Provingly Correct Optimizations in an Intrinsically Typed Compiler

Experimentation Project Report

Matthias Heinzel (1632256)

August 3, 2022

Contents

1 Introduction

When writing a compiler for a programming language, an important consideration is the treatment of binders and variables. A well-known technique when using dependently typed programming languages such as Agda [6] is to define an intrinsically typed syntax tree, where expressions are scope- and type-safe by construction and admit a total evaluation function [2]. This construction has featured in several papers, exploring basic operations like renaming and substitution [1] as well as compilation to different target languages [7, supplemental material].

Performing optimisations on intrinsically typed programs, on the other hand, has not received as much attention. However, optimisations play an important role in compilers and establishing their correctness is often not trivial, with ample opportunity for mistakes. In this setting, program *analysis* not only needs to identify optimisation opportunities, but provide a proof witness that the optimisation is safe, e.g. that some dead code is indeed not used. For the *transformation* of the intrinsically typed program, the programmer can then rely on the compiler to check the relevant invariants, but it can be cumbersome to make it sufficiently clear that type- and scope-safety are preserved, especially when manipulating binders and variables.

As a step towards a more general treatment of optimisations of intrinsically typed programs, we present an implementation of *dead binding elimination* for a simple language. It first annotates expressions with variable usage information and then removes bindings that turn out to be unused. We further prove that the optimisation is semantics-preserving. The Agda source code is available online at `https://git.science.uu.nl/m.h.heinzel/correct-optimisations`.

2 A Simple Expression Language

2.1 Expressions with Bindings

We define a simple, typed expression language with let-bindings, variables, primitive values (integers and Booleans), and a few binary operators. Since the optimisations we are interested in relate to variables and binders only, the choice of possible values and additional primitive operations on them is mostly arbitrary. Extending the language with further values and operators is trivial.

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

Expressions can be bound to a variable x using the **let** construction. Note that this makes the language equivalent to a restricted version of the simply typed λ-calculus, where λ-abstraction and application can only occur together as $(\lambda x.Q)P$. Encapsulating this pattern as **let** $x = P$ **in** Q simplifies parts of the analysis and avoids the need for allowing functions as values.

2.2 Representing Variables

The syntax specified above treats variables as letters, or more generally strings. To prevent complications with bindings of the same variable name shadowing each other and to make equality of terms independent of the specific names chosen (*α-equivalence*), compilers often represent variables in a different way. A popular choice are *de Bruijn indices*, where each variable is represented by a natural number, counting the number of bindings between variable occurence and its binding: 0 refers to the innermost binding, 1 to the next-innermost etc.

Still, there might be *free variables*, where the de Bruijn index is larger than the number of bindings it has access to (*in scope*). If this happens unexpectedly during evaluation, an error is raised. Similarly, the type of a bound expression might not match the expected type at the variable occurence where it is referenced. This makes the evaluation function partial; it should only be called after validating type- and scope-safety.

2.3 Intrinsically Typed Syntax Trees

When implementing a compiler in a dependently typed programming language, one does not need to accept partiality and the need for human vigilance. With *intrinsically typed syntax trees*, type- and scope-safety invariants are specified on the type level and verified by the type checker. We will demonstrate the approach in Agda and start by defining the types that terms can have.

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

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

To know if a variable occurence is valid, we must consider its *context*, the bindings that are in scope. With de Bruijn indices in an untyped setting, it would suffice to know the number of bindings in scope. In a typed settings, we also need to know the type of each binding, so we represent the context by a list of types: One for each binding in scope, from innermost to outermost.

```
Ctx = List U

variable
  Γ : Ctx
  σ τ : U
```

During evaluation, each variable in scope has a value. Together, we call these an *environment* in a given context.

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

A variable then is an index into its context, also guaranteeing that its type matches that of the binding. Since variable Ref σ Γ acts as a proof that the environment Env Γ contains an element of type σ, variable lookup is total.

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

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

Now we can define intrinsically typed expressions, where an Expr is indexed by both its type (σ : U) and context (Γ : Ctx). We can see how the context changes when introducing a new binding that is then available in the body of a Let.

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

This allows the definition of a total evaluator using an environment matching the expression's context.

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

3 Dead Binding Elimination

Note that an expression is not forced to make use of the whole context to which it has access. Specifically, a let-binding introduces a new element into the context, but it might never be used in the body. One commonly wants to identify such unused bindings so they can be removed from the program [5, Live Variable Analysis]. To that end, we annotate terms with the subset of their context that they actually use.

3.1 Subsets of a Context

A *sub-context* is not just any context. All its elements must come from the original context and we also require that their order is unchanged. This notion is known as a *thinning* or *order-preserving embedding* (OPE) [4] between the two contexts. We combine the sub-context with its OPE in a single data type Subset, which for each element of the context specifies whether to keep it or not. The uniquely determined sub-context can be extracted using $\lfloor \_ \rfloor$.

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

variable
  Δ Δ' Δ₁ Δ₂ : Subset Γ

⌊_⌋ : Subset Γ → Ctx
⌊ Empty ⌋           = []
⌊ Drop Δ ⌋          = ⌊ Δ ⌋
⌊ Keep {Γ} {τ} Δ ⌋  = τ :: ⌊ Δ ⌋
```

We can then define a subset-like order $\subseteq$:

```
_⊆_ : Subset Γ → Subset Γ → Set
Δ₁ ⊆ Keep Δ₂ = pop Δ₁ ⊆ Δ₂
Empty ⊆ Empty = ⊤
Drop Δ₁ ⊆ Drop Δ₂ = Δ₁ ⊆ Δ₂
Keep Δ₁ ⊆ Drop Δ₂ = ⊥
```

Dealing with Subset will simplify our task, especially the correctness proofs. While an OPE itself could be used to express a similar relation between contexts, its inhabitants are not unique. For example, [NAT] can be embedded into [NAT , NAT] either by keeping the first element or the second. This would complicate equality proofs on terms including witnesses of the relation. For similar reasons, it is unclear how we would want to define a union operator _∪_ : Ctx → Ctx → Ctx, but it is trivial on Subsets.

From now on, we will only consider expressions Expr ⌊ Δ ⌋ σ in some sub-context. Initially, we take Δ = all Γ : Subset Γ, the complete sub-context of the original context.

3.2 Live Variable Analysis

The next step is to annotate each expression with its *live variables*, the sub-context Δ' that is really used. To that end, we define annotated expressions LiveExpr Δ Δ' σ. Note that Δ is treated as Γ before, conceptually accumulating all defined variables top-down. On the other hand, Δ' accumulates used variables bottom-up, starting with a singleton sub-context at the variable occurences.

```
data LiveExpr {Γ : Ctx} : (Δ Δ' : Subset Γ) → (σ : U) → Set where
  Val :
    ⟦ σ ⟧ →
    LiveExpr Δ ∅ σ
  Plus :
    LiveExpr Δ Δ₁ NAT →
    LiveExpr Δ Δ₂ NAT →
    LiveExpr Δ (Δ₁ ∪ Δ₂) NAT
  Let :
    LiveExpr Δ Δ₁ σ →
    LiveExpr {σ :: Γ} (Keep Δ) Δ₂ τ →
    LiveExpr Δ (Δ₁ ∪ pop Δ₂) τ
  Var :
    (x : Ref σ ⌊ Δ ⌋) →
    LiveExpr Δ (sing Δ x) σ
```

To create such annotated expressions, we need to perform some static analysis of our source programs. The function analyse computes the live sub-context Δ' together with a matching annotated expression.

```
-- decide which variables are used or not
analyse : ∀ Δ → Expr ⌊ Δ ⌋ σ → Σ[ Δ' ∈ Subset Γ ] LiveExpr Δ Δ' σ
analyse Δ (Val v) = ∅ , Val v
analyse Δ (Plus e₁ e₂) with analyse Δ e₁ | analyse Δ e₂
... | Δ₁ , le₁ | Δ₂ , le₂ = (Δ₁ ∪ Δ₂) , Plus le₁ le₂
analyse Δ (Let e₁ e₂) with analyse Δ e₁ | analyse (Keep Δ) e₂
... | Δ₁ , le₁ | Δ₂ , le₂ = (Δ₁ ∪ pop Δ₂) , Let le₁ le₂
analyse Δ (Var x) = sing Δ x , Var x
```

The only requirement we have for it is that we can forget the annotations again.

```
-- forget the information about variable usage
forget : LiveExpr Δ Δ' σ → Expr ⌊ Δ ⌋ σ


-- forget ∘ analyse ≡ id
analyse-preserves : (e : Expr ⌊ Δ ⌋ σ) → forget (proj₂ (analyse Δ e)) ≡ e
```

3.3 Transformation

It is useful to realise that we can evaluate LiveExpr directly, almost identically to the previous eval function. The main difference is in the Let-case, where we match on Δ_2 to distinguish whether the bound variable is live. If it is not, we can directly evaluate the body, ignoring the bound declaration.

Another important detail is that evaluation works under an environment for any sub-context Δ_u containing (at least) the live variables Δ'. Requiring an Env $\lfloor\ \Delta\ \rfloor$ would force us to provide a value even for unused variables. Fixing it to exactly Env $\lfloor\ \Delta'\ \rfloor$, however, would make it necessary to project the environment for each recursive call, since the subexpressions generally only use parts of the context.

```
evalLive : ∀ Δᵤ → LiveExpr Δ Δ' τ → Env ⌊ Δᵤ ⌋ → .(Δ' ⊆ Δᵤ) → ⟦ τ ⟧
evalLive Δᵤ (Val v) env H = v
evalLive Δᵤ (Plus {Δ} {Δ₁} {Δ₂} e₁ e₂) env H =
    evalLive Δᵤ e₁ env (⊆∪₁-trans Δ₁ Δ₂ Δᵤ H)
  + evalLive Δᵤ e₂ env (⊆∪₂-trans Δ₁ Δ₂ Δᵤ H)
evalLive Δᵤ (Let {Δ = Δ} {Δ₁ = Δ₁} {Δ₂ = Drop Δ₂} e₁ e₂) env H =
  evalLive (Drop Δᵤ) e₂ env (⊆∪₂-trans Δ₁ Δ₂ Δᵤ H)
evalLive Δᵤ (Let {Δ = Δ} {Δ₁ = Δ₁} {Δ₂ = Keep Δ₂} e₁ e₂) env H =
  evalLive (Keep Δᵤ) e₂
    (Cons (evalLive Δᵤ e₁ env (⊆∪₁-trans Δ₁ Δ₂ Δᵤ H)) env)
    (⊆∪₂-trans Δ₁ Δ₂ Δᵤ H)
evalLive {Γ} {Δ} Δᵤ (Var x) env H = lookupLive Δ Δᵤ x env H
```

This *optimised semantics* hints at a similar program transformation and will be useful when proving it to be correct. The transformation simply maps each constructor to its counterpart in Expr, with some variable renaming (e.g. from $\lfloor\ \Delta_1\ \rfloor$ to $\lfloor\ \Delta_1 \cup \Delta_2\ \rfloor$) and the abovementioned case distinction, where we can omit unused bindings.

```
dbe : LiveExpr Δ Δ' σ → Expr ⌊ Δ' ⌋ σ
dbe (Val v) =
  Val v
dbe (Plus {Δ} {Δ₁} {Δ₂} e₁ e₂) =
  Plus (injExpr₁ Δ₁ Δ₂ (dbe e₁)) (injExpr₂ Δ₁ Δ₂ (dbe e₂))
dbe (Let {Δ₁ = Δ₁} {Δ₂ = Drop Δ₂} e₁ e₂) =
  injExpr₂ Δ₁ Δ₂ (dbe e₂)
dbe (Let {Δ₁ = Δ₁} {Δ₂ = Keep Δ₂} e₁ e₂) =
  Let
    (injExpr₁ Δ₁ Δ₂ (dbe e₁))
    (renameExpr (Keep Δ₂) (Keep (Δ₁ ∪ Δ₂)) (⊆∪₂ Δ₁ Δ₂) (dbe e₂))
dbe {Γ} {Δ} (Var x) =
  Var (sing-ref Δ x)

sing-ref : (Δ : Subset Γ) (x : Ref σ ⌊ Δ ⌋) → Ref σ ⌊ sing Δ x ⌋
```

As opposed to the type signature of forget, which returns to the original context, in dbe we drop unused variables, only keeping $\lfloor\ \Delta'\ \rfloor$.

3.4 Correctness

We want to show that dead binding elimination preserves semantics: eval ∘ dbe ∘ analyse ≡ eval. Since we know that forget ∘ analyse ≡ id, it is sufficient to show eval ∘ dbe ≡ eval ∘ forget. The proof gets simpler if we split it up using the optimised semantics:

$$\mathsf{eval} \circ \mathsf{dbe} \equiv \mathsf{evalLive} \equiv \mathsf{eval} \circ \mathsf{forget}$$

The actual proof statements are more involved, since they quantify over the expression and environment used. As foreshadowed in the definition of evalLive, the statements are also generalised to evaluation under any Env ⌊ Δ_u ⌋, as long as it contains the live sub-context. This gives us more flexibility when using the inductive hypothesis.

```
-- eval ∘ dbe ≡ evalLive
dbe-correct :
  (e : LiveExpr Δ Δ' σ) (Δᵤ : Subset Γ) (env : Env ⌊ Δᵤ ⌋) →
  .(H : Δ' ⊆ Δᵤ) →
  eval (renameExpr Δ' Δᵤ H (dbe e)) env ≡ evalLive Δᵤ e env H


-- evalLive ≡ eval ∘ forget
evalLive-correct :
  (e : LiveExpr Δ Δ' σ) (Δᵤ : Subset Γ) (env : Env ⌊ Δ ⌋) →
  .(H' : Δ' ⊆ Δᵤ) → .(H : Δᵤ ⊆ Δ) →
  evalLive Δᵤ e (prjEnv Δᵤ Δ H env) H' ≡ eval (forget e) env
```

Both proofs work inductively on the expression, with most cases being a straight-forward congruence. The interesting one is again Let, where we split cases on the variable being used or not and need some auxiliary facts about evaluation, renaming and sub-contexts.

An important detail is that the ⊆ proofs are *irrelevant arguments*, indicated by the preceding dot. This is possible since their witnesses are unique and makes it clear to the Agda type checker that e.g. prjEnv Δ' Δ H_1 and prjEnv Δ' Δ H_2 are equal.

3.5 Iterating the Analysis

The declaration of a binding that is removed can contain the only occurrences of some other variable. This makes another binding dead, allowing further optimisation when running the algorithm again. While in our simple setting all these bindings could be identified in a single pass using *strong live variable analysis*, in general it can be useful to simply iterate the optimisation until a fixpoint is reached.

This approach is not structurally recursive and the number of iterations is not known upfront, so Agda's termination checker needs our help. We

observe that the algorithm must terminate, since the number of bindings decreases with each iteration (except for the last) and cannot become negative. This corresponds to the ascending chain condition in program analysis literature [5]. To convince the termination checker, we use *well-founded recursion* [3] on the number of bindings.

The overall correctness follows directly from the correctness of each individual iteration step.

4 Further Work

4.1 Extending the Language

Since our language only contains let-bindings, it might be of interest to extend it with λ-abstractions (forming a simply-typed λ-calculus). Some increase in complexity seems necessary to eliminate applications of functions that do not use their argument, but we hope that our work is still largely applicable. The problem gets more challenging when introducing recursive bindings. Conversely, adding sum and product types might require more extensive bookkeeping, but should not pose fundamental difficulties.

4.2 Other Analyses

There are several other binding-related transformations to explore, such as moving bindings up or down in the syntax tree. Another interesting type of optimisation is avoidance of redundant computations using *available expression analysis*. An example is *common subexpression elimination*, where subexpressions get replaced by variables bound to equivalent declarations (pre-existing or newly created).

Ideally, further exploration will lead to the discovery of common patterns and useful strategies for performing optimisations on intrinsically typed syntax trees.

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018.

[2] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*, 1999.

[3] Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers – an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016.

[4] James Maitland Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2009.

[5] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Berlin, Heidelberg, Germany, 1st edition, 2014.

[6] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[7] Mitchell Pickard and Graham Hutton. Calculating dependently-typed compilers (functional pearl). *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.