Provingly Correct Optimisations on Intrinsically Typed Expressions

Extended Abstract

Matthias Heinzel
Utrecht University
Utrecht, Netherlands
m.h.heinzel@students.uu.nl

1 Introduction

When writing a compiler for a functional 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 [Norell 2007] is to define an intrinsically typed syntax tree, where expressions are scope- and type-safe by construction and admit a total evaluation function [Augustsson and Carlsson 1999]. This construction has featured in several papers, exploring basic operations like renaming and substitution [Allais et al. 2018] as well as compilation to different target languages [Pickard and Hutton 2021, 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.

Since our work is still in progress, we will mainly present a specific optimisation, *dead binding elimination*. It is implemented by first annotating expressions with variable usage information and then removing bindings that turn out to be unused. We further prove that the optimisation is semantics-preserving.

2 Dead Binding Elimination

2.1 Intrinsically Typed Expressions with Binders

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.

$$P, Q ::= v \mid P + Q \mid \ \dots \ \mid \textbf{let } x = P \textbf{ in } Q \mid x$$

In Agda, the type of expressions Expr is indexed by its return type (τ : U) and context (Γ : Ctx).

Each free variable is a de Bruijn index into the context and acts as a proof that the context contains an element of the matching type. We can see how the context changes when introducing a new binding:

```
data Expr (Γ : Ctx) : (τ : U) → Set where
  Let : Expr Γ σ → Expr (τ :: Γ) τ → Expr Γ τ
  ...
```

This allows the definition of a total evaluator using a matching environment:

```
eval : Expr Γ τ → Env Γ → ⟦τ⟧
```

2.2 Sub-contexts

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.

To reason about the part of a context that is live (actually used), we introduce *sub-contexts*. Conceptually, these are contexts that admit an *order-preserving embedding* (OPE) [Chapman 2009] into the original context, and we capture this notion in a single data type. For each element of a context, a sub-context specifies whether to Keep or Drop it.

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

The context uniquely described by a sub-context is then given by a function ⌊_⌋ : SubCtx Γ → Ctx, and we further know its embedding.

We now define ⊆ : SubCtx Γ → SubCtx Γ → Set, stating that one sub-context is a subset of the other. Its witnesses are unique, which simplifies the correctness proofs. A similar relation on Ctx does not have this property (e.g. [NAT] can be embedded into [NAT, NAT] either by keeping the first element or the second), which would complicate equality proofs on terms including witnesses of $\subseteq$.

From now on, we will only consider expressions Expr $\lfloor\Delta\rfloor\ \tau$ in some sub-context. Initially, we take $\Delta = \text{all}\ \Gamma : \text{SubCtx}\ \Gamma$, the complete sub-context of the original context.

2.3 Live Variable Analysis

Now we can annotate each expression with its *live variables*, the sub-context $\Delta' \subseteq \Delta$ that is really used. To that end, we define annotated expressions LiveExpr $\Delta\ \Delta'\ \tau$. While Δ is treated as Γ was before, Δ' now only contains live variables, starting with a singleton sub-context at the variable usage sites.

```
data LiveExpr : (Δ Δ′ : SubCtx Γ) (τ : U) → Set where
  Let : LiveExpr Δ Δ₁ σ →
    LiveExpr (Keep Δ) Δ₂ τ →
    LiveExpr Δ (Δ₂ ∪ pop Δ₂) τ
```

To create such annotated expressions, we need to perform some static analysis of our source programs. The function analyse computes an existentially qualified live sub-context Δ' together with a matching annotated expression. The only requirement we have for it is that we can forget the annotations again, with forget $\circ$ analyse $\equiv$ id.

```
analyse : Expr ⌊Δ⌋ τ → Σ[Δ′ ∈ SubCtx Γ] LiveExpr Δ Δ τ
forget : LiveExpr Δ Δ′ τ → Expr ⌊Δ⌋ τ
```

2.4 Transformation

Note that we can evaluate LiveExpr directly, differing from eval mainly in the Let-case, where we match on Δ_2 to distinguish whether the bound variable is live. If it is not, we directly evaluate the body, ignoring the bound declaration. Another important detail is that evaluation works under any environment containing (at least) the live context.

```
evalLive :
  LiveExpr Δ Δ′ τ → Env ⌊Δ_U⌋ → .(Δ ⊆ Δ_U) → ⟦τ⟧
```

This *optimised semantics* shows that we can do a similar program transformation and will be useful in its correctness proof. The implementation simply maps each constructor to its counterpart in Expr, with some renaming (e.g. from $\lfloor\Delta_1\rfloor$ to $\lfloor\Delta_1 \cup \Delta_2\rfloor$) and the abovementioned case distinction.

```
dbe : LiveExpr Δ Δ′ τ → Expr ⌊Δ′⌋ τ
dbe (Let {Δ₁} {Drop Δ₂} e₁ e₂) = injExpr₂ Δ₁ Δ₂ (dbe e₂)
dbe ...
```

As opposed to forget, which stays in the original context, here we remove unused variables, only keeping $\lfloor\Delta'\rfloor$.

2.5 Correctness

We want to show that dead binding elimination preserves semantics: eval $\circ$ dbe $\circ$ analyse $\equiv$ eval. Since we know that forget $\circ$ analyse $\equiv$ id, it is sufficient to show the following:

eval $\circ$ dbe $\equiv$ eval $\circ$ forget

The proof gets simpler if we split it up using the optimised semantics.

eval $\circ$ dbe $\equiv$ evalLive $=$ eval $\circ$ 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 $\lfloor\Delta_U\rfloor$, as long as it contains the live sub-context. This gives us more flexibility when using the inductive hypothesis.

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.

2.6 Iterating the Optimisation

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.

Such an iteration is not structurally recursive, so Agda's termination checker needs our help. We observe that the algorithm must terminate since the number of bindings decreases with each iteration (but the last) and cannot become negative. This is the same as the ascending chain condition in program analysis literature [Nielson et al. 2014]. To convince the termination checker, we use *well-founded recursion* [Bove et al. 2016] on the number of bindings.

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

3 Preliminary Results

The implementation and correctness proof of dead binding elimination are complete, the Agda source code is available online [1]. One interesting observation is that the correctness proof does not rely on how analyse computes the annotations. At first, this does not seem particularly useful, but for other optimisations the analysis might use complex, frequently changing heuristics to decide which transformations are worth it.

We are currently extending the expression language with λ-abstractions. While some increase in complexity is necessary to eliminate applications of functions that do not use their argument, the correctness proof seems to stay relatively simple.

[1] https://github.com/mheinzel/correct-optimisations

We are further investigating additional binding-related transformations, 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).

Between the different optimisations, we hope to discover common patterns and refine our approach, providing useful strategies for performing optimisations in intrinsically typed compilers.

References

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (jul 2018), 30 pages. https://doi.org/10.1145/3236785

Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming* (Gothenburg).

Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and Recursion in Interactive Theorem Provers – An Overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88. https://doi.org/10.1017/S0960129514000115

James Maitland Chapman. 2009. *Type checking and normalisation.* Ph.D. Dissertation. University of Nottingham. Advisor(s) Altenkirch, Thorsten.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2014. *Principles of Program Analysis* (1st ed.). Springer Berlin, Heidelberg, Germany. https://doi.org/10.1007/978-3-662-03811-6

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology. Advisor(s) Jansson, Patrik.

Mitchell Pickard and Graham Hutton. 2021. Calculating Dependently-Typed Compilers (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 82 (aug 2021), 27 pages. https://doi.org/10.1145/3473587