Analysis and Transformation of Intrinsically Typed Syntax

Master's Thesis

Matthias Heinzel

Utrecht University

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

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

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
    - e.g. GHC uses *the rapier* based on Barendregt convention
  - mistakes still happen
    - e.g. *the foil* created to "make it harder to poke your eye out"

De Bruijn Representation

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

$\textbf{let } x = 42 \textbf{ in}$	$\textbf{let } 42 \textbf{ in}$
$\quad \textbf{let } y = 99 \textbf{ in}$	$\quad \textbf{let } 99 \textbf{ in}$
$\quad\quad x$	$\quad\quad \langle 1 \rangle$

- 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
  Let  : Expr → Expr → Expr
  Num  : Nat → Expr
  Bln  : Bool → Expr
  Plus : Expr → 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)
  Let  : Expr σ → Expr τ → Expr τ
  Val  : ⟦ σ ⟧ → Expr σ
  Plus : Expr NAT → Expr NAT → Expr NAT
```

- 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)

eval : Expr σ Γ → Env Γ → ⟦ σ ⟧
```

Intrinsically Typed de Bruijn Representation

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

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

lookup : Ref σ Γ → Env Γ → ⟦ σ ⟧
lookup Top      (Cons v env)   = v
lookup (Pop i)  (Cons v env)   = lookup i env
```

- lookup is total

Intrinsically Typed de Bruijn Representation

```
eval : Expr σ Γ → Env Γ → ⟦ σ ⟧
eval (Var x)       env = lookup x env
eval (App e₁ e₂)  env = eval e₁ env (eval e₂ env)
eval (Lam e₁)      env = λ v → eval e₁ (Cons v env)
eval (Let e₁ e₂)  env = eval e₂ (Cons (eval e₁ env) env)
eval (Val v)       env = v
eval (Plus e₁ e₂) env = eval e₁ env + eval e₂ env
```

- evaluation is total

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
```

```
a ------ a      os
       - b      o'
c ------ c      os
                oz
```

```
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

```
_⨾_ : Γ₁ ⊑ Γ₂ → Γ₂ ⊑ Γ₃ → Γ₁ ⊑ Γ₃

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` $\uparrow \theta$ `: Expr` $\sigma \Uparrow \Gamma$

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 σ $\Uparrow$ Γ

```
record _⇑_ (T : List I → Set) (Γ : List I) : Set where
  constructor _↑_
  field
    {support} : List I
    thing : T support
    thinning : support ⊑ 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 (Val v) =
  Val v ↑ oe
```

- in values, no variable is live
- empty thinning

oe : [] $\sqsubseteq$ 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 (App e₁ e₂) =
  let e₁' ↑ θ₁ = dbe e₁   -- θ₁ : Δ₁ ⊑ Γ
      e₂' ↑ θ₂ = dbe e₂   -- θ₂ : Δ₂ ⊑ Γ
  in App (rename-Expr (un-∪₁ θ₁ θ₂) e₁')
         (rename-Expr (un-∪₂ θ₁ θ₂) e₂')
     ↑ (θ₁ ∪ θ₂)
```

- find minimal live context (if θ_1 or θ_2 keeps, keep!)
- rename subexpressions into that context

```
_∪_    : ∀ θ₁ θ₂ → ∪-domain θ₁ θ₂ ⊑ Γ
un-∪₁ : ∀ θ₁ θ₂ → Δ₁ ⊑ ∪-domain θ₁ θ₂
un-∪₂ : ∀ θ₁ θ₂ → Δ₂ ⊑ ∪-domain θ₁ θ₂
```

Dead Binding Elimination (direct approach)

```
dbe (Lam e₁) =
  let e₁' ↑ θ = dbe e₁  -- θ : Δ ⊑ (σ :: Γ)
  in Lam (rename-Expr (un-pop θ) e₁') ↑ pop θ
```

- pop off the top element
  - corresponding to variable bound by `Lam`

```
pop    : ∀ θ → pop-domain θ ⊑ Γ
un-pop : ∀ θ → Δ ⊑ (σ :: pop-domain θ)
```

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`
- values include functions, so we need extensional equality

```
postulate
  extensionality :
    {S : Set} {T : S → Set} (f g : (x : S) → T x) →
    (∀ x → f x ≡ g x) → f ≡ g
```

Dead Binding Elimination (direct approach)

```
project-Env : Δ ⊑ Γ → Env Γ → Env Δ

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 (App e₁ e₂) =
  let e₁' ↑ θ₁ = dbe e₁   -- θ₁ : Δ₁ ⊑ Γ
      e₂' ↑ θ₂ = dbe e₂   -- θ₂ : Δ₂ ⊑ Γ
  in App (rename-Expr (un-∪₁ θ₁ θ₂) e₁')
         (rename-Expr (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` $\sigma \ \theta$

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)
  Let : ...
  Val : ...
```

Dead Binding Elimination (annotated)

```
Let :
  {θ₁ : Δ₁ ⊑ Γ} {θ₂ : Δ₂ ⊑ (σ :: Γ)} →
  LiveExpr σ θ₁ → LiveExpr τ θ₂ →
  LiveExpr τ (combine θ₁ θ₂)
```

- in direct approach, handled in two cases
- for analysis, we have a choice:
  1. treat `Let` as an immediately `Applied` `Lam`

     ```
     combine θ₁ θ₂ = θ₁ ∪ pop θ₂
     ```

  2. custom operation for *strongly* live variable analysis

     ```
     combine θ₁ (o' θ₂) = θ₂
     combine θ₁ (os θ₂) = θ₁ ∪ θ₂
     ```

     (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 {σ} x) =
  [ σ ] , o-Ref x , Var x
analyse (App e₁ e₂) =
  let Δ₁ , θ₁ , le₁ = analyse e₁
      Δ₂ , θ₂ , le₂ = analyse e₂
  In ∪-domain θ₁ θ₂ , (θ₁ ∪ θ₂) , App le₁ le₂
...
```

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

```
dbe : Expr σ Γ → Expr σ ⇑ Γ
dbe e =
  let Δ , θ , le = analyse e
  in transform le oi ↑ θ
```

Dead Binding Elimination (annotated)

- no renaming anymore, directly choose desired context

```
transform (Var x) θ' = Var (ref-o θ')
transform (App {θ₁ = θ₁} {θ₂ = θ₂} e₁ e₂) θ' =
  App (transform e₁ (un-∪₁ θ₁ θ₂ ⨾ θ'))
      (transform e₂ (un-∪₂ θ₁ θ₂ ⨾ θ'))
transform (Lam {θ = θ} e₁) θ' =
  Lam (transform e₁ (un-pop θ ⨾ os θ'))
...
```

Dead Binding Elimination (annotated)

- for Let, again split on thinning (annotation)

```
...
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
- even harder for humans to reason about
  - constructing expressions basically performs LVA

Intrinsically Typed Co-de-Bruijn Representation

- how to deal with multiple subexpressions?
- basically, as with `LiveExpr` we need:
  - a suitable overall context Γ (like `_∪_`)
  - for each subexpression, a thinning into Γ
- building block: *relevant pair*

Intrinsically Typed Co-de-Bruijn Representation

```
record _×R_ (S T : List I → Set) (Γ : List I) : Set wher
  constructor pairR
  field
    outl  : S ⇑ Γ     -- S Δ1 and Δ1 ⊑ Γ
    outr  : T ⇑ Γ     -- T Δ2 and Δ2 ⊑ Γ
    cover : Cover (thinning outl) (thinning outr)
```

- usage: (Expr ($\sigma \Rightarrow \tau$) $\times_R$ Expr σ) Γ

Intrinsically Typed Co-de-Bruijn Representation

```
record _×R_ (S T : List I → Set) (Γ : List I) : Set wher
  constructor pairR
  field
    outl  : S ⇑ Γ     -- S Δ₁ and Δ₁ ⊑ Γ
    outr  : T ⇑ Γ     -- T Δ₂ and Δ₂ ⊑ Γ
    cover : Cover (thinning outl) (thinning outr)
```

- usage: (Expr (σ ⇒ τ) $\times_R$ Expr σ) Γ
- what is a cover?
  - we just have some overall context Γ
  - cover ensures that Γ is *relevant*, as small as possible

Intrinsically Typed Co-de-Bruijn Representation

- each element of Γ needs to be relevant
- i.e. at least one thinning keeps it

```
data Cover : Γ₁ ⊑ Γ → Γ₂ ⊑ Γ → Set where
  c's : Cover θ₁ θ₂ → Cover (o' θ₁) (os θ₂)
  cs' : Cover θ₁ θ₂ → Cover (os θ₁) (o' θ₂)
  css : Cover θ₁ θ₂ → Cover (os θ₁) (os θ₂)
  czz : Cover oz oz
```

Intrinsically Typed Co-de-Bruijn Representation

- how to deal with bindings?
- here, we allow multiple simultaneous bindings Γ '
  - requires talking about context concatenation (replaces `pop`)

Intrinsically Typed Co-de-Bruijn Representation

- how to deal with bindings?
- here, we allow multiple simultaneous bindings Γ'
  - requires talking about context concatenation (replaces pop)

- new construct (Γ' $\vdash$ T) Γ, consists of two things:

```
ψ : Δ' ⊑ Γ'          -- which new variables are used?
t : T (Δ' ++ Γ)      -- used variables added to context
```

Intrinsically Typed Co-de-Bruijn Representation

```
data Expr : U → Ctx → Set where
  Var :
    Expr σ [ σ ]
  App :
    (Expr (σ ⇒ τ) ×R Expr σ) Γ →
    Expr τ Γ
  Lam :
    ([ σ ] ⊢ Expr τ) Γ →
    Expr (σ ⇒ τ) Γ
  Let :
    (Expr σ ×R ([ σ ] ⊢ Expr τ)) Γ →
    Expr τ Γ
  ...
```

Conversion From Co-de-Bruijn Syntax

- take all those thinnings at the nodes
- only use them at the latest moment, variables

```
relax : Γ' ⊑ Γ → Expr σ Γ' → DeBruijn.Expr σ Γ
```

- keep composing the thinning
  - how do we deal with bindings (ψ `\\ e`)?

Concatenation of Thinnings

- thinnings have monoidal structure

```
_++⊑_ :
  Δ₁ ⊑ Γ₁ → Δ₂ ⊑ Γ₂ →
  (Δ₁ ++ Δ₂) ⊑ (Γ₁ ++ Γ₂)
```

- extends to covers

```
_++C_ :
  Cover θ₁ θ₂ → Cover ϕ₁ ϕ₂ →
  Cover (θ₁ ++⊑ ϕ₁) (θ₂ ++⊑ ϕ₂)
```

Conversion From Co-de-Bruijn Syntax

```
relax : Γ' ⊑ Γ → Expr σ Γ' → DeBruijn.Expr σ Γ
relax θ Var =
  -- eventually turn thinning into Ref
  DeBruijn.Var (ref-o θ)
relax θ (App (pairR (e₁ ↑ ϕ₁) (e₂ ↑ ϕ₂) cover)) =
  DeBruijn.App (relax (ϕ₁ ⨾ θ) e₁) (relax (ϕ₂ ⨾ θ) e₂)
relax θ (Lam (ψ \\ e)) =
  DeBruijn.Lam (relax (ψ ++⊑ θ) e)
relax θ (Let (pairR (e₁ ↑ θ₁) ((ψ \\ e₂) ↑ θ₂) c)) =
  -- combination of product and binding
  DeBruijn.Let (relax (θ₁ ⨾ θ) e₁) (relax (ψ ++⊑ (θ₂ ⨾ θ)
...
```

Conversion To Co-de-Bruijn Syntax

- other direction is harder
- we need to find all these thinnings
- resulting live context not known upfront, use `_⇑_`

```
tighten : DeBruijn.Expr σ Γ → Expr σ ⇑ Γ
```

Conversion To Co-de-Bruijn Syntax

```
_,R_ : S ⇑ Γ → T ⇑ Γ → (S ×R T) ⇑ Γ
```

- implementation similar to `_∪_`, but also constructs cover

```
_\\R_ : (Γ' : List I) → T ⇑ (Γ' ++ Γ) → (Γ' ⊢ T) ⇑ Γ
```

- relies on the fact that thinnings can be split

Conversion To Co-de-Bruijn Syntax

```
tighten : DeBruijn.Expr σ Γ → Expr σ ⇑ Γ
tighten (DeBruijn.Var x) =
  Var ↑ o-Ref x
tighten (DeBruijn.App e₁ e₂) =
  map⇑ App (tighten e₁ ,R tighten e₂)
tighten (DeBruijn.Lam e) =
  map⇑ Lam ([ _ ] \\R tighten e)
tighten (DeBruijn.Let e₁ e₂) =
  map⇑ Let (tighten e₁ ,R ([ _ ] \\R tighten e₂))
...

-- map⇑ f (t ↑ θ) = f t ↑ θ
```

Conversion To Co-de-Bruijn Syntax

- also prove that conversion agrees with semantics
  - DeBruijn.eval ∘ relax ≡ eval
  - eval ∘ tighten ≡ DeBruijn.eval

Dead Binding Elimination (co-de-Bruijn)

- co-de-Bruijn: all variables in the context must occur
- but let-bindings can still be dead (`o' oz \\` e_2)
  - 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 (`o' oz \\` e_2)
    - easy to identify now
    - remove them!

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

`dbe :` $\texttt{Expr}\ \tau\ \Gamma \rightarrow \texttt{Expr}\ \tau \Uparrow \Gamma$

Dead Binding Elimination (co-de-Bruijn)

```
dbe : Expr τ Γ → Expr τ ⇑ Γ
dbe Var =
  Var ↑ oi
dbe (App (pairR (e₁ ↑ ϕ₁) (e₂ ↑ ϕ₂) c)) =
  map⇑ App (thin⇑ ϕ₁ (dbe e₁) ,R thin⇑ ϕ₂ (dbe e₂))
dbe (Lam (_\\_ {Γ'} ψ e)) =
  map⇑ (Lam ∘ map⊢ ψ) (Γ' \\R dbe e)
...
```

- propagate livenss information using smart constructors

Dead Binding Elimination (co-de-Bruijn)

```
dbe (Let (pairR (e1 ↑ φ1) ((o' oz \\ e2) ↑ φ2) c)) =
  thin⇑ φ2 (dbe e2)
dbe (Let (pairR (e1 ↑ φ1) ((os oz \\ e2) ↑ φ2) c)) =
  map⇑ Let
    (  thin⇑ φ1 (dbe e1)
    ,R thin⇑ φ2 ([ _ ] \\R dbe e2)
    )
```

- 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')`
- for strong version:
  - `Let? p` semantically equivalent to `Let p`

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*
- problem:
  - any time you define a language, you need common operations (renaming, substitution, ...) and laws about them
  - for these, languages need variables and bindings, the rest is noise
  - `Ctrl+C`, `Ctrl+V`?

Syntax-generic Programming

- 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

    `foo : (d : Desc I) → Tm d` σ Γ `→ ...`

  - describe your language using `Desc`, get operations for free

Syntax-generic Programming

- 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

    `foo : (d : Desc I) → Tm d` σ Γ `→ ...`
  - describe your language using `Desc`, get operations for free
- authors created Agda package `generic-syntax`
  - we build on top of that
  - made it compile with recent Agda versions (had to remove sized types that were used to show termination)

Syntax-generic Programming

```
data Desc (I : Set) : Set₁ where
  `σ : (A : Set) → (A → Desc I)  → Desc I
  `X : List I → I → Desc I        → Desc I
  `■ : I                          → Desc I
```

- let's describe our language!

Syntax-generic Programming

```
`σ : (A : Set) → (A → Desc I) → Desc I
```

- `` `σ `` is for storing data, e.g. which constructor it is
  - variables are assumed, no need to describe them

```
data Tag : Set where
  `App  : U → U → Tag
  ...

Lang : Desc U
Lang = `σ Tag λ where
  (`App σ τ) → ...
  ...
```

Syntax-generic Programming

- `` `X `` is for recursion (subexpressions)
- also allows us to build product types

```
`X :
  List I →  -- new variables bound in subexpression
  I →       -- sort of subexpression
  Desc I →  -- (continue)
  Desc I
```

- `` `■ `` terminates description

```
`■ :
  I →       -- sort
  Desc I
```

```
  (`App σ τ) → `X [] (σ ⇒ τ) (`X [] σ (`■ τ))
  (`Lam σ τ) → `X [ σ ] τ (`■ (σ ⇒ τ))
```

Syntax-generic Programming

```
data Tag : Set where
  `App  : U → U → Tag
  `Lam  : U → U → Tag
  `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

- Allais et al. interpret `Desc` into de-Bruijn terms
- we interpret descriptions into co-de-Bruijn terms
  - using building blocks
  - McBride had something similar, but for different `Desc` type

```
_─Scoped : Set → Set₁
I ─Scoped = I → List I → Set
```

- something indexed by sort and context
  - e.g. `Expr : U ─Scoped`

Syntax-generic Co-de-Bruijn Representation

```
data Tm (d : Desc I) : I ─Scoped where
  `var : Tm d i [ i ]
  `con : ⟦ d ⟧ (Scope (Tm d)) i Γ → Tm d i Γ
```

- terms always have variables
- for the rest, interpret the description

Syntax-generic Co-de-Bruijn Representation

```
Scope : I ─Scoped → List I → I ─Scoped
Scope T    []      i = T i
Scope T Δ@(_ :: _) i = Δ ⊢ T i
```

- Scope roughly corresponds to bindings
- empty scopes are very common, avoid trivial `[] ⊢_`

Syntax-generic Co-de-Bruijn Representation

```
⟦_⟧ : Desc I → (List I → I ─Scoped) → I ─Scoped
⟦ `σ A d    ⟧ X i Γ = Σ[ a ∈ A ] (⟦ d a ⟧ X i Γ)
⟦ `X Δ j d  ⟧ X i = X Δ j ×R ⟦ d ⟧ X i
⟦ `■ j      ⟧ X i Γ = i ≡ j × Γ ≡ []
```

- context only contains live variables
  - enforced by relevant pair and constraints in `■

Syntax-generic Co-de-Bruijn Representation

- working generically, this works well
- but once description is concrete, there are unexpected indirections
- e.g. "unary product" $\llbracket$ `` `X `` Δ σ (`` ` ``■ τ) $\rrbracket$
  - trivial relevant pair (right side has empty context)

```
×R-trivial t =
  pairR (t ↑ oi) ((refl , refl) ↑ oe) cover-oi-oe
```

- in the other direction, we first have to "discover" that these extra thinnings are `oi` and `oe`

Generic Conversion From Co-de-Bruijn Syntax

- we convert between de Bruijn and co-de-Bruijn
  - completely generically!
- implementation is concise (few cases to handle)

```
relax :
  (d : Desc I) → Δ ⊑ Γ →
  CoDeBruijn.Tm d τ Δ →
  DeBruijn.Tm d τ Γ

tighten :
  (d : Desc I) →
  DeBruijn.Tm d τ Γ →
  CoDeBruijn.Tm d τ ⇑ Γ
```

Dead Binding Elimination (generic co-de-Bruijn)

- DBE can be done generically as well
- we need let-bindings, the rest does not matter

Dead Binding Elimination (generic co-de-Bruijn)

- DBE can be done generically as well
- we need let-bindings, the rest does not matter

- descriptions are closed under sums:
  `Tm (d ‵+ ‵Let)` τ Γ

Dead Binding Elimination (generic co-de-Bruijn)

```
_`+_ : Desc I → Desc I → Desc I
d `+ e = `σ Bool λ isLeft →
  if isLeft then d else e

`Let : Desc I
`Let {I} = `σ (I × I) $ uncurry $ λ σ τ →
  `X [] σ (`X [ σ ] τ (`■ τ))

-- ⟦ d ⟧ or ⟦ e ⟧ in ⟦ d + e ⟧
pattern inl t = true  , t
pattern inr t = false , t
```

Dead Binding Elimination (generic co-de-Bruijn)

- define mutually recursive functions

```
dbe :
  Tm (d `+ `Let) τ Γ →
  Tm (d `+ `Let) τ ⇑ Γ
dbe-⟦·⟧ :
  ⟦ d ⟧ (Scope (Tm (d' `+ `Let))) τ Γ →
  ⟦ d ⟧ (Scope (Tm (d' `+ `Let))) τ ⇑ Γ
dbe-Scope :
  (Δ : List I) →
  Scope (Tm (d `+ `Let)) Δ τ Γ →
  Scope (Tm (d `+ `Let)) Δ τ ⇑ Γ
```

Dead Binding Elimination (generic co-de-Bruijn)

- recognise parts from the concrete implementation?

```
dbe `var = `var ↑ oi
dbe (`con (inl t)) = map⇑ (`con ∘ inl) (dbe-⟦·⟧ t)
dbe (`con (inr t)) = bind⇑ Let? (dbe-⟦·⟧ {d = `Let} t)

Let? : ⟦ `Let ⟧ (...) τ Γ → Tm (d `+ `Let) τ ⇑ Γ
Let? t@(a , pairR (t₁ ↑ θ₁) (t₂ ↑ p) _)
  with ×R-trivial⁻¹ p
... | (o' oz \\ t₂) , refl = t₂ ↑ θ₂
... | (os oz \\ t₂) , refl = `con (inr t) ↑ oi
```

Dead Binding Elimination (generic co-de-Bruijn)

```
dbe-⟦·⟧ {d = `σ A d} (a , t) =
  map⇑ (a ,_) (dbe-⟦·⟧ t)
dbe-⟦·⟧ {d = `X Δ j d} (pairR (t₁ ↑ θ₁) (t₂ ↑ θ₂) c) =
  thin⇑ θ₁ (dbe-Scope Δ t₁) ,R thin⇑ θ₂ (dbe-⟦·⟧ t₂)
dbe-⟦·⟧ {d = `■ i} t =
  t ↑ oi

dbe-Scope [] t = dbe t
dbe-Scope (_ :: _) (ψ \\ t) =
  map⇑ (map⊢ ψ) (_ \\R dbe t)
```

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

Generic Co-de-Bruijn Representation

Discussion

- no correctness proofs yet
- using which semantics?
  - 1. prove it for a specific language
  - 2. use a generic notion of `Semantics` that is sufficient

Generic Co-de-Bruijn Representation

Discussion

- no correctness proofs yet
- using which semantics?
    - 1. prove it for a specific language
    - 2. use a generic notion of `Semantics` that is sufficient
- Allais et al. define generic `Semantics`
    - abstracts over traversal (similar to recursion schemes)
- defining a similar `Semantics` for co-de-Bruijn expressions seems more difficult
    - scopes change at each node, manipulating them requires re-constructing covers
    - probably easier when operating on thinned expressions (`_⇑_`)

Other Transformations

Let-sinking

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

Let-sinking

- results 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

- top level signature remains somewhat simple
- should look similar to
  `Let` : `Expr` σ Γ → `Expr` τ $(\sigma :: \Gamma)$ → `Expr` τ Γ
- but as we go under binders, the binding doesn't stay on top

```
sink-let :
  Expr σ (Γ₁ ++ Γ₂) →
  Expr τ (Γ₁ ++ σ :: Γ₂) →
  Expr τ (Γ₁ ++ Γ₂)
```

Let-sinking

- also requires renaming, partitioning context into 4 parts

```
rename-top-Expr :
  Expr τ (Γ₁ ++ Γ₂ ++ σ ∷ Γ₃) →
  Expr τ (Γ₁ ++ σ ∷ Γ₂ ++ Γ₃)
```

- this gets cumbersome
- especially for co-de-Bruijn:
  - need to partition (into four) 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
- progress with co-de-Bruijn proof, but messy and unfinished

Let-sinking

Discussion

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

- generally similar conclusions as from DBE
- re-ordering context does not interact nicely with thinnings
- maintaining the co-de-Bruijn structure is especially cumbersome

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

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