(w) = \{\{v_1, v_2\}, \{v_3\}\}$$



$$(W, \leq, \mathcal{N})$$

$$\mathcal{C} : (W \to \mathrm{Set}) \to (W \to \mathrm{Set})$$
$$\mathcal{C}A_w = \Sigma\alpha.\, \alpha \in \mathcal{N}(w) \times \{A_v\}_{v \in \alpha}$$

“Cover Modality”

$$(X \dot{\to} Y) \to (\mathcal{C}X \dot{\to} \mathcal{C}Y)$$

$$\alpha \in \mathcal{N}(w) \Rightarrow \forall v \in \alpha.\, w \leq v$$

$$\frac{}{\mathcal{N} \text{ is reachable}}$$

$$\emptyset \in \mathcal{N}(w)$$

$$\{w\} \in \mathcal{N}(w)$$

$$\mathcal{N} \text{ is transitive}$$

$$\Rightarrow$$

$$X \times \mathcal{C}Y \dot{\to} \mathcal{C}(X \times Y)$$

$$1 \dot{\to} \mathcal{C}X$$

$$X \dot{\to} \mathcal{C}X$$

$$\mathcal{C}(\mathcal{C}X) \dot{\to} \mathcal{C}X$$

$$\mathcal{N}(w) \neq \emptyset$$

$\mathcal{N}$ is closed under $\cap$

$$\alpha \in \mathcal{N}(w) \Rightarrow w \in \alpha$$

$\mathcal{N}$ is dense

$\Rightarrow$

$$1 \dot{\to} \mathcal{C}1$$

$$\mathcal{C}X \times \mathcal{C}Y \dot{\to} \mathcal{C}(X \times Y)$$

$$\mathcal{C}X \dot{\to} X$$

$$\mathcal{C}X \dot{\to} \mathcal{C}(\mathcal{C}X)$$

Define Residualization Operator for Maybe

$$(\{\Gamma, \Delta, \ldots\}, \leq, \mathcal{N})$$

$$\textbf{empty} : \emptyset \in \mathcal{N}(\Gamma) \qquad \textbf{single} : \{\Gamma\} \in \mathcal{N}(\Gamma)$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} t : \mathbb{M}A \qquad p : \alpha \in \mathcal{N}(\Gamma, A)}{\textbf{cons}\ t\ p : \alpha \in \mathcal{N}(\Gamma)}$$

$$\text{Obs. } \mathcal{C}X \cong [\![\mathbb{M}]\!]X$$

Show the Residualization Operator is a (Maybe) Monad

$$\mathcal{N} \text{ is reachable} \qquad X \times [\![\mathbb{M}]\!]Y \xrightarrow{\cdot} [\![\mathbb{M}]\!](X \times Y)$$

$$\emptyset \in \mathcal{N}(\Gamma) \qquad \Rightarrow \qquad 1 \xrightarrow{\cdot} [\![\mathbb{M}]\!]X$$

$$\{\Gamma\} \in \mathcal{N}(\Gamma) \qquad X \xrightarrow{\cdot} [\![\mathbb{M}]\!]X$$

$$\mathcal{N} \text{ is transitive} \qquad [\![\mathbb{M}]\!]([\![\mathbb{M}]\!]X) \xrightarrow{\cdot} [\![\mathbb{M}]\!]X$$

There are $(W, \leq, \mathcal{N})$ for which

- $\mathcal{C}$ residualizes algebraic effects
- $\mathcal{C}$ residualizes Err (with catch)
- $\mathcal{C}$ residualizes $\Box$ in D.C. modal λ-calculi
- $\mathcal{C}$ residualizes $\Diamond$ in Lax modal λ-calculi
- $\mathcal{C}$ residualizes 🔒 in F.S. modal λ-calculi

Under what conditions is $\mathcal{C}$ stable under sums?

$$(W, \leq, \mathcal{N})$$

$$\mathcal{C}A_w = \Sigma\alpha.\, \alpha \in \mathcal{N}(w) \times \{A_v\}_{v \in \alpha}$$

$$+ \text{ a library of properties of } \mathcal{C}$$

$$- \text{ generic collect and register}$$

github.com/nachivpn/frames

github.com/nachivpn/presheaves

github.com/nachivpn/cover

EOM