(Re-)Creating sharing in Agda's GHC backend

(RE-)CREATING SHARING IN AGDA'S GHC BACKEND

BY
NATALIE PERNA, B.A.Sc.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Master of Science (2017)
(Computer Science)

McMaster University
Hamilton, Ontario, Canada

TITLE:	(Re-)Creating sharing in Agda's GHC backend
AUTHOR:	Natalie Perna B.A.Sc. (Honours Computer Science) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Wolfram Kahl
NUMBER OF PAGES:	ix, 51

To my family

Abstract

Agda is a dependently-typed programming language and theorem prover, supporting proof construction in a functional programming style. Due to its incredibly flexible concrete syntax and support for Unicode identifiers, Agda can be used to construct elegant and expressive proofs in a format that is understandable even to those unfamiliar with the tool.

However, the semantics of Agda is lacking resource guarantees of the kind that Haskell programmers are used to with lazy evaluation, where multiple uses of function arguments and let-bound variables still result in the corresponding expressions to be evaluated at most once. With the current compiler backends of Agda, a mathematically-natural way to structure programs therefore frequently results in inefficient compiled programs, where the run-time complexity can be exponentional in cases where corresponding Haskell code executes in linear time.

This makes a highly-optimised compiler backend a particularly essential tool for practical development with Agda. The main contributions of this thesis are a series of compiler optimisations that inlines simple projections, removes some expressions with trivial evaluations that can be statically inferred, and reduces the need for repeated evaluations of the same expressions by increasing sharing.

We developed transformations that focus on the inherent "loss" of sharing that is frequently the result of compiling Agda programs. Where an Agda developer may imagine that value sharing should exist in the generated Haskell code, it often does not. We present several optimising transformations that re-introduce some of this "lost" sharing without affecting the type-theoretic semantics, then apply these optimisations to several typical Agda applications to examine the memory allocation and execution time effects.

In measuring the effects of these optimisations on Agda code we show that overall improvements in runtime on the order of 10-20% are possible. We hope that the development and discussion of these optimisations is useful to the Agda developer community, and may be helpful for future contributors interested in implementing new optimisations for Agda.

Acknowledgements

Thank you to Dr. Kahl, whose patience and guidance has helped me grow intellectually, both as an undergraduate and graduate student. I am truly grateful for my time learning from him.

Thank you also to the Natural Sciences and Engineering Research Council of Canada (NSERC) for the financial support that made my graduate studies possible.

Lastly, thank you to all of my family, friends, and loved ones, for their support and encouragement, especially my fiancé, Max.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we introduce Agda and give an overview of this project. In Section 1.1, we give an introduction to the Agda programming language and explain its unique characteristics, as well as a brief introduction to its compiler. In Section 1.2, we state the problem subject of our work. In Section 1.3, we give the motivation for the new optimisation strategies introduced here. In Section 1.4, we summarise our contributions, namely the optimisation strategies we implemented in the Agda compiler. Finally, in Section 1.5, we give the structure of the remainder of the thesis.

1.1 Agda

Agda (Norell, 2007) is a dependently-typed programming language and theorem prover, supporting proof construction in a functional programming style. Due to its incredibly flexible concrete syntax and support for Unicode identifiers (Bove et al., 2009), Agda can be used to construct elegant and expressive proofs in a format that is understandable even to those unfamiliar with the tool. As a result, many users of Agda, including our group, are quick to sacrifice speed and efficiency in our code in favour of proof clarity. This makes a highly-optimised compiler backend a particularly essential tool for practical development with Agda.

1.1.1 Dependent Types

Norell (2009) discusses the practical usage of dependent types for programming in Agda. Type signatures in Agda support dependent types, which means that types may depend on values. In traditional functional languages, types may depend on other types. For example, the Haskell type signature `xs :: Vector a` denotes a vector containing elements of type `a`, where `a` is a type variable. In a dependently-typed language like Agda types can depend not only on other types, but also on values. Consider the replicate function in Figure 1.1, which produces a Vector of elements of type A with length n. In many languages that don't support dependent types a programmer

Logic	Type theory
proposition	type
theorem	inhabited type
proof	program
implication	function space
conjunction	product
disjunction	sum
second order quantifier	type quantifier
truth	inhabited type
falsity	uninhabited type

Table 1.1: Correspondent concepts in logic and type theory.

can represent a parametrised vector type that contains elements of any arbitrary type, however, they cannot represent a parametrised vector type of any arbitrary specific length.

```
replicate : ∀ {A : Set} (n : ℕ) → (x : A) → Vec A n
replicate zero x = []
replicate (suc n) x = x :: replicate n x
```

Figure 1.1: The replicate function in Agda.

1.1.2 Type Theory

The core syntax of Agda is a dependently-typed lambda calculus, with a simple grammar as shown in Figure 1.3. Most functional languages, such as Haskell or ML, are built on a foundation of a simply-typed lambda calculus. This basis allows the expression of propositions in mathematical logic as types of a lambda calculus.

By the Curry–Howard correspondence (or the proofs-as-programs interpretation), we can then prove these propositions true by providing an element (program) of its corresponding type (Poernomo et al., 2005). Using constructive logic, a proof of a proposition is the construction of an object that is a member of the type representing that proposition. The correspondence between concepts in type theory and concepts in logic can be seen in Table 1.1

The ability for types to contain arbitrary values is significant because it expands the domain of theorems we can encode as types to the space of predicate logic. This allows us to encode almost any proposition or formula as a type.

In order to ensure this logic holds true for all Agda programs, the Agda type-checker requires that all programs be both total and terminating (Norell, 2009). Therefore, when an Agda program

passes type-checking[1], all of the specifications encoded as propositions in the types of functions are satisfied.

Agda also supports a flexible mix-fix syntax, as seen in Figure 1.2, and Unicode characters, such as the $\mathbb{N}$ to represent natural numbers in Figure 1.1. These features along with Agda's constructive functional style make Agda both an interesting programming language, but also a powerful proof assistant for generating elegant, expressive proofs.

```
if_then_else_ : ∀ {A : Set} → Bool → A → A → A
if true then t else f = t
if false then t else f = f
```

Figure 1.2: The if_then_else_ function in Agda.

$a ::=$	x	variable
	$\mid \lambda x \to a$	abstraction
	$\mid a\ a$	application
	$\mid (x : a) \to a$	function space
	$\mid Set[n]$	universe (at level n)
	$\mid (a)$	grouping

Figure 1.3: Agda core syntax grammar (Agda, 2017d).

1.1.3 Compiler

Agda has a number of available compilers and backends, but the one that is most efficient and most commonly used is the "GHC backend" (Benke, 2007), originally introduced under the name "MAlonzo"; this backend generates Haskell with extensions supported by GHC, the "Glasgow Haskell Compiler" by Marlow and Peyton Jones (2012). This backend has the goal of compiling Agda code with the performance of the generated code matching that of GHC, and it does so by translating Agda into Haskell, so that it can be compiled, and optimised by GHC. This is a practical and useful arrangement for real-world Agda usage because GHC has benefited from a massive development effort by a large community to create a highly performant compiler (Benke, 2007).

As discussed in the previous section, Agda provides a more expressive type system than Haskell. Because Agda supports dependent types and Haskell does not, in order for Agda generated code

[1](assuming a correct compiler)

to pass the Haskell type checker, it is necessary for the MAlonzo backend to wrap coercions around all function arguments and all function calls, which cast terms from one type to a different arbitrary type. Unfortunately, these potentially unsafe type coercions mean that there are many GHC optimisations which Agda's generated code is "missing out on" (Fredriksson and Gustafsson, 2011).

Some of the Agda optimisations described herein would typically be performed by GHC after translation to Haskell were it not for these coercions, so we instead ensure that we can still take advantage of these optimisations by implementing them in the Agda backend, before the translation to Haskell occurs.

1.1.4 Sharing

In several of our optimisations presented herein, our ultimate goal is to introduce sharing that was previously "lost". Take for example the simple module below:

```
module Sharing where

data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)

one = suc zero

four = let two = one + one
       in two + two
```

An Agda developer might, incorrectly, assume that in the calculation of four, the + function would only be called twice, with the evaluation of *two* stored and shared among its two callers. In actuality, the let binding is not necessarily preserved in the translation to Haskell, where sharing of the locally bound expression would be guaranteed. Unlike Haskell, Agda does not have any resource-aware semantics. In the compiled Haskell generated from the Sharing module, we can see that there are, in fact, three calls to the generated addition function *d8*, and the semantic "sharing" that was implied by the programmer who wrote the let binding has been "lost":

```
module MAlonzo.Code.Sharing where
import MAlonzo.RTE (coe, erased, addInt, subInt, mulInt, quotInt,
  remInt, geqInt, ltInt, eqInt, eqFloat)
import qualified MAlonzo.RTE
```

```
import qualified Data.Text
name2 = "Sharing.ℕ"
d2 = ()
data T2 a0 = C4 | C6 a0
name8 = "Sharing._+_"
d8 v0 v1
  = case coe v0 of
      C4 → coe v1
      C6 v2 → coe C6 (coe d8 v2 v1)
      _ → coe MAlonzo.RTE.mazUnreachableError
name16 = "Sharing.one"
d16 = coe C6 C4
name18 = "Sharing.four"
d18 = coe d8 (coe d8 d16 d16) (coe d8 d16 d16)
```

Though this particular illustrative example is not targeted by our optimizations, because it would require common subexpression elimination (CSE) to transform, we use examples like this as motivation for re-creating sharing through compiler optimisations.

1.2 Problem Statement

For practical development in Agda, a highly effective optimising compiler backend is a particularly essential tool to avoid performance concerns.

Our work aims to introduce a number of optimising transformations to the Agda internals and backend so that Agda users can continue to focus on elegant syntax and mathematical clarity, and leave the optimisations necessary to transform that code into a program that runs with acceptable heap allocation and execution time to the compiler.

The optimisations we focus on are specifically oriented towards aiding our team's most common, and most costly, uses of the Agda programming language, but they should be useful in a general context for most Agda users.

1.3 Motivation

As discussed above, the Agda language is highly conducive to writing elegant and expressive proofs, which leads many users of Agda, including our research group, to avoid code optimisations that may increase speed or efficiency if they have the side effect of reducing proof clarity (which they often do).

COST CENTRE	**%time**
Data.Product.Σ.proj$_2$	13.1
Data.Product.Σ.proj$_1$	7.5
Data.SUList.ListSetMap...	3.0
...	

Table 1.2: Profiler results of RATH-Agda execution.

As such, the optimisations presented herein are largely motivated by profiling data from our own real-world uses of Agda.

The first, and simplest, such example is the results of profiling a main execution of RATH-Agda. RATH-Agda is a library of category and allegory theories developed by Kahl (2017) which takes advantage of many features of the Agda programming language's flexibility. In order to achieve its primary goal of natural mathematical clarity and style, it faces, like many Agda programs, performance concerns.

Using the GHC built-in profiling system, we generated profiling data for the RATH-Agda library's execution and found that the time required to evaluate simple record projections combined to be the greatest cost-centres in the data. This representative usage of the RATH-Agda library spends more than 20% of execution time on just two types of projections (see Table 1.2).

Therefore, the first compiler optimisation we sought to add was an efficient automatic inlining of such projections.

1.4 Main Contributions

The main contributions to the Agda compiler include:

(i) automatic inlining of proper projections

(ii) removal of repeated case expressions

(iii) avoiding trivial case expressions with patterned let expressions

The Agda compiler's type checker allows us to identify proper projections, and we have developed a patch for automatically inlining such projections.

However, the pass commonly results in deeply nested case expressions, many of which are pattern matching on the same constructor as its ancestors, thereby duplicating variables that the compiler has already bound. By gathering the matched patterns throughout a pass over the nested terms, we are able to prune patterns that are unnecessarily repeated, and substitute in place the previously bound pattern variables.

We remove the need for more case analysis by identifying let expressions with a case expression body, which scrutinises the variable bound by the parent let. Often these case expressions in Agda's

generated code have only one case alternative, a constructor pattern match. In these situations, we replace the case expression with the body of that single alternative, and bind the constructor variable(s) using an as-pattern.

1.5 Structure of the Thesis

The remainder of this thesis is organised as follows:

Chapter 2 surveys some literature and projects relevant to our work.

Chapter 3 introduces the required compiler theory and logical background.

Chapters 4 to 7 describes the processes by which we optimise Agda programs, and gives a number of illustrative examples demonstrating the application of the implemented optimisations.

Chapter 8 discusses the contribution's strengths and weaknesses and draws some conclusions.

Chapter 2

Related Work

In this chapter, we provide a survey of the literature and discuss some existing techniques for improving functional code that relate closely to our work.

2.1 Common subexpression elimination

Common subexpression elimination (CSE) is a compiler optimisation that reduces execution time by avoid repeated computations of the same expression (Chitil, 1998). This is very similar to our goal with case squashing. As anyone familiar with the nature of purely functional programming languages might realise, identification of common subexpressions is much simpler in a functional language thanks to the expectation of referential transparency (Chitil, 1998). (As a reminder, referential transparency means that an expression always produces the same result regardless of the context in which it is evaluated.)

Appel (1992) first implemented CSE in the strict functional language ML's compiler, and Chitil (1998) first explored the implementation of CSE in a lazy functional language, Haskell. The difficulty with implementing CSE in a lazy language is that, although common subexpressions are easy to identify, determining which common subexpressions will yield a benefit by being eliminated is more challenging. To avoid complex data flow analysis on functional code, Chitil (1998) developed some simple syntactic conditions for judging when CSE is beneficial in Haskell code. We omit these from our survey, as such heuristics are unnecessary for our optimisation. We instead focus on the relevant "compilation by transformation" approach used when implementing CSE in GHC.

The GHC compilation process consists of translating Haskell code into a second-order λ-calculus language called Core, at which point a series of optimising transformation are performed, and the backend code generator transforms Core code into the final output (Chitil, 1998). This process is very similar to the Agda compilation process, which translates Agda code into Treeless code, applies a series of optimising transformations, and finally generates Haskell code through the backend, as discussed in Section 3.2.

The syntax of the Core intermediate language of Haskell is very similar to Treeless, with expressions consisting mainly of λ abstractions, **let** bindings, **case** expressions, constructors, literals and function applications, much like Agda's Treeless syntax.

CSE is implemented in GHC with a single recursive traversal of the Core program. For each expression, its subexpressions are first transformed, then it is determined whether the whole transformed expression has occurred already (Chitil, 1998).

Given the expression:

let $x = 3$ **in let** $y = 2 + 3$ **in** $2 + 3 + 4$

the result of the recursive trransformation is:

let $x = 3$ **in let** $y = 2 + x$ **in** $y + 4$.

2.2 Let-floating

Peyton Jones et al. (1996)'s "Let-floating: moving bindings to give faster programs" discusses the effects of a group of compiler optimisations for Haskell which move let bindings to reduce heap allocation and execution time. The cumulative impact of their efforts to move let-bindings around in Haskell code resulted in more than a 30% reduction in heap allocation and a 15% reduction in execution time.

Peyton Jones et al. (1996) explain that GHC (the Glasgow Haskell Compiler) is an optimising compiler whose guiding principle is *compilation by transformation*. Whenever possible, optimisations in GHC are implemented as correctness-preserving program transformations, a principle that is reflected in the Agda compiler as well. The optimisations that we present later in this thesis are also best thought of as "correctness-preserving program transformations".

Before we approach the optimisations presented by Peyton Jones et al. (1996) we discuss the operational interpretation of the Haskell Core language. Haskell Core expressions are built from the syntax shown in Figure 2.1.

NoDefault

In order to best understand the advantages of the let-floating transformations described below, there are two facts about the operational semantics of Core that are useful to know:

1. **let** bindings, and only **let** bindings, perform heap allocation.
2. **case** expressions, and only **case** expressions, perform evaluation.

In this paper, three types of let-floating transformations are presented:

1. "Floating inwards" to move bindings as far inwards as possible,

Expr →	*Expr Atom*	application
	| *Expr ty*	type application
	| λ $var_1 ... var_n$ `->` *Expr*	lambda abstraction
	| Λ *ty* `->` *Expr*	type abstraction
	| `case` *Expr* `of` *Alts*	case expression
	| `let` *var* `=` *Expr* `in` *Expr*	local definition
	| *con* $var_1 ... var_n$	constructor, $n \geqslant 0$
	| *prim* $var_1 ... var_n$	primitive, $n \geqslant 0$
	| *Atom*	literal
Atom →	*var*	variable
	| *Literal*	unboxed object
Alts →	$Calt_1; ...; Calt_n; Default$	$n \geqslant 0$
	| $Lalt_1; ...; Lalt_n; Default$	$n \geqslant 0$
Calt →	*con* $var_1 ... var_n$ `->` *Expr*	constructor alt, $n \geqslant 0$
Lalt →	*Literal* `->` *Expr*	literal alt
Default →	`NoDefault`	
	| *var* `->` *Expr*	

Figure 2.1: Syntax of the Haskell Core language.

2. "The full laziness transformation" which floats some bindings outside enclosing lambda abstractions, and
3. "Local transformations" which move bindings in several optimising ways (Peyton Jones et al., 1996).

Floating inwards

Floating inwards is an straightforward optimisation that aims to move all let bindings as far inward in the expression tree as possible. This accomplishes three separate benefits:

- It increases the chance that a binding will not be met in program execution, if it is not needed on a particular branch.

- It increases the chance that strictness analysis will be able to perform further optimisations.
- It makes it possible for further redundant expressions to be eliminated (Peyton Jones et al., 1996).

Consider as an example, the Haskell code:

$$
\begin{array}{l}
\textbf{let } x = \textbf{case } y \textbf{ of } (a, b) \rightarrow a \\
\textbf{in} \\
\textbf{case } y \textbf{ of} \\
\quad (p, q) \rightarrow x + p
\end{array}
$$

and its optimised version with let-bindings floated inward:

$$
\begin{array}{l}
\textbf{case } y \textbf{ of} \\
\quad (p, q) \rightarrow \textbf{let } x = \textbf{case } y \textbf{ of } (a, b) \rightarrow a \\
\qquad \textbf{in } x + p
\end{array}
$$

These two Haskell expressions are semantically-equivalent, but the second has potential to become more efficient than the first because later optimisations will be able to identify the opportunity to remove the redundant inner case expression which scrutinises the same variable as the outer case expression (Peyton Jones et al., 1996). Though it wasn't clear in the first expression, because the case expressions weren't nested, the second expression can now benefit from a GHC optimisation much like our "case squashing" optimisation.

Full laziness

While the above transformation attempts to push bindings inwards, the full laziness transformation does the reverse, floating bindings outwards. By floating some bindings out of lambda abstractions, we can avoid re-computing the same expression on repeated recursive calls to the same function (Peyton Jones et al., 1996).

The benefit from this increased sharing outweighs the detriment of increasing the let scope in most cases where:

- the expression being bound requires a non-negligible amount of computational work to evaluate; and
- the lambda abstraction it is used in is called more than once (Peyton Jones et al., 1996).

Consider as an example, the Haskell code (Peyton Jones et al., 1996):

$$
\begin{array}{l}
f = \lambda xs \rightarrow \\
\quad \textbf{let } g = \lambda y \rightarrow \textbf{let } n = \mathit{length}\ xs \\
\qquad \textbf{in } \ldots\ g\ \ldots\ n\ \ldots \\
\quad \textbf{in } \ldots\ g\ \ldots
\end{array}
$$

and its optimised version with let-bindings floated outward:

$f = \lambda xs \rightarrow$
  let $n = length\ xs$
  in let $g = \lambda y \rightarrow ...g\ ...\ n\ ...$
    in $...\ g\ ...$

In order to maximise the number of opportunities for this type of let-floating, it may be necessary to create dummy let bindings for free subexpressions in the lambda abstraction, so that they can be floated out as well (Peyton Jones et al., 1996).

Local transformations

The local transformations consist of a series of three small rewrite rules as follows:

1. (**let** $v = e$ **in** b) $a \quad \longrightarrow \quad$ **let** $v = e$ **in** $b\ a$

   Moving the let outside the application cannot have a negative effect, but it can have a positive effect by creating opportunities for other optimisations (Peyton Jones et al., 1996).

   Consider the case where b is a lambda function. Before floating the let outside the application, the function cannot be applied to a:

   (**let** $v = \langle\langle v\text{-}rhs \rangle\rangle$ **in** $(\lambda x \rightarrow ...x...))\ a$

   However, after floating the let outside, it is clear that a beta-reduction rule can be applied, substituting an a for ever x at compile-time:

   let $v = \langle\langle v\text{-}rhs \rangle\rangle$ **in** $(\lambda x \rightarrow ...x...)\ a$

2. **case** (**let** $v = e$ **in** b) **of** $alts \quad \longrightarrow \quad$ **let** $v = e$ **in case** b **of** $alts$

   Likewise for moving the let outside a case expression, it won't have a negative effect, but could have a positive effect (Peyton Jones et al., 1996).

   Consider the case where b is a constructor application. Before floating the let outside the case expression, there isn't a clear correspondence between the constructor and the alternatives:

   case (**let** $v = \langle\langle v\text{-}rhs \rangle\rangle$ **in** $con\ v1\ v2\ v3$) **of** $\langle\langle alts \rangle\rangle$

   However, after floating the let outside, it is clear that the case expression can be simplified to the body of the alternative with the same constructor, without any evaluation being performed:

   let $v = \langle\langle v\text{-}rhs \rangle\rangle$ **in case** $con\ v1\ v2\ v3$ **of** $\langle\langle alts \rangle\rangle$

3. **let** $x =$ **let** $v = e$ **in** b **in** $c \quad \longrightarrow \quad$ **let** $v = e$ **in let** $x = b$ **in** c

   Moving a let binding from the right-hand side of another let binding to outside it can have several advantages including potentially reducing the need for some heap allocation when the final form of the second binding becomes more clear (Peyton Jones et al., 1996).

   In the following example, floating the let binding out reveals a head normal form. Without floating, when x was met in the $\langle\langle body \rangle\rangle$, we would evaluate x by computing the pair (v, v) and overwriting the heap-allocated thunk for x with the result:

   let $x =$ **let** $v = \langle\langle v\text{-}rhs \rangle\rangle$ **in** (v, v)
   in $\langle\langle body \rangle\rangle$

   With floating, we would instead allocate a thunk for v and a pair for x, so that x is allocated in its final form:

   let $v = \langle\langle v\text{-}rhs \rangle\rangle$
   in let $x = (v, v)$
   in $\langle\langle body \rangle\rangle$

   This means that when x is met in the $\langle\langle body \rangle\rangle$ and evaluated, no update to the thunk would be needed, saving a significant amount of memory traffic (Peyton Jones et al., 1996).

Similar methods to these let-floating transformations are used in our patterned let-floating across function calls, with the same goal of increasing sharing and decreasing re-evaluation of the same expressions.

Chapter 3

Background

In this chapter, we introduce the necessary logical and compiler background concepts required for the understanding of the material presented in this thesis. In Section 3.1, we review some of the mathematical background useful for understanding our optimisations. In Section 3.2, we give an introduction to the Agda compiler.

3.1 Lambda Calculus

Lambda calculus (or λ-calculus) is a formal system for representing computational logic in terms of function abstractions and applications using variable binding and substitution. It warrants our understanding because concepts surrounding the Agda programming language and its compilation are inspired by, and can be elegantly explained by, the framework of the lambda calculi. In fact, the namesake of the default Agda GHC backend, MAlonzo, is Alonzo Church, the mathematician who first developed the lambda calculus (Agda, 2017d).

3.1.1 Pure λ-Calculus

In a pure λ-calculus, terms are built inductively from only variables, λ-abstractions and applications (Church, 1941), as shown in the following grammar:

$$
\begin{array}{rll}
t ::= & x & \text{variable} \\
| & \lambda x.t & \text{abstraction} \\
| & t\ t & \text{application}
\end{array}
$$

3.1.2 De Bruijn Index Notation

In order to eliminate the need for named variables in λ-calculus notation, De Bruijn indexed notation is used to represent bound terms (variables) with natural numbers, as presented by de Bruijn (1972). In any term, the positive integer n refers to the nth surrounding λ binder. In other words, the number is an index indicating the number of variable binders (or λ-abstractions) in scope between itself and the binder for the variable being referenced. The grammar of a De Bruijn indexed lambda calculus is shown below:

$$\begin{array}{rll} t ::= & \mathbb{N} & \text{variable} \\ | & \lambda\ t & \text{abstraction} \\ | & t\ t & \text{application} \end{array}$$

See Figure 3.1 for an illustration where the variable bindings and indices are coloured to indicate matches and the references are shown with arrows.

$$\lambda x.(\lambda y.(\lambda z.y)y)x$$

$$\lambda\ (\lambda\ (\lambda\ 1)\ 0)\ 0$$

Figure 3.1: An example of a (circumlocutious) identity function as a pure (above) and De Bruijn indexed (below) λ-calculus expression.

The internal representation of Agda code in the compiler is based on a De Bruijn indexed λ-calculus.

3.1.3 $\lambda\sigma$-Calculus

In order to perform the desired optimisations on the abstract syntax tree, we must be able to perform substitutions on terms. Treating the abstract syntax tree structure as a specialised λ-calculus, we can implement substitution as a function on terms.

Because our terms are built on De Bruijn indexed variables, we use Abadi et al. (1990)'s explicit substitution of a $\lambda\sigma$-calculus as a reference for understanding correct substitution on terms in the context of local variables bound by incrementing indices. The $\lambda\sigma$-calculus is a refinement of the λ-calculus where substitutions are manipulated explicitly, and substitution application is a term constructor rather than a meta-level notation.

3.1.4 Substitution

Take then, for instance, the classical application of the β-reduction rule in a pure λ calculus:

$$(\lambda x.t)s \rightarrow_\beta t[x := s]$$

Beta reduction is the process of simplifying an application of a function to the resulting substituted term. However, in order to β-reduce a De Bruijn indexed expression, like $(\lambda a)b$, it isn't sufficient to only substitute b into the appropriate occurrences in a. As the λ binding disappears, we must also decrement all remaining free indices in a (Abadi et al., 1990). This adapted form of the β-rule for De Bruijn indexed λ calculus can be represented by the following infinite substitution:

$$(\lambda t)s \rightarrow_\beta t[0 := s, 1 := 0, 2 := 1, ...]$$

However, the substitution in this adapted rule must be evaluated carefully to produce a correct result. Consider if the term t contains another λ binding. As the substitution is applied to that nested λ term, occurrences of 0 should not be replaced with s, because occurrences of 0 refer to the nested λ term's bound variable. Instead, occurrences of 1 should be replaced with s; likewise, occurrences of 2 should be replaced by 1, and so on. We thus "shift" the substitution (Abadi et al., 1990).

It is also important when applying substitutions to λ terms that we avoid the unintended capture of free variables in our terms being substituted in. Imagine again the nested λ term, with occurrences of 1 being replaced with s. Occurrences of 0 in s must be replaced with 1, else the nested λ binder will capture the index. We this "lift" the indices of s (Abadi et al., 1990). These two caveats result in the following substitution rule for De Bruijn indexed lambda terms:

$$(\lambda t)[0 := s, 1 := 0, ...] = \lambda t[1 := s[0 := 1, 1 := 2, ...], 2 := 1, ...]$$

Recognising the required index "shifting" and "lifting" in the substitution rule above should suffice as background for understanding the variable manipulation performed in our optimisation.

3.2 Agda

3.2.1 Compiler

The Agda programming language's first and most-used backend is MAlonzo, or more generically, the GHC backend (Benke, 2007). Given an Adga module containing a main function[1], the Agda

[1] An Agda module without a main file can be compiled with `--no-main`.

`--compile` option will compile the program using the GHC backend by default, which translates an Agda program into Haskell source. The generated Haskell source can then be automatically (default) or manually (with `--ghc-dont-call-ghc`) compiled to an executable program via GHC (Agda, 2017a).

There are several stages of translation and compilation in this process, as shown in Figure 3.2. The transition of primary interest for our optimisations is the conversion of compiled clauses to a "treeless" syntax. This translation occurs after Agda type-checking but before Haskell source is generated. Most Agda optimisations occur as alterations to the treeless terms.

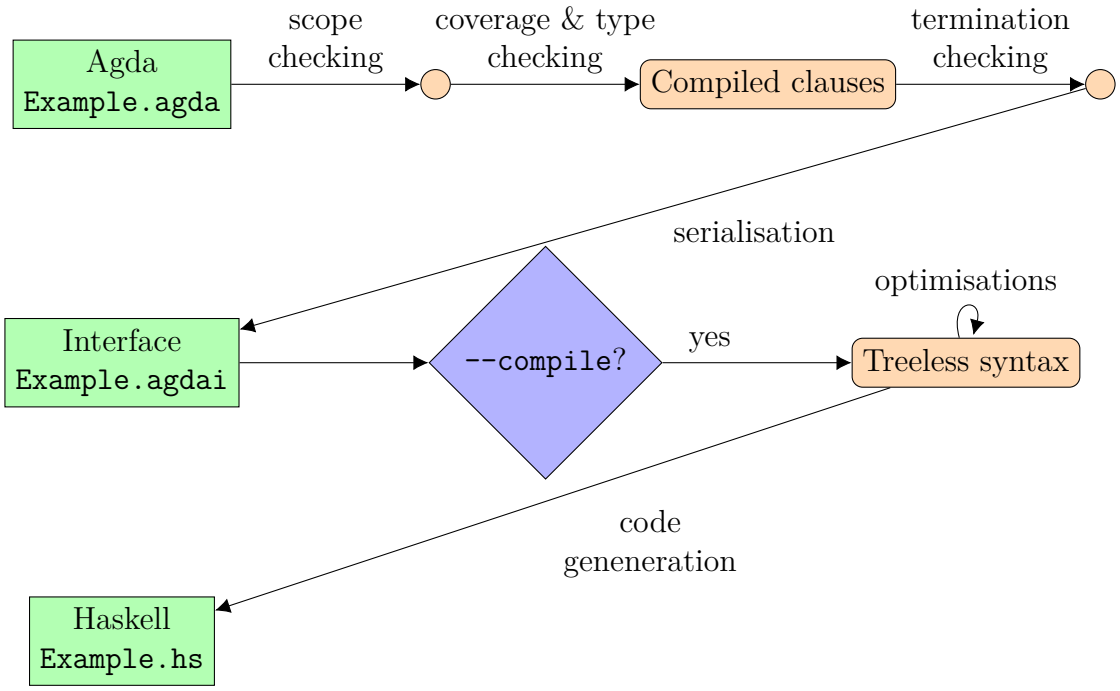

Figure 3.2: Stages of Agda compilation.

An Agda function is defined by declaring its type[2] and providing a definition in the form of one or more clauses. Functions on datatypes can be defined by pattern matching on the constructors of that datatype, describing a structurally recursive function (Agda, 2017d). This should sound familiar to users of functional programming languages like Haskell. Unlike Haskell, however, Agda does not permit partial functions. Therefore, functions defined by pattern matching must not exclude any possible cases from the pattern matching clauses (Agda, 2017d). Because function definitions in Agda are written as a series of one or more pattern matching clauses on possible variable inputs, we can construct an equivalent definition via case tree (Agda, 2017d). Once coverage checking and type checking is completed, pattern matching can be translated into case trees by successively splitting on each variable (Agda, 2017b). Compiled clauses are the first stage of compilation and they are, simply put, case trees.

[2]It is worth noting that type inference is an undecidable problem for definitions with dependent types, so type signatures must be provided in many cases, and by convention, should always be provided.

Take for example the simple not function on booleans below:

```
module Not where

data 𝔹 : Set where
  true : 𝔹
  false : 𝔹

not : 𝔹 → 𝔹
not true = false
not false = true
```

After successful scope, coverage and type checking, the following compiled clauses are produced for the not function:

```
case ru(0) of
  Not.𝔹.true -> done[] Not.𝔹.false
  Not.𝔹.false -> done[] Not.𝔹.true
```

and if the `--compile` flag is enabled, it will then be transformed into treeless syntax:

```
Not.not =
  λ a →
    case a of
      Not.𝔹.true → Not.𝔹.false
      Not.𝔹.false → Not.𝔹.true
```

The treeless syntax is the input to the compiler backend of Agda. It's a high-level internal syntax, the name for which is derived from its use of case expressions instead of case trees. The other notable difference between compiled clauses and treeless syntax is the absence of instantiated datatypes and constructors (Agda, 2017b). Note that internally, variables are represented only by their De Bruijn index, however for ease of illustration, we use named variables in our pretty-printed samples of treeless terms.

Treeless Syntax

This treeless syntax is constructed from the *TTerms* (**T**reeless **Terms**) data type and is the representation of the abstract syntax tree that we will refer to most frequently. It can be reasoned about as a lambda calculus with all local variables represented as De Bruijn indices. A listing of *TTerm* constructors is shown below:

```
type Args = [TTerm]
data TTerm = TVar Nat
  | TPrim TPrim
```

```
    | TDef QName
    | TApp TTerm Args
    | TLam TTerm
    | TLit Literal
    | TCon QName
    | TLet TTerm TTerm
    | TCase Nat CaseType TTerm [TAlt]
    | TUnit
    | TSort
    | TErased
    | TError TError
data TAlt = TACon QName Nat TTerm
    | TAGuard TTerm TTerm
    | TALit Literal TTerm
```

In more detail, the constructor alternatives for *TTerm*s are as follows (Agda, 2017b):

- *TVar* constructs a De Bruijn-indexed variable term.
- *TPrim* constructs a compiler-related primitive, such as addition, subtraction and equality on some primitive types.
- *TDef* constructs a qualified name identifying a function or datatype definition.
- *TApp* constructs a *TTerm* applied to a list of arguments, where each argument is itself a *TTerm*.
- *TLam* constructs a λ-abstraction with a body.
- *TLit* constructs a literal value, such as an integer or string.
- *TCon* constructs a qualified name identifying a constructor.
- *TLet* constructs a let expression, introducing a new local term binding in a term body.
- *TCase* constructs a case expression on a case scrutinee (always a De Bruijn indexed variable), a case type, a default value and a list of alternatives.

  The case alternatives, *TAlt*s, may be constructed from:

  - a *TACon*, which matches on a constructor of a given qualified name, binding the appropriate number of pattern variables to the body term if a match is made. Note that a *TCase*'s list of *Args* must have unique qualified names for each *TACon*.
  - a *TAGuard*, which matches on a boolean guard and binds no variables if matched against.
  - a *TALit*, which matches on a literal term.

- *TUnit*s are used for levels.
- *TSort* constructs a sort, as in the type of types.
- *TErased*s are used to replace irrelevant terms that are no longer needed.
- *TError*s are used to indicate a runtime error.

We also present below a simplified logical representation of the Agda treeless syntax as a grammar named using variables x instead instead of De Bruijn indices:

$$
\begin{array}{lll}
t ::= x & & \text{-- variable} \\
\quad \mid d & & \text{-- function or datatype name} \\
\quad \mid t\ t* & & \text{-- application} \\
\quad \mid \lambda x \to t & & \text{-- lambda abstraction} \\
\quad \mid l & & \text{-- literal} \\
\quad \mid \textbf{let}\ x = t\ \textbf{in}\ t & & \text{-- let} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ a*\ otherwise \to t & & \text{-- case} \\
a ::= d\ v1\ \ldots\ vn \to t & & \text{-- constructor alternative} \\
\quad \mid l \to t & & \text{-- literal alternative}
\end{array}
$$

We use this simplification in the following chapters to discuss our optimisations at a logical level of abstraction.

3.2.2 Module System

The Agda module is designed with simplicity in mind, with the primary goal of organising the way names are used in Agda programs into a hierarchical structure. By default, definitions and datatypes must be referenced unambiguously with both their qualified name and the module in which it is defined.

By this implementation, Agda modules don't have a "type", and scope checking can be accomplished entirely independently of type-checking. After type-checking, all definitions are lambda lifted (Agda, 2017a). However, because names are fully qualified and the concept of "scope" is removed from type-checking, information about potential sharing is lost once arguments are substituted into the types of module definitions.

Consider the following simple parametrized module in Agda:

```
module Composer {A : Set} (f : A → A) where
  twice : A → A
  twice x = f (f x)

  thrice : A → A
  thrice x = f (f (f x))
```

and the Haskell generated by compiling this module:

```
module MAlonzo.Code.Composer where
import MAlonzo.RTE (coe, erased, addInt, subInt, mulInt, quotInt,
  remInt, geqInt, ltInt, eqInt, eqFloat)
import qualified MAlonzo.RTE
import qualified Data.Text
name6 = "Composer.twice"
d6 v0 v1 v2 = du6 v1 v2
du6 v0 v1 = coe v0 (coe v0 v1)
name10 = "Composer.thrice"
d10 v0 v1 v2 = du10 v1 v2
du10 v0 v1 = coe v0 (coe v0 (coe v0 v1))
```

In this example the module's explicit parameters have been abstracted over the definitions to become explicit arguments to the module's functions.

Because arguments are inherited from all enclosing modules, in larger Agda projects, it is easy to create a situation where very large type signatures must be serialised many times when the same modules are referenced more than once (Agda, 2017c).

In Chapter 7 we discuss our attempts to re-introduce some of this lost sharing potential and reduce repeated computations.

3.2.3 Alternate method of case squashing

Following our own development of `--squash-cases` (see Chapter 5), an optimisation was added to the Agda compiler's Simplify stage which accomplishes the same goals as `--squash-cases` in a slightly different way. We examine here that method of removing repeated case expressions.

Immediately following the conversion of compiled clauses to treeless syntax in the Agda compiler, a series of optimising transformations are applied before the treeless expression is returned. One such step is the "simplify" group of transformations, which modify a *TTerm* in a variety of optimising ways.

As the expression is traversed, *simplify* is recursively called on each *TTerm* term, and *simpAlt* is called on each *TAlt* alternative. Given some expression casing on De Bruijn index x, for each alternative of the pattern *TACon name arity body*, the scrutinised variable index in the body, $x + arity$, is looked up in the variable environment. If the variable has already been bound, and therefore has a different De Bruijn index, y, a rewrite rule is added to the constructor. The rewrite rule indicates that every instance of *TApp* (*TCon name*) [*TVar i* | $i \leftarrow$ *reverse* \$ *take arity* [0 . .]] in the alternative's body should be replaced with a *TVar y*.

The rewrite rule is encoded as part of the wrapper *Reader* environment that is carried along with the *TTerm* throughout simplification, and is evaluated later by applying substitutions. It is at this point that all necessary De Bruijn index shifting is managed.

Chapter 4

Inlining Projections

In this chapter we present our projection inlining optimisation. In Section 4.1 we give usage instructions. In Section 4.2 we show a logical representation of the transformation. In Section 4.3 we provide some implementation details pertaining to the optimisation. Lastly, in Section 4.4 we apply projection inlining to a sample program and examine the results.

4.1 Usage

We added the option:

`--inline-proj`	`inline proper projections`

to our Agda branch which, when enabled, will replace every call to a function that is a proper projection with its function body.

4.2 Logical Representation

The logical representation of inlining is fairly straightforward. We recurse through the treeless representation of an Agda module. For every application of a function or datatype to a list of arguments, that is $d\ t_1 \ldots t_n$, where d is the name of a proper projection, and each t_i is a treeless term, we replace $d\ t_1 \ldots t_n$ with the function or datatype definition corresponding to d and substitute in the $t_1 \ldots t_n$ arguments.

4.3 Implementation

It is worth noting that the only projections which we identify and inline are "proper projections", that is, we do not include projection-like functions, or record field values, i.e. projections applied to an argument.

The only major complication in implementing the projection inlining optimisation is accounting for the potential for recursive inlining to loop, resulting in non-termination of compilation. Therefore, when inlining projections, we maintain an environment of previously inlined projections and avoid inlining the same projection more than one level deep.

For a complete listing of our implementation of the projection inlining optimisation, refer to Appendix A. The *Agda.Compiler.ToTreeless* module is responsible for converting Agda's internal syntax to the treeless syntax. It is during this translation that other manual forms of definition inlining are performed, so we introduce our optimisation as an additional guard on translating internal *Def*s, which checks whether the definition is a projection, and performs inlining if it is.

4.4 Application

RATH-Agda is a basic category and allegory theory library developed by Kahl (2017). It includes theories relating to semigroupoids, division allegories, typed Kleene algebras and monoidal categories, among other topics. The RATH-Agda repository also provides a set of test cases in a Main module, which can be used to test a variety of typical uses of the library's functions.

4.4.1 Before

In profiling the runtime of this Main module, we found that an inordinate amount of time was spent on evaluating simple record projections. The first few lines of the profiling report below indicate that the greatest cost centres in terms of time are the two simple record projections for the Σ data type, with a combined 17.6% of execution time spent evaluating them.

```
        Time and Allocation Profiling Report  (Final)

           Main +RTS -S -H7G -M7G -A128M -p -RTS

        total time  =        6.75 secs   (6755 ticks @ 1000 us, 1 processor)
        total alloc = 1,300,428,688 bytes  (excludes profiling overheads)

COST CENTRE                                                    %time %alloc

Data.Product.Σ.proj₂                                            10.4    0.0
```

```
Data.Product.Σ.proj₁                                          7.2    0.0
Categoric.KleeneCategory.DirectSum.SumStar.Square.E⋆′         4.9    7.0
Data.SUList.ListSetMap.RawLSM.RawLSM3.RawLSM3-comp.comp       4.1   10.4
Data.SUList.ListSetMap.RawLSM.RawLSM3.RawLSM3-comp.comp₀      3.9    3.7
...
```

Because enabling profiling does have an affect on execution, we also re-compiled the module without profiling and ran it six times, measuring execution time with the Unix `time` command, to determine its average runtime as 1.60 seconds.

4.4.2 After

By compiling Main with our new option, `--inline-proj`, enabled, we reduced total runtime and memory allocation, as can be seen in the second profiling report of the inlined code:

```
        Time and Allocation Profiling Report  (Final)

           Main +RTS -S -H7G -M7G -A128M -p -RTS

        total time  =        5.26 secs   (5261 ticks @ 1000 us, 1 processor)
        total alloc = 1,299,709,408 bytes  (excludes profiling overheads)

COST CENTRE                                                  %time %alloc

Data.SUList.ListSetMap.FinRel.Utils.FinId                     6.2    8.6
Data.SUList.ListSetMap.RawLSM.RawLSM3.MapImage.mapImage₀      5.6    5.5
Data.SUList.ListSetMap.RawLSM.RawLSM3.RawLSM3-comp.comp       4.9    8.8
Categoric.KleeneCategory.DirectSum.SumStar.Square.E⋆′         4.7    6.6
Data.SUList.ListSetMap.Semigroupoid.LSMJoinOp.\               3.8    8.3
...
```

We again re-compiled the module without profiling and ran it six times, measuring execution time with the Unix `time` command, to determine its average runtime with projections inlined as 1.44 seconds.

We therefore produced a speedup of 1.11× in the RATH-Agda Main module.

Chapter 5

Case Squashing

In this chapter we present our case squashing optimisation. In Section 5.1 we give usage instructions. In Section 5.2 we show a logical representation of the transformation. In Section 5.3 we provide some implementation details pertaining to the optimisation. Lastly, in Section 5.4 we apply case squashing to a sample program and examine the results.

Note that following our own development of the case squashing optimisation, a similar transformation was introduced to the Agda compiler as part of the Simplify pass on the treeless syntax. This alternate method of case squashing is discussed in Subsection 3.2.3. We discuss our case squashing transformation below as it affects code compiled by our branch that precedes this newly implemented alternate method of case squashing.

5.1 Usage

We added the option:

```
--squash-cases                                  remove unnecessary case expressions
```

to our Agda branch which, when enabled, will perform the case squashing optimisation described above.

5.2 Logical Representation

The goal of case squashing is to eliminate case expressions where the scrutinee has already been matched on by an enclosing ancestor case expression. Figure 5.1 shows the transformation from a case expression with repeated scrutinisations on the same variable, to the optimised "case squashed" version.

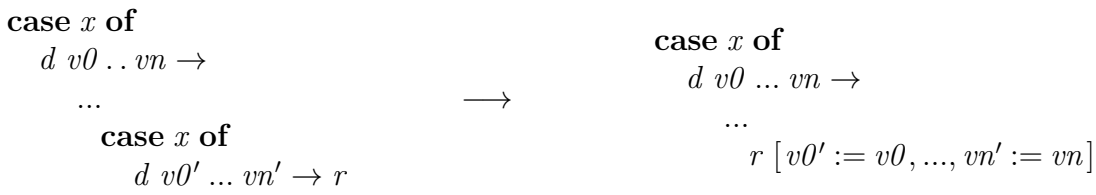

Figure 5.1: Case squashing rule.

For example, given the treeless expression with De Bruijn indexed variables in Figure 5.2, we can follow the De Bruijn indices to their matching variable bindings, to see that the first and third case expressions are scrutinising the same variable, the one bound by the outermost λ abstraction. Therefore, with only static analysis of the expression tree, we know that the third case expression must follow the *da* 2 1 0 branch, and we can thus safely transform the expression on the left into the substituted expression on the right.

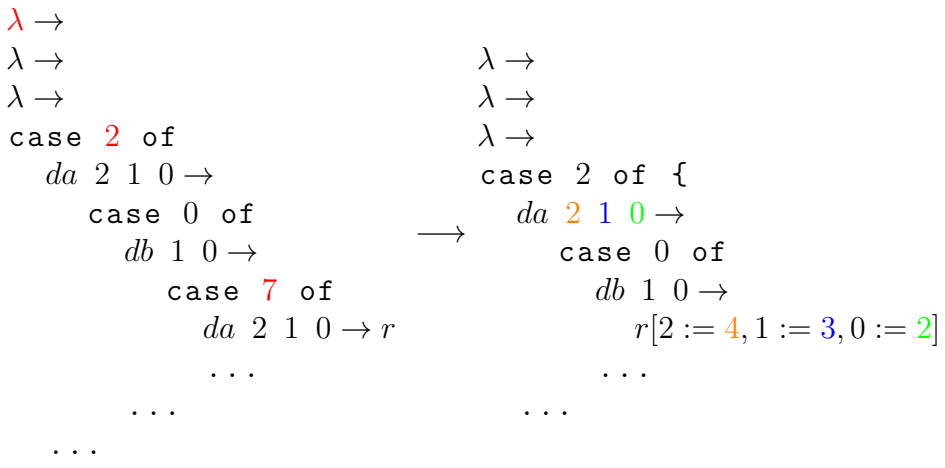

Figure 5.2: Case squashing example.

We perform this "case squashing" by accumulating an environment of all previously scrutinised variables as we traverse the tree structure (appropriately shifting De Bruijn indices in the environment as new variables are bound), and replacing case expressions that match on the same variable as an ancestor case expressions, with the corresponding case branch's body. Any variables in the body that refer to bindings in the removed branch should be replaced with references to the bindings in the matching ancestor case expression branch.

5.3 Implementation

The case squashing implementation is practically very similar to it's logical representation described above. While recursing through the treeless structure we accumulate an environment containing the relevant attributes of the case expressions in scope. As new variables are bound recursing down the structure, the indices stored in this environment are incremented accordingly.

For more details about how variable indices are replaced in the resulting term, refer to section 3.1.4.

For a complete listing of our implementation of the case squashing optimisation, refer to Appendix B.

5.4 Application

Take for example the simple usage of record projections in the Example1 module below.

```
module Example1 where

data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

record Pair (A : Set) (B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B

open Pair

f : Pair (ℕ → ℕ) ℕ → ℕ
f z = fst z (snd z)
```

When we compile this module once without `--inline-proj` on, and once again with `--inline-proj` enabled, a unified diff of the two generated Haskell files gives us what is shown in Figure 5.3a.

The compiled projection function Pair.snd, that is *d18* in the Haskell code, is replaced with a Haskell expression that cases on the pair (p in Agda, *d22* in Haskell) and returns the second field.

We then compile the same file with both `--inline-proj` and `--squash-cases`, and the difference between only inlining and both inlining and squashing can be seen in Figure 5.3b.

Figure 5.3c shows the overall unified diff from neither optimisation to both. In particular, note that after both inlining and squashing optimizations, there is only one case expression scrutinising *v0*.

```
--- compile/Example1.hs
+++ compile-inline/Example1.hs
@@ -24,4 +24,12 @@
 name28 = "Example1.f"
-d28 v0 = coe d22 v0 (coe d24 v0)
+d28 v0
+  = case coe v0 of
+      C26 v1 v2
+        -> coe
+             v1
+             (case coe v0 of
+                C26 v3 v4 -> coe v4
+                _ -> MAlonzo.RTE.mazUnreachableError)
+      _ -> coe MAlonzo.RTE.mazUnreachableError
```

(a) Unified difference of the Example1 module compiled without and then with `--inline-proj`.

```
--- compile-inline/Example1.hs
+++ compile-inline-squash/Example1.hs
@@ -26,10 +26,5 @@
 name28 = "Example1.f"
 d28 v0
   = case coe v0 of
-      C26 v1 v2
-        -> coe
-             v1
-             (case coe v0 of
-                C26 v3 v4 -> coe v4
-                _ -> MAlonzo.RTE.mazUnreachableError)
+      C26 v1 v2 -> coe v1 v2
       _ -> coe MAlonzo.RTE.mazUnreachableError
```

(b) Unified difference of the Example1 module compiled with `--inline-proj` and then also with `--squash-cases`.

```
--- compile/Example1.hs
+++ compile-inline-squash/Example1.hs
@@ -24,4 +24,7 @@
 name28 = "Example1.f"
-d28 v0 = coe d22 v0 (coe d24 v0)
+d28 v0
+  = case coe v0 of
+      C26 v1 v2 -> coe v1 v2
+      _ -> coe MAlonzo.RTE.mazUnreachableError
```

(c) Unified difference of the Example1 module compiled without either optimisation, then with both `--inline-proj` and `--squash-cases`.

Figure 5.3: Comparison of Example1 module compilations.

Chapter 6

Generating Pattern Lets

In this chapter we present our optimisation to generate pattern lets. In Section 6.1 we give usage instructions. In Section 6.2 we show a logical representation of the transformation. In Section 6.3 we provide some implementation details pertaining to the optimisation. Lastly, in Section 6.4 we apply pattern let generation to a sample program and examine the results.

6.1 Usage

We added the option:

```
--ghc-generate-pattern-let          make the GHC backend generate pattern lets
```

to our Agda branch which, when enabled, will generate pattern lets in the GHC backend during compilation.

6.2 Logical Representation

We can avoid generating certain trivial case expressions by identifying let expressions with the following attributes:

- the body of the **let** expression is a **case** expression;
- the case expression is scrutinising the variable just bound by the enclosing **let**;
- only one case alternative exists, a constructor alternative; and
- the default case is marked as *unreachable*.

Figure 6.1 shows the rule for generating an optimised Haskell expression given a treeless expression with the above properties.

let $x = e$
in case x **of**
  $d\ v0\ ...\ vn \rightarrow t$
  $otherwise \rightarrow u$

$\longrightarrow$

let $x@(d\ v0\ ...\ vn) = e$
in t

where $unreachable\ (u)$.

Figure 6.1: Generating pattern lets rule.

Note that branches may be marked *unreachable* if they are absurd branches or just to fill in missing case defaults which cannot be reached.

It is worth noting that this optimisation changes the evaluation sequence of subexpressions and, with Haskell semantics, could amount to the difference between a terminating and non-terminating expression. However, because we're operating on Haskell generated from an Agda program that has already been checked for termination, this semantics change is less dangerous.

Our treeless syntax does not support pattern matching, but when these cases are identified before transforming into Haskell expressions, we can replace them with "pattern lets", removing an unnecessary case expression, and immediately binding the appropriate constructor parameters in the enclosing **let** expression.

These generated pattern lets have two-fold benefits. Firstly, their use reduces the amount of case analysis required in execution, which saves both the time and space needed to run. Secondly, it creates significant opportunities for increasing sharing of expression evaluations which could not have been found when they were **case** expressions. This leads us to our next optimisation, pattern let floating, discussed in Chapter 7.

6.3 Implementation

The *Agda.Compiler.MAlonzo.Compiler* module is responsible for transforming Agda treeless terms into Haskell expressions. In the primary function for this compilation, we introduced a new alternative that matches on terms with potential to be transformed into pattern lets. In order to be a suitable candidate for this optimisation, a **let** expression must exhibit the properties described in Section 6.2. Because these Agda terms use De Bruijn indexed variables, that means the case expression should be scrutinising the 0 (most recently bound) variable, and the requirements can thus be represented with a pattern matching expression $TLet\ \_\ (TCase\ 0\ \_\ \_\ [\,TACon\ \_\ \_\ \_])$, followed by a check that the default branch is unreachable.

For a complete listing of our implementation of the pattern let generating optimisation, refer to Appendix C.

6.4 Application

The Agda module Triangle3sPB gives us an sample usage of mathematical pullbacks, by constructing triangle-shaped graphs and products of those graphs, as an example. These types of computations are relevant and important in many graph-rewriting calculations and can benefit from our optimisations.

When we compile this module once without `--ghc-generate-pattern-let` on, and once again with `--ghc-generate-pattern-let` enabled, a unified diff of the two generated Haskell files gives us what is shown in Figure 6.2. Both times, the module was compiled with `--inline-proj`.

As is shown by this difference, the case analysis on *v0* is no longer required and instead the constructor parameters are immediately bound in the enclosing **let** expression.

```
--- compile-inline/MAlonzo/Code/GraTra/Graphs.hs
+++ compile-inline-genplet/MAlonzo/Code/GraTra/Graphs.hs
@@ -15132,66 +14864,48 @@
 d4788 v0 v1
   = coe
       MAlonzo.Code.Categoric.FinColimits.Coproduct.C8807
-      (let v2
+      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
              = coe
                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       case coe v2 of
-         MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6
-           -> coe v3 v0 v1
-         _ -> MAlonzo.RTE.mazUnreachableError)
-      (let v2
+       coe v3 v0 v1)
+      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
              = coe
                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       case coe v2 of
-         MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6
-           -> coe v4 v0 v1
-         _ -> MAlonzo.RTE.mazUnreachableError)
-      (let v2
+       coe v4 v0 v1)
+      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
              = coe
                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       case coe v2 of
-         MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6
-           -> coe v5 v0 v1
-         _ -> MAlonzo.RTE.mazUnreachableError)
-      (let v2
+       coe v5 v0 v1)
+      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
              = coe
                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       case coe v2 of
-         MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6
-           -> coe v6 v0 v1
-         _ -> MAlonzo.RTE.mazUnreachableError)
+       coe v6 v0 v1)
```

Figure 6.2: Unified difference of the Triangle3sPB module compiled without and then with `--ghc-generate-pattern-let`.

Chapter 7

Pattern Let Floating

In this chapter we present our pattern let floating optimisation. In Section 7.1 we give usage instructions. In Section 7.2 we show a logical representation of the transformation. In Section 7.3 we provide some implementation details pertaining to the optimisation. Lastly, in Section 7.4 we apply pattern let floating to a sample program and examine the results.

7.1 Usage

We added the option:

```
--abstract-plet       abstract pattern lets in generated code
```

which splits generated function definitions into two functions, the first containing only the top-level pattern bindings and a call to the second, and the second containing only the original body inside those pattern bindings, dependent on the additional variables bound in those patterns.

We also added:

```
--float-plet          float pattern lets to remove duplication
```

to our Agda branch which, when enabled, will float the pattern lets up through the abstract syntax tree to join with other bindings for the same expression.

In combination with our option:

```
--cross-call-float          float pattern bindings across function calls
```

that is currently under development, bindings can also be shared across function calls.

7.2 Logical Representation

Figure 7.1 shows an example of floating pattern lets to a join point to increase sharing.

$$\begin{array}{l} f\ (\textbf{let}\ x@(d\ v0\ ...\ vn) = e\ \textbf{in}\ t1) \\ \quad (\textbf{let}\ x@(d\ v0\ ...\ vn) = e\ \textbf{in}\ t2) \end{array} \quad \longrightarrow \quad \begin{array}{l} \textbf{let}\ x@(d\ v0\ ...\ vn) = e \\ \textbf{in}\ f\ t1\ t2 \end{array}$$

Figure 7.1: Floating pattern lets example.

Pattern let floating combines the benefits of pattern lets, described in Chapter 6, with the benefits of floating described in Section 2.2. We take inspiration from Peyton Jones et al. (1996)'s "Full laziness" transformation in GHC and apply it to the code generated by the Agda compiler backend. In our pattern let floating optimisation, we float the pattern lets as far upwards in an expression tree if and until they can be joined with another floated pattern let on the same expression. By doing so, we avoid re-computing the same expression when it is used in multiple subexpressions.

7.3 Implementation

There are a couple of implementation-specific details of interest when implementing pattern let floating. Firstly, in order to float pattern lets, we first convert the *TTerm*s into a variant data type using named variables for ease of expression manipulation. Then the entire expression tree is recursed over, floating all λ bindings to the top of the expression and accumulating a list of variables in each definition is accumulated.

The *floatPatterns* function will only float pattern lets which occur in multiple branches, and they are floated to the least join point of those branches.

Further, it is worth noting that pattern let occurrences are duplicated at join points, indicating that matching pattern lets have "met" there. These matching patterns are then unified and later simplified away with the *squashFloatings* function. Patterns must have right-hand sides that are equivalent (up to α-conversion) in order to be considered matching.

For example, if the following two let bindings are found in separate branches of the expression tree:

$\textbf{let}\ a@(b@(c, d), e) = \langle\!\langle rhs \rangle\!\rangle\ \textbf{in}\ t1$
$\textbf{let}\ a@(b, c@(d, e)) = \langle\!\langle rhs \rangle\!\rangle\ \textbf{in}\ t2$

they will meet at the least join point of their two branches, and be unified into

$$\begin{array}{l} \textbf{let}\ f@(g@(h, i), j@(k, l)) = \langle\!\langle rhs \rangle\!\rangle\ \textbf{in} \\ \quad ...\ t1\ [a := f, b := g, c := h, d := i, e := j]\ ... \\ \quad ...\ t2\ [a := f, b := g, c := j, d := k, e := l]\ ... \end{array}$$

We are currently working on further expanding the pattern let floating optimisation such that they can not only be floated up expressions, but also across function calls. By floating pattern lets across function calls, we can avoid even more duplicated computation through sharing.

This feature is implemented by splitting the pattern lets at the root of functions into separate pattern lets and a body. By creating secondary functions that take the variables bound by pattern lets and make them explicit arguments to a separate function, we can abstract the patterns across function calls.

7.4 Application

The `--abstract-plet` feature is necessary to split functions into two, with the let-bound variables abstracted out into function arguments of the second function. An example of this is shown in Figure 7.2. This abstraction is what allows cross-call floating to occur.

```
--- compile/MAlonzo/Code/Data/Fin/VecCat/Pullback.hs
+++ compile-abstract/MAlonzo/Code/Data/Fin/VecCat/Pullback.hs
 name60 = "Data.Fin.VecCat.Pullback.FinPB\8320._.s"
 d60 v0 v1 v2 v3 v4 v5 v6 v7 = du60 v1 v2 v3 v4 v5
 du60 v0 v1 v2 v3 v4
   = let v5@(MAlonzo.Code.Data.Product.C30 v6 v7)
           = coe du58 v0 v1 v2 v3 v4 in
     coe v6
+dv60 v0 v1 v2 v3 v4 v5 v6 v7 = coe v6
 name70 = "Data.Fin.VecCat.PullbackB.FinPB\8320._.v"
 d70 v0 v1 v2 v3 v4 v5 v6 v7 = du70 v1 v2 v3 v4 v5
 du70 v0 v1 v2 v3 v4
   = let v5@(MAlonzo.Code.Data.Product.C30 v6
       v7@(MAlonzo.Code.Data.Product.C30 v8 v9))
           = coe du58 v0 v1 v2 v3 v4 in
     coe v9
+dv70 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 = coe v9
```

Figure 7.2: Unified difference of the Pullback module compiled without and then with `--abstract-plet`.

As readers may have noticed inspecting Figure 6.2 in the preceding chapter, there are 4 pattern let bindings for the same `v2` variable within the `d4788` function. This is a perfect opportunity for floating pattern lets, to create sharing where there formerly was none. Figure 7.3 shows the result of applying `--float-plet` to this compilation, resulting in the `v2` bindings floating above the shared function call.

```
--- compile-inline-genplet/MAlonzo/Code/GraTra/Graphs.hs
+++ compile-inline-genplet-float/MAlonzo/Code/GraTra/Graphs.hs
@@ -14862,32 +15090,16 @@
 name4788 = "GraTra.Graphs.Coproducts-Graph.coproduct"
 d4788 v0 v1
-  = coe
-      MAlonzo.Code.Categoric.FinColimits.Coproduct.C8807
-      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
+  = let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
               = coe
                   MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
                   MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
                   MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       coe v3 v0 v1)
-      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
+    dv4788 v0 v1 v2 v3 v4 v5 v6
+dv4788 v0 v1 v2 v3 v4 v5 v6
               = coe
-                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
-                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
-                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       coe v4 v0 v1)
-      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
-              = coe
-                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
-                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
-                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       coe v5 v0 v1)
-      (let v2@(MAlonzo.Code.Categoric.FinColimits.Coproduct.C9169 v3 v4 v5 v6)
-              = coe
-                  MAlonzo.Code.Data.Algebra.Generalised.Hom.Coproduct.du1112
-                  MAlonzo.Code.Data.Fin.VecCat.d26 MAlonzo.Code.Data.Fin.VecCat.d3920
-                  MAlonzo.Code.Data.Algebra.Signature.Graph.d20 in
-       coe v6 v0 v1)
+      MAlonzo.Code.Categoric.FinColimits.Coproduct.C8807 (coe v3 v0 v1)
+      (coe v4 v0 v1) (coe v5 v0 v1) (coe v6 v0 v1)
```

Figure 7.3: Unified difference of the Triangle3sPB module compiled without and then with `--float-plet`.

In Figure 7.4 we can see the result of cross-call floating on the Pullback module. Notice that without cross-call floating, a single call to `du78` would result in two unshared calls to `du58`, one via `du60` and one via `du70`. With cross-call floating, `du78` calls `du58` only once, then passes the results via the additional parameters to `dv78`, which in turn shares the values with both `dv60` and `dv70`.

```
--- compile-abstract/MAlonzo/Code/Data/Fin/VecCat/PullbackB.hs
+++ compile-cross/MAlonzo/Code/Data/Fin/VecCat/PullbackB.hs
 name78 = "Data.Fin.VecCat.PullbackB.FinPB\8320._.r\8320"
 d78 v0 v1 v2 v3 v4 v5 v6 v7 = du78 v1 v2 v3 v4 v5 v6
 du78 v0 v1 v2 v3 v4 v5
+  = let v6@(MAlonzo.Code.Data.Product.C30 v7
+        v8@(MAlonzo.Code.Data.Product.C30 v9 v10))
+            = coe du58 v0 v1 v2 v3 v4 in
+    dv78 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10
+dv78 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10
   = coe
-       addInt (coe du60 v0 v1 v2 v3 v4)
+       addInt (coe dv60 v0 v1 v2 v3 v4 v6 v7 v8)
        (coe
           MAlonzo.Code.Data.Fin.du18
           (coe
              MAlonzo.Code.Data.Fin.VecCat.PullbackVecUtils.du302
-             (coe du70 v0 v1 v2 v3 v4) v5))
+             (coe dv70 v0 v1 v2 v3 v4 v6 v7 v8 v9 v10) v5))
```

Figure 7.4: Unified difference of the Pullback module compiled without and then with `--cross-call`.

Chapter 8

Conclusion

In this chapter, we discuss various aspects of our optimisations in summary. In Section 8.1, we assess the strengths and weaknesses of the main contributions. In Section 8.2, we discuss future work that could follow this project. Finally, in Section 8.3, we draw conclusions from the thesis and give some closing remarks.

8.1 Assessment of the Contributions

The contributions described herein have a number of strengths that will serve the Agda development community.

Primarily, as shown by the results of the applications of our optimisations, our optimisations will reduce the runtime execution and heap allocation requirements of many Agda programs, and do not show adverse effects in any of the tests we have performed. Our projection inlining optimisation alone yields a 10-20% improvement in execution time in our tests. The additional optimisations for case squashing, pattern let generationg, and pattern let floating within functions do not yet result in measurably significant benefits in memory usage or run time, however, they are important transformations in preparation for cross-call floating. Our cross-call floating optimisation is currently in development, and we expect that the sharing benefit generated by its implementation will have the greatest impact.

Secondarily, this thesis may also serve as a detailed documentation of the portions of the Agda compiler and GHC backend that are necessary to understand for incorporating future optimising transformations. We hope that future contributors to Agda will find this presentation of our study of the Agda compiler useful in implementing their own desired optimisations.

Although the net effect of our optimisations is a positive one, there are still a number of weaknesses that warrant consideration.

A clear weakness of our case squashing optimisation is its isolated implementation. Because it was

developed as an independent transformation, it requires an additional traversal of the treeless terms to execute. Further, some of the logic built to deal with the handling of De Bruijn indices would have been avoidable had it been built as part of an existing set of optimisations that had similar optimisation helper functions already developed. As presented in Subsection 3.2.3, an independent version of case squashing has since been developed and introduced into the Agda compiler, as part of the *Agda.Compiler.Treeless.Simplify* module, which addresses both of these weaknesses.

Both the projection inlining and case squashing optimisations make use of accumulated environment parameters, which can be handled more modularly and appropriately using monads. This potential for code refactoring is discussed further in Section 8.2.

8.2 Future Work

In our implementations of projection inlining and case squashing, we noted that environments of relevant information were carried through graph traversal. For projection inlining, this was an environment of previously inlined projections is maintained to avoid looping on recursive inlining calls. For case squashing, this was an environment of previously met cases expressions. These environments are currently maintained as a list objects, passed from function call to function call. For modularity and maintainability of the code, these environments would be better refactored into reader monad transformers, which would allow an inherited environment to be bound to the function results and passed through to subcomputations via the given monad.

Work will also continue on the cross-call floating optimisation described in Chapter 7, which we expect to yield the most significant sharing benefits when it is fully functional.

Additionally, our floating optimisations would benefit from recognizing single-alternative case expressions which are not immediately nested within an enclosing let expression. In the future, we also seek to expand this optimisation's transformation to generate "dummy" pattern let expressions around said case expressions before executing the floating transformations.

Further testing of all optimisations on a greater variety of Agda codebases is also a necessary next step before they can be safely integrated with a stable release branch of the compiler.

8.3 Closing Remarks

We have implemented, tested, and profiled a series of optimisations to the Agda compiler which improve execution time and reduce memory usage for many of the Agda programs tested, and have no negative performance effects in any of our tests.

Our optimisations serve previously unmet needs of our team as well as many other Agda developers by re-introducing some of the "lost" value sharing without affecting the type-theoretic semantics of Agda programs.

We hope that the development and discussion of these optimisations is useful to the Agda developer community, and may be helpful for future contributors interested in implementing new optimisations for Agda.

Appendix A

ToTreeless.hs (abridged)

The abridged code listing below for the the *Agda.Compiler.ToTreeless* module documents our projection inlining optimisation. This optimisation replaces every call to a function that is a proper projection with its function body. The transformation occurs during the translation to *Treeless* syntax.

```
{-# LANGUAGE CPP #-}
module Agda.Compiler.ToTreeless
  (toTreeless
  , closedTermToTreeless
  ) where
```

closedTermToTreeless is called to transform the Agda internal syntax to *TTerm*s. It calls *substTerm*, which we show a segment of below indicating the point at which *maybeInlineDef* is called.

```
closedTermToTreeless :: I.Term → TCM C.TTerm
closedTermToTreeless t = do
  substTerm [] t `runReaderT` initCCEnv

substTerm :: ProjInfo → I.Term → CC C.TTerm
substTerm inlinedAncestors term = normaliseStatic term >>= λterm →
  case I.ignoreSharing $ I.unSpine term of
    {-... -}
    I.Def q es → do
      let args = fromMaybe __IMPOSSIBLE__ $ I.allApplyElims es
      maybeInlineDef inlinedAncestors q args
    {-... -}
```

We create the datatype *ProjInfo* for maintaining an environment of previously inlined definitions.

```
type ProjInfo = [(I.QName, (I.Args, Definition))]
```

By modifying *maybeInlineDef* with an additional guard for *isProperProjection fun* $\wedge$ *doInlineProj*, we call the existing *doinline* function, adapted to account for the already inlined ancestors environment.

```
maybeInlineDef :: ProjInfo → I.QName → I.Args → CC C.TTerm
maybeInlineDef inlinedAncestors q vs =
  ifM (lift $ alwaysInline q) (doinline inlinedAncestors) $ do
    lift $ cacheTreeless q
    def ← lift $ getConstInfo q
    doInlineProj ← optInlineProj <$> lift commandLineOptions
    case theDef def of
      fun@Function{} | fun ^. funInline → doinline []
                     | isProperProjection fun ∧ doInlineProj
                     → do
                       lift $ reportSDoc "treeless.inline" 20 $
                         text "- inlining projection" $$ prettyPure (defName def)
                       doinline inlinedAncestors
                     | otherwise → defaultCase
      _ → C.mkTApp (C.TDef C.TDefDefault q) <$> substArgs inlinedAncestors vs
  where
    updatedAncestors = do
      def ← lift $ getConstInfo q
      return $ (q, (vs, def)) : inlinedAncestors
    doinline inlinedAncestors = do
      ancestors ← updatedAncestors
      case (q `lookup` inlinedAncestors) of
        Nothing → C.mkTApp <$> inline q <*> substArgs ancestors vs
        Just _ → defaultCase
    inline :: QName → CC C.TTerm
    inline q = lift $ toTreeless' q
    defaultCase = do
          _ ← lift $ toTreeless' q
          used ← lift $ getCompiledArgUse q
          let substUsed False _ = pure C.TErased
              substUsed True arg = substArg inlinedAncestors arg
          C.mkTApp (C.TDef C.TDefDefault q) <$>
            sequence [substUsed u arg | (arg, u) ← zip vs $ used ++ repeat True]
substArgs :: ProjInfo → [Arg I.Term] → CC [C.TTerm]
substArgs = traverse ∘ substArg
substArg :: ProjInfo → Arg I.Term → CC C.TTerm
substArg inlinedAncestors x | erasable x = return C.TErased
  | otherwise                            = substTerm inlinedAncestors (unArg x)
```

Appendix B

Case Squash.hs

The following is a listing for our case squashing optimisation, which we developed as a separate module, *Agda.Compiler.Treeless.CaseSquash*. The *squashCases* function in this module removes repeated case expressions that are nested and match on the same variable. It is called as part of the pipeline of *ToTreeless* optimisations in the separate Agda compiler branch we maintain for this project, which does not include the later implemented case squashing simplification now present in the Agda stable branches.

```
{-# LANGUAGE CPP, PatternGuards #-}
module Agda.Compiler.Treeless.CaseSquash (squashCases) where
import Agda.Syntax.Abstract.Name (QName)
import Agda.Syntax.Treeless
import Agda.TypeChecking.Substitute
import Agda.TypeChecking.Monad as TCM
import Agda.Compiler.Treeless.Subst
# include "undefined.h"
import Agda.Utils.Impossible
```

Eliminates case expressions where the scrutinee has already been matched on by an enclosing parent case expression.

```
squashCases :: QName → TTerm → TCM TTerm
squashCases q body = return $ dedupTerm [] body
```

Case scrutinee (De Bruijn index) with alternative match for that expression, made up of qualified name of constructor and a list of its arguments (also as De Bruijn indices)

```
type CaseMatch = (Int, (QName, [Int]))
```

Environment containing *CaseMatch*es in scope.

```
type Env = [CaseMatch]
```

Recurse through *TTerm*s, accumulting environment of case alternatives matched and replacing repeated cases. De Bruijn indices in environment should be appropriatedly shifted as terms are traversed.

```
dedupTerm :: Env → TTerm → TTerm
  -- Increment indices in scope to account for newly bound variable
dedupTerm env (TLam tt) = TLam (dedupTerm (shiftIndices (+1) <$> env) tt)
dedupTerm env (TLet tt1 tt2) = TLet (dedupTerm env tt1)
  (dedupTerm (shiftIndices (+1) <$> env) tt2)
  -- Check if scrutinee is already in scope
dedupTerm env body@(TCase sc t def alts) = case lookup sc env of
    -- If in scope with match then substitute body
  Just match → caseReplacement match body
    -- Otherwise add to scope in alt branches
  Nothing → TCase sc t
    (dedupTerm env def)
    (map (dedupAlt sc env) alts)
  -- Continue traversing nested terms in applications
dedupTerm env (TApp tt args) = TApp (dedupTerm env tt) (map (dedupTerm env) args)
dedupTerm env body = body
```

Find the alternative with matching name and replace case term with its body (after necessary substitutions), if it exists.

```
caseReplacement :: (QName, [Int]) → TTerm → TTerm
caseReplacement (name, args) tt@(TCase _ _ _ alts)
  | Just (TACon _ ar body) ← lookupTACon name alts
  = varReplace [ar − 1, ar − 2 .. 0] args body
caseReplacement _ tt = tt
```

Lookup *TACon* in list of *TAlt*s by qualified name

```
lookupTACon :: QName → [TAlt] → Maybe TAlt
lookupTACon match ((alt@(TACon name ar body)) : alts) | match ≡ name = Just alt
lookupTACon match (_ : alts) = lookupTACon match alts
lookupTACon _ [] = Nothing
```

Introduce new constructor matches into environment scope

$$
\begin{aligned}
&\mathit{dedupAlt} :: \mathit{Int} \to \mathit{Env} \to \mathit{TAlt} \to \mathit{TAlt} \\
&\mathit{dedupAlt}\ \mathit{sc}\ \mathit{env}\ (\mathit{TACon}\ \mathit{name}\ \mathit{ar}\ \mathit{body}) = \\
&\quad \textbf{let}\ \mathit{env}' = (\mathit{sc} + \mathit{ar}, (\mathit{name}, [\mathit{ar} - 1, \mathit{ar} - 2 \,..\, 0])) : (\mathit{shiftIndices}\ (+\mathit{ar}) <\$> \mathit{env}) \\
&\quad \textbf{in}\ \mathit{TACon}\ \mathit{name}\ \mathit{ar}\ (\mathit{dedupTerm}\ \mathit{env}'\ \mathit{body}) \\
&\mathit{dedupAlt}\ \mathit{sc}\ \mathit{env}\ (\mathit{TAGuard}\ \mathit{guard}\ \mathit{body}) = \mathit{TAGuard}\ \mathit{guard}\ (\mathit{dedupTerm}\ \mathit{env}\ \mathit{body}) \\
&\mathit{dedupAlt}\ \mathit{sc}\ \mathit{env}\ (\mathit{TALit}\ \mathit{lit}\ \mathit{body}) = \mathit{TALit}\ \mathit{lit}\ (\mathit{dedupTerm}\ \mathit{env}\ \mathit{body})
\end{aligned}
$$

Shift all De Bruijn indices in a case match according to provided function on integers

$$
\begin{aligned}
&\mathit{shiftIndices} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{CaseMatch} \to \mathit{CaseMatch} \\
&\mathit{shiftIndices}\ f\ (\mathit{sc}, (\mathit{name}, \mathit{vars})) = (f\ \mathit{sc}, (\mathit{name}, \mathit{map}\ f\ \mathit{vars}))
\end{aligned}
$$

Substitute list of current De Bruijn indices for list of new indices in a term

$$
\begin{aligned}
&\mathit{varReplace} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{TTerm} \to \mathit{TTerm} \\
&\mathit{varReplace}\ (\mathit{from} : \mathit{froms})\ (\mathit{to} : \mathit{tos}) = \mathit{varReplace}\ \mathit{froms}\ \mathit{tos} \circ \mathit{subst}\ \mathit{from}\ (\mathit{TVar}\ \mathit{to}) \\
&\mathit{varReplace}\ [\,]\ [\,] = \mathit{id} \\
&\mathit{varReplace}\ \_\ \_ = \mathit{\_\_\_IMPOSSIBLE\_\_\_}
\end{aligned}
$$

Appendix C

Compiler.hs (abridged)

The following is an abridged subsection of *Agda.Compiler.MAlonzo.Compiler* module. This is the primary module of Agda's GHC backend, or MAlonzo, where we have implemented our generate pattern let optimisation.

The section of interest to us in this file is the *term* function, which translates *Treeless* terms to Haskell expressions. We have augmented this translation to perform pattern let generation, which avoids generating certain trivial case expressions in the Haskell output.

```
{-# LANGUAGE CPP, PatternGuards #-}
module Agda.Compiler.MAlonzo.Compiler where
```

The *term* function is called to extract the *TTerm* syntax to Haskell expressions, which modifying to include an additional guard for *TLet* _ (*TCase* 0 _ _ [*TACon* _ _ _]). The relevant segment of the *term* function is shown below.

```
term :: T.TTerm → CC HS.Exp
term tm0 = asks ccGenPLet >>= λgenPLet → case tm0 of
    {-... -}
  TLet _ (TCase 0 _ _ [TACon _ _ _])
    | genPLet
    , Just (PLet {pletNumBinders = numBinders, eTerm = TLet t1 tp}, tb) ← splitPLet tm0
    → do
      t1' ← term t1
      intros 1 $ λ[x] → do
        intros numBinders $ λxs → do
          tb' ← term tb
          p ← addAsPats (x : xs) 1 tp (HS.PVar x)
          return $ hsPLet p (hsCast t1') tb'
```

We introduce the following functions, *addAsPats* and *replacePats* to perform the pattern let generation.

In *addAsPats xs numBound tp pat*, recurse through *tp* to find all single constructor *TCase*s and replace *PVar*s of the same scrutinee with appropriate *PAsPat*s, until *TErased* is reached. *xs* contains all necessary variables introduce by the initial call in *term*, with *numBound* indicating the number of introduced of variables introduced by the caller used in *PApp*s and the top let.

```
addAsPats :: [HS.Name] → Nat → TTerm → HS.Pat → CC HS.Pat
addAsPats xs numBound
  tp@(TCase sc _ _ [TACon c cArity tp'])
  pat = case xs !!! (numBound − 1 − sc) of
    Just scName → do
      erased ← lift $ getErasedConArgs c
      hConNm ← lift $ conhqn c
      let oldPat = HS.PVar scName
      let vars = take cArity $ drop numBound xs
      let newPat = HS.PAsPat scName $ HS.PApp hConNm $
        map HS.PVar [x | (x, False) ← zip vars erased]
      let pat' = replacePats oldPat newPat pat
      addAsPats xs (numBound + cArity) tp' pat'
    Nothing → ___IMPOSSIBLE___
addAsPats _ _ TErased pat = return pat
addAsPats _ _ _ _ = ___IMPOSSIBLE___   -- Guaranteed by splitPLet
```

In *replacePats old new p*, replace all instances of *old* in *p* with *new*

```
replacePats :: HS.Pat → HS.Pat → HS.Pat → HS.Pat
replacePats old new p@(HS.PVar _) = if old ≡ p then new else p
replacePats old new (HS.PAsPat sc p) = HS.PAsPat sc $ replacePats old new p
replacePats old new p@(HS.PApp q pats) =
  HS.PApp q $ map (replacePats old new) pats
replacePats _ _ p = ___IMPOSSIBLE___   -- Guaranteed by addAsPats
```

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *17th POPL*, pages 31–46, New York, NY, USA, January 1990. ACM.

Agda. Agda documentation, 2017a. URL `http://agda.readthedocs.io/en/v2.5.2/index.html`.

Agda. The Agda package, 2017b. URL `https://hackage.haskell.org/package/Agda`.

Agda. Agda mailing list archives - [agda] about serialization, 2017c. URL `https://lists.chalmers.se/pipermail/agda/2017/009406.html`.

Agda. Agda documentation wiki, 2017d. URL `http://wiki.portal.chalmers.se/agda/agda.php`.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, England, 1992.

Marcin Benke. Alonzo—a compiler for Agda. In *Talk at Agda Implementors Meeting*, volume 6, 2007.

Ana Bove, Peter Dybjer, and Ulf Norell. *A Brief Overview of Agda – A Functional Language with Dependent Types*, pages 73–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-642-03359-9_6.

Olaf Chitil. Common subexpressions are uncommon in lazy functional languages. In Clack C., Hammond K., and Davie T., editors, *IFL '97: 9th International Workshop on Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 53–71. Springer, 1998. doi: 10.1007/BFb0055420.

Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

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*, 34: 381–392, 1972.

Olle Fredriksson and Daniel Gustafsson. A totally Epic backend for Agda. Chalmers University of Technology, 2011.

Wolfram Kahl. Relation-Algebraic Theories in Agda — RATH-Agda-2.2. Mechanically checked Agda theories, with 580 pages literate document output. http://relmics.mcmaster.ca/RATH-Agda/, January 2017. With contributions by Musa Al-hassy and Yuhang Zhao.

Simon Marlow and Simon Peyton Jones. *The Glasgow Haskell Compiler*. Lulu, January 2012. URL `https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/`.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, September 2007. See also `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

Ulf Norell. *Dependently Typed Programming in Agda*, pages 230–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-642-04652-0_5.

Simon Peyton Jones, Will Partain, and André Santos. Let-floating: Moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 1–12. ACM, May 1996. doi: 10.1145/232627.232630.

Iman Poernomo, John N Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs: The Curry–Howard Protocol*. Springer Science & Business Media, 2005.