CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Verified Function Inlining Optimization for the PureCake Compiler

Master's thesis in Computer science and engineering

Kacper Korban

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Verified Function Inlining Optimization for the PureCake Compiler

Kacper Korban

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Verified Function Inlining Optimization for the PureCake Compiler
Kacper Korban

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Verified Function Inlining Optimization for the PureCake Compiler
Kacper Korban
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis describes the implementation and formal verification of an inlining optimization pass for the PureCake compiler. PureCake is a verified optimizing compiler for a pure, lazy, functional programming language. The optimization is implemented and verified using the HOL4 theorem prover. The correctness proof of inlining is defined at two levels of abstraction. The meta-theory level and the compiler level. The implementation is heuristic agnostic and can be used with any inlining strategy. The thesis also describes a function specialization transformation. This transformation is unverified, but steps are described on how the current inlining implementation and methodology can be used to verify it.

Keywords: Formal Methods, Functional Programming, Formal Verification, Programming Languages, Compilers.

Acknowledgements

I would like to thank my supervisor Magnus Myreen for his support and guidance throughout this project. I would also like to thank the members of the PureCake team for sharing their knowledge of the project during weekly meetings. Especially Michael Norrish for his HOL4 tips and Hrutvik Kanabar for implementing the freshening pass.

Kacper Korban, Gothenburg, June 2023

Contents

List of Figures **xi**

1 Introduction **1**
1.1 Compilers . . . 1
1.1.1 Compiler Correctness . . . 1
1.1.2 PureCake . . . 1
1.1.3 Compiler optimizations . . . 1
1.2 Project Goal . . . 2
1.2.1 Function Inlining . . . 2
1.3 Thesis Outline . . . 2

2 Background **5**
2.1 HOL4 . . . 5
2.1.1 HOL4 Specific Definitions . . . 6
2.2 PureCake . . . 7
2.2.1 PureLang Definitions and Related Terms . . . 8
2.2.2 PureLang Expression Equivalence . . . 9
2.2.3 PureCake Proof Approach . . . 10

3 Verified Inlining at the Abstract Level **13**
3.1 Goal . . . 13
3.2 Proof strategy . . . 14
3.3 Single Inlining Relation . . . 15
3.3.1 Single inlining equivalence . . . 17
3.3.2 Deep Single Inlining Relation . . . 19
3.4 Multi-Inlining Relation . . . 21
3.4.1 Multi-inlining relation equivalence . . . 24
3.4.2 Multi-inlining relation correctness . . . 29

4 Verified Implementation of Inlining **31**
4.1 Implementation . . . 31
4.2 Implementation correctness proof . . . 33

5 Verified Specialization **37**
5.1 Goal . . . 37
5.2 Specializing approach . . . 38
5.3 Implementation . . . 38

6 Results **43**
6.1 Correctness Proof 43
6.2 Implemented Optimization Pass 44
6.3 Examples 44

7 Conclusions **47**
7.1 Summary of the completed work 47
7.2 Related Work 47
7.3 Future Work 48

Bibliography **51**

List of Figures

2.1 High-level summary of PureCake's intermediate languages and compilation passes. Used with the permission of the authors of the PLDI paper [2] 7

3.1 Single inlining relation 15
3.2 Deep single inlining relation 19
3.3 Multi-inlining relation 23
3.4 Definition of `no_shadowing` relation 26

4.1 `inline` function implementation 32

5.1 `const_call_args` function implementation 41

1

Introduction

1.1 Compilers

Compilers are critical parts of the software development process. And since we always assume that they work as expected, the goal of every compiler is to generate, in this order:

1. Correct code
2. Efficient code

1.1.1 Compiler Correctness

To address the first goal, research into verified compilers is becoming more and more prevalent. The first well-known verified compiler for a non-trivial programming language was CompCert C — a verified compiler for a subset of the C language. When it comes to functional programming languages, the best-known such compiler is CakeML [3] — a verified compiler for Standard ML.

1.1.2 PureCake

PureCake [2] is a verified compiler for a Haskell-like language. The language supported by PureCake is functional with lazy evaluation and monadic effects. The syntax supported by PureCake is indentation based to best mimic Haskell's syntax. PureCake also provides proof of its type soundness and the correctness of its Hindley-Milner style type inference. PureCake compiles the source code into CakeML code and it also contains formal proof of the correctness of its compilation process. So if we combine that with the proof of correctness for the source-to-machine-code compilation of CakeML, the whole PureCake compilation pipeline is formally verified to be correct.

1.1.3 Compiler optimizations

Research on compiler optimizations aims to address the second goal, *"Efficient code"*, from above. This part, despite being of a lesser priority, is more widely applied in compilers used in production. So the state-of-the-art on compiler optimizations is very high compared to compiler correctness verification. That is mainly because one

of the biggest reasons people moved away from low-level languages, is that compilers were able to generate more efficient low-level code than a human programmer.

1.2 Project Goal

The goal of this project is to add the function inlining optimization transformation and its correctness proof to the PureCake verified compiler.

1.2.1 Function Inlining

Pure lazy functional languages have the advantage of being easier to argue about thanks to referential transparency. But lazy evaluation can also take a toll on the runtime performance [8]. That is why it is crucial to perform more optimizations that will make their performance comparable to eagerly evaluated languages. More specifically in GHC — the most significant Haskell compiler — function inlining is one of the most essential optimizations [7].

Function inlining is a transformation that replaces a variable reference with a copy of its definition. For example, for the following expression written in Haskell:

```
let x = foo 1
in bar x
```

The definition of x can be copied and inserted into its use site as follows:

```
let x = foo 1
in bar (foo 1)
```

Such optimizations are important because they reduce the number of function calls and, perhaps more importantly, allow for other optimizations to work (e.g. dead code elimination). However, since every function body can be copied more than once, incautious implementations of the optimizations can come at the cost of program size. Therefore, heuristic algorithms are often used to determine when inlining is worth the cost of added code size.

Implementations of optimizations are a major source of bugs in compilers [9]. Therefore it is important for any such transformation to preserve the semantics of the program. And thanks to the fact that the PureCake compiler already consists of proof of its correctness, adding a semantics preservation proof of the optimization will make sure that the generated output code is still correct.

1.3 Thesis Outline

The report is structured in the following way: Chapter 2 introduces the necessary background to understand the technical aspects of the project. Next, Chapter 3 describes the approach to prove the correctness of the inlining optimization on an

abstract level — using a relation that encapsulates the syntactic transformation. Chapter 4 describes the implementation of the inlining optimization pass and the correctness proof of this implementation. Chapter 5 describes the approach of accommodating function specialization into the inlining pass and the implementation of the transformation. After that, Chapter 6 describes the results of the project and evaluates them. Finally, Chapter 7 summarises the work done and discusses related work as well as future work.

2

Background

This chapter describes the background necessary to understand the details of the project. Specifically, it provides a short introduction to HOL4, the interactive theorem prover used in the project, and the PureCake compiler, for which the optimization is designed.

2.1 HOL4

HOL4 is an interactive theorem prover for higher-order logic. It gives a programming environment to prove theorems and implement proof tools. It also provides automated provers and decision procedures, so that the developers can focus on the harder problems themselves.

HOL4 uses Standard ML as an interaction language, so defining datatypes, functions etc. is done using Standard ML code. All HOL definitions are essentially Standard ML expressions. HOL4 also has an extensive library of definitions and theorems, which are used in the development of proofs. This makes it faster for experienced users to develop proofs, as they can reuse common definitions and theorems. Though it also makes the entry barrier for new users higher, as they need to learn the library to be able to prove most theorems.

Proving theorems is done by combining sequences of higher-level combinators. Those combinators allow performing different types of rewrites on the assumptions and the goal. The most common tactics are the ones that allow for rewriting the goal using a given definition or theorem. Some common combinators allow for performing induction or case analysis on a term. Other noteworthy combinators match either the goal or the assumptions against a given theorem statement and give a specialized version of the theorem.

The interaction with HOL is mostly done via IDE plugins for Vim, Emacs and VSCode. The plugins allow for the management of the goal stack — the stack of currently active proofs. They also make the process of developing proofs more interactive, by allowing the developer to incrementally apply tactics to the sequent.

2.1.1 HOL4 Specific Definitions

The syntax of HOL4 is not necessary to understand the work done in this project since the theorems and definitions are shown in a mathematical notation. However, some HOL4-specific definitions and datatypes are used in the project, so it is worth explaining them.

- `list_rel R xs ys` — a predicate that defines a relation between two lists. It is true if the two lists `xs ys` are of the same length and the corresponding elements are related by the given relation `R`.
- `disjoint s1 s2` — a predicate between two sets. It is true if the two sets `s1 s2` are disjoint.
- `mem x xs` — a membership predicate between an element `x` and a list `xs`. It is true if the element `x` is in the list `xs`.
- `map2 f xs ys` — a function that takes a function `f` and two lists `xs ys` and applies the function to the corresponding elements of the two lists. It returns a list of the results.

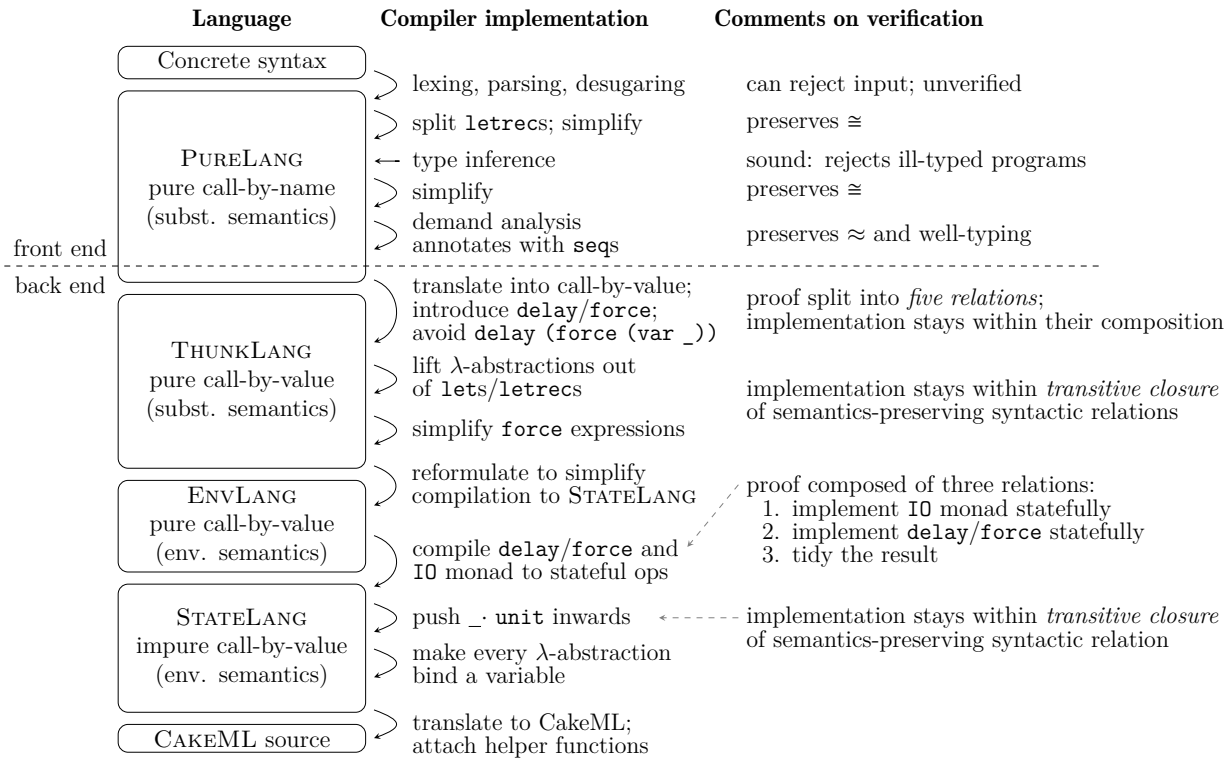


Figure 2.1: High-level summary of PureCake's intermediate languages and compilation passes. Used with the permission of the authors of the PLDI paper [2]

2.2 PureCake

PureCake is a verified compiler for a pure, lazy, functional programming language. It uses a Haskell-like syntax and type system. PureCake produces CakeML code, which allows for its compilation to be verified from the source to the machine code.

PureCake compilation pipeline consists of four intermediate languages. The overview of the compiler architecture is presented in Figure 2.1. For this project, the relevant language is the one used in the front end — PureLang. PureCake differentiates between the compiler language and the language used for transformation proofs. The declaration of the abstract syntax tree for the meta-theory level language is as follows:

Definition 1 *PureLang expressions*

```
exp =
    Var string
  | Prim op (exp list)
  | App exp exp
  | Lam string exp
  | Letrec ((string × exp) list) exp
```

The `App`, `Lam` and `Var` constructors are respectively used for function application, lambda abstraction and variable reference. They should be familiar to any reader

familiar with lambda calculus. `Prim` is used for performing primitive operations. Lastly, the `Letrec` constructor is a special kind of let binding, that is allowed to be recursive.

An additional thing of note is that the meta-theory level language does not have a constructor for `Let`. Non-recursive `Let` bindings are expressed as a combination of `Lam` and `App`. The following shows the original and translated version of the same expression:

```
let x = e1 in e2
-- translates to
(\x -> e2) e1
```

The abstract syntax tree used for compiler implementations is as follows:

Definition 2 *Compiler-level language expressions*

$$
\begin{aligned}
&\alpha\ \mathit{cexp} = \\
&\quad \mathit{Var}\ \alpha\ \mathit{mlstring} \\
&\mid \mathit{Prim}\ \alpha\ \mathit{cop}\ (\alpha\ \mathit{cexp}\ \mathit{list}) \\
&\mid \mathit{App}\ \alpha\ (\alpha\ \mathit{cexp})\ (\alpha\ \mathit{cexp}\ \mathit{list}) \\
&\mid \mathit{Lam}\ \alpha\ (\mathit{mlstring}\ \mathit{list})\ (\alpha\ \mathit{cexp}) \\
&\mid \mathit{Let}\ \alpha\ \mathit{mlstring}\ (\alpha\ \mathit{cexp})\ (\alpha\ \mathit{cexp}) \\
&\mid \mathit{Letrec}\ \alpha\ ((\mathit{mlstring} \times \alpha\ \mathit{cexp})\ \mathit{list})\ (\alpha\ \mathit{cexp}) \\
&\mid \mathit{Case}\ \alpha\ (\alpha\ \mathit{cexp})\ \mathit{mlstring} \\
&\quad ((\mathit{mlstring} \times \mathit{mlstring}\ \mathit{list} \times \alpha\ \mathit{cexp})\ \mathit{list}) \\
&\quad (((\mathit{mlstring} \times \mathit{num})\ \mathit{list} \times \alpha\ \mathit{cexp})\ \mathit{option}) \\
&\mid \mathit{NestedCase}\ \alpha\ (\alpha\ \mathit{cexp})\ \mathit{mlstring}\ \mathit{cepat}\ (\alpha\ \mathit{cexp}) \\
&\quad ((\mathit{cepat} \times \alpha\ \mathit{cexp})\ \mathit{list})
\end{aligned}
$$

The constructors are the same as the ones for the meta-theory level language, with small differences:

- `App` and `Lam` can take multiple arguments/parameters respectively.
- `Let` is added as a constructor.
- `Case` and `NestedCase` constructors are added for pattern matching. For the meta-theory level language, pattern matching is translated into a combination of primitive equality checks and non-recursive bindings.

2.2.1 PureLang Definitions and Related Terms

Several definitions are used in the proofs for this project that should be familiar to anyone with a background in compilers and programming languages. This section defines those terms and explains their meaning in the context of PureLang.

- `freevars` — a function that returns the set of free variables in an expression. A variable is free if it is not bound by a lambda abstraction or a `Letrec` binding. This function is defined for the meta-theory level language.

- `boundvars` — a function that returns the set of bound variables in an expression. This function is also defined for the meta-theory level language.
- `closed` — a predicate that returns true if an expression is closed, i.e. it has no free variables. This predicate is defined for the meta-theory level language.
- `exp_of` — a function that returns the expression corresponding to a compiler-level expression. There is no inverse function that returns the compiler-level expression corresponding to a meta-theory level expression.

2.2.2 PureLang Expression Equivalence

Before explaining the notion of expression equivalence, it is necessary to explain the notion of weak-head evaluation. Weak-head evaluation is a form of evaluation that only reduces to the outermost constructor or closure. For example, the following expression (where `Const` is a constructor):

```
(\x -> Const (x + 1)) 3
```

can be weak-head evaluated to:

```
Const (3 + 1)
```

The difference between weak-head evaluation and normal evaluation is that normal evaluation would reduce the expression further to:

```
Const 4
```

But since weak-head evaluation only reduces to the outermost constructor, it does not reduce the addition further.

Weak-head evaluation is the core of laziness in Haskell and PureCake. Whenever a term is forced in PureCake, it is weak-head evaluated. This way, computations are only evaluated when they are needed.

PureCake defines a notion of expression equivalence for PureLang. This relation is denoted with the name `exp_eq` or the infix operator ≅. This relation is defined using the notion of `applicative bisimilarity` — a notion of equivalence for lambda calculus. It informally states that, if two expressions are related by applicative bisimilarity, then whenever one of them can reduce its outermost redex and produce a weak-head value, the other expression should also be able to perform the same reduction and produce a corresponding weak-head value.

This relation was not used directly in the proofs for this project. Instead, other lemmas for this relation were used. One such lemma proved as part of the development of this project is the following:

Theorem 1 *exp_eq_subst_IMP_exp_eq*

$$\begin{array}{l}\vdash (\forall f.\ (\forall n\ v.\ \mathit{lookup}\ f\ n = \mathit{Some}\ v \Rightarrow \mathit{closed}\ v) \wedge \\ \quad \mathit{freevars}\ x \subseteq \mathit{domain}\ f \wedge \mathit{freevars}\ y \subseteq \mathit{domain}\ f \Rightarrow \\ \quad\ \mathit{subst}\ f\ x \cong \mathit{subst}\ f\ y) \Rightarrow \\ \quad x \cong y\end{array}$$

This lemma states that: for every map of expressions to substitute (`f`) if:

- it covers all the free variables of `x` and `y` and
- the results of substituting `x` and `y` with `f` are equivalent

then `x` and `y` are equivalent.

A similar lemma also exists for weak head evaluation:

Theorem 2 *eval_wh_IMP_exp_eq*

$$\begin{array}{l}\vdash (\forall f.\ (\forall n\ v.\ \mathit{lookup}\ f\ n = \mathit{Some}\ v \Rightarrow \mathit{closed}\ v) \wedge \\ \quad \mathit{freevars}\ x \subseteq \mathit{domain}\ f \wedge \mathit{freevars}\ y \subseteq \mathit{domain}\ f \Rightarrow \\ \quad\ \mathit{eval\_wh}\ (\mathit{subst}\ f\ x) = \mathit{eval\_wh}\ (\mathit{subst}\ f\ y)) \Rightarrow \\ \quad x \cong y\end{array}$$

This theorem states that for every `f` that also satisfies the same free variable condition from above, if `x` and `y` are equal after weak head evaluation, then `x` and `y` are equivalent.

Another important type of theorem used in expression equivalence proofs is the congruence theorem. Congruence theorems state that composing equivalent expressions gives expressions that are also equivalent. An example of a congruence theorem for the `App` constructor is the following:

Theorem 3 *exp_eq_App_cong* — *expression equivalence congruence theorem for App*

$$\vdash f \cong f' \wedge e \cong e' \Rightarrow \mathit{App}\ f\ e \cong \mathit{App}\ f'\ e'$$

This theorem states that if `f` and `f'` are equivalent, and `e` and `e'` are equivalent, then `App f e` and `App f' e'` are equivalent. Similar theorems are defined for all the other constructors. The congruence theorems are extremely useful for proving the equivalence of complex expressions, as they allow for the proof to be broken down into smaller, more manageable parts.

2.2.3 PureCake Proof Approach

When it comes to proofs, most of the proof logic is decoupled from the implementation functions. Instead of verifying specific functions that implement a compiler transformation, relations are defined that capture the core idea of the required

transformation. That way the correctness of the syntactic transformation performed by the compiler pass is proved, and whenever the implementation of the compilation phase has to change, the core of the proof stays the same.

For example, let's take a specific compiler transformation — the transformation that creates fresh variable names for all bound variables in an expression. The steps for designing this compiler transformation are as follows:

1. Define a relation `R` that captures the idea of the transformation. The meaning of this transformation will be: if `R e1 e2` holds, then `e2` can be the result of freshening variables in `e1`.
2. Prove that if `R e1 e2` holds, then `e1` is equivalent to `e2`.
3. Define the function `freshen` that implements the transformation.
4. Prove that the function satisfies the relation `R`. In other words, prove that `R e1 (freshen e1)` holds.
5. Use the previous two proofs to prove that `e1` is equivalent to `freshen e1`.

This way, the semantic part of the proof stays decoupled from the implementation proof and when the implementation has to be changed, only the last two steps have to be redone.

3

Verified Inlining at the Abstract Level

This chapter describes work done on the inlining optimization. It concentrates on the approach to prove the correctness of the inlining transformation on an abstract level. It shows three approaches for defining the correctness relation: a single inlining relation, a deep inlining relation and a multi-inlining relation.

3.1 Goal

The core idea for the inlining optimization is noticing that in certain cases the execution of a program will be faster if the runtime didn't have to make an unnecessary jump to a function call. Considering the following example:

```
let f = (\x -> 5)
in f 1
```

It can be easier to write code that uses multiple bindings, specifically ones that are referenced a single time. That is specifically the case with functional programming that utilize immutability.

Looking back at the example, one can notice that inlining is essentially beta-reduction since the given code can be expressed with just functional abstraction and function application, as follows:

```
(\f -> f 1) (\x -> 5)
```

Unfortunately, it is not always the case, that the compiler can perform this beta-reduction. When considering the following code snippet:

```
let f = (\x -> y + 5)
in
  let y = 3
  in f 1
```

It should be clear that even though f is only referenced once and looks like a prime candidate for inlining, performing the outer-most beta-reduction would change the semantics of the program. This example demonstrates the famous variable capture

problem, where inlining f would change y from being a free variable to a variable bound in an inner scope.

Another important intricacy that differentiates inlining from simple beta-reduction is that for a single binding, it can be possible for it to inline one of its occurrences and not the other. And because of that inlining should not remove the bindings to the original function definition. One example that can illustrate this case is the following:

```
let f = (\x -> 5)
in f 1 : map f xs ++ map f ys
```

In this case, the compiler should only inline the references in which the function is immediately applied. Otherwise instead of one function definition, the compiler would have to generate two. So the ideal inlined code snippet for this example would look like the following:

```
let f = (\x -> 5)
in (\x -> 5) 1 : map f xs ++ map f ys
```

Of course, if all references to a variable get inlined, the original declaration will become unused. In that case, a different optimization pass should be responsible for that — dead code elimination pass.

To summarise, inlining is a crucial optimization technique in functional programming that can significantly enhance the execution speed of a program by reducing unnecessary function calls. However, the process isn't as straightforward as it might appear due to complexities like the variable capture problem and the fact that a single binding can't always be inlined in all its occurrences.

3.2 Proof strategy

The strategy for proving the correctness of the inlining pass follows the same pattern as described in Section 2.2.

The steps are as follows:

1. Define a relation `subst_rel` that relates an expression `e` with every possible result of inlining variables in `e`.
2. Prove that for every `e1` and `e2`, if `subst_rel e1 e2` holds, then `e1` is equivalent to `e2`.
3. Write the `inline` function that implements the inlining pass.
4. Prove that for every `e` the result of `inline e` satisfies the relation `subst_rel e (inline e)`.
5. Use the previous theorem to prove that for every `e` the result of `inline e` is

Definition 3 *subst_rel_Var*

$\vdash$ *subst_rel* $v\ x\ (\mathit{Var}\ v)\ x$

Definition 4 *subst_rel_refl*

$\vdash$ *subst_rel* $v\ x\ t\ t$

Definition 5 *subst_rel_App*

$\vdash$ *subst_rel* $v\ x\ t_1\ u_1 \wedge$ *subst_rel* $v\ x\ t_2\ u_2 \Rightarrow$
subst_rel $v\ x\ (\mathit{App}\ t_1\ t_2)\ (\mathit{App}\ u_1\ u_2)$

Definition 6 *subst_rel_Lam*

$\vdash$ *subst_rel* $v\ x\ t\ u \wedge v \neq w \wedge w \notin$ *freevars* $x \Rightarrow$
subst_rel $v\ x\ (\mathit{Lam}\ w\ t)\ (\mathit{Lam}\ w\ u)$

Definition 7 *subst_rel_Letrec*

$\vdash$ *list_rel* $(\lambda(n,t_1)\ (m,u_1).\ n = m \wedge$ *subst_rel* $v\ x\ t_1\ u_1 \wedge n \notin$ *freevars* $x)\ xs\ ys \wedge$
subst_rel $v\ x\ t\ u \wedge \neg$*mem* $v\ ($*map fst* $xs) \Rightarrow$
subst_rel $v\ x\ (\mathit{Letrec}\ xs\ t)\ (\mathit{Letrec}\ ys\ u)$

Definition 8 *subst_rel_Prim*

$\vdash$ *list_rel* $($*subst_rel* $v\ x)\ xs\ ys \Rightarrow$ *subst_rel* $v\ x\ (\mathit{Prim}\ p\ xs)\ (\mathit{Prim}\ p\ ys)$

Figure 3.1: Single inlining relation

equivalent to `e`.

The next sections show alternative definitions of the `subst_rel` relation that have been defined and verified to preserve equivalence.

3.3 Single Inlining Relation

The first part of proving the correctness of inlining is, to show that the code after inlining a single variable is equivalent to the one before inlining. Provided of course that the inlined expression doesn't capture any unwanted variables. This proof can be expressed elegantly thanks to the expression equivalence relation defined for PureLang.

The base relation for single variable inlining is `subst_relv x t u`, an inductive relation representing the possibility to inline `x` for `v` in `t` to get `u`. The definition of this relation is shown in Figure 3.1.

The informal meaning of this inductive relation is that if an instance of `subst_rel v`

`x t u` can be constructed, it means that an expression `x` under the name of `v` can be inlined in `t` and get `u` as a result. The most important constructor is `subst_rel_Var`, which expresses the operation of inlining the given expression. The other constructors essentially just allow for performing inlining deeper inside the tree. Considering the example:

```
let f = (\x -> 5)
in f 1
```

Mixing concrete and abstract syntax to make the code easier to read. The following instance can be constructed:

```
subst_rel f (\x -> 5) (f 1) ((\x -> 5) 1)
```

To do so, the only constructors needed are the `refl`, `App` and `Var` ones. More specifically, to construct this relation one has to first use the `App` case:

Theorem 4 *subst_rel_App*

$$\vdash \mathit{subst\_rel}\ v\ x\ t_1\ u_1 \wedge \mathit{subst\_rel}\ v\ x\ t_2\ u_2 \Rightarrow \mathit{subst\_rel}\ v\ x\ (\mathit{App}\ t_1\ t_2)\ (\mathit{App}\ u_1\ u_2)$$

It in turn takes as an assumption,

```
subst_rel f (\x -> 5) f (\x -> 5)
```

and

```
subst_rel f (\x -> 5) 1 1
```

i.e. relations that show that the same substitution can be performed in both the function and the argument. The latter relation is trivial since we can just use reflexivity of `subst_rel` to prove it.

Theorem 5 *subst_rel_refl*

$$\vdash \mathit{subst\_rel}\ v\ x\ t\ t$$

The former is an interesting case because it's the one where inlining is performed. Here the `Var` constructor is used

Theorem 6 *subst_rel_Var*

$$\vdash \mathit{subst\_rel}\ v\ x\ (\mathit{Var}\ v)\ x$$

Its definition is quite simple, it just states that when inlining variable `v` the reference to `v` can be inlined i.e. replaced by the expression it binds. Using this rule concludes creating the desired relation instance.

The derivation tree for this relation instance is as follows:

```
———————————————— subst_rel_Var ———————————————— subst_rel_refl
  subst_rel                       subst_rel
  f (\x -> 5)                     f (\x -> 5)
  f (\x -> 5)                     1 1
———————————————————————————————————————————————— subst_rel_App
             subst_rel
             f (\x -> 5)
             (f 1) ((\x -> 5) 1)
```

3.3.1 Single inlining equivalence

Using this relation, the core of a single inlining of a specific expression is formulated.

Theorem 7 *Equivalence of inlining a single definition for* `subst_rel` *relation*

$\vdash v \notin \texttt{freevars}\ x \wedge \texttt{subst\_rel}\ v\ x\ t\ u \Rightarrow \texttt{Let}\ v\ x\ t \cong \texttt{Let}\ v\ x\ u$

This theorem expresses that, for any `v`, `x`, `t`, `u`, if `x` can be inlined for `v` in `t` to get `u`, then `Let v x t` is equivalent to `Let v x u`. Worth noticing is that this is specifically true for any `t`, `u` where `t == u`, which essentially means that this theorem shows that the equivalence is true whenever inlining can be performed and does not necessarily have to be performed.

The proof of this theorem is quite long and constitutes a big part of the work on this project. The following is a brief explanation of the proof.

The whole proof was done by induction on the `subst_rel` relation. The strategy used in most non-trivial cases was to first push down the `Let` constructor into every branch of the expression tree. This was done so that the subtrees will match the inductive hypotheses. Then use the congruence rule for that type of expression to reduce the proof case to having to prove the equivalence of said subtrees. Finally, use the induction hypothesis to prove the equivalence of the subtrees.

For example, starting with the original sequent for the `App` case:

```
0.  subst_rel v x t u
1.  v ∉ freevars x ⇒ Let v x t ≅ Let v x u
2.  subst_rel v x t' u'
3.  v ∉ freevars x ⇒ Let v x t' ≅ Let v x u'
4.  v ∉ freevars x
------------------------------------
    Let v x (App t t') ≅ Let v x (App u u')
```

This is how a single subgoal is presented in the HOL4 interactive terminal. The numbered lines are the assumptions and the last line is the goal.

The first step is to push down the `Let` constructor into every subtree of the `App` constructor. This is done by using the following theorem:

Theorem 8 *Let_App*

$\vdash \texttt{Let}\ w\ e\ (\texttt{App}\ e_1\ e_2) \cong \texttt{App}\ (\texttt{Let}\ w\ e\ e_1)\ (\texttt{Let}\ w\ e\ e_2)$

After performing this step on both sides of the equivalence we get the following sequent:

```
0.  subst_rel v x t u
1.  Let v x t ≅ Let v x u
2.  subst_rel v x t' u'
3.  Let v x t' ≅ Let v x u'
4.  v ∉ freevars x
-----------------------------------
    App (Let v x t) (Let v x t') ≅ App (Let v x u) (Let v x u')
```

Now the congruence rule for `App` can be used. This way the proof gets reduced to proving the equivalence of the subtrees.

Theorem 9 *exp_eq_App_cong* — *expression equivalence congruence theorem for App*

$$\vdash f \cong f' \wedge e \cong e' \Rightarrow \mathit{App}\ f\ e \cong \mathit{App}\ f'\ e'$$

After using this theorem the sequent looks like the following:

```
0.  subst_rel v x t u
1.  Let v x t ≅ Let v x u
2.  subst_rel v x t' u'
3.  Let v x t' ≅ Let v x u'
4.  v ∉ freevars x
-----------------------------------
    Let v x t' ≅ Let v x u' ∧ Let v x t ≅ Let v x u
```

Both cases of the goal now follow immediately from the induction hypotheses. This concludes the proof of this case.

Another interesting case that didn't follow the same approach was the `Var` case. A different strategy was used for it, as it is the case that performs the inlining. There is no need to push down the `Let` constructor in this case, since there are no subtrees. This denotes the edge case when a `Let` has already been pushed to the leaf of the expression tree. Instead, a different lemma is proved:

Theorem 10 *Let_Var3* — *inlining a variable in a Let*

$$\vdash w \notin \mathit{freevars}\ e \Rightarrow \mathit{Let}\ w\ e\ (\mathit{Var}\ w) \cong \mathit{Let}\ w\ e\ e$$

The proof of this lemma dives deeper into the evaluation of those expressions. It shows that the weak head evaluated forms of the expressions are equal. This together with the theorem shown before, `eval_wh_IMP_exp_eq`, is enough to prove the `Var` case.

Definition 9 *deep_subst_rel_Let*

$$\vdash \mathit{subst\_rel}\ v\ x\ y\ y' \wedge v \notin \mathit{freevars}\ x \Rightarrow \mathit{deep\_subst\_rel}\ (\mathit{Let}\ v\ x\ y)\ (\mathit{Let}\ v\ x\ y')$$

Definition 10 *deep_subst_rel_refl*

$$\vdash \mathit{deep\_subst\_rel}\ t\ t$$

Definition 11 *deep_subst_rel_App*

$$\vdash \mathit{deep\_subst\_rel}\ t_1\ u_1 \wedge \mathit{deep\_subst\_rel}\ t_2\ u_2 \Rightarrow \mathit{deep\_subst\_rel}\ (\mathit{App}\ t_1\ t_2)\ (\mathit{App}\ u_1\ u_2)$$

Definition 12 *deep_subst_rel_Lam*

$$\vdash \mathit{deep\_subst\_rel}\ t\ u \Rightarrow \mathit{deep\_subst\_rel}\ (\mathit{Lam}\ w\ t)\ (\mathit{Lam}\ w\ u)$$

Definition 13 *deep_subst_rel_Letrec*

$$\vdash \mathit{list\_rel}\ (\lambda\,(n, t_1)\ (m, u_1).\ n = m \wedge \mathit{deep\_subst\_rel}\ t_1\ u_1)\ xs\ ys \wedge \mathit{deep\_subst\_rel}\ t\ u \Rightarrow \mathit{deep\_subst\_rel}\ (\mathit{Letrec}\ xs\ t)\ (\mathit{Letrec}\ ys\ u)$$

Definition 14 *deep_subst_rel_Prim*

$$\vdash \mathit{list\_rel}\ \mathit{deep\_subst\_rel}\ xs\ ys \Rightarrow \mathit{deep\_subst\_rel}\ (\mathit{Prim}\ p\ xs)\ (\mathit{Prim}\ p\ ys)$$

Figure 3.2: Deep single inlining relation

3.3.2 Deep Single Inlining Relation

One question that can come to mind when reading the statement of this theorem is: Why is there a `Let` in the conclusion instead of just any expression? The reason for that is that the relation depends on the variable name and the expression that is being inlined. More specifically, considering a modified version of the example from before:

```
g (
  let f = (\x -> 5)
  in f 1
)
```

It is not possible to define `subst_rel` for this expression, even though nested inlining can be performed.

That is why another relation is defined, one that doesn't depend on the inlined data. This relation is shown in Figure 3.2.

The informal meaning of this relation is that: for any `t, u` it is possible to inline

any single variable in `t` to get `u`.

Using the new relation, it is possible to construct `deep_subst_rel` instance for the modified example. Which means

```
deep_subst_rel
  (g (let f = (\x -> 5) in f 1))
  (g (let f = (\x -> 5) in (\x -> 5) 1))
```

can be constructed, first using the `App` rule:

Theorem 11 *deep_subst_rel_App*

$$\vdash \textit{deep\_subst\_rel}\ t_1\ u_1 \wedge \textit{deep\_subst\_rel}\ t_2\ u_2 \Rightarrow \textit{deep\_subst\_rel}\ (\textit{App}\ t_1\ t_2)\ (\textit{App}\ u_1\ u_2)$$

It behaves just like a congruence over `App`, which is almost the same as the corresponding theorem in `subst_rel`. Similarly as with `subst_rel`, it takes as assumptions `deep_subst_rel g g` and

```
deep_subst_rel
  (let f = (\x -> 5) in f 1)
  (let f = (\x -> 5) in (\x -> 5) 1)
```

This time the former is trivial since it uses reflexivity. The latter on the other hand uses the `Let` case

Theorem 12 *deep_subst_rel_Let*

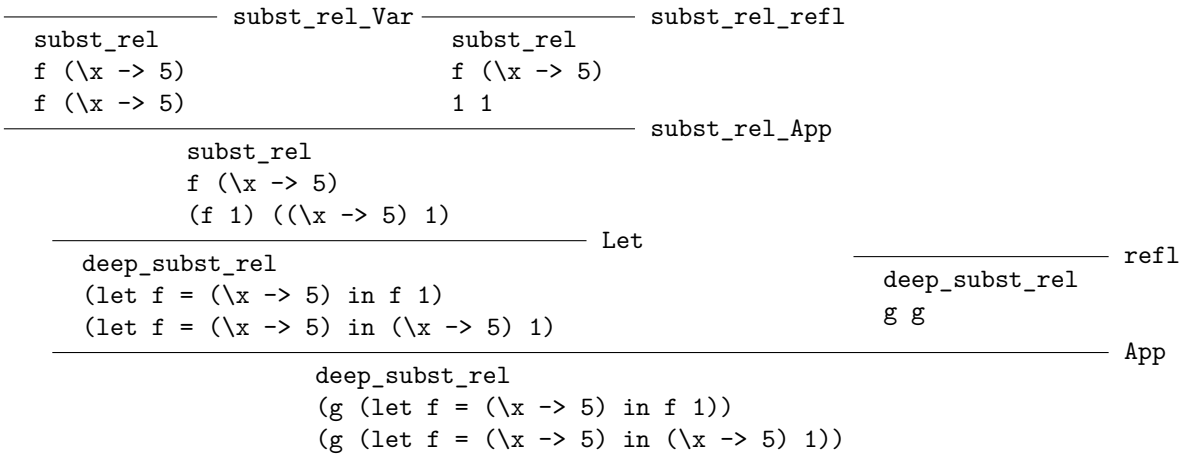
$$\vdash \textit{subst\_rel}\ v\ x\ y\ y' \wedge v \notin \textit{freevars}\ x \Rightarrow \textit{deep\_subst\_rel}\ (\textit{Let}\ v\ x\ y)\ (\textit{Let}\ v\ x\ y')$$

This theorem looks very similar to the one proved before for the equivalence of `Let` expressions based on `subst_rel`. And so `deep_subst_rel` reuses `subst_rel` in this case, using it as an assumption. For this example the specific relation that needs to be constructed is

```
subst_rel f (\x -> 5) (f 1) ((\x -> 5) 1)
```

which is the same instance that was constructed for the previous example.

The derivation for this example relation is as follows:



Using this new relation, a more general theorem was formulated and proved.

Theorem 13 *Equivalence of inlining any single definition for* ***deep_subst_rel*** *relation*

$\vdash$ ***deep_subst_rel*** $t\ u \Rightarrow t \cong u$

The proof of this theorem was much more straightforward than the one for `subst_rel`. Thanks to the reuse of the `subst_rel` relation, the proof was reduced to using congruence rules in most cases and just matching the previously proved theorem for `subst_rel` in the `Let` case.

This theorem is enough to express a transformation performed by an inlining pass. And so, it can be utilized to show the correctness of the pass. More specifically, based on this theorem, an implementation of an inlining pass `inline` can be verified by showing that `deep_subst_rel e (inline e)` holds for any expression `e`. This implementation would have to follow the syntactic transformations of the `deep_subst_rel` relation. This is a much more general theorem than the one for `subst_rel`. This theorem already satisfies the requirements for the correctness of an inlining pass, as described in the project goals.

The implementation would have to be implemented in a very specific way. It would have to traverse the AST and once a `Let` binding suitable for inlining is found, it would traverse the result subtree and replace all occurrences of the bound variable with the bound expression. After that, the traversal would continue. One can notice that this is not the most efficient way of implementing an inlining pass. Instead, it should be possible to traverse the AST and collect all the bindings suitable for inlining, while simultaneously performing the inlining.

3.4 Multi-Inlining Relation

An efficient implementation of an inlining pass can inline multiple expressions at the same time. Instead of inlining only a single expression whenever an appropriate binding is found, the algorithm should collect all declarations and be able to inline every one of them in one go. A new relation has been defined, this time indicating

the possible inlining of multiple declarations. The full definition of this inductive relation is shown in Figure 3.3.

For any `l`, `t`, `u`, `list_subst_rel l t u` expresses that declarations in `l` can be inlined in `t` possibly multiple times to get `u`.

This relation is slightly more complicated than the previous ones. But going through the rules:

1. `list_subst_rel_Let` — given a `Let` expression, allows adding it to the list of declarations to be inlined.
2. `list_subst_rel_LetrecInline` — given a non-mutually recursive `Letrec` expression, allows adding it to the list of declarations to be inlined. It is only relevant for specialization.
3. `list_subst_rel_VarSimp` — allows inlining a single non-recursive declaration from the list
4. `list_subst_rel_Var` — is a more general version of the previous rule, it allows for inlining a single non-recursive declaration in place of a variable reference. In addition, it allows performing other transformations on the inlined expression. This possibly allows for inlining inside the already inlined expression.
5. `list_subst_rel_LetrecIntro` — allows for inserting a definition of a recursive function anywhere in the tree. As in the previous case, it also allows for performing other transformations on the inlined declaration. This rule is also only relevant for specialization.
6. The rest of the rules behave similarly to their counterparts before — they allow for traversing the tree and inlining deeper.

Using this relation, it is possible to construct an instance for the following example:

```
let f = (\x -> 5)
in f 1
```

The following instance can be constructed:

```
list_subst_rel []
  (let f = (\x -> 5) in f 1)
  (let f = (\x -> 5) in (\x -> 5) 1)
```

To do so, the only constructors needed are the `refl`, `Let`, `App` and `Var` ones. More specifically, to construct this relation one has to first use the `Let` case:

Theorem 14 *list_subst_rel_Let*

$$\vdash \mathit{list\_subst\_rel}\ l\ x\ x' \wedge \mathit{list\_subst\_rel}\ (l +\!\!+ [(v, \mathit{Exp}\ x)])\ y\ y' \Rightarrow \mathit{list\_subst\_rel}\ l\ (\mathit{Let}\ v\ x\ y)\ (\mathit{Let}\ v\ x'\ y')$$

It in turn takes as assumptions,

Definition 15 `list_subst_rel_Let`

$$\vdash \texttt{list\_subst\_rel}\ l\ x\ x' \wedge \texttt{list\_subst\_rel}\ (l +\!\!+ [(v,\texttt{Exp}\ x)])\ y\ y' \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Let}\ v\ x\ y)\ (\texttt{Let}\ v\ x'\ y')$$

Definition 16 `list_subst_rel_LetrecInline`

$$\vdash \texttt{mem}\ (v,\texttt{Rec}\ t)\ l \wedge \texttt{list\_subst\_rel}\ l\ x\ y \wedge \texttt{Letrec}\ [(v,t)]\ y \cong z \Rightarrow \texttt{list\_subst\_rel}\ l\ x\ z$$

Definition 17 `list_subst_rel_Var`

$$\vdash \texttt{mem}\ (v,\texttt{Exp}\ x)\ l \wedge x \cong x_1 \wedge \texttt{no\_shadowing}\ x_1 \wedge \texttt{disjoint}\ (\texttt{boundvars}\ x_1)\ (\texttt{freevars}\ x_1) \wedge \texttt{disjoint}\ (\texttt{boundvars}\ x_1)\ (\texttt{vars\_of}\ l) \wedge \texttt{disjoint}\ (\texttt{boundvars}\ x_1)\ (\texttt{freevars\_of}\ l) \wedge \texttt{list\_subst\_rel}\ l\ x_1\ x_2 \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Var}\ v)\ x_2$$

Definition 18 `list_subst_rel_VarSimp`

$$\vdash \texttt{mem}\ (v,\texttt{Exp}\ x)\ l \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Var}\ v)\ x$$

Definition 19 `list_subst_rel_LetrecIntro`

$$\vdash \texttt{list\_subst\_rel}\ l\ x\ y \wedge \texttt{list\_subst\_rel}\ (l +\!\!+ [(v,\texttt{Rec}\ x)])\ t\ u \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Letrec}\ [(v,x)]\ t)\ (\texttt{Letrec}\ [(v,y)]\ u)$$

Definition 20 `list_subst_rel_refl`

$$\vdash \texttt{list\_subst\_rel}\ l\ t\ t$$

Definition 21 `list_subst_rel_Prim`

$$\vdash \texttt{list\_rel}\ (\texttt{list\_subst\_rel}\ l)\ xs\ ys \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Prim}\ p\ xs)\ (\texttt{Prim}\ p\ ys)$$

Definition 22 `list_subst_rel_App`

$$\vdash \texttt{list\_subst\_rel}\ l\ t_1\ u_1 \wedge \texttt{list\_subst\_rel}\ l\ t_2\ u_2 \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{App}\ t_1\ t_2)\ (\texttt{App}\ u_1\ u_2)$$

Definition 23 `list_subst_rel_Lam`

$$\vdash \texttt{list\_subst\_rel}\ l\ t\ u \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Lam}\ w\ t)\ (\texttt{Lam}\ w\ u)$$

Definition 24 `list_subst_rel_Letrec`

$$\vdash \texttt{list\_rel}\ (\lambda\,(n,t_1)\ (m,u_1).\ n = m \wedge \texttt{list\_subst\_rel}\ l\ t_1\ u_1)\ xs\ ys \wedge \texttt{list\_subst\_rel}\ l\ t\ u \Rightarrow \texttt{list\_subst\_rel}\ l\ (\texttt{Letrec}\ xs\ t)\ (\texttt{Letrec}\ ys\ u)$$

Figure 3.3: Multi-inlining relation

```
list_subst_rel [(f, (\x -> 5))] (f 1) ((\x -> 5) 1)
```

and

```
list_subst_rel [] (\x -> 5) (\x -> 5)
```

They perform respectively the inlining in the result expression of a `Let` and the inlining in the bound expression. The second assumption is trivial since it doesn't change the expression, so simply follows from reflexivity.

The first assumption is more interesting since it introduces a new expression to the list of declarations to be inlined. The next step is to use the `App` case, which takes as assumptions

```
list_subst_rel [(f, (\x -> 5))] f (\x -> 5)
```

and

```
list_subst_rel [(f, (\x -> 5))] 1 1
```

Again, the second assumption is trivial, since it follows from reflexivity. The first one performs the inlining using the `VarSimp` constructor which only requires the evidence that the variable is in the list of declarations:

Theorem 15 *list_subst_rel_VarSimp*

$$\vdash \mathit{mem}\ (v,\mathit{Exp}\ x)\ l \Rightarrow \mathtt{list\_subst\_rel}\ l\ (\mathit{Var}\ v)\ x$$

The derivation tree for this relation instance is as follows:

```
────────────────────────── VarSimp ──────────────────────────── refl
 list_subst_rel                     list_subst_rel
 [(f, (\x -> 5))]                   [(f, (\x -> 5))]
 f (\x -> 5)                        1 1
──────────────────────────────────────────────────────────────── App
       list_subst_rel                                           ────────────────────────── refl
       [(f, (\x -> 5))]                                          list_subst_rel []
       (f 1) ((\x -> 5) 1)                                       (\x -> 5) (\x -> 5)
      ──────────────────────────────────────────────────────────────────────────────────── Let
                            list_subst_rel []
                            (let f = (\x -> 5) in f 1)
                            (let f = (\x -> 5) in (\x -> 5) 1)
```

3.4.1 Multi-inlining relation equivalence

Similarly to the previous relation, an equivalence theorem was formulated and proved. A modification for this approach is the assumption that the inlining pass has to be preceded by a variable freshening pass — a pass ensuring that every variable name is unique. This way the reasoning behind tracking captures becomes much simpler. The freshening pass is not part of the scope of this thesis.

For the multi-inlining relation the core property for correctness is similar to the one declared for `subst_rel`. Then, it expressed equivalence between `Let v x t` and `Let v x u` with the assumption that `subst_rel v x t u`. This time instead of a single

`Let`, it expresses the equivalence of inlining multiple `Let` and `Letrec` expressions combined. And so the following theorem was formulated and proved:

Theorem 16 *Equivalence of inlining multiple definitions for* `list_subst_rel` *relation*

$$\vdash \mathit{list\_subst\_rel}\ xs\ x\ y \wedge \mathit{binds\_ok}\ xs \wedge \mathit{disjoint}\ (\mathit{boundvars}\ x)\ (\mathit{vars\_of}\ xs) \wedge \mathit{disjoint}\ (\mathit{boundvars}\ x)\ (\mathit{freevars}\ x) \wedge \mathit{disjoint}\ (\mathit{boundvars}\ x)\ (\mathit{freevars\_of}\ xs) \wedge \mathit{no\_shadowing}\ x \Rightarrow \mathit{Binds}\ xs\ x \cong \mathit{Binds}\ xs\ y$$

Where `Binds` is defined as follows:

Definition 25

$$\mathit{Binds}\ []\ e \stackrel{\text{def}}{=} e$$
$$\mathit{Binds}\ ((v, \mathit{Exp}\ x) :: xs)\ e \stackrel{\text{def}}{=} \mathit{Let}\ v\ x\ (\mathit{Binds}\ xs\ e)$$
$$\mathit{Binds}\ ((v, \mathit{Rec}\ x) :: xs)\ e \stackrel{\text{def}}{=} \mathit{Letrec}\ [(v, x)]\ (\mathit{Binds}\ xs\ e)$$

This theorem shows that for any expressions `x, y` and a list of declarations `xs`, if `list_subst_rel xs x y` then `x` with all declarations in `xs` is equivalent to `y` with all the same declarations. Some assumptions are also required. `binds_ok` ensures that the list of declarations is correct i.e. that no variable is bound twice and that older declarations do not depend on the newer ones. The `no_shadowing` relation is used to ensure that the freshening pass was performed. It gives the guarantee that no two bound variables have the same name. This inductive relation is defined in Figure 3.4.

Additionally, other assumptions that follow from performing freshening on the whole program and the program being closed are required. Those assumptions assert that the bound variables of `x` are not used in any context in the list of declarations `xs` and aren't used as free variables in `x`.

This is the most substantial proof done in this thesis. It is also the most complex one. The following is an abstract description of the proof strategy used. Before diving into the proof details, some notes are worth reminding. Firstly, there are two ways in which a new binding can be introduced to the inlining list: a `Let` expression or a `Letrec` with a single binding. Secondly, those bindings are inlined in two distinct ways. Nonrecursive expressions are inlined in place of their explicit references. Recursive expressions can be inlined in any place in the program.

The equivalence theorem was proved by induction on the `list_subst_rel` relation. Any cases that don't insert anything to the bindings list and don't look up anything from it were approached in a very similar way as in the `subst_rel_IMP_exp_eq` theorem. The `Binds` were pushed down into all subtrees of the expression. Next, the congruence rules for that constructor were used reducing the goal to only having to prove the equivalence of the subexpressions. After that, the induction hypothesis was applied and the goal was proved. This time there were no existing theorems

Definition 26 *no_shadowing_Var*

$$\vdash \mathit{no\_shadowing}\ (\mathit{Var}\ v)$$

Definition 27 *no_shadowing_Prim*

$$\vdash \mathit{every}\ \mathit{no\_shadowing}\ l \Rightarrow \mathit{no\_shadowing}\ (\mathit{Prim}\ p\ l)$$

Definition 28 *no_shadowing_App*

$$\vdash \mathit{no\_shadowing}\ x \wedge \mathit{no\_shadowing}\ y \wedge \mathit{disjoint}\ (\mathit{boundvars}\ x)\ (\mathit{boundvars}\ y) \Rightarrow \mathit{no\_shadowing}\ (\mathit{App}\ x\ y)$$

Definition 29 *no_shadowing_Lam*

$$\vdash \mathit{no\_shadowing}\ x \wedge v \notin \mathit{boundvars}\ x \Rightarrow \mathit{no\_shadowing}\ (\mathit{Lam}\ v\ x)$$

Definition 30 *no_shadowing_Letrec*

$$\begin{aligned}\vdash\ & \mathit{every}\\ & (\lambda\,(v, e).\\ & \quad \mathit{no\_shadowing}\ e \wedge \mathit{disjoint}\ (\mathit{boundvars}\ e)\ (\mathit{boundvars}\ x) \wedge\\ & \quad \mathit{disjoint}\ (\mathit{boundvars}\ e)\ (\mathit{freevars}\ e) \wedge\\ & \quad \mathit{disjoint}\ (\mathit{set}\ (\mathit{map}\ \mathit{fst}\ l))\ (\mathit{boundvars}\ e))\ l \wedge \mathit{no\_shadowing}\ x \wedge\\ & \mathit{disjoint}\ (\mathit{set}\ (\mathit{map}\ \mathit{fst}\ l))\ (\mathit{boundvars}\ x) \Rightarrow\\ & \mathit{no\_shadowing}\ (\mathit{Letrec}\ l\ x)\end{aligned}$$

Figure 3.4: Definition of `no_shadowing` relation

that allow for pushing the `Binds` into subgoals, so they had to be proved first. An example case that followed this schema is `Lam`, the sequent for it is as follows:

```
0.  list_subst_rel xs x y
1.  disjoint (boundvars x) (freevars x) ⇒ Binds xs x ≅ Binds xs y
2.  binds_ok xs
3.  disjoint (boundvars x) (vars_of xs)
4.  w ∉ vars_of xs
5.  disjoint (boundvars x) (freevars x DELETE w)
6.  disjoint (boundvars x) (freevars_of xs)
7.  w ∉ freevars_of xs
8.  no_shadowing x
9.  w ∉ boundvars x
------------------------------------
    Binds xs (Lam w x) ≅ Binds xs (Lam w y)
```

Next, the following theorem is used and a congruence rule for `Lam` is applied to the goal:

Theorem 17 *Binds_Lam*

$$\vdash \neg \mathit{mem}\ v\ (\mathit{map}\ \mathit{fst}\ xs) \wedge v \notin \mathit{freevars\_of}\ xs \Rightarrow \mathit{Binds}\ xs\ (\mathit{Lam}\ v\ x) \cong \mathit{Lam}\ v\ (\mathit{Binds}\ xs\ x)$$

The resulting sequent is the following:

```
0.  list_subst_rel xs x y
1.  disjoint (boundvars x) (freevars x) ⇒ Binds xs x ≅ Binds xs y
2.  binds_ok xs
3.  disjoint (boundvars x) (vars_of xs)
4.  w ∉ vars_of xs
5.  disjoint (boundvars x) (freevars x DELETE w)
6.  disjoint (boundvars x) (freevars_of xs)
7.  w ∉ freevars_of xs
8.  no_shadowing x
9.  w ∉ boundvars x
------------------------------------
    Binds xs x ≅ Binds xs y
```

This goal matches the inductive hypothesis, making the only thing left to do is some accounting with the assumptions.

The more interesting, cases are those that perform the inlining by inserting new definitions. Inlining both the recursive and nonrecursive expressions uses the same lemma. The lemma is as follows:

Theorem 18 *Binds_MEM*

$$\vdash \mathit{mem}\ e\ xs \wedge \mathit{binds\_ok}\ xs \Rightarrow \mathit{Binds}\ xs\ x \cong \mathit{Binds}\ (xs +\!\!+ [e])\ x$$

This lemma states that if a binding `e` is in the list of definitions `xs`, then it is possible to copy `e` and add this copy to the and of the declarations list. The proof of this lemma is briefly described in the following steps:

1. Induction on `xs` is used. The empty list case is trivial, since `e` has to be a member of the list.
2. For the other case, if the head of the list is not `h` then the inductive hypothesis is applied and the goal is proved.
3. Otherwise, the strategy is to show that `e` can be duplicated at the head of the list. Then next, the copy can be pushed to the end of the list. This is done by induction on the current tail. Where the base case is trivial again.
4. The inductive case then unwinds to be split into four different cases, depending on the type of `e` and the next element on the list. Proofs for each of those cases use slightly lower level transformations, like weak haed evaluation equality, similarly to the `Let_Var3` theorem used in the `subst_rel_IMP_exp_eq` proof. The statement of one of those lemmas is as follows:

   Theorem 19 *Let_Let_copy*

   $$\vdash v \neq w \wedge w \notin \mathit{freevars}\ x \wedge v \notin \mathit{freevars}\ x \Rightarrow \mathit{Let}\ v\ x\ (\mathit{Let}\ v\ x\ (\mathit{Let}\ w\ y\ e)) \cong \mathit{Let}\ v\ x\ (\mathit{Let}\ w\ y\ (\mathit{Let}\ v\ x\ e))$$

The `Bings_MEM` lemma is a very significant part of the whole proof and allows almost singlehandedly proving the `list_subst_rel_Var`, `list_subst_rel_VarSimp` and `list_subst_rel_LetrecInline` cases of the theorem.

Other interesting parts of the proof are the cases that introduce new bindings. Though surprisingly, those cases weren't much more complicated than the `list_subst_rel_App` once due to the way how the `Binds` relation is defined. A lot of accounting still has to be done to make the assumptions match the inductive hypothesis, but the actual proof is relatively simple. For example, the relevant part of the sequent for the `list_subst_rel_Let` case is the following:

```
...
1.  ... ⇒ Binds xs x ≅ Binds xs y
...
3.  ... ⇒
    Binds (xs ++ [(v,Exp x)]) x' ≅ Binds (xs ++ [(v,Exp x)]) y'
...
------------------------------------
  Binds xs (Let v x x') ≅ Binds xs (Let v y y')
```

Here assumptions `1` and `3` are the inductive hypotheses. And because the definition of `Binds` essentially defines a 'stack' of `Lets` and `Letrecs`, assumption `3` can be rewritten as follows:

```
...
```

```
1.  ... ⇒ Binds xs x ≅ Binds xs y
...
3.  ... ⇒
    Binds xs (Let v x x') ≅ Binds xs (Let v x y')
...
------------------------------------
  Binds xs (Let v x x') ≅ Binds xs (Let v y y')
```

From this point, the proof constitutes of the following steps:

1. Applying the inductive hypothesis from assumption `3` to the goal.
2. Using the `Binds_Let` theorem to pull the `Let` definition from inside of the `Binds`. The theorem is defined as follows:

   Theorem 20

   $$\vdash \neg \mathit{mem}\ v\ (\mathit{map}\ \mathit{fst}\ xs) \wedge v \notin \mathit{freevars\_of}\ xs \Rightarrow \mathit{Binds}\ xs\ (\mathit{Let}\ v\ x\ y) \cong \mathit{Let}\ v\ (\mathit{Binds}\ xs\ x)\ (\mathit{Binds}\ xs\ y)$$

3. Use the congruence rule for `Let` to match the goal to the other inductive hypothesis.
4. Apply the inductive hypothesis from assumption `1` to the goal.

The proof for the `list_subst_rel_LetrecIntro` case is very similar, apart from using the appropriate theorems for `Letrec` instead of `Let`.

3.4.2 Multi-inlining relation correctness

Finally, having proved the equivalence of inlining multiple declarations, a theorem was formulated and proved that shows the equivalence of inlining an empty list of declarations in a closed expression.

Theorem 21 *Equivalence of inlining for* `list_subst_rel` *relation*

$$\vdash \mathit{list\_subst\_rel}\ []\ x\ y \wedge \mathit{no\_shadowing}\ x \wedge \mathit{closed}\ x \Rightarrow x \cong y$$

This theorem is the core of the correctness proof for the inlining pass, as it shows the correctness of inlining for any expression. It immediately follows from the previous one, since it just instantiates the list of declarations to be empty. Proof of this theorem allows for proving the correctness of an efficient implementation of the inlining pass.

4

Verified Implementation of Inlining

This chapter continues the approach described in Section 3.2. The previous chapter defined the syntactic transformation of multi-inlining and showed that any implementation that the relation follows preserves the semantics of the program. This chapter focuses on the implementation of the inlining pass. It describes the transformation function as well as the correctness proof for the implementation. The correctness proof uses the previously introduced relation and the equivalence theorem proved for the relation.

4.1 Implementation

The implementation of the inlining pass follows the same pattern as the multi-inlining relation. It transforms the expression based on the given declarations. The implementation is shown in Figure 4.1. It is worth reminding that the implementation of the transformation uses different types of expressions than the relation-level proofs that were described so far. The implementation uses the `cexp` type, which contains some additional constructors — explicit `Let`, `Case` and currently unused `NestedCase`. The bridge that connects those two types of expressions is a function `exp_of` that transforms compiler expressions into PureLang expressions.

The `inline` function takes:

1. `m` — the map of declarations to be inlined.
2. `ns` — the set of names to avoid when freshening variables.
3. `h` — the heuristic function that decides whether a declaration is fit to be inlined.
4. Lastly, the expression to be transformed.

The need for passing the set of names might be surprising since the input to the function is already freshened. The reason for that is that when inlining an expression, it might be valuable to continue inlining inside the inserted definition body. Then the names of the variables in the body might clash with the names in `m`. And so the inserted body has to be freshened again. This isn't done in the current implementation of the pass, so it serves as a preparation for future improvements.

The implementation follows quite closely the same pattern as the multi-inlining relation.

Definition 31 *`inline` function implementation*

```
inline m ns h (Var a v) ≝
 case lookup m v of
   None ⇒ (Var a v,ns)
 | Some (cExp e) ⇒ if is_Lam e then (Var a v,ns) else (e,ns)
 | Some (cRec e') ⇒ (Var a v,ns)
inline m ns h (App a e es) ≝
 let (e1,ns1) =
   case get_Var_name e of
     None ⇒ inline m ns h e
   | Some v ⇒
     case lookup m v of
       None ⇒ inline m ns h e
     | Some (cExp v1) ⇒ (v1,ns)
     | Some (cRec v2) ⇒ inline m ns h e;
 (es2,ns2) = inline_list m ns1 h es
 in
   (App a e1 es2,ns2)
inline m ns h (Let a v e1 e2) ≝
 let m1 = heuristic_insert m h v e1;
 (e3,ns3) = inline m ns h e1;
 (e4,ns4) = inline m1 ns3 h e2
 in
   (Let a v e3 e4,ns4)
inline m ns h (Letrec a vbs e) ≝
 let m1 = heuristic_insert_Rec m h vbs;
 (vbs1,ns1) = inline_list m ns h (map snd vbs);
 (e2,ns2) = inline m1 ns1 h e
 in
   (Letrec a (map2 (λ(v,_) x. (v,x)) vbs vbs1) e2,ns2)
inline m ns h (Lam a vs e) ≝ let (e1,ns1) = inline m ns h e in (Lam a vs e1,ns1)
inline m ns h (Prim a op es) ≝
 let (es2,ns2) = inline_list m ns h es in (Prim a op es2,ns2)
inline m ns h (Case a e v bs f) ≝
 let (e1,ns1) = inline m ns h e;
 (bs2,ns2) = inline_list m ns1 h (map (λ(v,vs,e). e) bs);
 (f3,ns3) =
   case f of
     None ⇒ (None,ns2)
   | Some (vs,e') ⇒ let (e4,ns4) = inline m ns2 h e' in (Some (vs,e4),ns4)
 in
   (Case a e1 v (map2 (λ(v,vs,_) e. (v,vs,e)) bs bs2) f3,ns3)
inline m ns h (NestedCase a e v p e' bs) ≝ (NestedCase a e v p e' bs,ns)
inline_list m ns h [] ≝ ([],ns)
inline_list m ns h (e::es) ≝
 let (e1,ns1) = inline m ns h e; (es2,ns2) = inline_list m ns1 h es in (e1::es2,ns2)
```


Figure 4.1: `inline` function implementation

- `Var` case inlines the variable if it is in the map. It also prevents inlining if the expression to be inserted is a lambda abstraction. It is based on the reasoning presented before; there is no benefit from inlining a lambda that is not immediately applied to at least one argument.
- `App` case handles inlining in the function and the argument. It is slightly more interesting than the corresponding case in the relation. This case also handles inlining lambda abstractions, since if the callee is a variable reference, it is certain at this point that it will be called with at least one argument.
- `Let` case handles adding more declarations to the map. It is worth noting that the heuristic is checked at the point of adding the declaration to the map, instead of at the point of inlining.
- `Letrec` case behaves similarly to the `Let` case. It only adds non-mutually recursive declarations to the map of inline declarations.
- The rest of the cases simply traverse their children and perform inlining deeper. With one exception, `NestedCase` is not affected by the transformation. That is because PureCake doesn't support nested cases at present.

Using this declaration, a final function was defined that is correctly initialized and hides all the implementation details.

Definition 32 *`inline_all` function implementation*

$$\mathit{inline\_all} \stackrel{\text{def}}{=} \mathit{inline}\ \mathit{empty}\ \mathit{empty}$$

4.2 Implementation correctness proof

Using the multi-inlining relation, a theorem was formulated and proved that shows that the result of the inlining pass satisfies the relation. The statement of the theorem is as follows:

Theorem 22 *`inline` function correctness theorem*

$$\begin{aligned}\vdash\ & \mathit{memory\_inv}\ xs\ m \wedge \mathit{map\_ok}\ m \wedge \mathit{no\_shadowing}\ (\mathit{exp\_of}\ x) \wedge \\ & \mathit{disjoint}\ (\mathit{set}\ (\mathit{map}\ \mathit{fst}\ xs))\ (\mathit{boundvars}\ (\mathit{exp\_of}\ x)) \wedge \mathit{inline}\ m\ ns\ h\ x = (t, ns_1) \Rightarrow \\ & \mathit{list\_subst\_rel}\ xs\ (\mathit{exp\_of}\ x)\ (\mathit{exp\_of}\ t)\end{aligned}$$

This theorem is defined for any `m, ns, h, e` so it is heuristic-agnostic — the correctness of the implementation doesn't depend on the heuristic. The assumptions are quite similar to the ones for the equivalence of the relation. The function `exp_of` is the link between the two ASTs; it transforms the compiler expressions into PureLang expressions.

- `memory_inv` is a new addition. The relation used a simple linked list for the declaration storage. The implementation uses a map, to be more efficient. This assumption ensures that the contents of the map and the list are the same.

- `map_ok` is an artifact of using the specific map implementation.
- All the other assumptions give the same guarantees as the ones for the relation-level proof.

The proof of this theorem uses the induction theorem from the definition of the `inline` function. The approach for every case of the proof follows the same pattern, which is to just follow the implementation and for every code branch, instantiate the `list_subst_rel` relation with the appropriate constructor.

For example, the sequent for the `Var` case after expanding the definition of `inline` is:

```
0.  memory_inv xs m
1.  map_ok m
2.  no_shadowing (exp_of (Var a v))
3.  disjoint (set (map fst xs)) (boundvars (exp_of (Var a v)))
4.  (case lookup m v of
        None => (Var a v, ns)
      | Some (cExp e) => if is_Lam e then (Var a v, ns) else (e, ns)
      | Some (cRec e') => (Var a v, ns)) = (t,ns1)
------------------------------------
    list_subst_rel xs (exp_of (Var a v)) (exp_of t)
```

Next, the proof splits on the value of `lookup m v`. The `None` case is immediately solved by the `list_subst_rel_refl` constructor. The sequent for the `Some` part is as follows:

```
0.  memory_inv xs m
1.  map_ok m
2.  no_shadowing (exp_of (Var a v))
3.  disjoint (set (map fst xs)) (boundvars (exp_of (Var a v)))
4.  (case x of
        cExp e => if is_Lam e then (Var a v,ns) else (e,ns)
      | cRec e' => (Var a v,ns)) = (t,ns1)
5.  lookup m v = Some x
------------------------------------
    list_subst_rel xs (exp_of (Var a v)) (exp_of t)
```

And so the proof splits on the value of `x`. And this time the `cRec` case is immediately solved by the `list_subst_rel_refl` constructor. Then, after the split on the value of `is_Lam e`, the `list_subst_rel_refl` constructor solves the positive case. The sequent for the negative case is as follows:

```
0.  {explode a | ∃e. lookup m a = Some e} = set (map fst xs)
1.  ∀v e. lookup m v = Some e ⇒ mem (explode v,crhs_to_rhs e) xs
2.  map_ok m
3.  lookup m v = Some (cExp c)
4.  ¬is_Lam c
------------------------------------
```

```
list_subst_rel xs (Var (explode v)) (exp_of c)
```

This matches exactly the `list_subst_rel_VarSimp` constructor. After applying it and expanding some assumptions, the proof is complete.

A similar approach applies to the rest of the cases.

Finally, the correctness of the full inlining pass was proved by using the above theorem and the equivalence theorem for the relation.

Theorem 23 *Correctness theorem for the* `inline_all` *function*

$$\vdash \texttt{no\_shadowing}\,(\texttt{exp\_of}\ x) \wedge \texttt{closed}\,(\texttt{exp\_of}\ x) \wedge (t, ns_1) = \texttt{inline\_all}\ h\ x \Rightarrow \texttt{exp\_of}\ x \cong \texttt{exp\_of}\ t$$

Similarly to the relation-level proof, this theorem is defined for any expression that satisfies `no_shadowing` and is closed.

5

Verified Specialization

This chapter describes work done on the inlining specialization. It concentrates on the implementation of the specialization pass and the approach of introducing this optimization into the inlining pass. This optimization is not verified, the verification of this pass is left for future work.

5.1 Goal

The specialization optimization is slightly more complex than just inlining. Inlining is an optimization that substitutes `Let` bindings, whereas specialization uses `Letrec` — recursive declarations. The main idea of specialization is to be able to create specialized versions of recursive functions near the call site. Considering the following example:

```
letrec map = \f -> \lst -> case lst of
  [] -> []
  (x:xs) -> f x : map f xs
in
  let foo = (\x -> x + 1)
  in map foo
```

The specialization pass would ideally create a copy of the definition of `map` and insert it near the call site with all possible arguments specialized. So for the given example, a specializing pass would produce the following code:

```
letrec map = \f -> \lst -> case lst of
  [] -> []
  (x:xs) -> f x : map f xs
in
  let foo = (\x -> x + 1)
  in
    letrec map = \lst -> case lst of
      [] -> []
      (x:xs) -> foo x : map foo xs
    in map
```

Similarly to the inlining pass, this optimization should also only be used when the recursive function is immediately applied to an argument, otherwise the compiler would just end up adding extra code without any profit. That is also why the original binding should not be removed in this pass — only copied.

5.2 Specializing approach

The use of specialization is accommodated by the inlining pass. When looking back at the definition of `list_subst_rel_LetrecInline`, it is precisely the case when specialization should be applied.

Theorem 24

$$\vdash \mathit{mem}\,(v\texttt{,}\mathtt{Rec}\,t)\;l \wedge \mathit{list\_subst\_rel}\;l\;x\;y \wedge \mathtt{Letrec}\,[(v\texttt{,}t)]\;y \cong z \Rightarrow \mathit{list\_subst\_rel}\;l\;x\;z$$

This inlining case for any expression `x` performs inlining on it, with the result `y`, allows inserting a `Letrec` binding around the result and gives as the result any expression that is equivalent to `Letrec [(v,t)] y`. And so, if a specialization pass gives an equivalent output expression, it can be used during the inlining pass. Specialization can be split into two parts for a given term `f x`:

- First inline the definition for `f`. This transformation could give the following code:

```
letrec f = \a -> \b -> f a c
in f x
```

- Next, inline any argument that stays constant for every invocation of the function inside of its definition. The example could result in:

```
letrec f = \a -> \b -> f x c
in f x
```

Currently, the specialization pass is unverified, though the current methodology allows for it to be proven correct. The implementation level proof for inlining also makes it easy for the specialization proof to be introduced.

5.3 Implementation

The specialization is not implemented as a separate pass. Instead, it is simply a transformation function that is used by the inlining pass. The function is defined as follows:

Definition 33

$$
\begin{aligned}
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Lam}\ a\ \mathit{vs}\ e) \stackrel{\text{def}}{=} \\
&\quad \text{let } p = \mathit{const\_call\_args}\ f\ (\mathit{map}\ \mathit{Some}\ \mathit{vs})\ e; \\
&\quad m = \mathit{spec\_map}\ p\ \mathit{args} \\
&\quad \text{in} \\
&\qquad \text{if } m = \mathit{empty} \text{ then } \mathit{None} \\
&\qquad \text{else } \mathit{Some}\ (\mathit{Lam}\ a\ \mathit{vs}\ (\mathit{subst\_map}\ m\ e)) \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Var}\ v_4\ v_5) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Prim}\ v_6\ v_7\ v_8) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{App}\ v_9\ v_{10}\ v_{11}) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Let}\ v_{15}\ v_{16}\ v_{17}\ v_{18}) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Letrec}\ v_{19}\ v_{20}\ v_{21}) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{Case}\ v_{22}\ v_{23}\ v_{24}\ v_{25}\ v_{26}) \stackrel{\text{def}}{=} \mathit{None} \\
&\mathit{spec}\ f\ \mathit{args}\ (\mathit{NestedCase}\ v_{27}\ v_{28}\ v_{29}\ v_{30}\ v_{31}\ v_{32}) \stackrel{\text{def}}{=} \mathit{None}
\end{aligned}
$$

This function takes:

1. The `f` name of the `letrec` to specialize.
2. Arguments `args`, the function is called with.
3. The body of the `letrec`.

The output of this function is the specialized version of the provided `letrec` body.

The specialization can only be performed on `letrec`s with bodies that are top-level lambda abstractions, otherwise there are no arguments to specialize. The function works in the following steps:

1. It uses `const_call_args` to find for all "constant arguments" — arguments of `f` that are always only called with the references to themselves in recursive calls to `f`. The implementation of this function is shown in Figure 5.1. This function takes the name of the `Letrec`, the names of its top-level arguments and the body of the `Letrec`.
2. It then computes a lookup for all "constant arguments" and their appropriate values in `args`. This lookup is then used to substitute the constant arguments with their values in the body of the specialized `letrec`.
3. If the lookup is empty, no specialization is performed. Otherwise, it performs substitution of the constant arguments with their values inside of the provided `Letrec` body.
4. The result of the function is the specialized body of the `Letrec`.

Considering the following example:

```
letrec sum_map_to = \n -> \f -> case n of
  0 -> 0
  n -> f n + sum_map_to (n - 1) f
```

```
in
  sum_map_to 10 (\x -> x + 1)
```

The specialization algorithm will:

1. Search the body of `sum_map_to` for constant arguments. In this case, it will find that `f` is a "constant argument" and `n` is not. This is because `f` is only positionally called as itself in recursive calls, whereas `n` is called as `n - 1`.
2. It will then compute a lookup for the constant arguments. In this case, it will be `[f -> (x -> x + 1)]`.
3. Finally, it will substitute the constant arguments with their values.

When called within the inlining pass, the result will be:

```
letrec sum_map_to = \n -> \f -> case n of
  0 -> 0
  n -> f n + sum_map_to (n - 1) f
in
  letrec sum_map_to = \n -> \f -> case n of
    0 -> 0
    n -> f n + sum_map_to (n - 1) (\x -> x + 1)
  in
    sum_map_to 10 (\x -> x + 1)
```

Definition 34 *Function computing the "constant arguments" of a* `Letrec`

```
const_call_args f vs (App a e es) ≝
 case e of
   Var a' v ⇒
    if v = f then let vs1 = min_call_args vs es in const_call_args_list f vs1 es
    else const_call_args_list f vs es
 | Prim v32 v33 v34 ⇒ const_call_args_list f vs (e::es)
 | App v35 v36 v37 ⇒ const_call_args_list f vs (e::es)
 | Lam v38 v39 v40 ⇒ const_call_args_list f vs (e::es)
 | Let v41 v42 v43 v44 ⇒ const_call_args_list f vs (e::es)
 | Letrec v45 v46 v47 ⇒ const_call_args_list f vs (e::es)
 | Case v48 v49 v50 v51 v52 ⇒ const_call_args_list f vs (e::es)
 | NestedCase v53 v54 v55 v56 v57 v58 ⇒ const_call_args_list f vs (e::es)
const_call_args f vs (Var a v) ≝ if v = f then [] else vs
const_call_args f vs (Let a v e1 e2) ≝
 let vs1 = const_call_args f vs e1 in const_call_args f vs1 e2
const_call_args f vs (Lam a vss e1) ≝ const_call_args f vs e1
const_call_args f vs (Prim a p es) ≝ const_call_args_list f vs es
const_call_args f vs (Letrec a ves e1) ≝
 let vs1 = const_call_args_list f vs (map snd ves) in const_call_args f vs1 e1
const_call_args f vs (Case a e1 v bs d) ≝
 let vs1 = const_call_args f vs e1
 in
   case d of
     None ⇒ const_call_args_list f vs1 (map (λ(v,vs,e). e) bs)
   | Some (v2,d') ⇒
     let vs2 = const_call_args f vs1 d'
     in
       const_call_args_list f vs2 (map (λ(v,vs,e). e) bs)
const_call_args f vs (NestedCase a e v p e' bs) ≝ vs
const_call_args_list f vs [] ≝ vs
const_call_args_list f vs (e::es) ≝
 let vs1 = const_call_args f vs e in const_call_args_list f vs1 es
```

Figure 5.1: `const_call_args` function implementation

6

Results

This chapter summarises and evaluates the results obtained in this project. It provides details on the correctness of the pass, as well as the examples showing its quantitative results.

6.1 Correctness Proof

The correctness of inlining has been proven on two levels: the relation level and the implementation level. The relation-level proof is independent of the implementation, and it proves that any inlining pass that satisfies the same transformation schema preserves the semantics of the input program. The implementation-level proof is specific to the inlining pass implementation and proves that the implementation is correct. The correctness proofs of inlining are the main results of this project. They are:

Firstly, the correctness of the relation that encaptures the semantic transformation of the inlining pass:

Theorem 25

$$\vdash \mathit{list\_subst\_rel}\ []\ x\ y \wedge \mathit{no\_shadowing}\ x \wedge \mathit{closed}\ x \Rightarrow x \cong y$$

The equivalence proof for the multi-inlining relation is the most important part of the work. This proof establishes that the multi-inlining relation preserves the semantics of the original program. And because the multi-inlining relation encodes the transformation performed by the inlining pass, this theorem proves the correctness of inlining as a concept.

Secondly, the correctness of the inlining pass implementation:

Theorem 26

$$\vdash \mathit{no\_shadowing}\ (\mathit{exp\_of}\ x) \wedge \mathit{closed}\ (\mathit{exp\_of}\ x) \wedge (t, ns_1) = \mathit{inline\_all}\ h\ x \Rightarrow \mathit{exp\_of}\ x \cong \mathit{exp\_of}\ t$$

This theorem establishes the correctness of the full inlining pass. Like the relation-level proof, this theorem is defined for any expression that satisfies no_shadowing

and is closed. It's derived from the equivalence theorem for the relation, thereby providing a thorough proof of the full inlining pass.

Together those two theorems prove that the inlining pass does not alter the semantics of the input program.

6.2 Implemented Optimization Pass

The inlining pass was implemented in HOL4. The implementation is based on the formalization of the inlining relation. The implementation can be split into two distinctive parts: the inlining pass and the specialization function. The inlining pass is the main part of the implementation. It is responsible for traversing the AST and applying the inlining relation to each expression. The specialization transformation is a function that takes a `letrec` definition and a list of arguments and returns a specialized version of the definition. This function is used by the inlining pass to create specialized versions of `letrec` definitions.

There are two versions of the inlining pass: the fully verified version and the partially verified version. The fully verified version doesn't use the specialization transformation. It also doesn't recurse into the inserted expressions, due to the freshening function not being verified. The partially verified version uses both the specialization transformation and the freshening function. It is capable of inlining inside the inserted expressions and of inlining specialized recursive functions.

6.3 Examples

To showcase the capabilities of the inlining pass, a few examples are provided. These examples were transformed using the partially unverified pass. This version of the pass is unverified since it uses both the specialization transformation and the freshening function to inline inside the inserted expressions.

Example 1 — simple inlining of a constant:

```
let x = 7
in (\m -> x)
```

The expression after inlining:

```
let x = 7
in (\m -> 7)
```

This example shows the simplest case of inlining. The inlining pass simply replaces the variable `x` with its value `7`.

Example 2 — transitively inlining let bindings:

```
let f = (\x -> x + 5)
```

```
in let y = f 1
in 1 + y
```

The expression after inlining:

```
let f = (lam (x) (+ x (int 5)))
in let y = (let x = 1 in 1 + 5)
in 1 + (let x = 1 in 1 + 5)
```

This example shows the inlining of a let binding `f` that is used in another let binding `y`. The inlining pass first inlines the definition of `f`, which in turn creates a new let binding for `x`. Both of those bindings are then inlined into the reference of `y` in the resulting expression.

This showcases the inlining pass's ability to inline let bindings transitively. And continue with inlining inside of an inserted expression.

A thing of note in the above example is that the inlining pass paved the way for other optimizations to be applied. In this case, dead code elimination can remove both bindings for `x`. This would allow for the constant value to be computed at compile time by yet another optimization pass.

Example 3 — specialization:

```
letrec map = \f -> \lst -> case lst of
  [] -> []
  (x:xs) -> f x : map f xs
in
  let foo = (\x -> x + 1)
  in map foo
```

The expression after inlining:

```
letrec map = \f -> \lst -> case lst of
  [] -> []
  (x:xs) -> f x : map f xs
in
  let foo = (\x -> x + 1)
  in
    letrec map = \f -> \lst -> case lst of
      [] -> []
      (x:xs) -> foo x : map foo xs
    in map foo
```

This example showcases the inlining pass's ability to specialize recursive functions. The inlining pass first inlines the definition of `map` into its reference. It then specializes the definition of `map` by replacing the argument `f` with the specialized version `foo`.

Note that the specialization algorithm only specializes in the definition of `map`, without

removing the original argument. This optimization of removing unused bindings can be performed by another optimization pass, similar to dead code elimination.

7

Conclusions

This chapter provides a top-level summary of the completed work as well as thoughts on related and future work.

7.1 Summary of the completed work

As a result of this project, several significant milestones were accomplished:

Firstly, an inlining pass for PureCake was implemented. The pass was implemented using the first intermediate language — PureLang. The design of this pass is heuristic-agnostic, allowing it to be used with any inlining strategy. This makes the implementation open for future improvements without changing the overall implementation. This implementation also accommodates the use of specialization making it easy to incorporate into the pass.

Secondly, a correctness proof for the inlining transformation, which is independent of the implementation, was defined and proven correct. This proof shows that any inlining pass that satisfies the same transformation schema preserves the semantics of the input program. It can be applied universally to any inlining pass implementation that follows the same syntactic transformations. This approach provides an advantage in allowing for the proof to remain relevant and reusable even if the implementation changes.

A noteworthy milestone was the successful proof of a correctness theorem for the specific inlining pass implementation. This result gives the guarantee that the implementation is safe to use, as it does not alter the semantics of the input program.

Lastly, the project saw the successful development of an unverified `letrec` specialization pass. The implementation and correctness proof of the inlining pass already accommodates the use of this function specialization transformation. Furthermore, the verification methodology employed in this project paves the way for the function specialization pass to be proven correct.

7.2 Related Work

The topic of verified compilation has been getting more and more traction recently, so there are quite a few verified optimizing compilers out there. The most famous of

all CompCert [4] implements multiple optimization passes, one of which is function inlining. However, the main difference between CompCert's function inlining is that it is done for an imperative language, that only supports top-level function declarations.

Another well-known verified compiler is CakeML, which is a verified compiler for an ML-like, impure, eagerly evaluated, functional language. PureCake uses CakeML as a back-end, to generate machine code. CakeML being an optimizing compiler, has its implementation of function inlining [5]. Though the main difference with this work is that CakeML is still an impure and eagerly-evaluated language.

The research on the Pilsner compiler [6] also touches on topics relevant to this work. It is a compiler for an ML-like language that goes through a CPS-based intermediate language to an assembly-like target. Function inlining optimization is implemented for the intermediate language, because of that the inlining is limited to top-level functions.

7.3 Future Work

Future work on this project includes the following:

- The current implementation of inlining only uses a very simple size-based heuristic. This should be changed to a more sophisticated one, preferably one inspired by the GHC compiler [7]. It should use call graphs to analyze the dependencies and call frequencies of the bindings. It should distinguish between different types of inlinable bindings. For example:
  - Bindings that are unconditionally worth inlining, like non-recursive bindings that are referenced only once;
  - Bindings that are conditionally worth inlining, because it will allow for a specific expression to be further simplified;
  - Bindings that are small enough, so that the cost of inlining is cheaper than the cost of the function call.

  Detecting all of those cases can be hard and costly, so to simplify the conditions, empirically defined constants should be used to approximate the cost and benefit of inlining, just like in the GHC. A similar approach to [1] could then be used to find the best values for those magic numbers. A comprehensive set of benchmarks should be composed for this purpose, to be able to compare the performance of compiled programs with different parameters. A good starting point for such benchmarks is the set of programs used to evaluate the performance of the PureCake compiler.
- At the time of writing, the specialization pass is not verified. This should be done, to be able to introduce the whole inlining pass into the main compiler pipeline.

- The biggest benefit of an inlining pass is that it allows for other optimization to be triggered. And so, the more optimizations on the PureLang level are implemented, the more benefit the inlining pass will bring.

Bibliography

[1] Celeste Hollenbeck, Michael FP O'Boyle, and Michel Steuwer. Investigating magic numbers: improving the inlining heuristic in the glasgow haskell compiler. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pages 81–94, 2022.

[2] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. PureCake: A Verified Compiler for a Lazy Functional Language. In *Programming Language Design and Implementation (PLDI)*. ACM, 2023.

[3] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[5] Alexander Mihajlovic. Verifying Function Inlinining in CakeML. MSc. thesis, 2018.

[6] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. *SIGPLAN Not.*, 50(9):166–178, aug 2015.

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

[8] Nayeong Song. Comparison between lazy and strict evaluation. B.S. thesis, 2020.

[9] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software*, 174:110884, 2021.