Provingly Correct Optimisations on Intrinsically Typed Expressions

Extended Abstract

Matthias Heinzel

Utrecht University

Intrinsically typed syntax trees

Optimisations on intrinsically typed expressions

Dead binding elimination (DBE)

Correctness

What's next?

Intrinsically typed syntax trees

Intrinsically typed syntax trees

- use the Agda type system to define expressions that are correct by construction
- type- and scope-safe

Type-safety by construction

```
data U : Set where
  BOOL  : U
  NAT   : U

⟦_⟧ : U → Set
⟦ BOOL ⟧  = Bool
⟦ NAT ⟧   = Nat

data Expr : (σ : U) → Set where
  Val   : ⟦ σ ⟧ → Expr σ
  Plus  : Expr NAT → Expr NAT → Expr NAT
  ...
```

Scope-safety by construction

De-Bruijn-indices:

- keep track of bindings in scope (by position)

```
Ctx = List U
```

Scope-safety by construction

```
data Expr (Γ : Ctx) : (σ : U) → Set where
  Val   : ⟦ σ ⟧ → Expr Γ σ
  Plus  : Expr Γ NAT → Expr Γ NAT → Expr Γ NAT
```

Scope-safety by construction

```
data Expr (Γ : Ctx) : (σ : U) → Set where
  Val   : ⟦ σ ⟧ → Expr Γ σ
  Plus  : Expr Γ NAT → Expr Γ NAT → Expr Γ NAT

  Let   : Expr Γ σ → Expr (σ :: Γ) τ → Expr Γ τ
  Var   : Ref σ Γ → Expr Γ σ
```

Scope-safety by construction

```
data Expr (Γ : Ctx) : (σ : U) → Set where
  Val   : ⟦ σ ⟧ → Expr Γ σ
  Plus  : Expr Γ NAT → Expr Γ NAT → Expr Γ NAT

  Let   : Expr Γ σ → Expr (σ :: Γ) τ → Expr Γ τ
  Var   : Ref σ Γ → Expr Γ σ

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

Scope-safety by construction

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

- evaluation is total!

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

Intrinsically typed syntax trees

- well-known idea
- existing work on basic operations (evaluation, substitution, …)
- but little focus on optimisations

Optimisations on intrinsically typed expressions

Optimisations on intrinsically typed expressions

- optimisations are essential for most compilers
- many opportunities to introduce bugs

Optimisations on intrinsically typed expressions

- **Analysis** needs to identify optimisation opportunities *and* provide proof that they are safe
- **Transformation** needs to preserve type- and scope-safety
- finally, we want to prove preservation of semantics

Dead binding elimination (DBE)

Dead binding elimination (DBE)

- if bindings are not used, we want to remove them
- use live variable analysis (LVA) to annotate each expression with subset of context that is live

Sub-contexts

- a **sub-context** is a context with an order-preserving embedding (OPE)
- useful to bundle these into a single data type

```
data SubCtx : Ctx → Set where
  Empty  : SubCtx []
  Drop   : SubCtx Γ → SubCtx (τ :: Γ)
  Keep   : SubCtx Γ → SubCtx (τ :: Γ)
```

Sub-contexts

- we define some operations

```
_⊆_ : SubCtx Γ → SubCtx Γ → Set

_∪_ : SubCtx Γ → SubCtx Γ → SubCtx Γ
```

- we will only consider expressions in a context $\lfloor \Delta \rfloor$ determined by some sub-context Δ

Annotated expressions

- Δ: *defined* bindings, top-down (as Γ before)
- Δ': *used* bindings, bottom-up

```
data LiveExpr : (Δ Δ' : SubCtx Γ) (σ : U) → Set where
  Var :  (x : Ref σ ⌊ Δ ⌋) →
         LiveExpr Δ (sing Δ x) σ
  Plus : LiveExpr Δ Δ₁ NAT →
         LiveExpr Δ Δ₂ NAT →
         LiveExpr Δ (Δ₁ ∪ Δ₂) NAT
  ...
```

Live variable analysis

- we can relate ordinary and annotated expressions

```
analyse : Expr ⌊ Δ ⌋ σ → Σ[ Δ' ∈ SubCtx Γ ]
                          LiveExpr Δ Δ' σ

forget : LiveExpr Δ Δ' σ → Expr ⌊ Δ ⌋ σ
```

- reasonable requirement: `forget ∘ analyse ≡ id`

Dead binding elimination

```
dbe : LiveExpr Δ Δ' σ → Expr ⌊ Δ' ⌋ σ
```

- on a `Let`, it branches on whether the variable is live
- if not, just remove the binding

Correctness

Optimised semantics

- to split up the correctness proofs, we give semantics `evalLive` for annotated expressions
  - on a `Let`, it also branches
  - generalisation helps: accept any `Env` $\lfloor\ \Delta_u\ \rfloor$ with $\Delta'\ \subseteq\ \Delta_u$

Correctness

- the goal: `eval` $\circ$ `dbe` $\circ$ `analyse` $\equiv$ `eval`
- for the analysis, we know that: `forget` $\circ$ `analyse` $\equiv$ `id`
- so it is sufficient to do two straight-forward proofs:
  - `eval` $\circ$ `dbe` $\equiv$ `evalLive`
  - `evalLive` $\equiv$ `eval` $\circ$ `forget`
- both only require small lemmas for the `Let` case

What's next?

What's next?

- extend the language
  - lambda abstractions
  - recursive bindings
- implement more optimisations
  - *strong* live variable analysis
  - moving bindings up or down in the syntax tree
  - common subexpression elimination
- identify patterns, generalise the approach

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

I'm looking forward to your feedback!