Thesis for the degree of Master Of Science

Analysis and Transformation of Intrinsically Typed Syntax

Matthias Heinzel
1632256

Daily supervisor
Dr. Wouter Swierstra

Second supervisor
Dr. Marco Vassena

June 20, 2023

Software Technology Group
Department of Information and Computing Sciences
Universiteit Utrecht

Abstract

The correctness of variable representations used in compilers usually depends on the compiler writers' diligence to maintain complex invariants. Program transformations that manipulate the binding structure are therefore tricky to get right. In a dependently typed programming language such as Agda, we can however make use of intrinsically typed syntax trees to enforce type- and scope-safety by construction, ruling out a large class of binding-related bugs. We show how to perform (and prove correct) dead binding elimination and let-sinking using intrinsically typed de Bruijn indices. To avoid repeated traversals of the transformed expression, we include variable liveness information into the syntax tree and later employ a co-de-Bruijn representation. Finally, we perform transformations in this style syntax-generically.

Contents

Chapter 1

Introduction

When writing a compiler for a programming language, an important consideration is the treatment of binders and variables. They are part of most languages and there are several options for representing them, each with different implications for operating on and reasoning about programs. Often, it is possible to represent ill-formed syntax trees where variables do not refer to a suitable binding. This makes it easy to introduce compiler bugs that change the meaning of a program or make it invalid.

When using a dependently typed programming language such as Agda [16], intrinsically typed syntax trees can be used to make such ill-formed programs unrepresentable. Using this well-known technique, expressions become scope- and type-correct by construction, allowing for a total evaluation function [2]. Intrinsically typed constructions have featured in several papers, exploring basic operations like renaming and substitution [1] as well as compilation to different target languages [22, online supplementary material].

At the same time, there are large classes of important transformations that have not yet received much attention in an intrinsically typed setting. Optimisations, for example, play a central role in practical compilers, but establishing their correctness is often not trivial, with ample opportunity for binding-related mistakes [24, 12]. Letting the type checker keep track of invariants promises to remove common sources of bugs. A mechanised proof of semantics preservation can further increase confidence in the transformation's correctness.

In return for the guarantees provided, some additional work is required. Program *analysis* not only needs to identify optimisation opportunities, but potentially also provide a proof witness that the optimisation is safe, e.g. that some dead code is indeed unused. For the *transformation* of the intrinsically typed program, the programmer then has to convince the type checker that type- and scope-correctness invariants are preserved, which can be cumbersome. The goal of this thesis is to understand these consequences better and explore techniques for dealing with them.

A crucial aspect is that of *variable liveness*. Whether it is safe to apply a binding-related transformation usually depends on which parts of the program

make use of which binding. We employ several ways of providing and using variable liveness information for program transformations.

Structure Chapter 2 introduces the simple expression language we will work with and then gives some background information on program analysis and transformation, as well as different binding representations and their pitfalls.

In chapter 3 we start by showing a typical intrinsically typed de Bruijn representation of our expression language. We then explain thinnings and motivate their application to computing variable liveness. Equipped with these tools, we implement dead binding elimination and let-sinking, first on the standard de Bruijn representation, later more efficiently on a syntax tree annotated with the results of live variable analysis. We prove that both versions of dead binding elimination preserve semantics.

Chapter 4 continues the development by showing that variable liveness information can serve as the main mechanism for representing bindings, as witnessed by McBride's co-de-Bruijn representation [14]. After explaining how co-de-Bruijn terms work and can be constructed from de Bruijn terms, we again implement dead binding elimination and prove it correct. Finally, we also manage to implement let-sinking, but encounter several complications and struggle with the proof of correctness.

In chapter 5, we explain the idea of syntax-generic programming as presented by Allais, Atkey, Chapman, McBride and McKinna [1] and extend it with basic support for the co-de-Bruijn representation. This allows us to convert between de Bruijn and co-de-Bruijn terms and perform dead binding elimination *syntax-generically*.

In the end, chapter 6 discusses our main observations, open questions, and opportunities to continue the work presented here.

Contributions Our main contributions are:

- an implementation of (strongly) live variable analysis resulting in annotated intrinsically typed syntax trees (sections 3.3 and 3.4.2)
- an implementation of dead binding elimination on intrinsically typed syntax trees of three different flavours: de Bruijn (section 3.4.1), annotated de Bruijn (section 3.4.2), and co-de-Bruijn (section 4.3)
- proofs of correctness (preservation of semantics) for each implementations of dead binding elimination
- an implementation of let-sinking on intrinsically typed syntax trees of three different flavours (sections 3.5.1, 3.5.2 and 4.4)
- an incomplete proof of correctness for co-de-Bruijn let-sinking, with an explanation of the main challenges
- a generic interpretation of the syntax descriptions presented by Allais et al. [1] into co-de-Bruijn terms (section 5.2)

- syntax-generic conversion between de Bruijn and co-de-Bruijn terms (section 5.3)
- a syntax-generic implementation of dead binding elimination on co-de-Bruijn terms (section 5.4)

The Agda code and LaTeX source of this document are available online[1].

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

Chapter 2

Preliminaries

As a running example, we will consider a simple expression language based on the λ-calculus [3]. On top of variables with names $\{x, y, z, a, b, c, f, g, \ldots\}$, function application and λ-abstraction, we add let-bindings, primitive values $v \in \mathbb{B} \cup \mathbb{N}$ (with $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$) and a binary addition operator. Since we are primarily concerned with variables and binders, the choice of possible values and primitive operations on them is mostly arbitrary and can be extended easily.

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

To reduce the number of required parentheses, we give function application the highest and let-bindings the lowest precedence.

Let-bindings allow to bind a declaration P to a variable x. While any let-binding $\textbf{let}\ x = P\ \textbf{in}\ Q$ can be emulated using an immediately applied λ-abstraction $(\lambda x.\ Q)\ P$, they are very common and can benefit from transformations that target them specifically. We omit further constructs such as branching operators, recursive bindings or a fixpoint operator, but discuss some potential additions and their implications at the end (section 6.1.1).

2.1 Program Analysis and Transformation

We mainly consider transformations aimed at optimising functional programs. A large number of program analyses and optimisations are presented in the literature [15, 23] and used in production compilers such as the Glorious Haskell Compiler (GHC). We generally focus on transformations dealing with variable

binders, such as *inlining*, *let-floating*, *common subexpression elimination* and *dead binding elimination*.

Dead Binding Elimination An expression is not forced to make use of all bindings to which it has access. Specifically, a let-binding introduces a new variable, but it might never be used in the body. Consider for example the following expression:

$$\begin{aligned}&\textbf{let } x = 42 \textbf{ in}\\ &\quad\textbf{let } y = x + 6 \textbf{ in}\\ &\qquad\textbf{let } z = y + 7 \textbf{ in}\\ &\qquad\quad x\end{aligned}$$

Here, the binding for z is clearly unused, as the variable never occurs in the body. Such dead bindings can be identified by *live variable analysis* and consequently be removed.

Note that y is not needed either: Removing z will make y unused. Therefore, multiple iterations of live variable analysis and binding elimination might be required to remove as many bindings as possible. Alternatively, *strongly live variable analysis* can achieve the same result in a single pass by ignoring variable occurrences in the declaration of variables unless that variable is live itself.

Let-sinking Even when a binding cannot be removed, it can still be beneficial to move it to a different location. Several such strategies have for example been described and evaluated in the context of lazy functional programs [19]. Of those, we will focus on the *let-sinking* transformation (called let-floating in the paper). Generally, the further inward a let binding is moved, the better: other optimisations might get unlocked, and in the presence of branching, the declaration might never be evaluated.

Of course, we must ensure that the binding remains in scope for all of the variable's occurrences and should consider some exceptions to the rule of sinking as far as possible. We generally do not want to duplicate bindings or move them inside λ-abstractions, which can also duplicate work if the function is applied multiple times.

Let us look at what this means when sinking the binding for x in the following example with free variables f and g:

$$\textbf{let } x = f\ 42 \textbf{ in } (g\ 1)\ (f\ x + x)$$

$$\Downarrow$$

$$(g\ 1)\ (\textbf{let } x = f\ 42 \textbf{ in } f\ x + x)$$

The variable x is only used in the right side of the function application, but we cannot sink it any further, since it occurs on both sides of the addition.

Interestingly, let-sinking also covers a central part of *inlining*. When a variable only occurs once (and would thus benefit from inlining), the binding will be moved inwards until it reaches the single occurence, which can then be replaced by the binding's declaration.

$$\begin{array}{l}\textbf{let } x = f\ 42 \textbf{ in} \\ \quad \textbf{let } y = f\ 43 \textbf{ in} \\ \quad\quad f\ y + (y + x)\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}\textbf{let } y = f\ 43 \textbf{ in} \\ \quad f\ y + (y + \textbf{let } x = f\ 42 \textbf{ in } x)\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}\textbf{let } y = f\ 43 \textbf{ in} \\ \quad f\ y + (y + f\ 42)\end{array}$$

2.2 Binding Representations

The notation used so far treats variables as letters, or more generally strings. This is how humans usually write programs and makes it fairly natural to match a variable with its binding. For representing variables in a compiler or mechanised proof, however, different trade-offs apply.

Explicit names Using strings for variables is quite common in practical compilers, but comes with several disadvantages. For example, additional work is necessary if we want the equality of expressions to be independent of the specific variable names chosen (*α-equivalence*). Also, there are pitfalls like variable shadowing and variable capture during substitution, requiring the careful application of variable renamings [3]. Consider for example the following expression, where x is a free variable:

$$\textbf{let } y = x + 1 \textbf{ in } \lambda x.\ y$$

Naively inlining y here causes x to be captured by the λ-abstraction, incorrectly resulting in the program $\lambda x.\ (x + 1)$.

There have been various approaches to help compiler writers maintain the relevant invariants, such as GHC's *rapier* [18], but these are generally still error-prone. The developers of Dex for example used the rapier, but encountered multiple binding-related compiler bugs, leading them to create *the foil* to "make it harder to poke your eye out" [12].

De Bruijn indices With *de Bruijn indices* [11], one can instead adopt a *nameless* representation. Each variable is represented as a natural number, counting the number of nested bindings between variable occurrence and its binding: $\langle 0 \rangle$ refers to the innermost binding, $\langle 1 \rangle$ to the next-innermost etc. If we adapt the syntax for let-bindings to omit the unnecessary variable name, the example expression from dead binding elimination is represented as follows:

$$\begin{array}{ll} \textbf{let } x = 42 \textbf{ in} & \textbf{let } 42 \textbf{ in} \\ \quad \textbf{let } y = x + 6 \textbf{ in} & \quad \textbf{let } \langle 0 \rangle + 6 \textbf{ in} \\ \quad\quad \textbf{let } z = y + 7 \textbf{ in} & \quad\quad \textbf{let } \langle 0 \rangle + 7 \textbf{ in} \\ \quad\quad\quad x & \quad\quad\quad \langle 2 \rangle \end{array}$$

This makes α-equivalence of expressions trivial and avoids variable capture, but there are still opportunities for mistakes during transformations. Inserting or removing a binding requires us to traverse the binding's body and add or subtract 1 from all its free variables. We can see this in our example when removing the innermost (unused) let-binding. If we naively leave the variable $\langle 2 \rangle$ untouched, it will not refer to the declaration 42 anymore, but become a free variable:

$$\begin{array}{l} \textbf{let } 42 \textbf{ in} \\ \quad \textbf{let } \langle 0 \rangle + 6 \textbf{ in} \\ \quad\quad \langle 2 \rangle \quad \textit{– incorrect, should be 1} \end{array}$$

While useful for machines, de Bruijn representation can be unintuitive for humans to reason about. This can be alleviated by formally describing the necessary invariants and using tools to make sure they are upheld. An intrinsically typed de Bruijn representation is one possible way to achieve that, as demonstrated in section 3.1.

Co-de-Bruijn representation Another nameless option we only briefly mention here is the *co-de-Bruijn representation* [14]. It does not only admit a trivial α-equivalence, but its terms are also unchanged by adding or removing bindings in its context. On the other hand, it is even harder for humans to comprehend than de Bruijn syntax. McBride writes that "only a fool would attempt to enforce the co-de-Bruijn invariants without support from a typechecker" and makes heavy use of Agda's dependent type system. We follow his approach closely, as shown in section 4.1.

Other representations There are many other techniques[1] such as higher-order abstract syntax [21] and also combinations of multiple techniques, e.g. the locally nameless representation [7].

[1] There is an introductory blogpost by Jesper Cockx [8] comparing several approaches and their properties using Agda.

Chapter 3

De Bruijn Representation

The main objective of this chapter is to show how to manipulate the binding structure of intrinsically typed de Bruijn syntax. We start by demonstrating how the intrinsically typed representation enforces type- and scope-correctness by making the context of expressions explicit in their type. To talk about the relationship between contexts, we give an introduction to thinnings and some operations on them that will prove useful later. This leads us to the discovery that thinnings can nicely capture the notion of variable liveness, which is fundamental for manipulating bindings. Finally, we use them to describe program transformations and prove their correctness.

For brevity, we will make use of Agda's ability to quantify over variables implicitly. The types of these variables should be clear from their names and context.

3.1 Intrinsically Typed Syntax

Whether we use explicit names or de Bruijn indices, the language as seen so far makes it possible to represent expressions that are ill-typed (e.g. performing addition on Booleans) or -scoped. In Agda, we can similarly define expressions as follows:

```
data RawExpr : Set where
  Var  : Nat → RawExpr
  App  : RawExpr → RawExpr → RawExpr
  Lam  : RawExpr → RawExpr
  Let  : RawExpr → RawExpr → RawExpr
  Num  : Nat → RawExpr
  Bln  : Bool → RawExpr
  Plus : RawExpr → RawExpr → RawExpr
```

But how should expressions like Plus (Bln False) (Var 42) be evaluated? What is the result of adding Booleans and how do we ensure that a value (of

the right type) is provided for each variable used? Clearly, evaluating such an expression must lead to a runtime error.

Sorts The first problem can be addressed by indexing each expression with its *sort* U, the type that it should be evaluated to.

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

variable
  σ τ : U

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

data RawExpr : U → Set where
  Var  : Nat → RawExpr σ
  App  : RawExpr (σ ⇒ τ) → RawExpr σ → RawExpr τ
  Lam  : RawExpr τ → RawExpr (σ ⇒ τ)
  Let  : RawExpr σ → RawExpr τ → RawExpr τ
  Val  : ⟦ σ ⟧ → RawExpr σ
  Plus : RawExpr NAT → RawExpr NAT → RawExpr NAT
```

Note that the values not only consist of natural numbers and Booleans, but also functions between values, introduced by λ-abstraction. Sorts can further be interpreted as Agda types, which we use to represent values, for example during evaluation.

Context Sorts help, but to know if a variable occurrence is valid, one must also consider its *context*, the (typed) bindings that are in scope. We represent the context as a list of sorts: One for each binding in scope, from innermost to outermost.

```
Ctx = List U

variable
  Γ Δ : Ctx
```

De Bruijn indeces can then ensure that they reference an element of a specific type within the context.

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

By also indexing expressions with their context, the invariants can finally guarantee type- and scope-correctness by construction.

```
data Expr : (Γ : Ctx) (τ : U) → 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
```

Note how the context changes when introducing a new binding in the body of a Lam or Let.

Evaluation During evaluation, each variable in scope has a value. Together, these are called an *environment* for a given context.

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

Since variable Ref σ Γ acts as a proof that the environment Env Γ contains an element of type σ, variable lookup is total.

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

As a result, we can define a total evaluator that can only be called with an environment that matches the expression's context.

```
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
```

3.2 Thinnings

Since the context of an expression plays such an important role for its scope-safety, we want some machinery for talking about how different contexts relate to each other. One such relation, which will prove useful soon, is that of being a subcontext, or more precisely a context with an embedding into another. We formalise this notion in the form of *thinnings*, also called *order-preserving embeddings* (OPE) [6].

As several operations on thinnings are used pervasively throughout the rest of the thesis, we briefly introduce them here in a central location we can refer back to. Their applications will become apparent starting from section 3.4.

We closely follow the syntactic conventions of McBride [14], but grow our lists towards the left instead of using backwards lists and postfix operators.

```
data _⊑_ {I : Set} : List I → List I → Set where
  o′ : Δ ⊑ Γ →       Δ ⊑ (τ :: Γ)   -- drop
  os : Δ ⊑ Γ → (τ :: Δ) ⊑ (τ :: Γ)   -- keep
  oz : [] ⊑ []                       -- empty
```

Intuitively, a thinning tells us for each element of the target context whether it also occurs in the source target or not (*keep* or *drop*). As an example, let us embed the list `a :: c :: []` into `a :: b :: c :: []`:

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

Identity and composition Contexts and the thinnings between them form a category with the inital object `[]`. Concretely, this means that there is an empty and identity thinning (keeping none or all elements, respectively), as well as composition of thinnings in sequence, followingidentity and associativity laws.

```
oe : [] ⊑ Γ
oe {Γ = []}      = oz
oe {Γ = _ :: _}  = o′ oe

oi : Γ ⊑ Γ
oi {Γ = []}      = oz
oi {Γ = _ :: _}  = os oi

_⨟_ : Γ₁ ⊑ Γ₂ → Γ₂ ⊑ Γ₃ → Γ₁ ⊑ Γ₃
θ      ⨟ o′ ϕ  = o′ (θ ⨟ ϕ)
o′ θ   ⨟ os ϕ  = o′ (θ ⨟ ϕ)
os θ   ⨟ os ϕ  = os (θ ⨟ ϕ)
oz     ⨟ oz    = oz

law-⨟oi : (θ : Δ ⊑ Γ) → θ ⨟ oi ≡ θ
law-oi⨟ : (θ : Δ ⊑ Γ) → oi ⨟ θ ≡ θ
law-⨟⨟  : (θ : Γ₁ ⊑ Γ₂) (ϕ : Γ₂ ⊑ Γ₃) (ψ : Γ₃ ⊑ Γ₄) →
          θ ⨟ (ϕ ⨟ ψ) ≡ (θ ⨟ ϕ) ⨟ ψ
```

Concatenating thinnings Thinnings cannot just be composed in sequence, but also concatenated.

```
_++⊑_ : Δ₁ ⊑ Γ₁ → Δ₂ ⊑ Γ₂ → (Δ₁ ++ Δ₂) ⊑ (Γ₁ ++ Γ₂)
o′ θ ++⊑ ϕ = o′ (θ ++⊑ ϕ)
os θ ++⊑ ϕ = os (θ ++⊑ ϕ)
oz   ++⊑ ϕ = ϕ
```

This interacts nicely with sequential composition, specifically we prove that $(\theta_1 \fatsemi \theta_2) +\!\!+_{\sqsubseteq} (\phi_1 \fatsemi \phi_2) \equiv (\theta_1 +\!\!+_{\sqsubseteq} \phi_1) \fatsemi (\theta_2 +\!\!+_{\sqsubseteq} \phi_2)$.

Splitting thinnings If we have a thinning into a target context that is concatenated from two segments, we can also split the source context and thinning accordingly. To help the typechecker figure out what we want, we quantify over Γ_1 explicitly, Γ_2 can then usually be inferred.

```
record Split (Γ₁ : List I) (θ : Δ ⊑ (Γ₁ ++ Γ₂)) : Set where
  constructor split
  field
    {Δ₁} : List I
    {Δ₂} : List I
    θ₁ : (Δ₁ ⊑ Γ₁)
    θ₂ : (Δ₂ ⊑ Γ₂)
    eq : Σ (Δ ≡ Δ₁ ++ Δ₂) λ {refl → θ ≡ θ₁ ++⊑ θ₂}


_⊣_ : (Γ₁ : List I) (θ : Δ ⊑ (Γ₁ ++ Γ₂)) → Split Γ₁ θ
```

To show it in action, let us return to the previous example thinning and observe that we could have built it by concatenating two smaller thinnings:

```
θ₁ : (a :: []) ⊑ (a :: [])
θ₁ = os oz
θ₂ : (c :: []) ⊑ (b :: c :: [])
θ₂ = o′ (os oz)
θ : (a :: c :: []) ⊑ (a :: b :: c :: [])
θ = θ₁ ++⊑ θ₂  -- evaluates to os (o′ (os oz))
```

To go into the other direction, we split θ by calling $(\mathsf{a} :: []) \dashv \theta$, resulting in a $\mathsf{split}\ \theta_1\ \theta_2\ \mathsf{eq} : \mathsf{Split}\ (\mathsf{a} :: [])\ \theta$. The target context's first segment (a :: []) needs to be supplied explicitly to specify at which place the splitting should happen. The second segment is then determined by θ's target context.

Things with thinnings We will later deal with things (e.g. expressions) indexed by a context that we do not statically know. We will know, however, that it embeds into a specific context Γ via some thinning. As we have so far been careful to always use the context as the last argument to types, this concept of a thing with a thinning can be defined in a general way, to be used for a wide range of different datatypes.

```
record _⇑_ (T : List I → Set) (Γ : List I) : Set where
  constructor _↑_
  field
    {Δ} : List I
    thing : T Δ
    thinning : Δ ⊑ Γ
```

To avoid manual un- and re-packing, some combinators come in handy:

```
map⇑ : (∀ {Δ} → S Δ → T Δ)     → S ⇑ Γ → T ⇑ Γ
bind⇑ : (∀ {Δ} → S Δ → T ⇑ Δ) → S ⇑ Γ → T ⇑ Γ
thin⇑ : Δ ⊑ Γ                  → T ⇑ Δ → T ⇑ Γ
```

3.3 Variable Liveness

We want to compute an expression's *live variables*, i.e. the part of the context that is live. However, while an expression's context is just a list of sorts, a similar list is not sufficient as the result of this bottom-up analysis.

For example, knowing that two subexpressions both have a single live variable of sort NAT is not enough to deduce whether the combined expression has one or two live variables. We cannot know whether the two variables are the same, unless we have a way to connect them back to the context they come from. Another way of thinking about variable liveness is that for each variable in the context we want a binary piece of information: is it live or dead?

Thinnings support both of these interpretations: A thinning $\Delta \sqsubseteq \Gamma$ can be used to represent the live variables Δ together with an embedding into the context Γ. At the same time, looking at how it is constructed reveals for each element of the context whether it is live (os) or dead (o′).

We will now show for each constructor of our language how to compute its live variables, or rather their thinning into the context.

Values Starting with the most trivial case, values do not use any variables. The thinning from the (empty) list of their life variables consequently drops everything.

```
oe : [] ⊑ Γ
```

Variables A variable occurrence trivially has one live variable. To obtain a suitable thinning, we can make use of the fact that thinnings from a singleton context are isomorphic to references.

```
o-Ref : Ref σ Γ → (σ :: []) ⊑ Γ
o-Ref Top      = os oe
o-Ref (Pop x) = o′ (o-Ref x)
```

```
Ref-o : (σ :: []) ⊑ Γ → Ref σ Γ
Ref-o (o′ θ) = Pop (Ref-o θ)
Ref-o (os θ) = Top
```

Binary constructors Variables in the context are live if they are live in one of the subexpressions (i.e. some thinning is $\mathsf{os}\ \_$).

$$
\begin{array}{l}
\cup\text{-vars} : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \mathsf{List\ I} \\
\cup\text{-vars} \qquad\qquad (\mathsf{o}'\ \theta_1)\ (\mathsf{o}'\ \theta_2) = \cup\text{-vars}\ \theta_1\ \theta_2 \\
\cup\text{-vars}\ \{\Gamma = \sigma :: \_\}\ (\mathsf{o}'\ \theta_1)\ (\mathsf{os}\ \theta_2) = \sigma :: \cup\text{-vars}\ \theta_1\ \theta_2 \\
\cup\text{-vars}\ \{\Gamma = \sigma :: \_\}\ (\mathsf{os}\ \theta_1)\ (\mathsf{o}'\ \theta_2) = \sigma :: \cup\text{-vars}\ \theta_1\ \theta_2 \\
\cup\text{-vars}\ \{\Gamma = \sigma :: \_\}\ (\mathsf{os}\ \theta_1)\ (\mathsf{os}\ \theta_2) = \sigma :: \cup\text{-vars}\ \theta_1\ \theta_2 \\
\cup\text{-vars} \qquad\qquad \mathsf{oz} \qquad \mathsf{oz} \qquad = []
\end{array}
$$

To precisely describe the merged variable liveness information, we then construct a thinning from these combined live variables.

$$
\begin{array}{l}
\_\cup\_ : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \cup\text{-vars}\ \theta_1\ \theta_2 \sqsubseteq \Gamma \\
\mathsf{o}'\ \theta_1 \cup \mathsf{o}'\ \theta_2 = \mathsf{o}'\ (\theta_1 \cup \theta_2) \\
\mathsf{o}'\ \theta_1 \cup \mathsf{os}\ \theta_2 = \mathsf{os}\ (\theta_1 \cup \theta_2) \\
\mathsf{os}\ \theta_1 \cup \mathsf{o}'\ \theta_2 = \mathsf{os}\ (\theta_1 \cup \theta_2) \\
\mathsf{os}\ \theta_1 \cup \mathsf{os}\ \theta_2 = \mathsf{os}\ (\theta_1 \cup \theta_2) \\
\mathsf{oz} \quad \cup \mathsf{oz} \quad = \mathsf{oz}
\end{array}
$$

Furthermore, we can construct the two thinnings *into* the combined live variables and show that this is exactly what we need to reconstruct the original thinnings.

$$
\begin{array}{l}
\text{un-}\cup_1 : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \Delta_1 \sqsubseteq \cup\text{-vars}\ \theta_1\ \theta_2 \\
\text{un-}\cup_2 : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \Delta_2 \sqsubseteq \cup\text{-vars}\ \theta_1\ \theta_2
\end{array}
$$

$$
\begin{array}{l}
\text{law-}\cup_1\text{-inv} : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \\
\qquad\qquad \text{un-}\cup_1\ \theta_1\ \theta_2 \fatsemi (\theta_1 \cup \theta_2) \equiv \theta_1 \\
\text{law-}\cup_2\text{-inv} : (\theta_1 : \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 : \Delta_2 \sqsubseteq \Gamma) \to \\
\qquad\qquad \text{un-}\cup_2\ \theta_1\ \theta_2 \fatsemi (\theta_1 \cup \theta_2) \equiv \theta_2
\end{array}
$$

Binders When moving up over a binder, the bound variable gets removed from the context. In case it was part of the live variables, it also has to be removed there. This is done using pop, again with thinnings from and into the resulting context.

$$
\begin{array}{l}
\mathsf{pop\text{-}vars} : \Delta \sqsubseteq \Gamma \to \mathsf{List\ I} \\
\mathsf{pop\text{-}vars}\ \{\Delta = \Delta\} \qquad (\mathsf{o}'\ \theta) = \Delta \\
\mathsf{pop\text{-}vars}\ \{\Delta = \_ :: \Delta\}\ (\mathsf{os}\ \theta) = \Delta \\
\mathsf{pop\text{-}vars} \qquad\qquad \mathsf{oz} \qquad = [] \\
\mathsf{pop} : (\theta : \Delta \sqsubseteq (\sigma :: \Gamma)) \to \mathsf{pop\text{-}vars}\ \theta \sqsubseteq \Gamma \\
\mathsf{pop}\ (\mathsf{o}'\ \theta) = \theta \\
\mathsf{pop}\ (\mathsf{os}\ \theta) = \theta
\end{array}
$$

$$
\begin{array}{l}
\mathsf{un\text{-}pop} : (\theta : \Delta \sqsubseteq (\sigma :: \Gamma)) \to \Delta \sqsubseteq (\sigma :: \mathsf{pop\text{-}vars}\ \theta) \\
\mathsf{law\text{-}pop\text{-}inv} : (\theta : \Delta \sqsubseteq (\sigma :: \Gamma)) \to \mathsf{un\text{-}pop}\ \theta \fatsemi \mathsf{os}\ (\mathsf{pop}\ \theta) \equiv \theta
\end{array}
$$

Let-bindings For let-bindings, one option is to treat them as an immediate application of a λ-abstraction, combining the methods we just saw. This corresponds to weakly live variable analysis, since even if the variable is dead, we end up considering other variables to be live if they are used in its declaration.

$$\begin{aligned}
&_\cup_{\mathsf{let}}_ \;:\; (\theta_1 \;:\; \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 \;:\; \Delta_2 \sqsubseteq (\sigma :: \Gamma)) \;\to\; \cup\text{-}\mathsf{vars}\;\theta_1\;(\mathsf{pop}\;\theta_2) \sqsubseteq \Gamma\\
&\theta_1 \;\cup_{\mathsf{let}}\; \theta_2 \;=\; \theta_1 \;\cup\; \mathsf{pop}\;\theta_2
\end{aligned}$$

The other option is to do strongly live variable analysis with a custom operation $\_\cup_{\mathsf{let}}\_$ that ignores the declaration's context if it is unused in the body.

$$\begin{aligned}
&\cup_{\mathsf{let}}\text{-}\mathsf{vars} \;:\; (\theta_1 \;:\; \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 \;:\; \Delta_2 \sqsubseteq (\sigma :: \Gamma)) \;\to\; \mathsf{Ctx}\\
&\cup_{\mathsf{let}}\text{-}\mathsf{vars}\;\{\Delta_2 \;=\; \Delta_2\}\;\theta_1\;(\mathsf{o}'\;\theta_2) \;=\; \Delta_2\\
&\cup_{\mathsf{let}}\text{-}\mathsf{vars}\qquad\qquad\;\; \theta_1\;(\mathsf{os}\;\theta_2) \;=\; \cup\text{-}\mathsf{vars}\;\theta_1\;\theta_2\\
&_\cup_{\mathsf{let}}_ \;:\; (\theta_1 \;:\; \Delta_1 \sqsubseteq \Gamma)\,(\theta_2 \;:\; \Delta_2 \sqsubseteq (\sigma :: \Gamma)) \;\to\; \cup_{\mathsf{let}}\text{-}\mathsf{vars}\;\theta_1\;\theta_2 \sqsubseteq \Gamma\\
&\theta_1 \;\cup_{\mathsf{let}}\; (\mathsf{o}'\;\theta_2) \;=\; \theta_2\\
&\theta_1 \;\cup_{\mathsf{let}}\; (\mathsf{os}\;\theta_2) \;=\; \theta_1 \;\cup\; \theta_2
\end{aligned}$$

We do not need the composed thinnings into the live variables, as we will always distinguish the two cases of θ_2 anyways and can then rely on the thinnings defined for $\_\cup\_$.

To illustrate the difference, let us return to an example shown earlier:

$$\begin{array}{ll}
\textbf{let}\ x = 42\ \textbf{in} & \textbf{let}\ 42\ \textbf{in}\\
\quad \textbf{let}\ y = x + 6\ \textbf{in} & \quad \textbf{let}\ \langle 0\rangle + 6\ \textbf{in}\\
\quad\quad \textbf{let}\ z = y + 7\ \textbf{in} & \quad\quad \textbf{let}\ \langle 0\rangle + 7\ \textbf{in}\\
\quad\quad\quad x & \quad\quad\quad \langle 2\rangle
\end{array}$$

If we focus on the subexpression in the last two lines, we see that in our syntax representation it is an $\mathsf{Expr}\ \mathsf{NAT}\ (\mathsf{NAT} :: \mathsf{NAT} :: [])$, where the first element of the context correspondes to y, the second to x.

$$\begin{aligned}
&\mathsf{Let}\ (\mathsf{Plus}\ (\mathsf{Var}\ \mathsf{Top})\ (\mathsf{Val}\ 7))\\
&\quad (\mathsf{Var}\ (\mathsf{Pop}\ (\mathsf{Pop}\ \mathsf{Top})))
\end{aligned}$$

In the declaration, only the innermost binding y is live, so we have a thinning $\mathsf{os}\ (\mathsf{o}'\ \mathsf{oz})$. In the body (with an additional binding in scope), we have $\mathsf{o}'\ (\mathsf{o}'\ (\mathsf{os}\ \mathsf{oz}))$. With the weak version of $\_\cup_{\mathsf{let}}\_$ we get

$$\mathsf{os}\ (\mathsf{o}'\ \mathsf{oz})\ \cup_{\mathsf{let}}\ \mathsf{o}'\ (\mathsf{o}'\ (\mathsf{os}\ \mathsf{oz}))\ =\ \mathsf{os}\ (\mathsf{o}'\ \mathsf{oz})\ \cup\ \mathsf{o}'\ (\mathsf{os}\ \mathsf{oz})\ =\ \mathsf{os}\ (\mathsf{os}\ \mathsf{oz})$$

stating that both variables in scope are live. With the strong version, on the other hand, only y is considered live:

$$\mathsf{os}\ (\mathsf{o}'\ \mathsf{oz})\ \cup_{\mathsf{let}}\ \mathsf{o}'\ (\mathsf{o}'\ (\mathsf{os}\ \mathsf{oz}))\ =\ \mathsf{os}\ (\mathsf{o}'\ \mathsf{oz})$$

3.4 Dead Binding Elimination

3.4.1 Direct Approach

To make the decision whether a binding can be removed, we need to find out if it is used or not. This can be achieved by returning liveness information as part of the transformation's result and use that after the recursive call on the body. Precisely, we return an $\mathsf{Expr}\ \sigma\ \Uparrow\ \Gamma$, the transformed expression in a reduced context of only its live variables, together with a thinning into the original one.

Transformation The transformation proceeds bottom-up. Once all subexpressions have been transformed and we know their live variables, we calculate the overall live variables with their corresponding thinnings. Since the constructors of Expr require the subexpressions' context to match their own, we need to rename the subexpressions accordingly.

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

Each renaming traverses the expression and we end up renaming the same parts of the expressions repeatedly (at each binary constructor). While this is clearly inefficient, it cannot be avoided easily with the approach shown here. If we knew upfront which context the expression should have in the end, we could immediately produce the result in that context, but we only find out which variables are live *after* doing the recursive call. Separately querying liveness before doing the recursive calls would also require redundant traversals, but we will show a solution to this issue in the next section.

Most cases of the implementation keep the expression's structure unchanged, only manipulating the context:

```
dbe : Expr σ Γ → Expr σ ⇑ Γ
dbe (Var x) =
  Var Top ↑ o-Ref x
dbe (App e₁ e₂) =
  let e₁′ ↑ θ₁ = dbe e₁
      e₂′ ↑ θ₂ = dbe e₂
  in App (rename-Expr (un-∪₁ θ₁ θ₂) e₁′) (rename-Expr (un-∪₂ θ₁ θ₂) e₂′)
       ↑ (θ₁ ∪ θ₂)
dbe (Lam e₁) =
  let e₁′ ↑ θ = dbe e₁
  in Lam (rename-Expr (un-pop θ) e₁′) ↑ pop θ
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)
```

```
dbe (Val v) =
  (Val v) ↑ oe
dbe (Plus e₁ e₂) =
  ...   -- just as App
```

For Let, we split on the binding being live or dead in dbe e_2. Only if it is dead will the typechecker allow us to return e_2' without the binding. Finally, note that checking liveness *after* already removing dead bindings from the body corresponds to *strongly* live variable analysis.

Correctness We prove preservation of semantics based on the total evaluation function. Since we allow functions as values, reasoning about equality requires us to postulate extensionality. This does not impact the soundness of the proof and could be avoided by moving to a different setting, such as homotopy type theory [25].

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

As the transformed expression generally has a different context than the input expression, they cannot be evaluated under the same environment. We project the environment to match the smaller context, dropping all its unneeded elements.

```
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
```

Alternatively, it is possible to rename the expression (and law-eval-rename-Expr witnesses that both approaches are exchangable), but in this case it turns out to be more convenient to reason about context projection.

```
law-eval-rename-Expr :
  (e : Expr σ Δ) (θ : Δ ⊑ Γ) (env : Env Γ) →
  eval (rename-Expr θ e) env ≡ eval e (project-Env θ env)
```

The inductive proof requires combining a large number of laws about evaluation, renaming, environment projection and the thinnings we constructed. The Lam case exemplifies that. We omit most of the proof terms except for the inductive hypothesis, as they are rather long. The intermediate terms still demonstrate how we need to apply several lemmas.

```
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 env))
     ≡⟨ law-eval-rename-Expr e₁′ (un-pop θ₁) _ ⟩
       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)
     ■
...
```

The cases for binary operators have a similar structure, but are even longer, as they need to apply laws once for each subexpression. Since the implementation uses a **with**-abstraction for the Let-case, the proof does the same:

```
dbe-correct (Let e₁ e₂) env with dbe e₁ | dbe e₂ | dbe-correct e₁ | dbe-correct e₂
... | e₁′ ↑ θ₁ | e₂′ ↑ o′ θ₂ | h1 | h2 =
  h2 (Cons (eval e₁ env) env)
... | e₁′ ↑ θ₁ | e₂′ ↑ os θ₂ | h1 | h2 =
  let v = eval (rename-Expr (un-∪₁ θ₁ θ₂) e₁′) (project-Env (θ₁ ∪ θ₂) env)
  in
    eval (rename-Expr (os (un-∪₂ θ₁ θ₂)) e₂′)
      (Cons v (project-Env (θ₁ ∪ θ₂) env))
  ≡⟨ ... ⟩
    ...   -- long proof
  ≡⟨ ... ⟩
    eval e₂ (Cons (eval e₁ env) env)
  ■
```

Note that we also **with**-abstract over the inductive hypothesis. When abstracting over e.g. dbe e_1, the statement we need to prove gets generalised and then talks about e_1'. However, dbe-correct e_1 talks about dbe e_1 and Agda is not aware of their connection. Generalising dbe-correct e_1 makes it refer to e_1' as well. [1]

3.4.2 Using Annotations

In compilers, it is a common pattern to perform separate analysis and transformation passes, for example with strictness and occurrence analysis in GHC [20]. We can do the same to make variable liveness information available without repeatedly having to compute it on the fly. For dead binding elimination, this allows us to avoid the redundant renaming of subexpressions.

[1] `https://agda.readthedocs.io/en/v2.6.3/language/with-abstraction.html`

Liveness annotations To annotate each part of the expression with its live variables, we first need to define a suitable datatype of annotated expressions LiveExpr $\tau\ \theta$. The thinning θ here captures liveness information in the same way as during the direct transformation in section 3.4.1. Its target context Γ plays the same role as in Expr $\sigma\ \Gamma$, but Δ only contains the live variables.

```
data LiveExpr {Γ : Ctx} : U → {Δ : Ctx} → Δ ⊑ Γ → Set where
  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 :
    {θ₁ : Δ₁ ⊑ Γ} {θ₂ : Δ₂ ⊑ (σ :: Γ)} →
    LiveExpr σ θ₁ →
    LiveExpr τ θ₂ →
    LiveExpr τ (θ₁ ∪let θ₂)
  Val :
    ⟦ σ ⟧ →
    LiveExpr σ oe
  Plus :
    {θ₁ : Δ₁ ⊑ Γ} {θ₂ : Δ₂ ⊑ Γ} →
    LiveExpr NAT θ₁ →
    LiveExpr NAT θ₂ →
    LiveExpr NAT (θ₁ ∪ θ₂)
```

The operator $\_\cup_{\mathsf{let}}\_$ used here can refer to either one of the two versions we introduced, but for the remainder of this thesis we will use the strongly live version.

Analysis To create such an annotated expressions, we need to perform strongly live variable analysis. As we do not know the live variables upfront, analyse computes an existentially qualified context and thinning, together with a matching annotated expression. The implementation is straightforward, directly following the expression's structure.

```
analyse : Expr σ Γ → Σ[ Δ ∈ Ctx ] Σ[ θ ∈ Δ ⊑ Γ ] LiveExpr σ θ
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 ∪-vars θ₁ θ₂ , (θ₁ ∪ θ₂) , App le1 le2
...
```

It is sensible to assume that the only thing analysis does is to attach annotations without changing the structure of the expression. We capture this property by stating that we can always forget the annotations to obtain the original expression (forget ∘ analyse ≡ id).

```
forget : {θ : Δ ⊑ Γ} → LiveExpr τ θ → Expr τ Γ
analyse-preserves :
  (e : Expr τ Γ) →
  let _ , _ , le = analyse e
  in forget le ≡ e
```

Note that we can evaluate LiveExpr directly, differing from eval in two points: Firstly, since the annotations make it easy to identify dead let-bindings, we can skip their evaluation. Secondly, evaluation works under any environment containing *at least* the live variables. This makes it possible to get by with the minimal required environment, but still gives some flexibility. For example, we can avoid projecting the environment for each recursive call, just manipulating the thinning instead.

```
evalLive : {θ : Δ ⊑ Γ} → LiveExpr τ θ → Env Γ′ → Δ ⊑ Γ′ → ⟦ τ ⟧
...
evalLive (Let {θ₁ = θ₁} {θ₂ = o′ θ₂} e₁ e₂) env θ′ =
  evalLive e₂ env θ′
evalLive (Let {θ₁ = θ₁} {θ₂ = os θ₂} e₁ e₂) env θ′ =
  evalLive e₂
    (Cons (evalLive e₁ env (un-∪₁ θ₁ θ₂ ⨟ θ′)) env)
    (os (un-∪₂ θ₁ θ₂ ⨟ θ′))
...
```

We will later use this to split the correctness proof into multiple small parts.

Transformation The second pass we perform is similar to dbe in the direct approach, but with a few key differences. Firstly, it operates on annotated expressions LiveExpr τ θ for a known thinning $\theta : \Delta \sqsubseteq \Gamma$ instead of discovering the thinning and returning it with the result. However, the transformed expression will not necessarily be returned in the smallest possible context Δ, but any chosen larger context Γ'. This way, instead of inefficiently having to rename afterwards, the result gets created in the desired context straight away. Most cases now simply recurse while accumulating the thinning that eventually gets used to create the variable reference.

```
transform : {θ : Δ ⊑ Γ} → LiveExpr τ θ → Δ ⊑ Γ′ → Expr τ Γ′
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′))
transform (Let {θ₁ = θ₁} {θ₂ = o′ θ₂} e₁ e₂) θ′ =
  transform e₂ 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′)))
transform (Val v) θ′ =
  Val v
transform (Plus {θ₁ = θ₁} {θ₂ = θ₂} e₁ e₂) θ′ =
  Plus (transform e₁ (un-∪₁ θ₁ θ₂ ⨟ θ′))
       (transform e₂ (un-∪₂ θ₁ θ₂ ⨟ θ′))
```

Finally, we can compose analysis and transformation into an operation with the same signature as the direct implementation.

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

Correctness The goal is again to show that dead binding elimination preserves semantics, which we can express with the same statement as before, or conceptually as eval ∘ dbe ≡ eval. We could again immediately attempt a proof by structural induction, but each case would require cumbersome massaging of the thinnings supplied to various operations. Instead, we aim to simplify the proof by breaking it down into smaller parts using the optimised semantics:

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

Both proofs work inductively on the expression, with most cases being a straightforward 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 contexts.

After doing two relatively simple proofs, we can combine them and do the remaining reasoning without having to handle each constructor separately. Conceptually, we pre-compose analyse on both sides and remove forget ∘ analyse (which we know forms an identity) to obtain the desired equality.

eval ∘ transform	≡	eval ∘ forget
eval ∘ transform ∘ analyse	≡	eval ∘ forget ∘ analyse
eval ∘ dbe	≡	eval

Just as transform itself, the proof statements in Agda are generalised to evaluation under any Env Γ', as long as it contains the live variables. This gives us more flexibility when using the inductive hypothesis, showing that it can sometimes be easier to prove something more general.

3.5 Let-sinking

As outlined in section 2.1, we want to move a single let-bindings as far inward as possible without duplicating it or pushing it into a λ-abstraction. Again, we will first show a direct implementation and then employ the annotated expression type.

3.5.1 Direct Approach

Type signature We want to replace a let-binding Let decl e with the result of the transformation sink-let decl e, which suggests a signature like sink-let : Expr σ Γ $\rightarrow$ Expr τ (σ :: Γ) $\rightarrow$ Expr τ Γ. However, while we initially deal with the topmost entry in the context, this changes when going under other binders. The solution chosen here uses list concatenation in the context to allow σ to occur at any position.

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

Choosing [] as the prefix then again results in the signature above.

Transformation Just as dead binding elimination, let-sinking heavily relies on variable liveness information. To know where a binding should be moved, we need to know where it is used. As we are working with a plain (unannotated) syntax tree in this section, we need to query the subexpressions' variable usage on demand, which repeatedly traverses the expression. This is difficult to avoid, since usage information is computed bottom-up, but let-sinking needs to proceed top-down.

More concretely, we need to find out for each subexpression whether it uses the binding we are let-sinking or not. If the binding is unused, we need to make that clear to the typechecker by removing it from the subexpression's context. Therefore, we combine querying and the context change into a single operation we refer to as *strengthening*.

```
strengthen : Expr τ (Γ₁ ++ σ :: Γ₂) → Maybe (Expr τ (Γ₁ ++ Γ₂))
```

We now give the complete implementation of let-sinking before highlighting specific parts.

```
sink-let : Expr σ (Γ₁ ++ Γ₂) → Expr τ (Γ₁ ++ σ :: Γ₂) → Expr τ (Γ₁ ++ Γ₂)
sink-let decl (Var x) with rename-top-Ref x
... | Top = decl
... | Pop x' = Var x'
sink-let decl e@(App e₁ e₂) with strengthen e₁ | strengthen e₂
... | just e₁' | just e₂' = App e₁' e₂'
... | nothing | just e₂' = App (sink-let decl e₁) e₂'
... | just e₁' | nothing = App e₁' (sink-let decl e₂)
... | nothing | nothing = Let decl (rename-top e)
sink-let decl e@(Lam e₁) =
  Let decl (rename-top e)  -- Do not sink into λ-abstractions!
sink-let decl e@(Let e₁ e₂) with strengthen e₁ | strengthen e₂
... | just e₁' | just e₂' = Let e₁' e₂'
... | nothing | just e₂' = Let (sink-let decl e₁) e₂'
... | just e₁' | nothing = Let e₁' (sink-let (weaken decl) e₂)
... | nothing | nothing = Let decl (rename-top e)
sink-let decl (Val v) =
  Val v
sink-let decl e@(Plus e₁ e₂) with strengthen e₁ | strengthen e₂
... | just e₁' | just e₂' = Plus e₁' e₂'
... | nothing | just e₂' = Plus (sink-let decl e₁) e₂'
... | just e₁' | nothing = Plus e₁' (sink-let decl e₂)
... | nothing | nothing = Let decl (rename-top e)
```

Variables When sinking a binding into a variable, there are two possible cases:

1. If the variable references exactly the let-binding we are sinking, we can replace it by the declaration, effectively inlining it.
2. If the variable references a different element of the context, the declaration is unused and we only need to rename the variable into the smaller context.

To distinguish the two cases, we rename the reference (moving the variable in question to the front).

```
rename-top-Ref : Ref τ (Γ₁ ++ σ :: Γ₂) → Ref τ (σ :: Γ₁ ++ Γ₁)
```

If the result is Top, we learn that $\sigma \equiv \tau$ and can return the declaration. If it is Pop x', we can return x', as it does not have the variable of the declaration in its context anymore.

Creating the binding Once we stop sinking the let-binding (e.g. when we reach a λ-abstraction), we insert the declaration. However, the typechecker will not accept a plain Let decl e. It is still necessary to rename the expression, since it makes use of the newly created binding, but expects it at a different de Bruijn index.

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

Binary constructors For binary operators, we need to check which subexpressions make use of the declaration. There are four possible cases:

1. Both of the subexpressions can be strengthened. This means that we are sinking a dead let-binding, which normally should not happen. Nevertheless, we need to handle the case, simply dropping the binding.
2. The right subexpression can be strengthened. We recurse into the left one.
3. The left subexpression can be strengthened. We recurse into the right one.
4. Neither subexpression can be strengthened, as both use the declaration. To avoid duplicating code, we do not sink further, but create a let-binding at the current location.

Binders If we recurse into the body of a let-binding, an additional variable comes into scope. This means that we need to add it to the context prefix Γ_1 and weaken the declaration.

```
weaken : Expr τ Γ → Expr τ (σ :: Γ)
weaken = rename-Expr (o′ oi)
```

This traverses the declaration for each binder it is moved across, but in the next section we will use a simple trick to avoid that.

3.5.2 Using Annotations

Perhaps unsurprisingly, we can again avoid the repeated querying using liveness annotations. As during dead binding elimination, we first do an analysis pass and later simply look at the annotated thinnings to find out where a declaration is used.

The structure of the implementation is the same as for the direct approach, so we will only highlight a few differences.

Type signature Similarly to section 3.4.2, we first perform the analysis and then a transformation that results in a plain Expr again. Combined, this has the same signature as the direct version.

```
transform :
  {θ : Δ ⊑ (Γ₁ ++ σ :: Γ₂)} →
  Expr σ ⇑ (Γ₁ ++ Γ₂) →
  LiveExpr τ θ →
  Expr τ (Γ₁ ++ Γ₂)


sink-let : Expr σ (Γ₁ ++ Γ₂) → Expr τ (Γ₁ ++ σ :: Γ₂) → Expr τ (Γ₁ ++ Γ₂)
sink-let decl e = let _ , θ , le = analyse e in transform (decl ↑ oi) le
```

Note that only the body is annotated, as we do not need liveness information for the declaration. The declaration however is passed with a thinning. This change is independent of the others, but will avoid repeatedly having to rename the declaration when going under binders.

Binary constructors The Let case shows all major changes. The main one is that instead of traversing the subexpressions (attempting to strengthen them), it is sufficient to work with the thinnings found during analysis. We make use of the ability to split and concatenate them, as shown in section 3.2.

```
transform {Γ₁ = Γ₁} decl e@ (Let {θ₁ = θ} {θ₂ = ϕ} e₁ e₂)
  with Γ₁ ⊣ θ | (_ :: Γ₁) ⊣ ϕ
... | split θ₁ (o′ θ₂) (refl , refl) | split ϕ₁ (o′ ϕ₂) (refl , refl) =
  Let (DBE.transform e₁ (θ₁ ++⊑ θ₂)) (DBE.transform e₂ (ϕ₁ ++⊑ ϕ₂))
... | split θ₁ (os θ₂) (refl , refl) | split ϕ₁ (o′ ϕ₂) (refl , refl) =
  Let (transform decl e₁) (DBE.transform e₂ (ϕ₁ ++⊑ ϕ₂))
... | split θ₁ (o′ θ₂) (refl , refl) | split ϕ₁ (os ϕ₂) (refl , refl) =
  Let (DBE.transform e₁ (θ₁ ++⊑ θ₂)) (transform (thin⇑ (o′ oi) decl) e₂)
... | split θ₁ (os θ₂) (refl , refl) | split ϕ₁ (os ϕ₂) (refl , refl) =
  Let (rename-Expr⇑ decl) (rename-top (forget e))
...
```

Focusing on the first subexpression, we use $\_\dashv\_$ to split the annotated thinning $\theta : (\Delta_1 +\!\!+ \Delta_2) \sqsubseteq (\Gamma_1 +\!\!+ \sigma :: \Gamma_2)$ from the context of e_1 into two thinnings $\theta_1 : \Delta_1 \sqsubseteq \Gamma_1$ and $\theta_2 : \Delta_2 \sqsubseteq \sigma :: \Gamma_2$. If the declaration is unused, we obtain a smaller θ_2, which we can use to construct $\theta_1 +\!\!+_\sqsubseteq \theta_2 : (\Delta_1 +\!\!+ \Delta_2) \sqsubseteq (\Gamma_1 +\!\!+ \Gamma_2)$, which shows that σ is not required in the context of e_1. To then turn the annotated e_1 into an Expr σ $(\Gamma_1 +\!\!+ \Gamma_2)$, we could forget the annotations followed by renaming, but we instead use the already defined DBE.transform, which does the job in a single traversal.

Binders Instead of weakening the declaration every time we go under a binder, we manipulate the thinning it is wrapped in thin⇑ (o′ oi). As a result, we only need to rename the declaration once at the end, when the binding is created.

```
rename-Expr⇑ : Expr σ ⇑ Γ → Expr σ Γ
rename-Expr⇑ (e ↑ θ) = rename-Expr θ e
```

3.6 Discussion

We used thinnings to implement live variable analysis and two program transformations. In both cases, the approach of directly performing the transformation on de Bruijn syntax required us to traverse a number of syntax nodes roughly quadratic in the size of the tree. At the cost of a single analysis pass upfront (and some additional code), we were able to replace the redundant traversals with simple operations on thinnings.

Reordering the context When changing the order of let-bindings during let-sinking, the order of the variables in the context changes as well. As thinnings present *order-preserving* embeddings, they are not suited to describe such a change of context. Consequently, we had to resort to concatenation and define an additional set of operations, such as for renaming expressions. The complexity of the transformation was significantly higher than for dead binding elimination.

We will continue using the order-preserving flavour of thinnings, but discuss potential alternatives in chapter 6.

3.6.1 Alternative Designs

Iterating transformations As discussed in section 2.1, more than one pass of dead binding elimination might be necessary to remove all unused bindings. While in our simple setting all these bindings could be identified in a single pass using strongly live variable analysis, in general it can be beneficial to iterate optimisations a fixed number of times or until a fixpoint is reached. For example, it is reported that GHC's simplifier pass is iterated up to 4 times [20].

As an example, we defined a function that keeps applying dbe as long as the number of bindings in the expression decreases. 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 clearly can never be negative. This is an instance of to the ascending chain condition in program analysis literature [15]. To convince the termination checker, we used the technique of well-founded recursion [4] on the number of bindings. The correctness was then straightforward to prove, as it follows directly from the correctness of each individual iteration step.

Signature of let-sinking We remind ourselves of the type signature of sink-let. To talk about removing an element from the context at a specific position, we used list concatenation.

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

Note that we could alternatively have used other ways to achieve the same, such as insertion at a position `n : Fin (length Γ)` or removal of σ at a position `i : Ref σ Γ`.

```
pop-at : (Γ : Ctx) → Ref σ Γ → Ctx
pop-at (σ :: Γ) Top = Γ
pop-at (τ :: Γ) (Pop i) = τ :: pop-at Γ i

Expr σ (Γ₁ ++ Γ₂) → Expr τ (Γ₁ ++ σ :: Γ₂) → Expr τ (Γ₁ ++ Γ₂)
Expr σ (pop-at Γ i) → Expr τ Γ → Expr τ (pop-at Γ i)
Expr σ Γ → Expr τ (insert n σ Γ) → Expr τ Γ
```

Using list concatenation, however, seems more principled and allows us to make use of general operations and properties of the concatenation of contexts and thinnings.

Keeping annotations In both transformations, we used annotated expressions for the input, but returned the result without annotations. When performing multiple different transformations in sequence (or the same one multiple times), each pass requires us to do live variable analysis anew, just to then throw away the results.

If instead transformations computed updated liveness annotations as they are constructing the resulting expression, we could stay in LiveExpr world all the time. However, each transformation would then effectively need to include analysis, making it more complex. This could potentially be factored out, but a first attempt for let-sinking encountered various practical issues. In addition, indexing LiveExpr by *two* different contexts seems redundant. Could a representation considering only the live variables be simpler, while providing the same benefits? The next chapter will feature such a representation.

Chapter 4

Co-de-Bruijn Representation

After showing that de Bruijn representation can be made type- and scope-correct by indexing expressions with their context (the variables in scope), we found out how useful it is to also know the live variables. The type of annotated expressions we created was therefore indexed (perhaps redundantly) by both of these, as well as the thinning between them. Here however, we will work with McBride's co-de-Bruijn syntax [14], another nameless intrinsically typed representation, which is indexed by its weakly live variables alone.

In this representation, bindings can be added or removed without having to traverse their body to rename the variables. The bookkeeping required is relatively complex, but Agda's typechecker helps us maintain the invariants.

We will begin by giving an intuition for the co-de-Bruijn representation and show how it translates into a few core building blocks, each with a convenient smart constructor. Based on these, we define a co-de-Bruijn-based syntax tree for our expression language and demonstrate that it can be converted to and from our original de Bruijn expressions. Once the foundations are in place, we again perform dead binding elimination and let-sinking, making use of the variable liveness information inherent in co-de-Bruijn terms.

4.1 Intrinsically Typed Syntax

Intuition The intuition that McBride gives (and uses to motivate the name *co-de-Bruijn*) is that de Bruijn representation keeps all introduced bindings in its context and only selects one of them *as late as possible*, when encountering a variable, i.e. de Bruijn index. Co-de-Bruijn representation follows the dual approach, shrinking down the context *as early as possible* to only those variables that occur in the respective subexpression. When reaching a variable, only a singleton context remains, so there is no need for an index.

After dealing with live variable analysis in the previous chapter, we also think about it in a way similar to liveness annotations: starting from the variables, the live variables get collected and turned into annotations, bottom-up.

Relevant pairs The most insightful situation to consider is that of handling multiple subexpressions, for example with addition. Assuming we have e_1 : Expr NAT Δ_1 and e_2 : Expr NAT Δ_2, each indexed by their live variables, how do we construct the syntax node representing $e_1 + e_2$? It should be indexed by the smallest Γ with thinnings $\theta_1 : \Delta_1 \sqsubseteq \Gamma$ and $\theta_2 : \Delta_2 \sqsubseteq \Gamma$. For LiveExpr, we specified the resulting context using $\_\cup\_$, ensuring that it is the smallest context into which we can embed both Δ_1 and Δ_2. Here, we achieve the same using a *cover* of the thinnings to ensure that every element of Γ is part of Δ_1, Δ_2, or both ("everybody's got to be somewhere"). Fundamentally, we can never construct a Cover (o′ _) (o′ _).

```
data Cover : Δ₁ ⊑ Γ → Δ₂ ⊑ 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
```

As each binary operator will in some form contain two thinnings and their cover, we combine them into a reusable datatype called *relevant pair*.

```
record _×ᵣ_ (S T : List I → Set) (Γ : List I) : Set where
  constructor pairᵣ
  field
    outl  : S ⇑ Γ   -- containing S Δ₁ and Δ₁ ⊑ Γ
    outr  : T ⇑ Γ   -- containing T Δ₂ and Δ₂ ⊑ Γ
    cover : Cover (thinning outl) (thinning outr)
```

As an example, let us construct the relevant pair of the two expressions e_1 : Expr NAT $(\sigma :: [])$ and e_2 : Expr NAT $(\tau :: [])$, each with a (different) single live variable in their context. The combined live variables then contain both variables, so one thinning will target the first element, and one the other: pair_R $(\mathsf{e}_1 \uparrow \mathsf{os}\ (\mathsf{o}'\ \mathsf{oz}))$ $(\mathsf{e}_2 \uparrow \mathsf{o}'\ (\mathsf{os}\ \mathsf{oz}))$ c : (Expr NAT $\times_R$ Expr NAT) $(\sigma :: \tau :: [])$. The cover $\mathsf{c} = \mathsf{cs}'\ (\mathsf{c's}\ \mathsf{czz})$ follows the same structure.

Manually finding the combined live variables and constructing the cover everytime a relevant pair gets constructed quickly gets cumbersome. We can delegate the work to a smart constructor, but note that nothing about e_1 and e_2 tells us which element should come first in the context – the choice was made (arbitrarily) by creating the thinnings. As part of the input, we therefore require thinnings into a shared context. Any shared context will do, since we only need it to relate the two subexpressions' contexts and can still shrink it down to the part that is live.

```
_,ᵣ_ : S ⇑ Γ → T ⇑ Γ → (S ×ᵣ T) ⇑ Γ
```

We will not show the implementation here, but it is generally similar to that of $\_\cup\_$, recursing over each element of Γ to check which of the thinnings use it, decide whether to include it in the resulting context, and construct the matching thinnings and cover.

Bindings Another important consideration are bindings. Not all bound variables are required to be used, they can be dropped from the context of their subexpressions immediately. For example, λ-abstractions that don't use their argument are perfectly valid and cannot be removed as easily as dead let-bindings.

With the goal of creating another general building block that can be used for a wide range of language constructs, we allow a list of multiple simultaneous bindings. Instead of an operation like pop dealing with a single binding, we now use a thinning $\phi : \Delta' \sqsubseteq \Gamma'$ to express which of the bound variables Γ' are used, and concatenate the live variables Δ' to the context.

```
record _⊢_ (Γ′ : List I) (T : List I → Set) (Γ : List I) : Set where
  constructor _⨄_
  field
    {Δ′}     : List I
    thinning : Δ′ ⊑ Γ′
    thing    : T (Δ′ ++ Γ)
```

Given an expression, wrapping it into this datatype requires us to find out which part of its context is bound here. Luckily, with the right thinning at hand, this can be handled by a general smart constructor.

```
_⨄ᴿ_ : (Γ′ : List I) → T ⇑ (Γ′ ++ Γ) → (Γ′ ⊢ T) ⇑ Γ
```

Again, we will not spend much time explaining the implementation, but briefly mention that it relies on the ability to split the thinning that goes into $\Gamma' +\!\!+ \Gamma$ into two parts using $\_\dashv\_$, as seen in section 3.2.

Expression language Using the building blocks defined above, our expression language can be defined pretty concisely.

```
data Expr : (σ : U) (Γ : 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
  Val  : ⟦ σ ⟧ → Expr σ []
  Plus : (Expr NAT ×ᴿ Expr NAT) Γ → Expr NAT Γ
```

Let-bindings make use of both a relevant pair and binding, without us having to think much about what the thinnings involved should look like. Since the context of the declaration is considered relevant irrespective of the let-binding itself being live, it corresponds to the *weakly* live variables. Achieving *strong* variable liveness would require a custom combinator, but more importantly, we will show that it is not necessary for an efficient implementation of the strong version of dead binding elimination.

Evaluation To later be able to talk about preservation of semantics, we first need to define a semantics, which we again do in form of a total evaluation function. As with evalLive, we allow supplying an environment that is larger than strictly needed. This allows us to compose a thinning instead of having to project the environment for each recursive call.

```
eval : Expr τ Δ → Δ ⊑ Γ → Env Γ → ⟦ τ ⟧
eval Var θ env =
  lookup (ref-o θ) env
eval (App (pair_R (e₁ ↑ θ₁) (e₂ ↑ θ₂) cover)) θ env =
  eval e₁ (θ₁ ⨟ θ) env
    (eval e₂ (θ₂ ⨟ θ) env)
eval (Lam (ψ ⧵ e)) θ env =
  λ v → eval e (ψ +⊑ θ) (Cons v env)
eval (Let (pair_R (e₁ ↑ θ₁) ((ψ ⧵ e₂) ↑ θ₂) c)) θ env =
  eval e₂ (ψ +⊑ (θ₂ ⨟ θ))
    (Cons (eval e₁ (θ₁ ⨟ θ) env) env)
...
```

At the variable occurrences, the expression's context is a singleton and we can convert the thinning into an index (Ref) to do a lookup on the environment. The thinning ψ for bindings that get introduced needs to concatenated with the accumulated binding. Finally note that despite all the similarities to evalLive, we do not skip the declaration's evaluation when encountering a dead let-binding.

4.2 Conversion from/to de Bruijn Syntax

The conversion between de Bruijn and co-de-Bruijn representation is very similar to computing and forgetting annotations in the second part of section 3.4.

To de Bruijn Since the converted expression often needs to be placed in a larger context than just its live variables, we allow to specify the desired context using a thinning.

```
toDeBruijn : Δ ⊑ Γ → CoDeBruijn.Expr σ Δ → DeBruijn.Expr σ Γ
```

The recursive calls work the exact same way as during evaluation, composing the thinning on the way down to turn it into into a de Bruijn index when reaching a variable. The only difference is that instead of computing a value, the resulting expressions are just packaged up into a syntax tree again.

From de Bruijn The other direction is slightly more work, as it effectively needs to perform live variable analysis. Luckily, we benefit greatly from the smart constructors. As we do not know the context of the resulting co-de-Bruijn expression upfront, we once more return the result with a thinning.

```
fromDeBruijn : DeBruijn.Expr σ Γ → CoDeBruijn.Expr σ ⇑ Γ
fromDeBruijn (Var x) =
  Var ↑ o-Ref x
fromDeBruijn (App e₁ e₂) =
  map⇑ App (fromDeBruijn e₁ ,R fromDeBruijn e₂)
fromDeBruijn (Lam e) =
  map⇑ Lam (_ \\R fromDeBruijn e)
fromDeBruijn (Let e₁ e₂) =
  map⇑ Let (fromDeBruijn e₁ ,R (_ \\R fromDeBruijn e₂))
fromDeBruijn (Val v) =
  Val v ↑ oe
fromDeBruijn (Plus e₁ e₂) =
  map⇑ Plus (fromDeBruijn e₁ ,R fromDeBruijn e₂)
```

After using the smart constructors to obtain a relevant pair or binder (with a thinning), it only remains to wrap things up using the right constructor.

Correctness While the conversion is pretty straightforward, mapping constructors one-to-one to their counterparts, we can prove that the semantics of the two representations agree.

```
toDeBruijn-correct :
  (e : CoDeBruijn.Expr τ Δ) (env : Env Γ) (θ : Δ ⊑ Γ) →
  let e′ = toDeBruijn θ e
  in DeBruijn.eval e′ env ≡ CoDeBruijn.eval e θ env


fromDeBruijn-correct :
  (e : DeBruijn.Expr τ Γ) (env : Env Γ) →
  let e′ ↑ θ = fromDeBruijn e
  in CoDeBruijn.eval e′ θ env ≡ DeBruijn.eval e env
```

The first proof works by a completely straightforward structural induction, without requiring any further lemmas. The other direction is slightly more interesting: As the smart constructors are defined using **with**-abstraction, the case for each constructor first requires us to mirror that structure before being able to use the induction hypothesis.

4.3 Dead Binding Elimination

Co-de-Bruijn expressions enforce that every variable in their context must occur somewhere. However, there can still be dead let-bindings: the declaration of type σ bound by $\psi \setminus\!\setminus \mathsf{e}_2 : ((\sigma :: []) \vdash \mathsf{Expr}\ \tau)\ \Gamma$ can be immediately discarded in ψ, never to occur in e_2. We need to identify such dead let-bindings and eliminate them.

Since an expression's context contains its *weakly* live variables and removing dead bindings can make some of them dead, we return the result in a (generally) smaller context with a thinning.

$$\mathsf{dbe} : \mathsf{Expr}\ \tau\ \Gamma \to \mathsf{Expr}\ \tau \Uparrow \Gamma$$

Transformation The weakly live variables are already present as part of the co-de-Bruijn representation, so no further analysis is necessary. For the weak version of dead binding elimination, we simply need to find all let-bindings in the input expression that have a thinning $\mathsf{o}'\ \mathsf{oz} : [] \sqsubseteq (\sigma :: [])$.

The change in context caused by the transformation has several consequences: Firstly, these new thinnings coming from each recursive call need to be composed with the existing ones on the way up (e.g. using $\mathsf{thin}\Uparrow$). Secondly, we need to rebuild the variable usage information, i.e. calculate new contexts and covers at each node using the smart constructors $\_ \setminus\!\setminus_R \_$ and $\_ ,_R \_$.

$$\begin{aligned}
&\mathsf{dbe} : \mathsf{Expr}\ \tau\ \Gamma \to \mathsf{Expr}\ \tau \Uparrow \Gamma \\
&\mathsf{dbe}\ \mathsf{Var} = \\
&\quad \mathsf{Var} \uparrow \mathsf{oi} \\
&\mathsf{dbe}\ (\mathsf{App}\ (\mathsf{pair}_R\ (\mathsf{e}_1 \uparrow \phi_1)\ (\mathsf{e}_2 \uparrow \phi_2)\ \mathsf{c})) = \\
&\quad \mathsf{map}\Uparrow\ \mathsf{App}\ (\mathsf{thin}\Uparrow\ \phi_1\ (\mathsf{dbe}\ \mathsf{e}_1)\ ,_R\ \mathsf{thin}\Uparrow\ \phi_2\ (\mathsf{dbe}\ \mathsf{e}_2)) \\
&\mathsf{dbe}\ (\mathsf{Lam}\ (\psi \setminus\!\setminus \mathsf{e})) = \\
&\quad \mathsf{map}\Uparrow\ (\mathsf{Lam} \circ \mathsf{thin}\vdash\ \psi)\ (_ \setminus\!\setminus_R\ \mathsf{dbe}\ \mathsf{e}) \\
&\mathsf{dbe}\ (\mathsf{Let}\ (\mathsf{pair}_R\ (\mathsf{e}_1 \uparrow \phi_1)\ ((\mathsf{o}'\ \mathsf{oz} \setminus\!\setminus \mathsf{e}_2) \uparrow \phi_2)\ \mathsf{c})) = \\
&\quad \mathsf{thin}\Uparrow\ \phi_2\ (\mathsf{dbe}\ \mathsf{e}_2) \\
&\mathsf{dbe}\ (\mathsf{Let}\ (\mathsf{pair}_R\ (\mathsf{e}_1 \uparrow \phi_1)\ ((\mathsf{os}\ \mathsf{oz} \setminus\!\setminus \mathsf{e}_2) \uparrow \phi_2)\ \mathsf{c})) = \\
&\quad \mathsf{map}\Uparrow\ \mathsf{Let}\ (\mathsf{thin}\Uparrow\ \phi_1\ (\mathsf{dbe}\ \mathsf{e}_1)\ ,_R\ \mathsf{thin}\Uparrow\ \phi_2\ (_ \setminus\!\setminus_R\ \mathsf{dbe}\ \mathsf{e}_2)) \\
&\mathsf{dbe}\ (\mathsf{Val}\ \mathsf{v}) = \\
&\quad \mathsf{Val}\ \mathsf{v} \uparrow \mathsf{oz} \\
&\mathsf{dbe}\ (\mathsf{Plus}\ (\mathsf{pair}_R\ (\mathsf{e}_1 \uparrow \phi_1)\ (\mathsf{e}_2 \uparrow \phi_2)\ \mathsf{c})) = \\
&\quad \mathsf{map}\Uparrow\ \mathsf{Plus}\ (\mathsf{thin}\Uparrow\ \phi_1\ (\mathsf{dbe}\ \mathsf{e}_1)\ ,_R\ \mathsf{thin}\Uparrow\ \phi_2\ (\mathsf{dbe}\ \mathsf{e}_2)) \\
\\
&\mathsf{thin}\vdash : \Gamma_1 \sqsubseteq \Gamma_2 \to (\Gamma_1 \vdash \mathsf{T})\ \Gamma \to (\Gamma_2 \vdash \mathsf{T})\ \Gamma \\
&\mathsf{thin}\vdash\ \phi\ (\theta \setminus\!\setminus \mathsf{t}) = (\theta \fatsemi \phi) \setminus\!\setminus \mathsf{t}
\end{aligned}$$

To get the strong version, we can do the recursive calls first and check the thinnings *afterwards*. For that we use a small helper function $\mathsf{Let?}$, which behaves like the constructor Let if the binding is live, but otherwise removes the declaration. The other cases are the same as in the previous section.

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

dbe (Let (pairR (e1 ↑ φ1) ((ψ \\ e2) ↑ φ2) c)) =
  bind⇑ Let?
    (thin⇑ φ1 (dbe e1) ,R thin⇑ φ2 (map⇑ (thin⊢ ψ) (_ \\R dbe e2)))
```

Due to the combinators applying and mapping over thinnings, the definition is concise, but opaque. To give a better feeling for how much plumbing is involved, we can also inline all combinators and compose the thinnings manually:

```
dbe (Let (pairR (e1 ↑ φ1) ((ψ \\ e2) ↑ φ2) c)) =
  let e1′          ↑ φ1′ = dbe e1
      (ψ′ \\ e2′) ↑ φ2′ = _ \\R dbe e2
      p′ ↑ θ = (e1′ ↑ (φ1′ ⨟ φ1)) ,R (((ψ′ ⨟ ψ) \\ e2′) ↑ (φ2′ ⨟ φ2))
      e′ ↑ θ′ = Let? p′
  in
    e′ ↑ (θ′ ⨟ θ)
```

Additionally inlining the smart constructors to show how they construct their thinnings would make the code even more noisy and difficult to follow.

Correctness As seen in the inlined version, a lot of nontrivial operations are involved in constructing the thinnings and covers in the result. This also makes the proofs more complicated. Often, conceptually simple parts of the proof require extensive massaging of deeply buried thinnings using commutative laws or helpers with types like $\theta_1' \fatsemi \theta_1 \equiv \theta_2' \fatsemi \theta_2 \to \theta_1' \fatsemi (\theta_1 \fatsemi \phi) \equiv \theta_2' \fatsemi (\theta_2 \fatsemi \phi)$.

```
dbe-correct :
  (e : Expr τ Δ) (env : Env Γ) (θ : Δ ⊑ Γ) →
  let e′ ↑ θ′ = dbe e
  in eval e′ (θ′ ⨟ θ) env ≡ eval e θ env
```

While the combinators and smart constructors used in the implementation compose seamlessly, the same is not quite true for proofs about them. We were able to factor out lemmas dealing with $\_\vdash\_$ and $\_\times_R\_$, such that e.g. the cases for application and addition in dbe-correct are both instances of the same proof, using either _$_ or _+_ as an argument. However, defining these building blocks in a way that allowed composing them to handle Let proved more difficult than expected. In the end, we simply handled Let as a similar but separate special case. The main issue for proof composability seems to be that some type equalities need to be matched on (e.g. using **with**-abstraction) just for the required call to the lemma or induction hypothesis to typecheck. Some attempts to break the proof into components also failed to satisfy the termination checker.

For let-bindings in the strong version, we additionally made use of a lemma stating that Let? p and Let p are semantically equivalent.

4.4 Let-sinking

In addition to the expected benefits of readily available variable liveness information, we will see that co-de-Bruijn representation also affords us more precision when specifying the inputs to the transformation. On the flipside, the reordering of the context inherent to let-sinking causes even more complications than in the de Bruijn version.

Type signature We again start from the observation that the signature for let-sinking should be similar to the Let constructor, but allowing for a prefix in context that we need when going under binders. But here, declaration and binding form a relevant pair, each in their own context with a thinning into the overall context. To make it clear what this type consists of, we flatten the structure:

$$
\begin{aligned}
&\mathsf{sink\text{-}let}\ : \\
&\quad (\mathsf{decl}\ :\ \mathsf{Expr}\ \sigma\ \Uparrow\ (\Gamma_1\ +\!\!+\ \Gamma_2))\ \to \\
&\quad (\mathsf{body}\ :\ \mathsf{Expr}\ \tau\ \Uparrow\ (\Gamma_1\ +\!\!+\ \sigma :: \Gamma_2))\ \to \\
&\quad \mathsf{Cover}\ (\mathsf{thinning\ decl})\ (\mathsf{thinning\ body})\ \to \\
&\quad \mathsf{Expr}\ \tau\ (\Gamma_1\ +\!\!+\ \Gamma_2)
\end{aligned}
$$

For now, we will ignore the cover and return the result with a thinning (i.e. without having to show that the whole context $\Gamma_1\ +\!\!+\ \Gamma_2$ is relevant). As we will see later, this avoids complicated reasoning about covers.

$$
\begin{aligned}
&\mathsf{sink\text{-}let}\ : \\
&\quad \mathsf{Expr}\ \sigma\ \Uparrow\ (\Gamma_1\ +\!\!+\ \Gamma_2)\ \to \\
&\quad \mathsf{Expr}\ \tau\ \Uparrow\ (\Gamma_1\ +\!\!+\ \sigma :: \Gamma_2)\ \to \\
&\quad \mathsf{Expr}\ \tau\ \Uparrow\ (\Gamma_1\ +\!\!+\ \Gamma_2)
\end{aligned}
$$

However, this type is imprecise in a different way: The context of the body is thinned into a precisely specified overall context, but its own structure is opaque. We know that it consists of two parts (thinned into Γ_1 and Γ_2 respectively), but that information first needs to be discovered. Furthermore, we want to require that the declaration is live in the body we want to move it into, so we know even more about the context. To make that structure more apparent, we can make stronger assumptions about the context of the body (not just the context it is thinned into). The structure of the overall context on the other hand is less important to us.

$$
\begin{aligned}
&\mathsf{sink\text{-}let}\ : \\
&\quad \mathsf{Expr}\ \sigma\ \Uparrow\ \Gamma\ \to \\
&\quad \mathsf{Expr}\ \tau\ (\Gamma_1\ +\!\!+\ \sigma :: \Gamma_2)\ \to \\
&\quad \Gamma_1\ +\!\!+\ \Gamma_2\ \sqsubseteq\ \Gamma\ \to \\
&\quad \mathsf{Expr}\ \tau\ \Uparrow\ \Gamma
\end{aligned}
$$

The knowledge that the binding is used eliminates some edge cases we previously had to deal with. Using a simple wrapper, we can still get back the less restrictive type signature that can be applied to any let-binding:

```
sink-let-top : (Expr σ ×_R ((σ :: []) ⊢ Expr τ)) Γ → Expr τ ⇑ Γ
sink-let-top (pair_R (decl ↑ φ) ((os oz \\ e) ↑ θ) c) =
  sink-let [] _ (decl ↑ φ) e refl θ
sink-let-top (pair_R decl ((o′ oz \\ e) ↑ θ) c) =
  e ↑ θ   -- binding is unused, why bother with let-sinking?
```

Variables We immediately observe this when dealing with variables. Since we know that a variable's context contains exactly one element, and also that the declaration is part of that of the context, the variable *must* refer to the declaration. After making this clear to Agda using a pattern match with an absurd case, we can replace the variable with the declaration.

Creating the binding As in the de Bruijn setting, we need to rename the body of the newly created binding. However, it becomes more cumbersome here:

```
reorder-Ctx :
  Expr τ Γ → (Γ ≡ Γ₁ ++ Γ₂ ++ Γ₃ ++ Γ₄) →
  Expr τ (Γ₁ ++ Γ₃ ++ Γ₂ ++ Γ₄)
```

The context is split into four segments (where Γ_3 is $(\sigma :: [])$) that get reordered, which means that we also need to split every thinning and cover into four parts and carefully reassemble them.

```
cover++⊑4 :
  (θ₁ : Γ′₁ ⊑ Γ₁) (θ₂ : Γ′₂ ⊑ Γ₂) … →
  Cover (θ₁ +⊑ θ₂ +⊑ θ₃ +⊑ θ₄) (φ₁ +⊑ φ₂ +⊑ φ₃ +⊑ φ₄) →
  Cover (θ₁ +⊑ θ₃ +⊑ θ₂ +⊑ θ₄) (φ₁ +⊑ φ₃ +⊑ φ₂ +⊑ φ₄)
```

The need for such an operation suggests that co-de-Bruijn representation, and in particular the notion of thinnings it is based on, might not be very well suited for the reordering of bindings.

Binary constructors Variable usage information is immediately available: We split and examine the thinnings of the subexpressions to see where the declaration is used. Using the cover, we can rule out the case where no subexpression uses the declaration. In contrast to the previous implementation of let-sinking, no strengthening is necessary: discovering that a variable is unused is enough. The subexpressions are then combined using $\_,_R\_$, creating new thinnings and a cover for us.

Binders No weakening of the declaration is necessary when going under a binder, as we simply update its thinning. But recursing into the body of another let-binding still complicates things: Although we know that the liveness of the bound variable should be unaffected by let-sinking, the imprecise type signature allows for changes in context, so we need to find out again whether the binding is used or not.

Correctness Work on the proof is incomplete, as the sheer number of operations and bindings involved makes it messy. There are many lemmas about splitting thinnings and reordering the context that are cumbersome to state and prove correct. It seems like some of the complications could be avoided if we managed to avoid the usage of $\_,_R\_$ as explained in section 4.5.1.

4.5 Discussion

Useful properties Co-de-Bruijn expressions generally seem promising for defining transformations, with several useful properties:

1. Liveness information is present at each syntax node.
2. Changing the context in a way expressible with thinnings does not require a traversal of the expression.
3. The type of an expression only depends on the expression itself, not on the (potentially unused) bindings around it. One situation where this can be useful is when identifying identical expressions in different parts of the expression, as is needed for common subexpression elimination.

Complications On the other hand, the elaborate bookkeeping that is part of co-de-Bruijn syntax trees makes the construction of expressions more complicated. While this complexity can be hidden behind smart constructors, it leaks easily. For example, the proofs about transformed expressions are significantly more complex than their de Bruijn counterparts, which especially became apparent for let-sinking. It might be possible to create a general set of proof combinators that mirror the structure of the smart constructors, but this kind of abstraction over proofs is usually difficult (or impossible) to achieve.

4.5.1 Alternative Designs

Covers As hinted at when choosing a type signature for the let-sinking transformation in section 4.4, it should not be necessary to return the result with a thinning. If all variables occur in either declaration or body, they will still occur in the result, no matter where exactly the declaration is moved. Therefore, the context should always remain the same, just the thinnings and covers have to be rebuilt. This could potentially simplify parts of the implementation and especially the proof, since directly constructing a relevant pair creates fewer indirections than using $\_,_R\_$ to discover new thinnings.

It seems desirable to reflect this observation in the type signature, specifying the result more precisely. However, making it clear to the typechecker involves relatively complex manipulation of the covers.

The issue boils down to quite fundamental associative and commutative operations on relevant pairs: we want to be able to turn any $(\mathsf{S} \times_R (\mathsf{T} \times_R \mathsf{U}))\ \Gamma$ into $((\mathsf{S} \times_R \mathsf{T}) \times_R \mathsf{U})\ \Gamma$ or also $(\mathsf{T} \times_R (\mathsf{S} \times_R \mathsf{U}))\ \Gamma$ etc., and intuitively it's clear that the live variables Γ are unaffected by these operations.

Chapter 5

Syntax-generic Co-de-Bruijn Representation

So far, we worked with specialised types for the syntax trees of the language we defined. Modifying the language or defining a new one would require us to also modify the implementation of each transformation. However, the core of the transformation would likely remain unchanged: dead binding elimination for example only needs to know where variables are bound and occur in the expression, and then exclusively modifies let-bindings. All other parts of the syntax tree get traversed in a highly uniform way.

This problem is addressed by Allais et al. [1] with the concept of syntax-generic programming, although based on a de Bruijn representation. The main idea is to:

1. define a datatype of syntax descriptions Desc
2. describe a family of terms Tm d for each (d : Desc I)
3. implement operations *once*, generically over descriptions
4. describe your language using Desc to get access to the generic operations

To define the syntax-generic co-de-Bruijn terms, we build on top of the `generic-syntax` [1] Agda package, which is an artefact of the abovementioned paper. It failed to compile with recent versions of Agda, mainly due to issues with sized types, which were used to show termination. Therefore, we trimmed the package down to the parts interesting to us and removed the size information from all types. The paper still serves as a great introduction to the topic, but we will start with a short overview of the main constructions we use.

[1] `https://github.com/gallais/generic-syntax`

5.1 Descriptions of Syntax

At the core of this chapter is the the type of syntax descriptions, taken verbatim from Allais et al.

```
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
```

The argument I represents the type associated with each expression and variable brought into scope, typically their sort. Variable occurrences do not need to be modeled in the description, as they are part of any language implicitly.

The constructor ‘σ is then used to store data of some type A. Since the remaining description can then depend on the value of the data, it can be used as a tag deciding which constructor of the syntax tree is present. ‘X can be used for recursion (i.e. subexpressions) with a list of variables that come into scope and specified sort. After building a product-like structure (including sums by using the dependent product ‘σ), the descriptions are terminated with ‘■, stating their sort.

Example Let us give a description of our expression language to get a feeling for syntax descriptions. We start by defining a type of tags for each type of syntax node (except variable occurrences, as noted above). Each tag also carries the sorts it will use.

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

Once we plug the type into ‘σ, we can give a description for each of the constructors. Those are typically a product of multiple subexpressions. While the details can be hard to follow, some similarities with the original Expr type we defined should become apparent.

```
Lang : Desc U
Lang = ‘σ ‘Lang 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 ⟦ τ ⟧ λ _ → ‘■ τ
  ‘Plus      → ‘X [] NAT (‘X [] NAT (‘■ NAT))
```

5.2 Intrinsically Typed Syntax

The `generic-syntax` package only interprets syntax descriptions into de Bruijn terms. McBride shows a syntax-generic interpretation into co-de-Bruijn terms [14], but it is based on a different structure of syntax descriptions. Since we want to interpret a single description type into both types of terms, it is not directly reusable. However, the building blocks for bindings and relevant pairs still help us to create our own co-de-Bruijn interpretation.

We start be interpreting descriptions into flat (non-recursive) types representing a single syntax node, where the argument X marks the recursive occurrences and can later be used to form a fixpoint.

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


⟦_⟧ : 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 ≡ []
```

The interpretation of `$\sigma$ is exactly the same as for de Bruijn terms, storing a value of type A and (based on its value) continuing with the remaining description. The other two cases however need to enforce the invariants on the live variables $\Gamma$ by which the expressions are indexed: `X uses *relevant* pairs and `■ requires that the context starts out empty until something explicitly extends it.

Based on that, we can build the recursive type of terms. At each recursive occurrence, a new scope is introduced. While this could be done as $\Delta \vdash$ T i independent of the bound variables Δ, a single case distinction avoids a trivial wrapper with a thinning [] ⊑ [].

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


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

We also see that variables still always have a single live variable in their context.

Instantiation the terms We can obtain co-de-Bruijn terms of our expression language using the description we created.

```
Expr : U -Scoped
Expr = Tm Lang
```

These are isomorphic to the co-de-Bruijn syntax tree we defined manually. However, there are some practical differences coming from the way relevant products are used in the interpretation. At the end, each description is terminated by a ‘■ resulting in an expression indexed by an empty list of live variables, which means that even constructing a unary product ⟦ ‘X Δ i (‘■ j) ⟧ requires trivial thinnings and covers.

This is not an issue when working generically, but when constructing a term for a concrete description, it causes some boilerplate, even for a simple program such as foo = f 1.

```
  -- de Bruijn
foo :: Expr BOOL ((NAT ⇒ BOOL) :: [])
foo = App (Var Top) (Val 1)
  -- co-de-Bruijn
foo :: Expr BOOL ((NAT ⇒ BOOL) :: [])
foo = App (pair_R (os oz ↑ Var) (o′ oz ↑ Val 1) (cs' czz))
  -- syntax-generic co-de-Bruijn
foo : Expr BOOL ((NAT ⇒ BOOL) :: [])
foo =
  ‘con (‘App NAT BOOL ,
    pair_R
      (‘var ↑ os oz)
      (pair_R ((‘con ((‘Val NAT) , (1 , (refl , refl)))) ↑ oz)
              ((refl , refl) ↑ oz)
              czz
        ↑ o′ oz)
      (cs' czz))
```

Luckily, the boilerplate during construction of terms can be reduced using smart constructors (e.g. pattern synonyms App, Lam, ...) or a general helper of this shape:

```
×_R-trivial : {T : List I → Set} →
  T Γ → (T ×_R λ Δ → τ ≡ τ × Δ ≡ []) Γ
×_R-trivial t = pair_R (t ↑ oi) ((refl , refl) ↑ oe) cover-oi-oe
```

However, when deconstructing a term, it is not clear to the typechecker that the redundant thinnings must be identity and empty thinning (oi and oe) respectively, so we cannot just use pattern synonyms to hide them away. We first need to make the fact obvious in a **with**-abstraction calling a helper function:

```
×_R-trivial⁻¹ : {T : List I → Set} →
  (T ×_R λ Δ → τ ≡ τ′ × Δ ≡ []) Γ → T Γ × τ ≡ τ′
×_R-trivial⁻¹ (pair_R (t ↑ θ_1) ((refl , refl) ↑ θ_2) cover)
  with refl ← cover-oi-oe⁻¹ cover =
    t , refl
```

This affects any operation on our language by introducing additional **with**-abstractions. Evaluation or also converting into the concrete co-de-Bruijn representation, for example, should be absolutely trivial, but end up somewhat verbose.

5.3 Conversion from/to de Bruijn Syntax

The conversion between de Bruijn and co-de-Bruijn terms can be done generically for any description.

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

While the operations used in the implementation are generally the same as in the concrete setting, the structure is noticeably different. Instead of handling each constructor of the language, we use three mutually recursive functions to handle scopes, variables and constructors, respectively. We will see the same approach in more detail when doing dead binding elimination in the next section.

5.4 Dead Binding Elimination

Dead binding elimination can be written in a way that abstracts over most of the language, but it does not quite work for any description. The description at least needs to feature let-bindings, but how can we express this requirement in the type signature? While it would be useful to be able to directly plug in our description type together with some kind of witness of how it contains let-bindings, this comes with some complexity. Since this is mainly an issue of ergonomics and not directly relevant to our goal of performing transformations, we adopt a simpler solution from Allais et al.'s paper.

Type signature We make use of the fact that descriptions are closed under sums. This allows to describe constructors of a language separately and then combine them.

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


pattern inl t = true , t
pattern inr t = false , t
```

For our use case, we describe let-bindings and then define dead binding elimination for any description that has explicitly been extended with them.

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


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

The observations made about the return type in the concrete setting still apply, so the result again comes with a thinning.

Transformation The implementation is mostly similar to the concrete version, but we split it into three mutually recursive functions, each handling a different "layer" of the term datatype.

```
dbe :
  Tm (d '+ 'Let) σ Γ →
  Tm (d '+ 'Let) σ ⇑ Γ
dbe-⟦·⟧ :
  (d' : Desc I) →
  ⟦ 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
```

The last two of these functions traverse a single syntax node, apply dbe to each subexpression and combine the resulting live variables using the co-de-Bruijn smart constructors.

```
dbe-⟦·⟧ ('σ A d) (a , t) =
  map⇑ (a , _) (dbe-⟦·⟧ (d a) t)
dbe-⟦·⟧ ('X Δ j d) (pair_R (t1 ↑ θ1) (t2 ↑ θ2) c) =
  thin⇑ θ1 (dbe-Scope Δ t1) ,_R thin⇑ θ2 (dbe-⟦·⟧ d t2)
dbe-⟦·⟧ ('■ i) t =
  t ↑ oi
dbe-Scope [] t = dbe t
dbe-Scope (_ :: _) (ψ \\ t) = map⇑ (map⊢ ψ) (_ \\_R dbe t)
```

The implementation of dbe itself is split into a case for variables, a case for the description d and finally a case for let-bindings, where most of the work happens. We would like to write the following:

```
dbe 'var = 'var ↑ oi
dbe ('con (inl t)) = map⇑ ('con ∘ inl) (dbe-⟦·⟧ _ t)
dbe ('con (inr t@(a , pair_R (t1 ↑ θ1) (p ↑ θ2) _)))
  with ×_R-trivial⁻¹ p
```

```
... | (o′ oz \\ t2) , refl =
  thin⇑ θ₂ (dbe t2)
... | (os oz \\ t2) , refl =
  map⇑ (`con ∘ inr) (dbe-⟦·⟧ `Let t)
```

However, this definition fails Agda's termination checker. This can for example be solved by manually inlining the call to dbe-⟦·⟧ `Let in the last line, resulting in a more verbose implementation.

For the strong version, we again introduce a helper function Let? and now everything works nicely: since it only checks for dead bindings afterwards, the recursive call happens directly on the subexpression from the input, which is clearly structurally smaller. Therefore, the following definition is accepted by the termination checker:

```
Let? : ⟦ `Let ⟧ (Scope (Tm (d `+ `Let))) τ Γ → Tm (d `+ `Let) τ ⇑ Γ
Let? t@(a , pair_R (t1 ↑ θ₁) (p ↑ θ₂) _)
  with ×_R-trivial⁻¹ p
... | (o′ oz \\ t2) , refl = t2 ↑ θ₂
... | (os oz \\ t2) , refl = `con (inr t) ↑ oi
dbe `var          = `var ↑ oi
dbe (`con (inl t)) = map⇑ (`con ∘ inl) (dbe-⟦·⟧ _ t)
dbe (`con (inr t)) = bind⇑ Let? (dbe-⟦·⟧ `Let t)
```

5.5 Discussion

Given the generality of the dead binding elimination presented in this chapter, its implementation turned out concise and relatively readable. However, the syntax-generic representation adds considerable complexity, be it when describing a language's syntax through the Desc type, defining mutually recursive functions to operate on terms, or dealing with the trivial relevant pairs that get introduced by the interpretation into co-de-Bruijn syntax. Furthermore, there remain some questions that would need to be answered to migrate the remaining work from chapter 4 to the syntax-generic setting.

Let-sinking Making the concrete version of let-sinking syntax-generic is not quite as straightforward as for dead binding elimination, since some rules for when to stop let-sinking depend on specific language constructs. The rule of not duplicating bindings can be implemented generically, but the constraint on not moving a let-binding into λ-abstractions is more ad-hoc. The presence of bound variables is not enough to make a conclusion: we *do* want to sink into other let-bindings and we can imagine a number of other language constructs for which it is not immediately obvious whether it is beneficial to sink through them or not. This suggests the need for a mechanism by which the desired behaviour can be specified as an additional input to the let-sinking transformation.

Correctness We did not attempt to prove preservation of semantics for syntax-generic transformations, but note that it raises the question of which semantics to use for that. We can certainly prove that the syntax-generic dead binding elimination behaves correctly for our example language with its semantics, but we do not have a semantics for any arbitrary description at hand.

Ideally, it should be possible to do the proof for *any* semantics with some property, like being defined as a fold of the syntax tree. However, the exact constraints are not obvious. A good candidate is the notion of Semantics as defined by Allais et al. [1], which indeed is based on a fold (catamorphism with an algebra) with the constraint that the values it operates on are *thinnable*.

Unfortunately, "applying" a Semantics to co-de-Bruijn terms becomes more complicated than it is for de Bruin terms, as the relationship between the context of an expression and the context of its subexpressions is more involved. It seems worthwhile to attempt finding a solution to this issue, which then might make it possible to prove that a transformation preserves any Semantics for any syntax description.

Chapter 6

Discussion

Summary As demonstrated in this thesis, binding-related program transformations can be performed on intrinsically typed syntax trees. To avoid redundant traversals, it is beneficial to compute variable liveness information upfront, e.g. providing it as part of the syntax tree.

Thinnings proved to be useful for reasoning about variable liveness of expressions and, in simple cases, manipulating their context. This is also witnessed by the connection between weakly live variable analysis and the co-de-Bruijn representation, which is itself built around thinnings.

Finally, it seems promising to define binding-related transformations syntax-generically, as they often handle most language constructs in a uniform way. This was showcased by performing dead binding elimination generically on co-de-Bruijn expressions of any language containing let-bindings.

Reordering bindings While thinnings worked well for dead binding elimination, the let-sinking transformation suffered from their order-preserving nature, requiring another mechanism to describe the reordering of bindings.

One could attempt to address this issue by extending thinnings to allow for permutations, but this might just move the complexity elsewhere, as the operations on thinnings then become more difficult to define. For example, it is not obvious how to perform the liveness union operation $\_ \cup \_$ without a way of ensuring that the reorderings in both its arguments agree with each other.

Another approach that is also used by Allais et al. is to define thinnings as a function $(\forall\ \sigma\ \to\ \mathsf{Ref}\ \sigma\ \Delta\ \to\ \mathsf{Ref}\ \sigma\ \Gamma)$ on references. This has some advantages, as it naturally supports reordering and inherits associativity and identity laws by virtue of being a function. However, the representation as a function is opaque and not even guaranteed to be injective (so the source context could be larger than the target context). This makes it difficult to talk about equality and define operations on thinnings.

6.1 Further Work

In addition to the challenges around reordering bindings, there are several other open ends that we could not address due to lack of time. For example, attempting a proof of correctness for the de Bruijn version of let-sinking might be simpler than for the co-de-Bruijn variant we failed to complete in section 4.4. However, it might be wiser to first improve the implementation (regarding reordering bindings) before trying to prove it correct.

The syntax-generic co-de-Bruijn approach could also benefit from further exploration. As noted in section 5.5, syntax-generic let-sinking comes with some interesting questions. Another large topic we only discussed briefly is that of correctness, which first requires a suitable notion of semantics. The most promising avenue for that is to extend the work by Allais et al. [1] not just with basic support for co-de-Bruijn terms, but also for notions such as Semantics.

There are several techniques that are related to aspects of the work shown here. For example, we saw several distinct definitions of syntax trees, each with a different amount of extra information: raw expressions, intrinsically typed expressions with some invariants, and annotated expressions that also contain the results of program analysis. However, the language they describe is fundamentally the same. Making this relationship explicit using *ornamentations* [9] could replace ad-hoc definitions like forget. Similarly, many program analyses can be studied through the common framework of *coeffects* [17].

Finally, the existing work can be extended with additional language constructs and transformations, each posing new challenges.

6.1.1 Extending the Language

Recursive bindings In a recursive let-binding, the bound variable is available in its own declaration. While this only requires a small change in the definition of the syntax tree, evaluation can now diverge. The treatment of semantics requires significant changes to account for this partiality [4, 5, 13, 10]. Program analysis of recursive functions poses additional challenges [15] and transformations are affected as well: In the presence of effects such as non-termination, removing or reordering let-bindings is only semantics-preserving if their declarations can be shown to be pure (i.e. terminating).

Mutually recursive binding groups Since mutual recursion allows multiple bindings to refer to each other, the current approach of handling one binding at a time is not sufficient. Instead, there is a list of simultaneous declarations where the scope of each is extended with variables for all the declarations. This can be represented in the syntax tree without too much effort, even using the generic syntax descriptions seen before. Manipulating this structure, e.g. splitting binding groups into strongly connected components, is expected to be challenging, but potentially instructive.

6.1.2 Other Transformations

Local rewrites There is a number of local transformations that simply rewrite a specific pattern into an equivalent one. Most examples can always be performed safely:

- constant folding and identities,
  e.g. $P + 0 \;\Rightarrow\; P$
- turning beta redexes into let-bindings,
  i.e. $(\lambda x.\, Q)\; P \;\Rightarrow\; \textbf{let}\; x = P\; \textbf{in}\; Q$
- floating let-bindings out of function application,
  i.e. $(\textbf{let}\; x = P\; \textbf{in}\; Q)\; R \;\Rightarrow\; \textbf{let}\; x = P\; \textbf{in}\; Q\; R$

A general fold-like function should allow to specify a single instance of a rewrite and then apply it wherever possible in a pass over the whole program. Similarly, the overall preservation of semantics should follow from that of a single rewrite instance.

Common subexpression elimination The aim of this transformation is to find subexpressions that occur multiple times and replace them with a variable refering to a single matching declaration, reducing both code size and work performed during evaluation. For a basic implementation it suffices to try finding occurrences of expressions that are the same as the declaration of a binding already in scope. A more powerful approach is to put more work into identifying common subexpressions and making the transformation itself introduce shared bindings at suitable points.

In de Bruijn representation, identifying identical subexpressions is challenging, since their context (and thus their type) may differ. This problem is avoided by co-de-Bruijn representation.

Bibliography

[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. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, jul 2018. doi:10.1145/3236785.

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

[3] Henk P. Barendregt. The lambda calculus: its syntax and semantics. In *Studies in Logic and the Foundations of Mathematics*, volume 103, 1985.

[4] 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, nov 2014. doi:10.1017/s0960129514000115.

[5] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), jul 2005. doi:10.2168/lmcs-1(2:1)2005.

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

[7] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, may 2011. doi:10.1007/s10817-011-9225-2.

[8] Jesper Cockx. 1001 representations of syntax with binding, 2021. Accessed: 2022-12-16. URL: https://jesper.sikanda.be/posts/1001-syntax-representations.html.

[9] Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, apr 2014. arXiv:1201.4801, doi:10.1017/s0956796814000069.

[10] Nils Anders Danielsson. Operational semantics using the partiality monad. *ACM SIGPLAN Notices*, 47(9):127–138, oct 2012. doi:10.1145/2398856.2364546.

[11] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

[12] Dougal Maclaurin, Alexey Radul, and Adam Paszke. The foil: Capture-avoiding substitution with no sharp edges. October 2022. arXiv:2210.04729.

[13] Conor McBride. Turing-completeness totally free. In *Lecture Notes in Computer Science*, pages 257–275. Springer International Publishing, 2015. doi:10.1007/978-3-319-19797-5_13.

[14] Conor McBride. Everybody's got to be somewhere. *Electronic Proceedings in Theoretical Computer Science*, 275:53–69, jul 2018. doi:10.4204/EPTCS.275.6.

[15] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin, 1999. doi:10.1007/978-3-662-03811-6.

[16] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation - TLDI '09*. ACM Press, 2008. doi:10.1145/1481861.1481862.

[17] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. *ACM SIGPLAN Notices*, 49(9):123–135, nov 2014. doi:10.1145/2692915.2628160.

[18] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, jul 2002. doi:10.1017/s0956796802004331.

[19] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. *ACM SIGPLAN Notices*, 31(6):1–12, jun 1996. doi:10.1145/232629.232630.

[20] Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, sep 1998. doi:10.1016/s0167-6423(97)00029-4.

[21] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23(7):199–208, jul 1988. doi:10.1145/960116.54010.

[22] Mitchell Pickard and Graham Hutton. Calculating dependently-typed compilers (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–27, aug 2021. doi:10.1145/3473587.

[23] André L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995. URL: https://theses.gla.ac.uk/74568/.

[24] Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. Embracing a mechanized formalization gap, 2019. doi:10.48550/ARXIV.1910.11724.

[25] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.