Utrecht University

Analysis and Transformation of Intrinsically Typed Syntax

Master's Thesis

Matthias Heinzel

Utrecht University

Analysis and Transformation

Variable Representations

Intrinsically Typed de Bruijn Representation

Intrinsically Typed Co-de-Bruijn Representation

Syntax-generic Co-de-Bruijn Representation

Other Transformations

Discussion

Analysis and Transformation

Expression Language

$$
\begin{aligned}
P, Q ::= {} & x \\
\mid {} & P\ Q \\
\mid {} & \lambda x.\ P \\
\mid {} & \textbf{let}\ x = P\ \textbf{in}\ Q \\
\mid {} & v \\
\mid {} & P + Q
\end{aligned}
$$

- based on λ-calculus
    - well studied notion of computation
- we add let-bindings, Booleans, integers and addition

Analysis and Transformation

- fundamental part of compilers
- we focus on those dealing with bindings
- in this presentation: dead binding elimination (DBE)

Dead Binding Elimination (DBE)

- remove dead (unused) bindings
- which bindings exactly are dead?
  - x occurs in its body
  - but only in declaration of y

$$
\begin{array}{l}
\textbf{let } x = 42 \textbf{ in} \\
\quad \textbf{let } y = x \textbf{ in} \\
\qquad 1337
\end{array}
$$

Live Variable Analysis (LVA)

- collect live variables, bottom up
- for *strongly* live variable analysis, at let-binding:
  - only consider declaration if its binding is live

$$
\begin{array}{l}
\textbf{let } x = 42 \textbf{ in} \\
\quad \textbf{let } y = x \textbf{ in} \\
\qquad 1337
\end{array}
$$

Variable Representations

Named Representation

- what we have done so far, just use strings
- pitfall: shadowing, variable capture
    - e.g. inline y in expression **let** $y = x + 1$ **in** $\lambda x.\ y$
    - usually avoided by convention/discipline
    - mistakes still happen

De Bruijn Representation

- no names, de Bruijn indices are natural numbers
- *relative* reference to binding (0 = innermost)

$$\begin{array}{l} \textbf{let } x = 42 \textbf{ in} \\ \quad \textbf{let } y = 99 \textbf{ in} \\ \quad\quad x \end{array}$$

$$\begin{array}{l} \textbf{let } 42 \textbf{ in} \\ \quad \textbf{let } 99 \textbf{ in} \\ \quad\quad \langle 1 \rangle \end{array}$$

- pitfall: need to rename when adding/removing bindings
- not intuitive for humans

Other Representations

- co-de-Bruijn
- higher-order abstract syntax (HOAS)
- combinations of multiple techniques
- ... [1]

[1] http://jesper.sikanda.be/posts/1001-syntax-representations.html

Intrinsically Typed de Bruijn Representation

Naive Syntax

```
data Expr : Set where
  Var  : Nat → Expr
  App  : Expr → Expr → Expr
  Lam  : Expr → Expr
  ...
```

- What about `App (Bln False) (Var 42)`?
- error-prone, evaluation is partial

Sorts

- solution: index expressions by their sort (type of their result)

```
data U : Set where
  _⇒_ : U → U → U
  BOOL : U
  NAT  : U

⟦_⟧ : U → Set
⟦ σ ⇒ τ ⟧ = ⟦ σ ⟧ → ⟦ τ ⟧
⟦ BOOL ⟧   = Bool
⟦ NAT ⟧    = Nat
```

Sorts

```
data Expr : U → Set where
  Var  : Nat → Expr σ
  App  : Expr (σ ⇒ τ) → Expr σ → Expr τ
  Lam  : Expr τ → Expr (σ ⇒ τ)
  ...
```

- helps, e.g. can only apply functions to matching arguments
- but variables are still not safe!

Context

- always consider *context*, i.e. which variables are in scope

```
Ctx = List U

data Ref (σ : U) : Ctx → Set where
  Top  : Ref σ (σ :: Γ)
  Pop  : Ref σ Γ → Ref σ (τ :: Γ)
```

- a reference is both:
  - an index (unary numbers)
  - proof that the index refers to a suitable variable in scope

Intrinsically Typed de Bruijn Representation

```
data Expr : U → Ctx → Set where
  Var  : Ref σ Γ → Expr σ Γ
  App  : Expr (σ ⇒ τ) Γ → Expr σ Γ → Expr τ Γ
  Lam  : Expr τ (σ :: Γ) → Expr (σ ⇒ τ) Γ
  Let  : Expr σ Γ → Expr τ (σ :: Γ) → Expr τ Γ
  Val  : ⟦ σ ⟧ → Expr σ Γ
  Plus : Expr NAT Γ → Expr NAT Γ → Expr NAT Γ
```

- *intrinsically* typed
- well-typed and well-scoped *by construction*!

Intrinsically Typed de Bruijn Representation

- evaluation requires an *environment*
    - a value for each variable in the context

```
data Env : List I → Set where
  Nil   : Env []
  Cons  : ⟦ σ ⟧ → Env Γ → Env (σ :: Γ)
```

- lookup and evaluation are total

```
lookup : Ref σ Γ → Env Γ → ⟦ σ ⟧
```

```
eval : Expr σ Γ → Env Γ → ⟦ σ ⟧
```

Variable Liveness

- we want to talk about the *live* context (result of LVA)
- conceptually: for each variable in scope, is it live or dead?
- we use *thinnings*

Thinnings

```
data _⊑_ : List I → List I → Set where
  o' : Δ ⊑ Γ →       Δ  ⊑ (τ :: Γ)  -- drop
  os : Δ ⊑ Γ → (τ :: Δ) ⊑ (τ :: Γ)  -- keep
  oz : [] ⊑ []                       -- done

os (o' (os oz)) : [ a , c ] ⊑ [ a , b , c ]
```

- can be seen as “bitvector”
- or as order-preserving embedding from source into target

Thinnings, Categorically

`_⨾_` : $\Gamma_1 \sqsubseteq \Gamma_2 \rightarrow \Gamma_2 \sqsubseteq \Gamma_3 \rightarrow \Gamma_1 \sqsubseteq \Gamma_3$

```
a ------ a    a ------ a     a ------ a
            ⨾        - b  =         - b
       - c    c ------ c            - c
```

- composition is associative
- composition has an identity `oi` : $\Gamma \sqsubseteq \Gamma$

Dead Binding Elimination (direct approach)

- first, we attempt DBE in a single pass
- we want to return result in its live context Δ
  - not known upfront, but should embed into original context Γ
- precisely, we want to return
  - expression `e : Expr` σ Δ
  - thinning θ : $\Delta \sqsubseteq \Gamma$
- wrapped into a datatype
  - `e` $\uparrow$ θ : `Expr` $\sigma \Uparrow \Gamma$

```
dbe : Expr σ Γ → Expr σ ⇑ Γ
```

Dead Binding Elimination (direct approach)

- most of the expression structure stays unchanged
- generally:
    - transform all subexpressions, find out their live context
    - find combined live context (and thinnings)
    - rename subexpressions into that

```
rename-Ref  : Δ ⊑ Γ → Ref σ Δ  → Ref σ Γ
rename-Expr : Δ ⊑ Γ → Expr σ Δ → Expr σ Γ
```

Dead Binding Elimination (direct approach)

```
dbe (Var x) =
  Var Top ↑ o-Ref x
```

- variables have exactly one live variable [σ]
- thinnings from singleton context are isomorphic to references

```
o-Ref : Ref σ Γ → [ σ ] ⊑ Γ
```

Dead Binding Elimination (direct approach)

```
dbe (Let e₁ e₂) with dbe e₁ | dbe e₂
... | e₁' ↑ θ₁  | e₂' ↑ o' θ₂ =
  e₂' ↑ θ₂
... | e₁' ↑ θ₁  | e₂' ↑ os θ₂ =
  Let (rename-Expr (un-∪₁ θ₁ θ₂) e₁')
      (rename-Expr (os (un-∪₂ θ₁ θ₂)) e₂')
  ↑ (θ₁ ∪ θ₂)
```

- most interesting case
- look at live context of transformed subexpressions:
  - if `o'`, eliminate dead binding!
  - if `os`, we cannot remove it (Agda won't let us)
- this corresponds to *strongly* live variable analysis

Dead Binding Elimination (direct approach)

Correctness

- intrinsically typed syntax enforces some invariants
- correctness proof is stronger, but what does "correctness" mean?

Dead Binding Elimination (direct approach)

Correctness

- intrinsically typed syntax enforces some invariants
- correctness proof is stronger, but what does "correctness" mean?

- preservation of semantics (based on `eval`)
  - conceptually: `eval` $\circ$ `dbe` $\equiv$ `eval`

Dead Binding Elimination (direct approach)

Correctness

- intrinsically typed syntax enforces some invariants
- correctness proof is stronger, but what does “correctness” mean?

- preservation of semantics (based on `eval`)
  - conceptually: `eval ∘ dbe ≡ eval`

```
dbe-correct :
  (e : Expr σ Γ) (env : Env Γ) →
  let e' ↑ θ = dbe e
  in eval e' (project-Env θ env) ≡ eval e env
```

Dead Binding Elimination (direct approach)

```
dbe-correct :
  (e : Expr σ Γ) (env : Env Γ) →
  let e' ↑ θ = dbe e
  in eval e' (project-Env θ env) ≡ eval e env
```

- proof by structural induction
- requires laws about evaluation, renaming, environment projection, operations on thinnings, …

Dead Binding Elimination (direct approach)

```
dbe-correct (Lam e₁) env =
  let e₁' ↑ θ₁ = dbe e₁
  in extensionality _ _ λ v →
      eval (rename-Expr (un-pop θ₁) e₁') (project-Env (os (pop θ₁)) (Cons v en
    ≡⟨ ... ⟩
      eval e₁' (project-Env (un-pop θ₁) (project-Env (os (pop θ₁)) (Cons v env
    ≡⟨ ... ⟩
      eval e₁' (project-Env (un-pop θ₁ ⨾ os (pop θ₁)) (Cons v env))
    ≡⟨ ... ⟩
      eval e₁' (project-Env θ₁ (Cons v env))
    ≡⟨ dbe-correct e₁ (Cons v env) ⟩
      eval e₁ (Cons v env)
    ∎
```

- binary constructors similarly with (for each subexpression)
- for Let, distinguish cases again

Dead Binding Elimination (direct approach)

```
dbe (Let e₁ e₂) with dbe e₁ | dbe e₂
... | e₁' ↑ θ₁  | e₂' ↑ o' θ₂ =
  e₂' ↑ θ₂
... | e₁' ↑ θ₁  | e₂' ↑ os θ₂ =
  Let (rename-Expr (un-∪₁ θ₁ θ₂) e₁')
      (rename-Expr (os (un-∪₂ θ₁ θ₂)) e₂')
  ↑ (θ₁ ∪ θ₂)
```

- remember: repeated renaming for each binary constructor
- inefficient! (quadratic complexity)
- hard to avoid
  - in which context do we need the transformed subexpressions?
  - we can query it upfront, but that's also quadratic

Dead Binding Elimination (annotated)

- repeated renaming can be avoided by an analysis pass
  - so we know upfront which which context to use
- common in compilers
- we define annotated syntax tree
  - again using thinnings, constructed as before
  - for $\{\theta : \Delta \sqsubseteq \Gamma\}$, we have `LiveExpr` σ θ

Dead Binding Elimination (annotated)

```
data LiveExpr {Γ : Ctx} : {Δ : Ctx} → U → Δ ⊑ Γ → Set
  Var :
    (x : Ref σ Γ) →
    LiveExpr σ (o-Ref x)
  App :
    {θ₁ : Δ₁ ⊑ Γ} {θ₂ : Δ₂ ⊑ Γ} →
    LiveExpr (σ ⇒ τ) θ₁ →
    LiveExpr σ θ₂ →
    LiveExpr τ (θ₁ ∪ θ₂)
  Lam :
    {θ : Δ ⊑ (σ :: Γ)} →
    LiveExpr τ θ →
    LiveExpr (σ ⇒ τ) (pop θ)
  ...
```

Dead Binding Elimination (annotated)

```
Let :
  {$\theta_1$ : $\Delta_1 \sqsubseteq \Gamma$} {$\theta_2$ : $\Delta_2 \sqsubseteq$ ($\sigma$ :: $\Gamma$)} →
  LiveExpr $\sigma$ $\theta_1$ → LiveExpr $\tau$ $\theta_2$ →
  LiveExpr $\tau$ (combine $\theta_1$ $\theta_2$)
```

- in direct approach, handled in two cases
- for strong analysis, same:

  ```
  combine $\theta_1$ (o' $\theta_2$) = $\theta_2$
  combine $\theta_1$ (os $\theta_2$) = $\theta_1 \cup \theta_2$
  ```

  (only consider declaration if binding is live!)

Dead Binding Elimination (annotated)

- now, construct an annotated expression

```
analyse :
  Expr σ Γ →
  Σ[ Δ ∈ Ctx ]
    Σ[ θ ∈ (Δ ⊑ Γ) ]
      LiveExpr σ θ
```

- annotations can also be forgotten again

```
forget : {θ : Δ ⊑ Γ} → LiveExpr σ θ → Expr σ Γ
```

- `forget ∘ analyse ≡ id`

Dead Binding Elimination (annotated)

- implementation does not surprise

```
analyse (Var {$\sigma$} x) =
  [ $\sigma$ ] , o-Ref x , Var x
analyse (App $e_1$ $e_2$) =
  let $\Delta_1$ , $\theta_1$ , $le_1$ = analyse $e_1$
      $\Delta_2$ , $\theta_2$ , $le_2$ = analyse $e_2$
  In $\cup$-domain $\theta_1$ $\theta_2$ , ($\theta_1 \cup \theta_2$) , App $le_1$ $le_2$
...
```

Dead Binding Elimination (annotated)

- after analysis, do transformation
- caller can choose the context (but at least live context)

```
transform : {θ : Δ ⊑ Γ} →
  LiveExpr σ θ → Δ ⊑ Γ' → Expr σ Γ'
```

- `dbe` ≡ `transform ∘ analyse`
- together, same type signature as direct approach

Dead Binding Elimination ()

- for Let, again split on thinning (annotation)
- no renaming anymore, directly choose desired context

```
...
transform (Let {θ1 = θ1} {θ2 = o' θ2} e1 e2) θ' =
  transform e2 (un-∪2 θ1 θ2  ⨾ θ')
transform (Let {θ1 = θ1} {θ2 = os θ2} e1 e2) θ' =
  Let (transform e1 (un-∪1 θ1 θ2 ⨾ θ'))
      (transform e2 (os (un-∪2 θ1 θ2 ⨾ θ')))
...
```

Dead Binding Elimination (annotated)

Correctness

- specification is the same as for direct approach
- but this time, we start proving another thing:

```
eval ∘ transform ≡ eval ∘ forget
  -- precompose analyse on both sides
eval ∘ transform ∘ analyse ≡ eval ∘ forget ∘ analyse
  -- apply definition of dbe, law about analyse
eval ∘ dbe ≡ eval
```

- less shuffling to be done for each constructor

Intrinsically Typed de Bruijn Representation

Discussion

- analysis requires an extra pass, but pays off
- currently, transformations get rid of annotations
  - maintaining them would require more effort
- `LiveExpr` is indexed by two contexts, which seems redundant

Intrinsically Typed Co-de-Bruijn Representation

Intrinsically Typed Co-de-Bruijn Representation

- “dual” to de Bruijn indices, due to Conor McBride:
    - de Bruijn indices pick from the context “as late as possible”
    - co-de-Bruijn gets rid of bindings “as early as possible”
        - using thinnings
- our intuition:
    - expressions indexed by their (weakly) live context

Intrinsically Typed Co-de-Bruijn Representation

- complex bookkeeping
  - each subexpression has its own context, connected by thinnings
  - constructing expressions basically performs LVA
- building blocks with smart constructors hide complexity

Dead Binding Elimination (co-de-Bruijn)

- co-de-Bruijn: all variables in the context must occur
- but let-bindings can still be dead
    - easy to identify now
    - remove them!

Dead Binding Elimination (co-de-Bruijn)

- co-de-Bruijn: all variables in the context must occur
- but let-bindings can still be dead
  - easy to identify now
  - remove them!

- this might make some (previously weakly live) bindings dead
  - context gets smaller

```
dbe : Expr τ Γ → Expr τ ⇑ Γ
```

Dead Binding Elimination (co-de-Bruijn)

```
dbe (Let (pair$_R$ (e$_1$ $\uparrow \phi_1$) ((o' oz \\ e$_2$) $\uparrow \phi_2$) c)) =
  thin$\Uparrow$ $\phi_2$ (dbe e$_2$)
dbe (Let (pair$_R$ (e$_1$ $\uparrow \phi_1$) ((os oz \\ e$_2$) $\uparrow \phi_2$) c)) =
  ...
```

- option 1: check liveness in input
- binding might still become dead in dbe e$_2$
- correspondes to *weakly* live variable analysis

Dead Binding Elimination (co-de-Bruijn)

```
Let? : (Expr σ ×ᵣ ([ σ ] ⊢ Expr τ)) Γ → Expr τ ⇑ Γ
Let?   (pairᵣ _ ((o' oz \\ e₂) ↑ θ₂) _) = e₂ ↑ θ₂
Let? p@(pairᵣ _ ((os oz \\ _)  ↑ _)  _) = Let p ↑ oi

dbe (Let (pairᵣ (e₁ ↑ ϕ₁) ((_\\_ {Γ'} ψ e₂) ↑ ϕ₂) c)) =
  bind⇑ Let?
    (  thin⇑ ϕ₁ (dbe e₁)
    ,ᵣ thin⇑ ϕ₂ (map⇑ (map⊢ ψ) (Γ' \\ᵣ dbe e₂))
    )
```

- option 2: check liveness after recursive call
- correspondes to *strongly* live variable analysis

Dead Binding Elimination (co-de-Bruijn)

Correctness

- correctness proof allows larger environment than needed
  - gives flexibility for inductive step
- complex:
  - requires extensive massaging of thinnings
  - laws about `project-Env` with `_⨾_` and `oi`
  - laws about thinnings created by `_,`$_R$`_`
  - `(θ ⨾ θ') ++⊑ (ϕ ⨾ ϕ') ≡ (θ ++⊑ ϕ) ⨾ (θ' ++⊑ ϕ')`

Intrinsically Typed Co-de-Bruijn Representation

Discussion

- co-de-Bruijn representation keeps benefits of `LiveExpr`
  - liveness information available by design
- some parts get simpler (just a single context)
  - building blocks (e.g. relevant pair) allow code reuse
- some parts get more complicated (mainly proofs)
  - thinnings in result require reasoning about them a lot
  - operations on thinnings get quite complex

Syntax-generic Co-de-Bruijn Representation

Syntax-generic Programming

- based on work by Allais et al.
  - *A type- and scope-safe universe of syntaxes with binding: their semantics and proofs*
- main idea:
  - define a datatype of syntax descriptions `Desc`
  - each `(d : Desc I)` describes a language of terms `Tm d` σ Γ
  - implement operations *once*, generically over descriptions
  - describe your language using `Desc`, get operations for free

Syntax-generic Programming

- description of our language (looks cryptic)

```
data Tag : Set where
  `App `Lam `Let : U → U → Tag
  `Val  : U → Tag
  `Plus : Tag

Lang : Desc U
Lang = `σ Tag λ where
  (`App σ τ) → `X [] (σ ⇒ τ) (`X [] σ (`■ τ))
  (`Lam σ τ) → `X [ σ ] τ (`■ (σ ⇒ τ))
  (`Let σ τ) → `X [] σ (`X [ σ ] τ (`■ τ))
  (`Val τ)    → `σ Core.⟦ τ ⟧ λ _ → `■ τ
  `Plus       → `X [] NAT (`X [] NAT (`■ NAT))
```

Syntax-generic Co-de-Bruijn Representation

- we interpret descriptions into co-de-Bruijn terms
  - using building blocks
- we convert between de Bruijn and co-de-Bruijn
  - completely generically!
- we do DBE for all languages with let-bindings

```
dbe :
  Tm (d `+ `Let) τ Γ →
  Tm (d `+ `Let) τ ⇑ Γ
```

Generic Co-de-Bruijn Representation

Discussion

- generic code is more reusable
- in some sense nice to write
    - fewer cases to handle (abstraction)
- but also more complex

Other Transformations

Let-sinking

- move let-binding as far inwards as possible without
  - duplicating it
  - moving it into a λ-abstraction

Let-sinking

- pretty similar to DBE
  - also requires liveness information to find location
  - can be done directly, with repeated liveness querying
  - annotations make it more efficient
- but it gets more complex
  - instead of just removing bindings, they get reordered
  - also reorders the context, but thinnings are *order-preserving*
  - requires another mechanism to talk about that
- to keep it manageable, we focus on one binding at a time

Let-sinking

- also requires renaming, partitioning context into 4 parts

```
rename-top-Expr :
  (Γ' : Ctx) →
  Expr τ (Γ' ++ Γ₁ ++ σ ∷ Γ₂) →
  Expr τ (Γ' ++ σ ∷ Γ₁ ++ Γ₂)
```

- this gets cumbersome
- especially for co-de-Bruijn:
  - need to partition and re-assemble thinnings

Let-sinking

Discussion

- implemented for de Bruijn (incl. annotated) and co-de-Bruijn
    - exact phrasing of signatures has a big impact
- maintaining the co-de-Bruijn structure is especially cumbersome
- progress with co-de-Bruijn proof, but messy and unfinished

Discussion

Observations

- semantics: total evaluator makes it relatively easy
  - what about recursive bindings or effects?
- reordering context not a good fit for thinnings
  - use a more general notion of embedding?
    - Allais et al. use (`∀` σ `→ Ref` σ Δ `→ Ref` σ Γ)
    - opaque, harder to reason about

Further Work

- unfinished proofs for let-sinking
- generic let-sinking
    - which constructs not to sink into?
- correctness of generic transformations
    - using which semantics?

Further Work

- more language constructs
    - recursive bindings
    - non-strict bindings
    - branching
    - ...
- more transformations
    - let-floating (e.g. out of λ)
    - common subexpression elimination
        - co-de-Bruijn is useful for that, not indexed by variables in scope
    - ...

`https://github.com/mheinzel/correct-optimisations`

extended slides

thesis

implementation