Verified Inlining and Specialisation for PureCake

Hrutvik Kanabar[1] (✉), Kacper Korban[2], and Magnus O. Myreen[2]

[1] University of Kent, Canterbury, UK
`hrk32@cantab.ac.uk`

[2] Chalmers University of Technology, Gothenburg, Sweden
`kacper.f.korban@gmail.com` `myreen@chalmers.se`

Abstract. Inlining is a crucial optimisation when compiling functional programming languages. This paper describes how we have implemented and verified function inlining and loop specialisation for PureCake, a verified compiler for a Haskell-like (purely functional, lazy) programming language. A novel aspect of our formalisation is that we justify inlining by pushing and pulling `let`-bindings. All of our work has been mechanised in the HOL4 interactive theorem prover.

Keywords: verified compilation · function inlining · loop optimisation · functional programming · machine-checked proofs

1 Introduction

It can be tricky to generate high-quality code from lazy, purely functional programs for a number of reasons. One of these reasons is that functional programming encourages a brief declarative style that makes heavy use of shorthands (*e.g.*, for partially-applied functions) and higher-order functions [8]. Producing good code from such input requires a well-developed inliner, as noted [17] by the developers of the Glasgow Haskell Compiler (GHC):

> "One of the trickiest aspects of a compiler for a functional language is the handling of inlining. [...] Effective inlining is particularly crucial in getting good performance."

This paper is about implementing and verifying an inliner that can specialise loops for PureCake, an end-to-end verified compiler for a Haskell-like language [10].

The inliner by example. The following simple example demonstrates what our inliner does. Imagine that a programmer is to write a function that increments every element of a list of integers. The programmer should write:

```
suc_list = map (+1)
```

Here, the programmer has relied on the library function `map` below to perform the necessary list traversal.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

To generate high-quality code for `suc_list`, the compiler must *both* inline *and* specialise `map`. Our inliner takes the definition of `suc_list` above and produces the following code.

```
suc_list =
  let map' xs =
    case xs of
      []     -> []
      (y:ys) -> y + 1 : map' ys
  in map'
```

In particular, the inliner has combined the following code transformations:

– selective expansion of function definitions at call sites; and
– loop specialisation of recursive functions with known arguments (*e.g.*, argument `f` to `map` is always `(+1)` in `suc_list`).

Contributions. Our work adds verified inlining and loop specialisation to PureCake. Our inliner is capable of optimisations such as the one above. More specifically, we make the following contributions:

1. We define and prove sound a relation that encapsulates an envelope of semantics-preserving inlinings (§ 4). This relation is independent of the heuristics of any real implementation. It is proved sound using a novel formalisation of inlining as pushing/pulling of `let`-bindings.
2. We derive sound equational principles that allow us to lift out arguments which remain constant during recursion, such as `f` in `map` in the example above (§ 5). These principles are phrased such that they can be used in the relation above and have the effect of specialising loops.
3. We implement an inliner that can specialise loops and verify that its action preserves semantics, relying on the formalisations above (§ 6).
4. We integrate our inliner into the PureCake compiler and its verification (§ 7).

All of our work is mechanised using the HOL4 interactive theorem prover, and our development is open-source.[3] To the best of our knowledge, ours is the first verified inliner for a lazy functional programming language, and the first verified loop specialiser for any functional language.

2 The Inliner by Example

We begin with a high-level explanation of how our inliner works, before diving into verification details in later sections. We will show the transformations the

[3] https://github.com/cakeml/pure, see also our artifact hosted on Zenodo [9].

inliner performs step-by-step. As a running example, we use the code from the previous section with one modification: we lift `(+1)` to a separate function `add1` for clarity. The input code after this modification is as follows:

```
suc_list = map add1

add1 i = i + 1

map f [] = []
map f (x:xs) = f x : map f xs

main = ...
```

Our inliner is installed very early in the PureCake compiler, directly after parsing and *binding group analysis*. Binding group analysis processes the program above to the code below, breaking up the mutually recursive bindings into a nesting of `let`-expressions. Note that there is no dependency between `add1` and `map`, so their definitions could be reordered; for this example we put `add1` first.

```
18  let add1 i = i + 1 in
19  let map f l = case l of
20                  []     -> []
21                  (x:xs) -> f x : map f xs in
22  let suc_list = map add1 in
23  let main = ... in main
```

The inliner receives this program as input. As it traverses the program, it records known definitions that it may wish to inline later on. In particular, it maintains a mapping from names to their definitions, which starts off empty. Therefore, after processing line 18 (*i.e.*, the definition of `add1`), the mapping contains only the definition of `add1`, that is, `\i -> i + 1`.

The inliner then moves to line 19, the `let`-expression that defines `map`. The definition of `map` is recursive, so the inliner analyses it to determine whether any of its arguments remain constant over all recursive calls. In the case of `map`, it finds that the first argument, `f`, remains constant. This means that it can *loop specialise* `map` to produce the following equivalent definition.

```
let map f =
    let map' l = case l of
                   []     -> []
                   (x:xs) -> f x : map' xs
    in map'
in ...
```

Our inliner does not alter the definition of `map` in the program, but it does add this equivalent definition to its mapping of known definitions. We will very soon see why it is useful to pull out the constant argument `f`.

The inliner moves on to the definition of `suc_list` on line 22.

```
let suc_list = map add1 in ...
```

After pulling out the constant argument `f` above, the inliner considers `map` to be a single-argument function. Therefore, the application `map add1` here seems fully applied and the inliner will rewrite it. First, it transforms `map add1` into the following.

```
let suc_list =
    let f = add1 in
    let map' l = case l of
                   []     -> []
                   (x:xs) -> f x : map' xs
    in map'
in ...
```

Notice the use of a binding `let f = add1` to assign the constant argument `f` of `map`. Then, the inliner recurses into this expression, replacing `f` by `add1` in the second row of the pattern match:

```
                   (x:xs) -> add1 x : map' xs
```

The inliner recurses again into the modified subexpression `add1 x`, and realises that `add1` (which is mapped to `\i -> i + 1`) is fully applied. Therefore, it inlines `add1` too:

```
                   (x:xs) -> (let i = x in i + 1) : map' xs
```

Once again, the inliner recurses on the modified subexpression, turning the innermost `i` into `x`:

```
                   (x:xs) -> (let i = x in x + 1) : map' xs
```

The final code produced by the inliner is below. The definition of `suc_list` has been rewritten so extensively that it now resembles a copy of `map` which has been specialised to the `add1` function.

```
let add1 i = i + 1 in
let map f l = case l of
                []     -> []
                (x:xs) -> f x : map f xs in
let suc_list =
    let f = add1 in
    let map' l = case l of
                   []     -> []
                   (x:xs) -> (let i = x in x + 1) : map' xs
```

```
50       in map'
51   in let main = ... in main
```

Some dead code remains, *e.g.*, `let f = add1` (line 46) and `let i = x` (line 49). We perform a simple dead code elimination pass immediately after the inliner to remove these.

Single-pass optimisation. Note that our inliner does not make multiple passes over input code, in contrast to the presentation above. It performs a single top-down pass over its input, calling itself recursively only on function applications or variables that it has successfully rewritten. The depth of this recursion is bounded by a simple user-configurable recursion limit.

3 Setting: PureCake

We implement and verify our inlining and specialisation optimisations as part of the verified compiler PureCake. In this section, we describe both the PureCake project at a high level, and the key aspects of its formalisation on which we rely.

What is PureCake? PureCake [10] is an end-to-end verified compiler for a Haskell-like language known as PureLang. Here, a "Haskell-like" language is one which: is purely functional with monadic effects; evaluates lazily; and has a syntax resembling that of Haskell. PureCake compiles PureLang to the CakeML language, which is call-by-value and ML-like, and has an end-to-end verified compiler [12,14]. CakeML targets machine code, so PureCake and CakeML can be composed to produce end-to-end guarantees for the compilation of PureLang to machine code [10, §6].

The PureCake compiler is designed to be realistic: it accepts a featureful input language and generates performant code. This makes it an ideal setting for verified inlining and specialisation optimisations. We add these to PureCake as PureLang-to-PureLang transformations.

Formalisation details. PureLang is formalised using two ASTs: *compiler* expressions and *semantic* expressions, denoted ce and e respectively [10, §3.2]. The compiler implementation uses compiler expressions, and their semantics is given by desugaring into semantic expressions (denoted `desugar`, of type $ce \to e$).

The call-by-name operational semantics of PureLang is defined over its simpler semantic expressions [10, §3.3]. This semantics admits an equational theory [10, §3.4] which is sound and complete with respect to contextual equivalence. Its equivalence relation, $e_1 \cong e_2$, is based on an untyped applicative bisimulation from Abramsky's lazy λ-calculus [1] and is proved congruent via Howe's method [7], *i.e.*, expressions composed of equivalent subexpressions are themselves equivalent.

PureCake's compiler passes are verified in two stages.

1. A binary syntactic relation is defined over semantic expressions ($e_1 \mathcal{R} e_2$). The relation is proved to imply $e_1 \cong e_2$, so e_1 and e_2 have identical observable behaviour in all contexts. Intuitively, the syntactic relation carves out an envelope of possible valid transformations, independent of the heuristics of any real implementation.
2. The implementation is then defined over compiler expressions, with concrete heuristics. It is verified to perform only those valid transformations expressed by the syntactic relation.

Composition of the two stages produces the overall proof that the action of the compiler implementation preserves semantics. A key benefit of this approach is that heuristics remain an implementation detail in stage 2, and can be changed without incurring the significant proof obligations of stage 1.

Approach and paper outline. We can now describe more precisely the steps we took to add inlining and loop specialisation to the PureCake compiler.

§ 4 *(stage 1)* We defined a relation which captures an envelope of valid inlining transformations, and proved that this relation preserves semantics.
§ 5 We formalised loop specialisation using PureLang's equational theory such that it can be used in the envelope mentioned above.
§ 6 *(stage 2)* We implemented the overall inlining and specialisation transformations over compiler expressions, verifying that they fit the envelopes.
§ 7 We integrated our inliner into the PureCake compiler pipeline and its top-level correctness result.
§ 8 We benchmarked the performance of the output of the inliner.

4 Inlining as a Relational Envelope

In this section, we define a relation which characterises all the inlinings that we wish to perform. We then prove that any code transformation contained within this relational envelope must preserve semantics.

4.1 Understanding the relation

We begin by describing the intuition behind our relation.

Inlining is not substitution. Inlining is a more complex transformation than substitution or β-conversion. If we were to view inlining as a special case of these, we would generate unsatisfactory code. In particular, consider the example below: inlining based on substitution must replace all three occurrences of `f` with its definition; inlining based on β-conversion would remove the `let`-binding.

```
let f i = 5 in f 1 : map f xs ++ map f ys
```

By contrast, a real inliner must be able to choose whether to inline a definition *per use of that definition*. In other words, the inliner should decide which usages of a given definition are rewritten on a case-by-case basis. For the example above, a real inliner should produce the code below. Note that it chooses to inline the function `f` only at the usage which fully applies it.

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

Of course, a real inliner would further transform `(\i -> 5) 1` into `5` (this is in fact a β-conversion). For clarity in this example, we do not show that step.

Inlining is a series of `let` transformations. The key intuition behind our inlining transformations is as follows. We push **let**-bindings into expressions as far as possible, rewrite the result, then pull the bindings out again. We illustrate this by example below, starting from the same initial code as above.

```
let f i = 5 in f 1 : map f xs ++ map f ys
```

We now push in the `let`-binding which defines `f` to produce a series of equivalent expressions. First, we push it in one step past the list constructor `(:)`:

```
(let f i = 5 in f 1) :
    (let f i = 5 in map f xs ++ map f ys)
```

Next, we push it in through the function application `f 1`:

```
(let f i = 5 in f) (let f i = 5 in 1) :
    (let f i = 5 in map f xs ++ map f ys)
```

Now, we choose to rewrite the use of `f` under the first `let f i = 5` to `\i -> 5`:

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

Note that we have chosen *not* to perform any other rewrites of `f`, because other uses of `f` are not fully applied.

We can now reverse the pushing in of `let`-bindings, *i.e.*, we pull them out instead. The final result is as follows, where `f` is inlined exactly as we wanted:

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

Stacking `let` transformations. Above, our example shows how we can inline a *single* `let`-binding: we push it inwards, use it for rewriting, and pull it outwards back to its original position. We can generalise this straightforwardly to handle a *list* of `let`-bindings. This mimics the implementation of a real inliner, which must carry with it a collection of definitions it may wish to inline.

Consider the following example, in which an inliner attempts to rewrite the expression `g 3 + 7` and carries definitions `f i = 5`; `h i = 2`; `g i = f i + 1`.

```
let f i = 5 in
let h i = 2 in
let g i = f i + 1 in
  g 3 + 7
```

Just as with a single `let`-binding, we can push in the stack of `let`-bindings, rewrite, and pull them out again. This produces the following expression.

```
let f i = 5 in
let h i = 2 in
let g i = f i + 1 in
  (\i -> (\i -> 5) i + 1) 3 + 7
```

The only complication in generalising to a stack of `let`-bindings is that some definitions can depend on others. In the example above, the definition of `g` depends on `f`. This is why we model the bindings as a *list*: this preserves scoping correctly, ensuring we do not break any dependencies between definitions.

Note that this intuition of pushing in and pulling out of `let`-bindings applies only to the formalisation that justifies our inlining rewrites. The implementation of our inliner performs no such push/pull transformations: as one might expect, it merely carries around a simple (unordered) map of variable names to their definitions. This map represents exactly the set of definitions that the inliner may wish to use for rewriting at usage sites.

4.2 Defining a Semantics-Preserving Envelope

We now describe an inductive relation, $l \Vdash e_1 \rightsquigarrow e_2$, which characterises all of the inlining transformations that we perform. We prove that any transformation described by the relation lies within the equational theory of PureLang ($\cong$, § 3). Therefore, the relation describes only semantics-preserving transformations.

The relation $l \Vdash e_1 \rightsquigarrow e_2$ should be read as follows: expression e_1 can be transformed into expression e_2 under the definitions in the list l. Both e_1 and e_2 are PureLang semantic expressions, and l is a list of definitions. Each such definition is of the form $x \leftarrow e$, associating name x with semantic expression e. We will first describe the formal *meaning* of $l \Vdash e_1 \rightsquigarrow e_2$, which is best understood via its soundness theorem, Theorem 1. Then in following subsections, we describe key parts of the *definition* of $\rightsquigarrow$.

Theorem 1 relates derivations of $l \Vdash e_1 \rightsquigarrow e_2$ with $\cong$, PureLang's equational theory, assuming pre and lets_ok. The definitions of pre and lets_ok are shown in Figure 1—they enforce distinct variable names between both the expression e_1 and each of the definitions in l to avoid inadvertent clashes or capture.

$$\mathsf{vars\_of}\ l \overset{\text{def}}{=} \bigcup \Big\{ \{x\} \cup \mathsf{freevars}\ e \ \Big|\ \mathsf{mem}\ (x \leftarrow e)\ l \Big\}$$

$$\mathsf{pre}\ l\ e \overset{\text{def}}{=} \mathsf{barendregt}\ e\ \wedge\ \mathsf{boundvars}\ e \# \mathsf{vars\_of}\ l$$

$$\mathsf{lets\_ok}\ [] \overset{\text{def}}{=} \mathsf{T}$$
$$\mathsf{lets\_ok}\ ((x \leftarrow e) :: l) \overset{\text{def}}{=}$$
$$x \notin \mathsf{freevars}\ e\ \wedge\ \big(\{x\} \cup \mathsf{freevars}\ e\big) \# \{x \mid \exists e.\ \mathsf{mem}\ (x \leftarrow e)\ l\}\ \wedge\ \mathsf{lets\_ok}\ l$$

Fig. 1. The definition of pre and lets_ok. Here, the # predicate returns true only for disjoint sets: $s_1 \# s_2 \overset{\text{def}}{=} (s_1 \cap s_2 = \varnothing)$.

Theorem 1. Soundness of $l \Vdash e_1 \rightsquigarrow e_2$.

$$\vdash\ l \Vdash e_1 \rightsquigarrow e_2\ \wedge\ \mathsf{pre}\ l\ e_1\ \wedge\ \mathsf{lets\_ok}\ l\ \Rightarrow\ \mathbf{lets}\ l\ e_1 \cong \mathbf{lets}\ l\ e_2$$

where $\mathbf{lets}\ []\ e \overset{\text{def}}{=} e$ and $\mathbf{lets}\ ((x \leftarrow e') :: l)\ e \overset{\text{def}}{=} \mathbf{let}\ x = e'\ \mathbf{in}\ (\mathbf{lets}\ l\ e)$

In particular, expressions e_1 and e_2 related in the context of definitions l produce equal expressions (according to $\cong$) under the stack of **let**-bindings corresponding to l. The latter correspondence is encapsulated by the definition of **lets**, which nests **let**-bindings. This theorem is proved by induction over the derivation of $l \Vdash e_1 \rightsquigarrow e_2$. In upcoming subsections, we will examine key rules of $\rightsquigarrow$ and their cases in this inductive proof.

When the inliner is first invoked, it is passed an entire PureLang program and has no knowledge of any definitions. In other words, its mapping of variable names to known definitions is empty, corresponding to the list l being empty ($[]$). In this case, we can simplify Theorem 1 by instantiating $l \mapsto []$, and unfolding the definitions of pre l and lets_ok l. This produces the following theorem:

Theorem 2. Soundness of $[] \Vdash e_1 \rightsquigarrow e_2$.

$$\vdash\ [] \Vdash e_1 \rightsquigarrow e_2\ \wedge\ \mathsf{barendregt}\ e_1\ \wedge\ \mathsf{closed}\ e_1\ \Rightarrow\ e_1 \cong e_2$$

We can read this as follows: if we can transform some closed e_1 which satisfies barendregt to some e_2 according to $\rightsquigarrow$, then e_1 and e_2 are equivalent. The barendregt predicate restricts the variable naming convention within e_1 to avoid problems with variable capture, because PureLang has explicit names. In particular, barendregt is the well known Barendregt variable convention that enforces unique free/bound variable names across an entire program [3].

The precise definition of barendregt is not necessary here. Suffice it to say that in order to discharge this assumption, our inliner implementation will rely on a freshening pass. This pass α-renames programs such that they obey the Barendregt variable convention, and therefore satisfy barendregt.

Reflexivity. We must allow the inliner to choose whether to rewrite a usage site on a case-by-case basis (§ 4.1). Therefore, the inliner must be allowed *not* to inline, *i.e.*, it must be able to leave an expression unchanged. Therefore the $\rightsquigarrow$ relation has a reflexivity rule:

$$\frac{}{l \Vdash e \rightsquigarrow e}\ \textsc{refl}$$

The REFL case of the proof of Theorem 1 boils down to showing the equation $\mathbf{lets}\, l\; e \;\cong\; \mathbf{lets}\, l\; e$, which is trivial due to reflexivity of $\cong$.

Inlining. The simplest rule for inlining uses a definition found in the list l (where mem denotes list membership) to rewrite a variable:

$$\frac{\mathsf{mem}\ (x \leftarrow e)\ l}{l \Vdash \mathbf{var}\, x \rightsquigarrow e}\ \textsc{inline}$$

In particular, if l associates name x with definition e, then the variable $\mathbf{var}\, x$ can be replaced by expression e. The INLINE case of Theorem 1 requires establishing:

$$\vdash\ \mathsf{mem}\ (x \leftarrow e)\ l\ \wedge\ \mathsf{lets\_ok}\ l\ \wedge\ \mathsf{pre}\ l\ \ (\mathbf{var}\, x) \Rightarrow \mathbf{lets}\, l\ (\mathbf{var}\, x)\ \cong\ \mathbf{lets}\, l\; e$$

Proof outline. We first derive a lemma that allows us to duplicate a **let**-binding from l, assuming $\mathsf{lets\_ok}$ (defined in Figure 1 such that it enables this lemma):

$$\vdash\ \mathsf{mem}\ (x \leftarrow e)\ l \wedge \mathsf{lets\_ok}\ l\ \Rightarrow\ \mathbf{lets}\, l\; e'\ \cong\ \mathbf{lets}\, l\ (\mathbf{let}\, x = e\ \mathbf{in}\ e') \qquad \textsc{Let-dup}$$

Equipped with the LET-DUP lemma, we proceed as follows:

$$\begin{aligned} \mathbf{lets}\, l\ (\mathbf{var}\, x)\ &\cong\ \mathbf{lets}\, l\ (\mathbf{let}\, x = e\ \mathbf{in}\ \mathbf{var}\, x) && (\textsc{Let-dup}) \\ &\cong\ \mathbf{lets}\, l\; e && (\text{trivial}) \end{aligned}$$

□

Let. We can now inline known definitions, but we must be able to learn those definitions in the first place. The rule LET allows us to add a **let**-bound definition to the stack l, using the append operator ($+\!\!+$).

$$\frac{l \Vdash e_1 \rightsquigarrow e_1' \qquad l +\!\!+ (x \leftarrow e_1) \Vdash e_2 \rightsquigarrow e_2'}{l \Vdash (\mathbf{let}\, x = e_1\ \mathbf{in}\ e_2) \rightsquigarrow (\mathbf{let}\, x = e_1'\ \mathbf{in}\ e_2')}\ \textsc{Let}$$

Proof outline. LET case of Theorem 1.

$$\begin{aligned} & \mathbf{lets}\, l\ (\mathbf{let}\, x = e_1\ \mathbf{in}\ e_2) \\ \cong\ & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_1))\ e_2 && (\text{definition of } \mathbf{lets}) \\ \cong\ & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_1))\ e_2' && (\textsc{ih} \text{ for } e_2) \\ \cong\ & \mathbf{lets}\, l\ (\mathbf{let}\, x = e_1\ \mathbf{in}\ e_2') && (\text{definition of } \mathbf{lets}) \\ \cong\ & \mathbf{let}\, x = (\mathbf{lets}\, l\; e_1)\ \ \mathbf{in}\ \ (\mathbf{lets}\, l\; e_2') && (\text{push in } \mathbf{lets}) \\ \cong\ & \mathbf{let}\, x = (\mathbf{lets}\, l\; e_1')\ \ \mathbf{in}\ \ (\mathbf{lets}\, l\; e_2') && (\textsc{ih} \text{ for } e_1) \\ \cong\ & \mathbf{lets}\, l\ (\mathbf{let}\, x = e_1'\ \mathbf{in}\ e_2') && (\text{pull out } \mathbf{lets}) \end{aligned}$$

□

Above, we can push and pull **lets** through **let** because the precondition pre enforces sufficiently distinct variable names.

Note that this rule records the unmodified expression e_1 in the stack of known definitions l. It could instead use the $\rightsquigarrow$-transformed expression e_1'. The proof strategy with this modification is essentially unchanged, except we must reverse our applications of the inductive hypotheses.

Congruences. We must be able to apply $\rightsquigarrow$ within subexpressions. Therefore, we have several *congruence* rules, such as the following:

$$\frac{l \Vdash e_1 \rightsquigarrow e_1' \qquad l \Vdash e_2 \rightsquigarrow e_2'}{l \Vdash (e_1 \cdot e_2) \rightsquigarrow (e_1' \cdot e_2')}\ \textsc{App-cong} \qquad \frac{l \Vdash e \rightsquigarrow e'}{l \Vdash (\lambda x.\ e) \rightsquigarrow (\lambda x.\ e')}\ \textsc{Lam-cong}$$

$$\frac{\forall i.\ l \Vdash e_i \rightsquigarrow e_i' \qquad l \Vdash e \rightsquigarrow e'}{l \Vdash (\textbf{letrec}\ \overline{x_n = e_n}\ \textbf{in}\ e) \rightsquigarrow \left(\textbf{letrec}\ \overline{x_n = e_n'}\ \textbf{in}\ e'\right)}\ \textsc{Letrec-cong}$$

Each such case in Theorem 1 requires showing that we can push/pull **lets** into/out of subexpressions. Once again, the precondition pre permits this by enforcing sufficiently distinct variable names. The remainder of the proof follows from congruence of $\cong$.

Simplification. The following rule allows $\rightsquigarrow$ to carry out any transformation that preserves $\cong$:

$$\frac{l \Vdash e_1 \rightsquigarrow e_2 \qquad e_2 \cong e_2'}{l \Vdash e_1 \rightsquigarrow e_2'}\ \textsc{simp}$$

The $\textsc{simp}$ case in Theorem 1 is a direct consequence of the transitivity of $\cong$.

This rule permits the inliner to modify (and in particular, simplify) generated expressions during its operation. There are two important uses of this ability:

– Turning fully applied λ-abstractions into a stack of **let**-bindings. This allows recursive applications of inlining (see rule $\textsc{trans}$ below).

$$(\lambda x_1.\ \lambda x_2.\ \ldots\ \lambda x_n.\ e) \cdot e_1 \cdot e_2 \cdot \ldots \cdot e_n \cong \textbf{lets}\ (x_1 \leftarrow e_1 :: x_2 \leftarrow e_2 :: \ldots :: x_n \leftarrow e_n)\ e \quad (1)$$

– Freshening names of bound variables (*i.e.*, α-renaming). This happens directly before application of the rule $\textsc{trans}$ below.

Transitivity. To permit recursion into recently inlined expressions, $\rightsquigarrow$ has a transitivity rule:

$$\frac{l \Vdash e_1 \rightsquigarrow e_2 \qquad l \Vdash e_2 \rightsquigarrow e_3 \qquad \mathsf{pre}\ l\ e_2}{l \Vdash e_1 \rightsquigarrow e_3}\ \textsc{trans}$$

In particular, e_1 can be transformed to e_3 if there is some intervening e_2 which can act as a stepping stone.

Unusually, we require precondition pre to hold of intermediate expression e_2. This is demanded by the proof of Theorem 1, in which we can only instantiate inductive hypotheses if we first establish pre. Unfortunately, $l \Vdash e_1 \rightsquigarrow e_2$ and $\mathsf{pre}\ l\ e_1$ are not enough to derive $\mathsf{pre}\ l\ e_2$. Fortunately, we can freshen bound variable names (*i.e.*, α-rename) sufficiently to establish pre, and justify this freshening using rule SIMP above.

Specialisation. The $\rightsquigarrow$ relation must be able to support loop specialisation, as described for the `map` function in § 2. Therefore, it has a rule SPEC which permits conversion of a **letrec** into a **let**, as long as there is a proof that the conversion preserves $\cong$.

$$\frac{\begin{array}{cc} l \Vdash e_1 \rightsquigarrow e_1' & (\forall e.\ \textbf{letrec}\,x = e_1\ \textbf{in}\ e \cong \textbf{let}\,x = e_2\ \textbf{in}\ e) \\ l +\!\!+ (x \leftarrow e_2) \Vdash e_3 \rightsquigarrow e_3' \quad \mathsf{disjoint\_names}\ e_2\ e_3 & x \notin \mathsf{freevars}\ e_2 \end{array}}{l \Vdash \textbf{letrec}\,x = e_1\ \textbf{in}\ e_3 \rightsquigarrow \textbf{letrec}\,x = e_1'\ \textbf{in}\ e_3'}\ \textsc{spec}$$

That is, if we can $\cong$-convert some **letrec** $x = e_1$ to some **let** $x = e_2$, then we can append $x \leftarrow e_2$ to the stack of known definitions when processing **letrec** body e_3. Again, we require restrictions on variable naming: the variables bound in e_2 and e_3 must be disjoint, and the bound variable x must not appear free in e_2.

Proof outline. SPEC case of Theorem 1.

$$\begin{array}{lll}
& \textbf{lets}\,l\ (\textbf{letrec}\,x = e_1\ \textbf{in}\ e_3) & \\
\cong & \textbf{lets}\,l\ (\textbf{let}\,x = e_2\ \textbf{in}\ e_3) & \text{(assumption of rule)} \\
\cong & \textbf{lets}\ (l +\!\!+ (x \leftarrow e_2))\ e_3 & \text{(definition of } \textbf{lets}\text{)} \\
\cong & \textbf{lets}\ (l +\!\!+ (x \leftarrow e_2))\ e_3' & (\textsc{ih} \text{ for } e_3) \\
\cong & \textbf{lets}\,l\ (\textbf{let}\,x = e_2\ \textbf{in}\ e_3') & \text{(definition of } \textbf{lets}\text{)} \\
\cong & \textbf{lets}\,l\ (\textbf{letrec}\,x = e_1\ \textbf{in}\ e_3') & \text{(ass. of rule, symmetry of } \cong\text{)} \\
\cong & \textbf{letrec}\,x = (\textbf{lets}\,l\,e_1)\ \textbf{in lets}\,l\,e_3' & \text{(push } \textbf{lets}\text{)} \\
\cong & \textbf{letrec}\,x = (\textbf{lets}\,l\,e_1')\ \textbf{in lets}\,l\,e_3' & (\textsc{ih} \text{ for } e_1) \\
\cong & \textbf{lets}\,l\ (\textbf{letrec}\,x = e_1'\ \textbf{in}\ e_3') & \text{(pull out } \textbf{lets}\text{)}
\end{array}$$

□

5 Specialisation of Recursive Bindings

Our example in § 2 showed that our inliner can specialise applications of recursive functions such as `map` to known arguments such as `add1`. This is possible whenever constant arguments such as `f` can be pulled out of the recursion. That is, whenever we can transform recursive functions like `map` (left) into equivalent code which makes the constant argument explicit using `map'` (right):

```
let map f l =
  case l of
    []     -> []
    (x:xs) -> f x : map f xs
```

```
let map f = let map' l =
              case l of
                []     -> []
                (x:xs) -> f x : map' xs
            in map'
```

In this section, we describe how we prove correctness of such transformations. Critically, our proofs can be used in the SPEC rule of $\rightsquigarrow$ from the previous section.

5.1 Understanding Specialisation

Like $\rightsquigarrow$, our specialisation transformation is justified using equational reasoning. We illustrate the equational steps below, again noting that the implementation is much more direct. We use the `map` example of § 1, eliding parts not relevant to specialisation. The input is therefore as follows:

```
let map f l = ... f x ... map f xs ...
```

We first make a local copy of the recursive definition `map`, named `map'`:

```
let map = let map' f l = ... f x ... map' f xs ...
          in map'
```

We then η-expand the final usage of the copy `map'`:

```
let map = let map' f l = ... f x ... map' f xs ...
          in \f l -> map' f l
```

Next, we pull out the new λ-abstractions to the top-level:

```
let map f l = let map' f l = ... f x ... map' f xs ...
              in map' f l
```

We then α-rename the constant argument in the copy (here, `f` becomes `g`):

```
let map f l = let map' g l = ... g x ... map' g xs ...
              in map' f l
```

The first major step (*transform 1*) replaces the constant argument `g` with the known value to which the function `map'` is always applied, `f`:

```
let map f l = let map' g l = ... f x ... map' f xs ...
              in map' f l
```

The second major step (*transform 2*) deletes the now unused argument `g`. It removes the argument from *both* the definition of `map'` *and* all calls to `map'`:

```
let map f l = let map' l = ... f x ... map' xs ...
              in map' l
```

We push back in some of the top-level λ-abstractions, in this case just `l`:

```
let map f = let map' l = ... f x ... map' xs ...
            in \l -> map' l
```

Finally, η-contraction removes the λ-abstraction over `l`:

```
let map f = let map' l = ... f x ... map' xs ...
            in map'
```

Most of the steps are straightforwardly justified in PureLang's equational theory. However, the steps marked *transform 1* and *transform 2* are more involved. We discuss these below.

5.2 Key Lemmas for Specialisation

Both *transform 1* and *transform 2* require a substitution-like traversal of the entire subexpression under consideration. It is not clear how to justify these traversals using simple equational reasoning in PureLang's theory. Therefore, we resort to more cumbersome simulation proofs to establish $\cong$ by appealing to its definition in terms of PureLang's operational semantics.

For *transform 1*, we prove a theorem of the following form. Here call_with_arg holds only if every application of f in e is applied to **var** y after n arguments, and the names f and y are never rebound within e.

$$\begin{aligned} &\vdash \textsf{call\_with\_arg}\ f\ \overline{x_n}\ y\ e\ \wedge\ \ldots \\ &\quad\Rightarrow\ \textbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ e)\ \ \textbf{in}\ \ ((\textbf{var}\, f)\cdot \overline{e_{1\,n}} \cdot (\textbf{var}\, w) \cdot \overline{e_{2\,m}}) \\ &\qquad\cong\ \textbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ e[{}^{\textbf{var}\, w}/_{y}])\ \ \textbf{in}\ \ ((\textbf{var}\, f)\cdot \overline{e_{1\,n}} \cdot (\textbf{var}\, w) \cdot \overline{e_{2\,m}}) \end{aligned}$$

Though the variable w is free in the theorem above, it is a closed constant expression in most parts of the proof, which simplifies the derivation of this theorem. This is because $\cong$ is defined over open terms in terms of closing substitution and a relation over closed terms. The proof of this theorem is a large simulation based on the semantics of PureLang.

For *transform 2*, we prove a theorem with a similar shape. This time, remove_call_arg is an inductive relation that ensures y never appears in e_1 and relates e_1 to a second expression e_2 in which the relevant argument has been removed from each application of f.

$$\begin{aligned} &\vdash \textsf{remove\_call\_arg}\ f\ \overline{x_n}\ y\ \overline{z_m}\ e_1\ e_2\ \wedge\ \ldots \\ &\quad\Rightarrow\ \textbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ \lambda\, \overline{z_m}\,.\ e_1)\ \ \textbf{in}\ \ ((\textbf{var}\, f)\cdot \overline{e_{3\,n}} \cdot (\textbf{var}\, y) \cdot \overline{e_{4\,m}}) \\ &\qquad\cong\ \textbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda\, \overline{z_m}\,.\ e_2)\ \ \textbf{in}\ \ ((\textbf{var}\, f)\cdot \overline{e_{3\,n}} \cdot \overline{e_{4\,m}}) \end{aligned}$$

We prove this theorem by a large simulation too. The simulation strategy is necessary because **letrec** causes (potentially non-terminating) recursion.

6 Implementing a Correct Inliner

In this section, we describe the implementation of our inliner and the proof that its action lies within the $\rightsquigarrow$ relation described in § 4. We also touch on three other transformations mentioned previously: specialisation, freshening of bound variables, and dead code elimination. Our inliner relies on all three.

6.1 Preliminaries

We implement our inliner within a state monad with the following type:

$$\alpha\ \mathsf{M} \overset{\mathsf{def}}{=} \mathsf{name\ set} \rightarrow (\alpha,\ \mathsf{name\ set})$$

Here, name set is a set of variable names; we will see its usage shortly. This monad has standard return/bind operators, and we will use Haskell-style do-notation to show definitions written within the monad.

The inliner itself has the following signature:

$$\mathsf{inline}\ :\ (h : \mathsf{heuristic}) \rightarrow (k : \mathsf{num}) \rightarrow (m : (\mathsf{name} \mapsto ce)) \rightarrow ce \rightarrow ce\ \mathsf{M}$$

In other words, the inliner transforms compiler expressions to compiler expressions within the state monad, requiring several other inputs:

– An *unordered* mapping m from names to expressions. This is the "memory" of the inliner: the set of known definitions which it can use for rewriting.
– Heuristic h decides whether to "remember" a definition for future inlining. It accepts an expression ce and returns a boolean: if true, the definition should be remembered.
– Natural number k is the recursion limit for the inliner, used to bound its recursion into rewritten expressions.
– The name set parameter hidden within the monad keeps track of all variable names (whether bound or free) in input expression ce. It is used to ensure that sufficiently fresh variable names are chosen when freshening the names of bound variables.

6.2 Inliner implementation

The inliner traverses compiler expressions top-down. During the traversal, it performs two key operations: rewriting a variable to a known definition from memory, and adding a new definition to memory.

Rewriting a variable. There are two kinds of expressions in which the inliner will attempt to rewrite a variable. The first is a lone variable (of the form $\mathbf{var}\,x$), and the second is an application of a variable to some arguments (of the form $(\mathbf{var}\,x) \cdot \ldots$). The latter case is used to inline fully applied functions only.

In the lone variable case, the inliner is defined as follows:

$$\mathsf{inline}_h^k\ m\ (\mathbf{var}\ x) \overset{\mathsf{def}}{=} \begin{cases} \mathsf{return}\ (\mathbf{var}\ x) & x \notin \mathsf{domain}\ m\ \vee\ k = 0\ \vee \\ & \quad m(x) = \lambda y.\ \ldots \\ \mathsf{inline}_h^{k-1}\ m\ ce & m(x) = ce \end{cases}$$

That is, on encountering a free variable x the inliner does one of the following:

- *Leaves the variable unchanged* if the definition of x is unknown, or the recursion limit has been reached, or the definition of x is known to be a λ-abstraction. The last case may seem unusual, but note we do not rewrite variables to λ-abstractions unless the result will be fully applied. This is handled in the application case below.
- *Rewrites the variable* by inserting the expression ce found in memory, and then recurses into ce with a decremented recursion limit.

In the application case, the inliner is defined as follows:

```
$\mathsf{inline}_h^k\ m\ ((\mathbf{var}\ x) \cdot ce_1 \cdot \ldots \cdot ce_n) \overset{\mathsf{def}}{=}$ do
  $[ce'_1,\ \ldots,\ ce'_n] \leftarrow \mathsf{mapM}\ (\mathsf{inline}_h^k\ m)\ [ce_1,\ \ldots,\ ce_n]$;
  if $x \notin \mathsf{domain}\ m\ \vee\ k = 0$ then return $((\mathbf{var}\ x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$ else do
    $ce \leftarrow \mathsf{freshen}\ (m(x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$;          (2)
    case convert_to_lets $ce$ of
    | None $\rightarrow$ return $((\mathbf{var}\ x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$
    | Some $ce'$ $\rightarrow$ $\mathsf{inline}_h^{k-1}\ m\ ce'$
```

That is, on encountering a free variable x applied to n arguments the inliner does the following:

1. Recurses into the arguments to produce n new arguments.
2. Searches for variable x in memory and checks the recursion limit. If x is not found or the recursion limit has been reached, the inliner returns variable x applied to the n *new* arguments.
3. Rewrites x using its definition from memory, $m(x)$.
4. Freshens the resulting application of $m(x)$ to the n new arguments.
5. Attempts to convert the freshened application to a series of **let**-bindings. This is precisely the conversion shown in eq. (1) (pg. 11). Note that the conversion fails (returns None) if $m(x)$ is not fully applied, in which case the inliner bails out of inlining the definition of x.
6. Recurses into the newly produced series of **let**-bindings with a decremented recursion limit.

The conversion into **let**-bindings is critical: it allows the inliner to learn the definitions of the applied arguments $ce'_1,\ \ldots,\ ce'_n$ for future inlining within the function body of $m(x)$. Note that we only decrement the recursion limit when the size of the input expression may not have strictly decreased. This happens only when performing non-structural recursions, which only occur when we recurse into a definition rewritten from memory.

Remembering a new definition. The inliner can remember **let**- or **letrec**-bound expressions.
In the **let** case, it is defined as follows:

```
$\mathsf{inline}^k_h\ m\ (\mathbf{let}\, x = ce_1\ \mathbf{in}\ ce_2) \overset{\mathsf{def}}{=} \mathsf{do}$
  $ce_1' \leftarrow \mathsf{inline}^k_h\ m\ ce_1;$
  $\mathsf{let}\ m' = \mathsf{remember}_h\ m\ (x \leftarrow ce_1)\,;$
  $ce_2' \leftarrow \mathsf{inline}^k_h\ m'\ ce_2;$
  $\mathsf{return}\ (\mathbf{let}\, x = ce_1'\ \mathbf{in}\ ce_2')$
```

$$\mathsf{remember}_h\ m\ (x \leftarrow ce) \overset{\mathsf{def}}{=} \mathsf{if}\ \ \mathsf{cheap}\ ce\ \wedge\ h\ ce\ \ \mathsf{then}\ \ m[x \mapsto ce]\ \ \mathsf{else}\ \ m$$

That is, the inliner recurses into ce_1 (without decrementing the recursion limit), before memorising the definition $x \leftarrow ce_1$ and recursing into ce_2 with the augmented memory. The function remember records the definition only when two conditions are satisfied: the definition is cheap, and heuristic h returns true.

As the name suggests, cheap is a predicate that determines whether a definition is cheap to compute, and so will not slow the program down or cause loss of value sharing when inlined. The definition of cheap is as follows:

$$\mathsf{cheap}\ (\mathbf{var}\, x)\ =\ \mathsf{cheap}\ (\lambda x.\ e)\ =\ \mathsf{cheap}\ (op[\,])\ \overset{\mathsf{def}}{=}\ \mathsf{T} \qquad \mathsf{cheap}\ \_\ \overset{\mathsf{def}}{=}\ \mathsf{F}$$

In the **letrec** case, the inliner must also perform specialisation. Its action is defined as follows:

```
$\mathsf{inline}^k_h\ m\ (\mathbf{letrec}\, x = ce_1\ \mathbf{in}\ ce_2) \overset{\mathsf{def}}{=} \mathsf{do}$
  $ce_1' \leftarrow \mathsf{inline}^k_h\ m\ ce_1;$
  $\mathsf{let}\ m' = \mathsf{remember\_rec}_h\ m\ (x \leftarrow ce_1)\,;$
  $ce_2' \leftarrow \mathsf{inline}^k_h\ m'\ ce_2;$
  $\mathsf{return}\ (\mathbf{letrec}\, x = ce_1'\ \mathbf{in}\ ce_2')$

$\mathsf{remember\_rec}_h\ m\ (x \leftarrow ce) \overset{\mathsf{def}}{=}$
  $\mathsf{if}\ \neg\, \mathsf{can\_specialise}\ (x \leftarrow ce)\ \vee\ \neg\, \mathsf{h}\ ce\ \ \mathsf{then}\ \ m\ \ \mathsf{else}$
    $\mathsf{let}\ ([w_1^{a_1}\ \ldots\ w_n^{a_n}],\ \lambda\,\overline{y_m}\,.\ ce') = \mathsf{extract\_const\_args}\ (x \leftarrow ce)$
    $\mathsf{in}\ [x\ \mapsto\ \mathsf{specialise}\ x\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ (\lambda\,\overline{y_m}\,.\ ce')]$
```

This mirrors the **let** case almost exactly. The key difference is the use of remember_rec instead of remember: this does not check cheap, but does attempt specialisation (and bails out if it fails). We examine specialisation in the upcoming subsection.

Heuristics. So far, we have only implemented one heuristic based on expression size: the inliner only remembers definitions that are smaller than a user-configurable bound. Our implementation can accept any heuristic function as an input, making it straightforward to support new kinds of heuristic.

Implementing specialisation. Above, $\mathsf{specialise}$ transforms a **letrec**-binding into a **let**-binding before adding it to memory. We rely on two helper functions: $\mathsf{can\_specialise}$ and $\mathsf{extract\_const\_args}$.

The test $\mathsf{can\_specialise}$ simply checks if we are able to specialise a recursive body. The body must be a λ-abstraction with some constant arguments. Then, $\mathsf{extract\_const\_args}$ will extract these constant arguments. It accepts a definition $x \leftarrow ce$, where we know ce is a λ-abstraction of the form $\lambda\, \overline{x_n}\, .\; ce$. It splits the formal parameters $\overline{x_n}$ into $x_1 \ldots x_m$ and $x_{m+1} \ldots x_n$, where m is the minimum number of arguments that x is invoked with recursively in body ce. It further annotates the $x_1 \ldots x_m$ with annotations $a_1 \ldots a_m$, which describe whether the arguments remain constant for each recursive call. In the implementation of inline above, this has produced the annotated variables $w_i^{a_i}$ and left the remainder of the λ-abstraction untouched $(\lambda\, \overline{y_m}\, .\; ce')$.

Then, $\mathsf{specialise}$ is defined as follows.

$$\begin{array}{l}
\mathsf{specialise}\ f\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ \ ce \overset{\mathsf{def}}{=} \\
\quad \mathsf{let}\ (\overline{x_n}, ce') = \mathsf{specialise\_each}\ x\ \ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ \ ce\ \mathsf{in} \\
\quad \mathsf{let}\ (\overline{y_i}, \overline{z_j}) = \mathsf{drop\_common\_suffix}\ \ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ \ \overline{x_n}\ \mathsf{in} \\
\quad \lambda\, \overline{y_i}\, .\; \mathbf{letrec}\, f = (\lambda\, \overline{x_n}\, .\; ce')\ \ \mathbf{in}\ \ (\mathbf{var}\, f) \cdot (\mathbf{var}\, z_1) \cdot \ldots \cdot\ (\mathbf{var}\, z_j)
\end{array}$$

That is, it processes each annotated variable in turn, updating their call sites in body ce (*i.e.*, performing *transform 1* and *transform 2* from § 5 simultaneously using $\mathsf{specialise\_each}$), producing a new set of formal parameters $\overline{x_n}$. It determines which of these can be η-contracted (the final step in § 5) with a call to $\mathsf{drop\_common\_suffix}$, and then returns the new **letrec** which accepts constant arguments $\overline{y_i}$ at the top-level, and has η-contracted constant arguments $\overline{z_j}$ applied directly already.

Freshening and Dead-Let Elimination. Our inliner assumes that its input expression has a variable naming convention which is sufficient to prevent it from accidentally capturing variables during operation. Therefore, we only give the inliner expressions which obey the Barendregt variable convention, which asserts unique bound variable names and disjoint bound/free names [3]. This is achieved by freshening (α-renaming) bound variables directly before inlining, and further freshening before recursing into subexpressions taken from the inliner's memory. For example, the inliner invokes $\mathsf{freshen}$ in eq. (2) (pg. 16) above. This is precisely why the inliner carries around a $\mathsf{name\ set}$ in its state monad: this set contains all variable names (whether bound or free) of the input expression. Freshening avoids names in this set when inventing fresh names, and returns an updated set each time it runs.

The output of the inliner also contains various unused **let**-bindings. We showed such bindings in the example of § 1 (namely, `f` and `i`). To remove such bindings, we run a dead-**let** elimination pass directly after the inliner.

Including these two auxiliary passes, the top-level definition of the inliner is as follows:

$$\begin{aligned}&\mathsf{inliner}_h^k\ ce \overset{\mathsf{def}}{=}\\ &\quad \mathsf{let}\ (ce',\ names) = \mathsf{freshen}\ ce\ (\mathsf{boundvars}\ ce)\ \mathsf{in}\\ &\quad \mathsf{let}\ (ce_i,\ \_) = \mathsf{inline}_h^k\ \varnothing\ ce'\ names\ \mathsf{in}\\ &\quad\ \ \mathsf{dead\_let}\ ce_i\end{aligned} \tag{3}$$

That is, the inliner freshens names, inlines definitions top-down starting with an empty (∅) memory, then removes dead **let**s. Note that the top-level definition expects to receive only closed expressions, which is why it only passes bound variables (boundvars) to freshen. This respects our invariant that the name set contains all bound and free variable names, as there are no free variables.

6.3 Inliner correctness

In this section, we prove that the inliner implementation is correct. In the context of PureCake's proof strategy as described in § 3:

– *(stage 1)* Theorem 2 above (pg. 9) proved that $\rightsquigarrow$ preserves semantics.
– *(stage 2)* Theorem 3 below will prove that any transformation performed by the inliner lies within the $\rightsquigarrow$ relation of § 4.

We then compose these results to produce our final soundness theorem: the output expression of the inliner is equivalent to its corresponding input.

Theorem 3. inline satisfies $\rightsquigarrow$.

$$\begin{aligned}&\vdash \mathsf{inline}_h^k\ m\ ce\ ns = (ce',\ ns')\ \wedge\ \mathsf{memory\_rel}_{ns}\ l\ m\ \wedge\\ &\quad \mathsf{barendregt}\ \ (\mathsf{desugar}\ ce)\ \wedge\ \mathsf{boundvars}\ ce \mathrel{\#} \mathsf{domain}\ m\ \wedge\\ &\quad \mathsf{freevars}\ ce\ \cup\ \mathsf{boundvars}\ ce \subseteq ns\ \wedge\ \mathsf{wf}\ ce\\ &\qquad \Rightarrow\ l \Vdash (\mathsf{desugar}\ ce) \rightsquigarrow (\mathsf{desugar}\ ce')\end{aligned}$$

That is, after desugaring compiler expressions into semantic expressions (desugar, see § 3), the action of the inliner for input ce, memory m, and name set ns lies within $\rightsquigarrow$ for some stacked **let**s l when the following hold:

– (memory_rel) m and l contain the same definitions, and each such definition *both* satisfies wf below *and* has bound/free variables within ns;
– (barendregt) bound names in ce are unique, and disjoint from free names;
– the bound variables of ce do not shadow (are disjoint from, #) any variables with known definitions, *i.e.*, those in the domain of m;
– all bound/free variables of ce are within ns; and
– (wf) ce is well-formed.

Proof outline. Induction over the implementation function inline. For each case of the proof, we apply rules of $\rightsquigarrow$ to justify each atomic inlining operation. □

Theorem 4. Top-level correctness of $\mathsf{inliner}$.

$$\vdash \mathsf{wf}\ ce\ \wedge\ \mathsf{closed}\ ce\ \Rightarrow\ (\mathsf{desugar}\ ce)\ \cong\ \big(\mathsf{desugar}\ (\mathsf{inliner}_h^k\ ce)\big)$$

Proof outline. Composition of Theorem 3 above with Theorem 2 (pg. 9), the soundness theorem for $\rightsquigarrow$. Unfolding the definition of $\mathsf{inliner}$, we use the soundness theorem of $\mathsf{freshen}$, the closed assumption, and the application of inline to empty memory $\varnothing$ to discharge the preconditions on Theorem 3. □

7 Integration into the PureCake Compiler

We insert the inliner and its associated cleanup of dead **let**-bindings as PureLang-to-PureLang transformations early in the PureCake compiler. In particular, directly after parsing and binding group analysis, as shown in Figure 2. Elimination of dead **let**s happens directly afterwards.

Unusually, the inliner runs before type inference. Ideally, it would take place afterwards: it changes program structure significantly, and type inference should execute on code resembling user input to allow direct error-reporting. The reasoning behind this design choice is PureCake's demand analysis, which facilitates strictness optimisations by annotating variables that can be evaluated eagerly. We found that running the inliner before demand analysis produces significantly better performance (§ 8, Figure 4). However, the soundness proof for demand analysis requires it to receive only well-typed input code. To run the inliner after type inference and before demand analysis, we would have to prove that it preserves well-typing, which is a significant undertaking due to PureLang's untyped AST. Future iterations of PureLang's AST are intended to be typed; therefore, we could consider proving type preservation in future work.

To update PureCake's compiler correctness theorem after integrating our inliner, we must establish that the inliner preserves both semantics and various syntactic invariants. We have already presented our proof of semantics preservation in § 6. The latter syntactic invariants guarantee that compiler expressions are closed and satisfy well-formedness properties which are checked as part of parsing. For example, PureLang forbids degenerate function applications to zero arguments: this can be expressed in the AST for PureLang compiler expressions but is ill-formed. Establishing preservation of the invariants is mostly mechanical, but quite tedious and long-winded.

8 Benchmarks

In this section we measure the efficacy of our inliner. In particular, we benchmark code generated by PureCake to determine how much the addition of the inliner improves runtime and memory overhead.

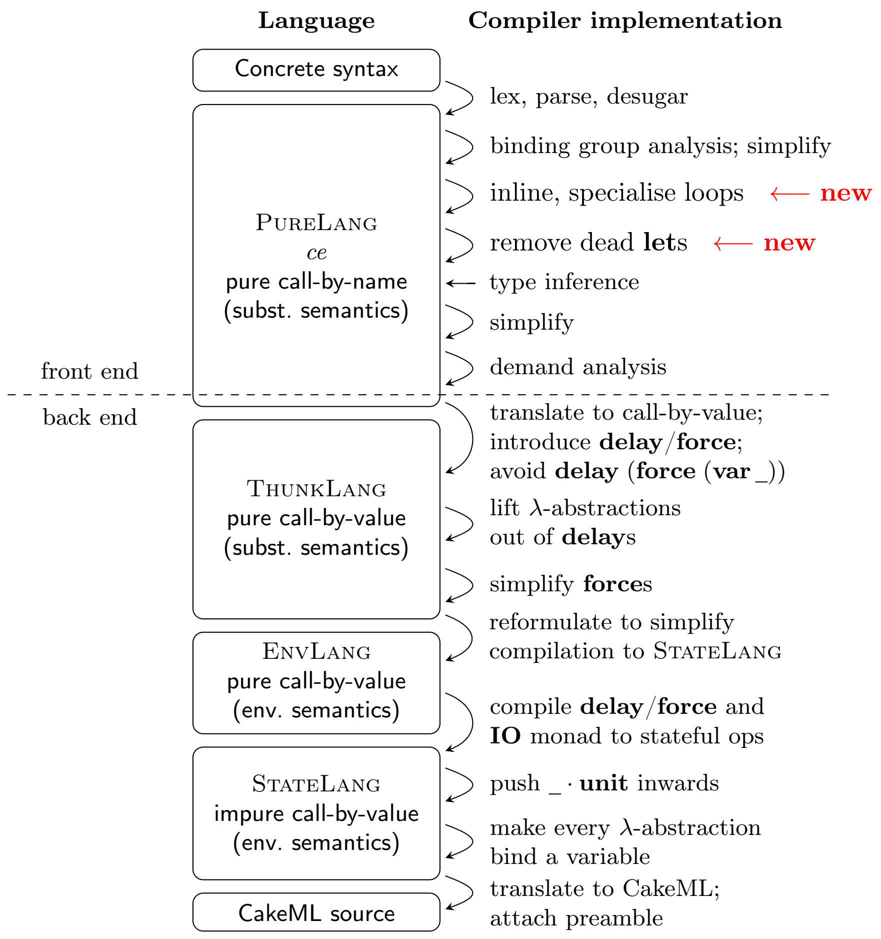

Fig. 2. High-level structure of the PureCake compiler. The inliner and its associated clean up are PureLang-to-PureLang passes which take place immediately after binding group analysis and before type inference.

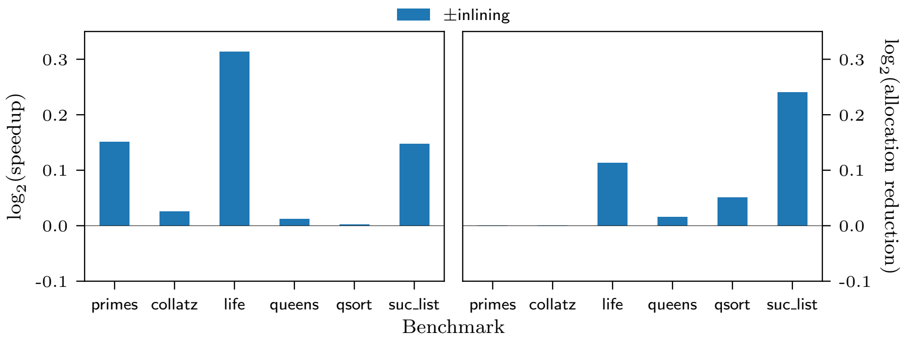

Fig. 3. Graphs showing the performance impact of our inliner: the base-2 logarithm of a ratio of measurements (execution time or heap allocations) with/without the inliner enabled: $\log_2\left({}^{m_{\text{disabled}}}/_{m_{\text{enabled}}}\right)$. Error bars are too small to be visible.

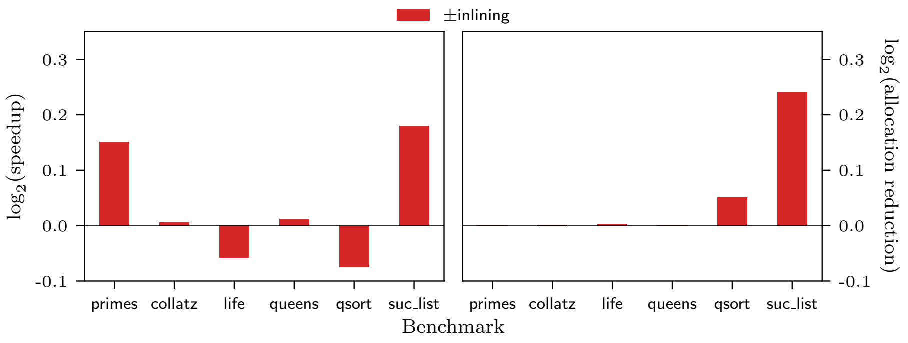

Fig. 4. Graphs showing the performance impact of our inliner when executed *after* PureCake's demand analysis. Performance is clearly worse compared to Figure 3; *therefore we do not pursue this approach.*

Methodology. We evaluate the performance of several benchmark programs with and without the inliner enabled, using an Intel® Xeon® E-2186G and 64 GB RAM. We consider the same programs as presented by the PureCake developers in prior work [10, §7.1]. We also add a new `suc_list` program, which repeatedly applies the `suc_list` function shown in § 1 to a list of natural numbers. Like the PureCake developers, we measure wall-clock runtime and total heap allocations as reported by the CakeML runtime. Our measurements are facilitated by existing benchmarking scripts found in the PureCake development.

Results. Figure 3 shows our results, plotted as two bar graphs: the left shows runtime speedup, the right shows allocation reduction. In many cases, our inliner significantly improves performance; in all cases it does not worsen performance. The value for each plot is obtained by taking the base-2 logarithm of a ratio: the measurement without the inliner enabled (*i.e.*, the longer duration or greater allocation) divided by the measurement with the inliner enabled. Expressed as

Table 1. Line counts for each part of our development.

Part of development	kLoC
Syntactic relation ($\rightsquigarrow$) and its soundness (§ 4)	2.6
Equational theory behind specialisation (§ 5)	4.0
Implementation of inliner (incl. specialisation) (§ 6.2)	0.6
Correctness of the implementation (§ 6.3)	3.7
Freshening and its correctness proof	3.1
Elimination of dead lets and its correctness proof	0.5
Total	**∼15**

a percentage, the most significant improvements are: a ∼20% reduction in the runtime of life, a ∼15% reduction in the allocations of suc_list.

Inliner placement. We noted in § 7 that our inliner should run before PureCake's demand analysis. Here, we justify that design choice. In particular, we benchmark a version of the PureCake compiler which runs our inliner directly *after* demand analysis. The results are shown in Figure 4. The improvements in runtime and memory overhead are reduced for several benchmarks, and in some cases runtime even worsens overall. Therefore, our inliner should run before demand analysis for maximum benefit.

Code size and compile times. Simple measurements of code size show that our inliner can produce significantly larger CakeML programs (∼50% increase); however CakeML's efficient handling of inserted **let**s reduces the effect for binaries ($<$ 15% overall increase). Compile times are unaffected: these remain dominated by PureCake's type-checking and CakeML's register allocation.

Line counts. Our work adds to PureCake significantly. Table 1 shows line counts for each part of our development, measured using `wc -l`.

9 Related Work

Verified inlining in functional languages. CakeML [12] compiles a subset of Standard ML (strict, impure) to several mainstream architectures with end-to-end guarantees. It performs function inlining in its second intermediate language, ClosLang, which has first-class closures. A flow analysis discovers invocations of known functions, and simultaneously inlines closed functions which themselves do not contain closures. Use of de Bruijn indices sidesteps reasoning about shadowing and freshening. As in our work, recursive applications of inlining improve the performance of higher-order functions; we go one step further with specialisation and the inlining of open terms which can contain λ-abstractions.

CertiCoq [2] verifiably compiles Gallina (the metalanguage of Coq) to C light, an intermediate language early in CompCert's pipeline. One of its passes [4] performs several *shrink reductions* simultaneously: transformations that only reduce code size. One such reduction is the inlining of functions which are applied exactly once; in this case, inlining *is* β-reduction, contrary to our discussion in § 4.1. Restriction to shrink reductions further removes the need for a recursion limit as code size strictly decreases on each recursive call. Their verification relies on a more general rewrite system which permits inlining of functions which are used multiple times. A separate pass [16] further inlines small non-recursive functions which can be applied multiple times; here a key concern is maintenance of A-normal form expressions. In all proofs, the Barendregt variable convention (*i.e.*, barendregt) is used to avoid name clashes.

Pilsner [15] compiles a strict impure language to an idealised assembly, inlining select top-level functions in its intermediate representation. Recursive functions can be unrolled in this way, but not specialised. Again, the Barendregt variable convention is enforced. The focus here is on the novel proof technique of parametric inter-language simulations (PILS) to enable compositional compiler correctness, where PureCake focuses on mechanised whole-program compiler correctness for a realistic language.

Other verified inlining passes. CompCert [13] compiles a subset of C99, performing function inlining in its register transfer language (RTL). This control flow graph (CFG) representation differs considerably from the functional PureLang; inlining considers only top-level function declarations in the RTL setting. Rather than using a recursion limit, CompCert guarantees termination by forbidding inlining of functions within their own bodies.

CompCert also performs *lazy code motion* [19] within RTL. A special case of this transformation is loop-invariant code motion, which loosely resembles our specialisation: both are concerned with moving constant expressions out of loops, but in our functional setting loops are expressed as recursive functions. Their verification uses translation validation [18]: an unverified tool transforms code, and then per-run automation proves that semantics has been preserved.

The Plutus Tx language from the Cardano blockchain platform resembles a subset of Haskell, and is compiled to a custom language known as Plutus Core. The compiler is implemented as a GHC plugin: GHC machinery first lowers Plutus Tx to a System F-like language, which is then optimised and compiled further. The compiler is verified using *translation certification* [11], which aims to make translation validation approaches less brittle by combining automated and manual proof. As in PureCake, syntactic relations are used to encapsulate semantics-preserving transformations: automated proof shows that unverified code transformations inhabit the relations, and manual proof shows that the relations preserve semantics. Translation certification is robust to evolving compiler implementations because the syntactic proofs are more amenable to automated verification than the semantic ones. A syntactic relation akin to § 4 justifies inlining; however, semantic verification is ongoing work at the time of writing. The Barendregt variable convention is enforced in this work too.

Verified optimisation of realistic Haskell-like languages. The CoreSpec project[4] tackles verified variants of Haskell as implemented by GHC. For example, GHC's dependent types extensions were proposed using formal specifications of the syntax, semantics, and typing rules of GHC's Core language [20]. The unverified tool `hs-to-coq` [6] translates Haskell code to Gallina (Coq's metalanguage), leveraging Coq's logic to enable equational reasoning about real-world programs. A future aim of the project is to derive Coq models of Core automatically from GHC's implementation, prove correctness of optimisations within Coq, and integrate the resulting verified code back into GHC as a plugin. Where CoreSpec focuses on accurate modelling of GHC with the loss of some trust, PureCake instead sacrifices faithfulness for end-to-end guarantees.

GHC's arity analysis pass [5] η-expands functions to avoid excessive thunk allocations. Its mechanised proof of correctness for a simplified Core language relies on an explicitly call-by-need semantics to show performance preservation, *i.e.*, that η-expansion does not reduce value-sharing.

10 Summary and Future Work

This paper has described our work on a verified inlining and loop specialisation pass for PureLang, a lazy functional programming language. First, we verified a syntactic relation which defines an envelope of permitted inlining transformations, independent of heuristic choices. We used a novel phrasing of inlining as the pushing in and pulling out of `let`-bindings to prove the relation sound using PureLang's equational theory. Our inliner implementation is then proven to remain within this envelope. We have integrated our work into the PureCake compiler, an end-to-end verified compiler, and demonstrated significant performance improvements. To the best of our knowledge, ours is the first verified function inliner for a lazy functional programming language, and the first verified loop specialiser for any functional language.

In future work, we intend to support loop unrolling and develop better heuristics that decide when to do inlining. Loop unrolling will probably involve augmenting the definition of **lets** so that it can hold both **let** expressions and **letrec**s. Developing good heuristics will require some careful experimentation with the compiler implementation. We do not expect adjustment to the inliner's heuristics to impact our correctness proofs in any significant way, since the proofs are designed to be independent of heuristic choices.

Acknowledgements. Hrutvik Kanabar was supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS). Magnus Myreen was supported by Swedish Research Council grant 2021-05165.

Data availability statement. An artifact supporting the results presented in this paper is openly available on Zenodo [9]. The latest development version of PureCake is available on GitHub (https://github.com/cakeml/pure).

[4] https://deepspec.org/entry/Project/Haskell+CoreSpec

References

1. Abramsky, S.: The lazy λ-calculus. In: Research Topics in Functional Programming. Addison Wesley (1990)
2. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.Z.: CertiCoq: A verified compiler for Coq. In: Workshop on Coq for Programming Languages (CoqPL) (2017), https://popl17.sigplan.org/details/main/9/CertiCoq-A-verified-compiler-for-Coq
3. Barendregt, H.P.: The lambda calculus - its syntax and semantics, Studies in logic and the foundations of mathematics, vol. 103. North-Holland (1985)
4. Bélanger, O.S., Appel, A.W.: Shrink fast correctly! In: Principles and Practice of Declarative Programming (PPDP). ACM (2017). https://doi.org/10.1145/3131851.3131859
5. Breitner, J.: Formally proving a compiler transformation safe. In: Symposium on Haskell. ACM (2015). https://doi.org/10.1145/2804302.2804312
6. Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). Proc. ACM Program. Lang. **2**(ICFP) (2018). https://doi.org/10.1145/3236784
7. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. **124**(2) (1996). https://doi.org/10.1006/inco.1996.0008
8. Hughes, J.: Why functional programming matters. Comput. J. **32**(2) (1989). https://doi.org/10.1093/comjnl/32.2.98
9. Kanabar, H., Korban, K., Myreen, M.O.: Artifact for "Verified Inlining and Specialisation for PureCake" (2024). https://doi.org/10.5281/zenodo.10456887
10. Kanabar, H., Vivien, S., Abrahamsson, O., Myreen, M.O., Norrish, M., Åman Pohjola, J., Zanetti, R.: PureCake: A verified compiler for a lazy functional language. In: Programming Language Design and Implementation (PLDI). ACM (2023). https://doi.org/10.1145/3591259
11. Krijnen, J.O.G., Chakravarty, M.M.T., Keller, G., Swierstra, W.: Translation certification for smart contracts. In: Functional and Logic Programming (FLOPS). Lecture Notes in Computer Science, vol. 13215. Springer (2022). https://doi.org/10.1007/978-3-030-99461-7_6
12. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL). ACM (2014). https://doi.org/10.1145/2535838.2535841
13. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009). https://doi.org/10.1145/1538788.1538814
14. Myreen, M.O.: The CakeML project's quest for ever stronger correctness theorems (invited paper). In: Interactive Theorem Proving (ITP). LIPIcs, vol. 193. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.1
15. Neis, G., Hur, C., Kaiser, J., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: a compositionally verified compiler for a higher-order imperative language. In: International Conference on Functional Programming (ICFP). ACM (2015). https://doi.org/10.1145/2784731.2784764
16. Paraskevopoulou, Z., Li, J.M., Appel, A.W.: Compositional optimizations for CertiCoq. Proc. ACM Program. Lang. **5**(ICFP) (2021). https://doi.org/10.1145/3473591

17. Peyton Jones, S.L., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming **12** (2002)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1384. Springer (1998). https://doi.org/10.1007/BFb0054170
19. Tristan, J., Leroy, X.: Verified validation of lazy code motion. In: Programming Language Design and Implementation (PLDI). ACM (2009). https://doi.org/10.1145/1542476.1542512
20. Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. Proc. ACM Program. Lang. **1**(ICFP) (2017). https://doi.org/10.1145/3110275