# Simple Lambda Lifting: Formalisation in Lean and a new efficient algorithm

Tom Levy
University of Waikato
Hamilton, New Zealand
tomlevy93@gmail.com

Steve Reeves
University of Waikato
Hamilton, New Zealand
steve.reeves@waikato.ac.nz

## Abstract

Lambda lifting is a technique used in compilers to convert nested function definitions to top-level function definitions. A series of papers has led to an $O(n^2)$ algorithm, however it is complex. We present a simple $O(n^2)$ algorithm for lambda lifting and prove its correctness. We also formalise a lambda lifting specification from the literature in Lean 4, and use that to prove some of the properties and test our algorithm on generated test cases. One of our contributions is to formalise the notion of a "complete" and "minimal" lifting, addressing a small issue with the handling of unused functions that to our knowledge affects all previous algorithms.

## CCS Concepts

• **Software and its engineering** → **Compilers**; **Procedures, functions and subroutines**; **Software verification**; • **Theory of computation** → *Graph algorithms analysis.*

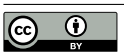
## Keywords

lambda lifting

**ACM Reference Format:**
Tom Levy and Steve Reeves. 2026. Simple Lambda Lifting: Formalisation in Lean and a new efficient algorithm. In *IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3793656.3793692

## 1 Introduction

Improving long-standing algorithms, as in our example here where we show how to make a widely used algorithm faster, carries with it the burden of showing that the improvement still matches the specification. In this paper we show by example how existing, widely used and very well established algorithms and implementations might be re-verified by taking a formal specification regarded as the canonical definition of an algorithm—in this case given as a set of logical rules in natural deduction form—and formalising it in a proof assistant (Lean 4 in our case); then a combination of machine-checked and "pen and paper" proofs can be used to demonstrate that the improved algorithm still meets the definition given by the original specification. We suggest that this might be a generally good thing to do even when making "obvious" improvements to widely used algorithms, since much can be at stake.




*FormaliSE '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2478-7/26/04
https://doi.org/10.1145/3793656.3793692

Many programming languages allow functions to be declared inside other functions, a feature known as "nested functions" or "local functions". It is useful because it allows programmers to avoid namespace pollution, group code logically, and most importantly because it allows the local functions to refer to identifiers from the enclosing scope. Access to the enclosing scope can make the code more concise and improve readability, since values from the enclosing scope do not need to be passed explicitly to the local functions. Further, in languages that allow functions to be used as values, nested functions offer a convenient way to create functions that have associated data, such as a callback with associated state.

However, it can be challenging to handle nested functions in compilers due to the non-local references to variables from enclosing scopes. In particular, the use of nested functions as values is incompatible with stack-based function calling; this is known as the "funarg" problem [25, 17].

Lambda lifting [9] is a technique that converts a program containing nested functions into an equivalent program that does not make use of nested functions. This is useful because it means later compilation passes do not need to handle nested functions. The idea is to "lift" each local function to the top-level scope, and introduce additional parameters as necessary to satisfy references to free variables. We refer to these additional parameters as "extraneous parameters" (following the terminology of [3, 15]).

For example, consider the following code to calculate the fifth triangular number ("@" denotes function application):

```
letrec triangular_number = λ n .
  letrec sum_from = λ i .
    if i > n then 0 else i + sum_from @ (i + 1)
  in sum_from @ 1
in triangular_number @ 5
```

Note that the body of `sum_from` refers to `n`, which is a parameter of the enclosing function. In order to convert the function `sum_from` to a global function, we need to introduce a parameter to `sum_from` allowing the value of `n` to be passed in. The result of lifting is:

```
letrec
  sum_from = λ n . λ i .
    if i > n then 0 else i + sum_from @ n @ (i + 1)
  triangular_number = λ n . sum_from @ n @ 1
in triangular_number @ 5
```

Note how the argument `n` must be added to all the call sites of `sum_from`. If the function is used as a value (in a language that supports higher-order functions / first-class functions), then in general the added arguments would need to be passed using partial application, producing a function value with the same signature

as the original function. In this paper we focus on approaches that avoid partial application (except when functions are used as values), because it requires special support from the language, such as closures.

Also note the subtle change to the variable bindings: even though the expression `i > n` in `sum_from` did not change textually, the binding of `n` in that expression has changed—in the original program it was bound to the parameter `n` of `triangular_number`, while in the lifted program it is bound to the parameter of `sum_from`.

The extraneous parameters of `sum_from` in this example are simply the free variables of the original definition. But determining the extraneous parameters in general is non-trivial for two reasons:

- A local function may call other local functions. If those functions require extraneous parameters, then corresponding arguments will need to be introduced in the calls to those functions, which will in turn produce new free variables (unless those variables are declared by the calling function). This applies transitively.
- Local functions may be mutually recursive. So an algorithm for determining the extraneous parameters needs to be able to handle programs whose call graph contains cycles.

Further, we wish to find the smallest set of extraneous parameters, to produce more efficient code.

Determining the extraneous parameters is the core of lambda lifting. Once that is done, there are a few of remaining tasks, but they are straightforward manipulations of the abstract syntax tree:

(1) insert the extraneous parameters in the parameter lists and argument lists;
(2) rename identifiers as necessary to avoid naming conflicts;
(3) move all local function definitions to the top-level scope ("block floating").

There are algorithms for determining the minimal extraneous parameters in the literature, but they have a suboptimal time complexity or are complex. The algorithm we present in this paper is simple, and its *worst-case* time complexity is $O(n^2)$ like the fastest existing algorithm [16].

We go further and give a tight time complexity analysis for *all* cases by considering the size of the output program. The worst-case size of the output program is $\Theta(n^2)$ where $n$ is the size of the input program;[1] but if each top-level function only contains a small[2] number of nested function or a small number of parameters, then the size of the output program is $\Theta(n)$. We show that the time complexity of our algorithm is linear in the size of the output program (Theorem 4.8); hence the time complexity of our algorithm is optimal on all cases if the lifted program is to be output. (In contrast, the fastest existing algorithm has an optimal *worst-case* time complexity, but there exist non-worst-case inputs on which its time complexity is suboptimal.)

We prove that our algorithm is correct with respect to an existing formal specification from the literature [6] and with respect to our notion of a "complete" and "minimal" lifting, making it the first algorithm proven to compute the *minimal* extraneous parameters.

[1] This is true both when the program size is measured in tokens and when it is measured in bytes.
[2] By "small" we mean less than an arbitrary constant.

## 2 Literature

Church's lambda calculus [2] is one of the earliest examples of the notion of nested functions; the reduction rules handle nested functions using capture-avoiding substitution.

LISP had nested functions but used what later became known as "dynamic scoping", which is generally considered undesirable [22] and was regarded as a bug [14].

Algol 60 had nested functions that could be passed as parameters but could not be returned from a function. This was implemented using special stack frame links [24][21, §2.2].

Closures were first described by Landin in [10]. Closures are used by PAL [5] and Scheme [24, 23].

Hughes [8] describes a method for converting $\lambda$-expressions containing free variables to supercombinators (functions that contain no free variables). The method relies heavily on partial application. These ideas are also described in [19] in more detail.

Johnsson [9] describes a different method that does not make use of partial application (except when functions are used as values). As described in Section 1, this "lambda lifting" method eliminates free variables by assigning each function a set of extraneous parameters and inserting corresponding extraneous arguments in each call site (and at each reference if the function is used as a value). The implementation is somewhat complicated because inserting the extraneous arguments may produce new free variables, so computing the final sets of extraneous parameters requires computing a variation of the transitive closure. Johnsson's approach is to set up a system of set equations and solve it using substitution for each `letrec` expression, with time complexity $O(n^3)$.

The code generated by Johnsson-style lifting has different performance characteristics compared to supercombinator/closure conversion [19, 7], and both are still widely used [3]. Johnsson-style lifting also been used to optimise partial evaluators [15].

Fischbach and Hannan give a formal specification of lambda lifting [6] and prove it preserves operational semantics. Their specification is quite general: it permits both Hughes-style and Johnsson-style liftings, as well as other forms of lifting. They also describe an algorithm similar to Johnsson's and prove its correctness with respect to the specification. However, they do not specify any notion of a "complete" lifting that eliminates all free variables, and they do not prove their algorithm computes the minimal extraneous parameters.

CertiCoq [18] is a compiler for Gallina (the specification language of Coq) that includes a verified lambda lifting transformation; the time complexity is not specified but appears to be at least $O(n^3)$.

There has been a series of papers on improving the time complexity of Johnsson-style lifting using graph-based approaches. The most obvious approach for computing the extraneous parameters of a function is to take the union of its free variables and the extraneous parameters of the functions it calls. However, a simple top-down recursive traversal of the call graph would fail due to infinite recursion if there are any cycles (mutually recursive functions). It is possible to avoid infinite recursion by stopping the recursion when it reaches functions that have already been visited, and it can be shown that this will produce the correct extraneous parameters for the function that the recursion initially started from, however the sets of extraneous parameters computed for other functions

visited during the recursion may be incomplete. An algorithm that performs independent traversals starting from each function would produce the correct extraneous parameters, but its overall time complexity would be $O(n^3)$.

Danvy and Schultz [3] describe an $O(n^2)$ algorithm which eliminates cycles in the call graph by coalescing strongly-connected components (SCCs) and assigning the same extraneous parameters to all functions within each SCC. However, this approach is flawed; one analysis is given in [15], and we give our own brief analysis.[3]

Morazán and Schultz [16] describe an $O(n^2)$ algorithm that solves the issue. The algorithm is complex—it uses dominator trees in addition to SCCs.

Newer lambda lifting implementations such as in GHC [7] continue to use $O(n^3)$ algorithms, suggesting a need for a simple, fast, and correct algorithm. Such an algorithm will be useful to future compiler implementors, and has applications beyond nested functions—for example, it can be used to lift local types inside generic functions in the Go programming language (which has fast compilation as a core goal [20]).

## 3 Syntax

We use the syntax for expressions and types from [6]:

$$e ::= c \mid x \mid \lambda x.e \mid e_1 \mathbin{@} e_2 \mid \texttt{letrec } \overline{f_i = e_i} \texttt{ in } e$$
$$\tau ::= \iota \mid \tau \to \tau$$

These symbols denote (in order) expressions, pre-defined constants, identifiers, abstractions, function application, `letrec` expressions with mutually-recursive bindings, types, base types, and function types. We refer the reader to [6] for a more detailed explanation and for the specification of the type system.

For the lambda lifting specification, $\tau$ is extended using singleton types:

$$\tau ::= \iota \mid \tau \to \tau \mid \{e\}_\tau \to \tau$$

$\{e\}_\tau$ is a singleton type; the only term of that type is the expression $e$ of type $\tau$. We restrict the expression $e$ in singleton types to be an identifier, as done in the version of the lambda lifting specification in [6, §3.2]. The more general specification in [6, §4] does not have this restriction and allows lifting arbitrary expressions (enabling full laziness), but in this paper our focus is solely on parameter lifting. The type $\{x\}_{\tau_1} \to \tau_2$ denotes the type of the function that results when $x$ is inserted as an extraneous parameter of a function of type $\tau_2$.

Note that the grammar does not allow "bare" singleton types—singleton types may only appear as part of a lifted function type. This restriction is subtle but critical; as stated in [6], it implies that in the rules (abs) and (app) of the lifting specification, $\tau_1$ cannot be a singleton type. This is necessary for the proofs of type correctness and operational correctness in [6].

We treat $\lambda x_1.\cdots.\lambda x_n.e$ as a single multi-parameter function (for the purpose of the call graph and analysis of free variables).

There are three types of bindings:

- A `letrec` binding of the form $f = \lambda x.e$ is a *function binding*.
- A `letrec` binding of the form $f = e$ where $e$ is not an abstraction is a *variable binding*.
- A binding created by a binder $\lambda x$ is a *parameter binding*.

A *variable* is an identifier bound by a variable or parameter binding, and a *function name* is an identifier bound by a function binding.

A function $\lambda\ldots$ that occurs as the right-hand side of a function binding is a *named function*, while a function that occurs elsewhere is an *anonymous function*.

*Shadowing* is when an identifier in a binding is already bound in the enclosing scope. The Fischbach–Hannan specification forbids shadowing to avoid some issues [6, p. 513].[4] For the purpose of our algorithm, we assume all identifiers are distinct; that can be ensured using a preprocessing step that renames identifiers ($\alpha$-conversion),[5] or by assigning the identifiers unique IDs as in Section 4.5.

## 4 The algorithm

### 4.1 Definitions

We use the standard definitions for **scope**, **free variables**, and **closed expressions**. We use the term **extraneous parameters** to refer to the parameters added by the lambda lifting process (following the terminology of [3, 15]).

*Definition 4.1 (Declaring function).* The **declaring function** of an identifier binding is the innermost function that encloses the binding, or null if the binding is not enclosed by any function.

*Definition 4.2 (Referencing).* An occurrence of an identifier is **referenced directly** by a function if the function is the innermost function that encloses the occurrence. (We will sometimes abbreviate this to "referenced" for readability.)

*Definition 4.3 (Function reference graph).* The **function reference graph** is analogous to the call graph. The nodes are the functions (including anonymous functions), and there is an edge from $f$ to $g$ if $g$ is referenced directly by $f$ (either because $f$ calls $g$ or because $f$ uses $g$ as a value). When $g$ is an anonymous function, we say that it is referenced directly by $f$ if $f$ is the innermost enclosing function of $g$.

*Definition 4.4 (Non-local variables).* An occurrence of a variable in a function is **non-local** if the function is not the declaring function of the variable and the occurrence is not a reference to a global variable.

[3] The Danvy–Schultz algorithm [3, Figure 9] uses a top-down recursive traversal of the input program. For each `letrec` expression, it builds a call graph (restricted to the functions bound in the `letrec`), coalesces SCCs, then uses the resulting condensation graph to propagate free variables; all functions in a component are assigned the same extraneous parameters.

The issue is that it does not consider calls to functions in parent scopes (they are explicitly excluded in **foreach** $g \in \mathrm{FF}(f_i) \cap \{f_1, \ldots, f_k\}$ **do** and are not considered elsewhere); this causes it to miss required extraneous parameters in some cases.

The issue can be fixed by identifying calls to functions in parent scopes and adding their extraneous parameters, specifically by changing **let** $V = \ldots$ to **let** $V = \ldots \cup (\bigcup\{V_g \mid g \in C, (g, V_g) \in S\})$ where $C = \bigcup_{f \in p} \mathrm{FF}(f) \setminus \{f_1, \ldots, f_k\}$ are the calls to functions in parent scopes. (As written, this change increases the time complexity $O(n^3)$. However, it is possible to improve this to $O(n^2)$ by propagating the variables eagerly.)

[4] An alternative is to generate unique names for the extraneous parameters and update references accordingly.

[5] Renaming identifiers also avoids name clashes when functions are block-floated, and ensures a `letrec` expression does not contain multiple bindings with the same identifier (the Fischbach–Hannan specification allows that, and it does not cause issues there, but it does cause issues when block-floating is performed).

*Definition 4.5 (Referenced non-local variables).* A variable $x$ is a **referenced non-local variable** of a function if there exists an occurrence of $x$ which is referenced directly by the function and is non-local in the function.

The referenced non-local variables of a function are a subset of the function's free variables, restricted to ensure that the extraneous parameters derived from them are minimal:

- We exclude global variables because they remain accessible after lifting, so they never need to be added as extraneous parameters.
- We exclude function references because we are interested in block-floating all functions to the top-level scope, so after block-floating all functions will be in scope.
- We only include variables referenced directly by the function to avoid unnecessary extraneous parameters in the presence of unused functions.

To illustrate the last point, consider the following example:

```
letrec main = λ x .
  letrec f = λ y .
    letrec g = λ z . x
    in 1
  in f @ 2
in main @ 3
```

The variable x is a non-local variable in f but is not referenced directly by f. It becomes clear that f does not require x as an extraneous parameter (unlike g) after we perform block-floating:

```
letrec g = λ x . λ z . x
       f = λ y . 1
       main = λ x . f @ 2
in main @ 3
```

Once g is block-floated out of f, f does not reference x; and f does not reference any function that has x as an extraneous parameter, so x does not need to be added as an extraneous parameter of f.

Alternatively, the issue can be avoided by eliminating dead code in a preprocessing step.[6]

*Definition 4.6 (Solution).* The **solution** for a program is a mapping $EP$ from each function to a set of extraneous parameters satisfying the following conditions, for all functions $f$ and $g$ and for all identifiers $v$:

(1) $v$ is a referenced non-local variable of $f \implies v \in EP[f]$
(2) $v \in EP[f] \wedge g$ references $f \wedge g$ is not the declaring function of $v \implies v \in EP[g]$
(3) Nothing else is in $EP$.

In other words, the referenced non-local variables in the original program must be included as extraneous parameters, and extraneous parameters need to be propagated to callers and other referencing functions (except the declaring function) because they will be inserted at call sites and references; this is somewhat similar to [16]. A variable is never inserted as an extraneous parameter of its declaring function, ensuring that variables never escape their original scope when they are inserted as extraneous parameters and arguments (see Theorem 6.4).

This definition is novel, and is one of the key insights behind our algorithm.

We prove in Section 6 that the solution gives the minimal extraneous parameters in the class of liftings we are interested in.

(Note that anonymous functions are included in this definition. We assume each function is assigned a distinct map key.)

## 4.2 Example

The conditions from the definition of the solution are straightforward to apply. We demonstrate the process using the following example; our analysis mirrors the steps performed by our algorithm. (An underscore indicates an unused parameter.)

```
letrec main = λ x . λ y .
  letrec f = λ _ .
    letrec g = λ _ . x + f @ 0
           h = λ _ . y + z + f @ 0
           z = 1
    in h @ 0
  in f @ 0
in main @ 2 @ 3
```

The variable x is a referenced non-local of g, so it is added to $EP$[g]. The function g is unused, so x is not added to other sets.

The variable y is a referenced non-local of h, so it is added to $EP$[h]. The function f calls h and y $\in EP$[h], so y is added to $EP$[f]. Likewise g calls f so y is added to $EP$[g]. The function main calls f, but main is the declaring function of y so y is not added to $EP$[main]. The function h calls f (a cycle), so y needs to be added to $EP$[h]; but y is already in $EP$[h] so there is no change.

The variable z is a referenced non-local of h, so it is added to $EP$[h]. The function f calls h, but f is the declaring function of z so z is not added to $EP$[f].

Therefore, $EP$ = {main : {}, f : {y}, g : {x, y}, h : {y, z}}.

## 4.3 Algorithm 1

We now describe an algorithm and prove it computes the solution defined above.

We present two versions of the algorithm. The first version is simple and intuitive, but proving directly that it computes the solution is challenging due to inconveniences with reasoning about recursion. So we later present an iterative version of this algorithm and prove that this second version computes the solution.

The key properties of our algorithm are that it traverses the function reference graph in a bottom-up fashion and that it processes each variable separately. This makes it easy to handle cycles—we simply skip functions that have already been visited. Processing each variable separately might seem inefficient, because it requires multiple passes over the function reference graph; however that is not the case (algorithms that perform a single pass over the call graph will in general spend more time processing each node, because they need to merge sets containing multiple variables).

[6] Algorithm-specific fixes also exist. For example, the simple parameter lifting algorithm in [6, Fig. 5] can be fixed by changing $\bigcup \theta_i \cup \theta$ to $\theta$ in the rule (letrec); the proof of Theorem 5 continues to hold trivially—specifically, Lemma 3 continues to hold because $\bigcup \theta_i \cup \theta \subseteq \mathrm{dom}(\Gamma)$ implies $\theta \subseteq \mathrm{dom}(\Gamma)$; and the algorithm continues to produce a complete lifting (excluding anonymous functions)—the premise $\Lambda^* \triangleright e : (\tau, \theta) \Rightarrow m$ guarantees that if $f_i$ is referenced in $e$ then $\theta$ will be a superset of $\theta_i$, so any parameters in $\bigcup \theta_i$ that are not already in $\theta$ are unnecessary.

```
1  define DFS(v: Identifier, f: Identifier, traversed_funcs: Set):
2      if f not in traversed_funcs:
3          add f to traversed_funcs
4          for g in functions that reference f:
5              if g is not the declaring function of v:
6                  DFS(v, g, traversed_funcs)
7
8  define multi_source_DFS(v: Identifier, starting_funcs: Set) → Set:
9      traversed_funcs := new set
10     for f in starting_funcs:
11         DFS(v, f, traversed_funcs)
12     return traversed_funcs
13
14 for v in all_variables:
15     starting_funcs := functions where v is a referenced non-local
16     traversed_funcs := multi_source_DFS(v, starting_funcs)
17     for f in traversed_funcs:
18         add v to the extraneous parameters of f
```

**Algorithm 1**

```
1  EP := mapping from each function to a set of extraneous parameters
2        (all sets initially empty)
3  stack := new stack
4  for v in all_variables:
5      for f in functions where v is a referenced non-local:
6          add v to EP[f]
7          push (f, v) onto stack
8  while stack is not empty:
9      (f, v) := pop stack
10     for g in functions that reference f:
11         if g is not the declaring function of v and v ∉ EP[g]:
12             add v to EP[g]
13             push (g, v) onto stack
14 output EP
```

**Algorithm 1b**

For each variable, Algorithm 1 performs a depth-first traversal (DFS) starting from the functions where the variable is a referenced non-local and propagating to the functions that reference them. The set *traversed_funcs* records the functions traversed by the DFS. If the DFS encounters a function that has already been traversed, it skips that function; otherwise it adds the function to the set. As a special case, the DFS does not propagate to the declaring function of the variable. When the traversal ends, the variable is added as an extraneous parameter of each of the traversed functions.

We assume a preprocessing step has been used to compute the (reverse) function reference graph and the referenced non-local variables of the functions.

## 4.4 Algorithm 1b

Algorithm 1b is an iterative version of Algorithm 1, produced by converting the recursion to iteration using an explicit stack. It is equivalent to Algorithm 1 apart from some harmless reordering due to the stack being last-in first-out. (Proof sketch of the equivalence: If we trivially refactor Algorithm 1 by moving lines 2–3 from the beginning of *DFS* to immediately before each call to *DFS*, then a call to *DFS* in Algorithm 1 with arguments $(v, f, *)$ corresponds to pushing $(f, v)$ onto the stack in Algorithm 1b.)

The variable *stack* contains pairs $(f, v)$, denoting that the variable $v$ needs to be propagated to the functions that reference $f$. Whenever a variable $v$ is added to $EP[f]$, the pair $(f, v)$ is pushed onto *stack*. All the variables are processed one after another using a single **while** loop. This allows conditions (1) and (2) from the definition of the solution to be neatly translated to a loop invariant.

Theorem 4.7. *The output of Algorithm 1b is the solution.*

Proof. The loop invariant of the **while** loop on line 8 is as follows. It is almost exactly the same as the conditions from the definition of the solution; the important difference is that we added $(f, v) \notin stack$ to the second clause, to relax condition (2) for the pairs currently in the stack.

For all functions $f$ and $g$ and for all identifiers $v$, the invariant is the conjunction of the following clauses:

(1) $v$ is a referenced non-local variable of $f \implies v \in EP[f]$
(2) $v \in EP[f] \wedge (f, v) \notin stack \wedge g$ references $f \wedge g$ is not the declaring function of $v \implies v \in EP[g]$
(3) The only elements in $EP$ are those required by conditions (1) and (2) from the definition of the solution.

*Proof that the invariant holds immediately before the **while** loop is entered:*

The code before the **while** loop (specifically lines 4–7) adds $v$ to $EP[f]$ for all functions $f$ where $v$ is a referenced non-local variable (and only for those functions), so clauses 1 and 3 are satisfied.

The code before the **while** loop pushes $(f, v)$ onto *stack* whenever it adds $v$ to $EP[f]$, so clause 2 is satisfied vacuously.

*Proof that the invariant holds at the end of each iteration of the **while** loop:*

The body of the **while** loop does not remove any identifiers from $EP$, so clause 1 will continue to be satisfied.

Removing the pair $(f, v)$ from *stack* may cause clause 2 to become false for $f$, but the following **for** loop (lines 10–13) will immediately restore the invariant by adding $v$ to $EP[g]$ for all $g$ that reference $f$ (excluding the declaring function of $v$), so at the end of the iteration clause 2 will hold for $f$.

Adding $v$ to $EP[g]$ does not violate clause 2 because $(g, v)$ is also pushed onto *stack*.

Observe that if $(f, v) \in stack$ then $v \in EP[f]$. This follows because the algorithm always adds $v$ to $EP[f]$ before it pushes $(f, v)$ onto *stack*, and it never removes any identifiers from $EP$.

At the beginning of the iteration $(f, v)$ was in *stack*, so $v \in EP[f]$ and condition (2) requires $v \in EP[g]$, satisfying clause 3.

Hence the invariant holds at the end of each iteration.

*Therefore,* the invariant will also hold at the end of the **while** loop. In addition, *stack* will be empty, so $(f, v) \notin stack$ will be true

for all pairs and hence the loop invariant will be logically equivalent to the conditions for a solution. □

Theorem 4.8. *The time complexity of Algorithm 1b is linear in the size of the output program.*

Proof. We assume the availability of map and set data structures providing $O(1)$ access, $O(1)$ initialisation, and iteration linear in the number of elements; see Section 4.5.

The preprocessing step to compute the function reference graph and the referenced non-local variables of the functions can be done using a single linear-time pass over the input program, because each identifier occurrence can be referenced directly by at most one function. This is also linear in the size of the output program, because the output program is at least as long as the input program (they are the same apart from the additional parameters and arguments, and the reordering caused by block floating).

Each **push** operation corresponds to a unique extraneous parameter of some function. They will all be inserted in the output program, so the total number of **push** operations (and hence also **pop** operations) is linear in the size of the output program.

Each iteration of the **for** loop on line 10 corresponds to a unique extraneous argument in some call site or reference, and these arguments will be inserted in the output program, so the total number of iterations is linear in the size of the output program. □

## 4.5 Data structure implementation

Finding map and set data structures that satisfy the assumptions of Theorem 4.8 is not trivial. The most practical approach is to use a hash table, offering an *average* time complexity of $O(1)$ and a worst-case time complexity of $O(n)$ for access operations. Hash tables are highly likely to perform well in practice; but to tighten the theoretical worst-case time complexity of the algorithm, we describe another approach.

We convert all identifiers in the input program to integer IDs in the range 0 to $|I| - 1$ (where $|I|$ is the number of bindings). This can be done using a prefix tree, with time complexity linear in the size of the input program. A map can then be represented as an array indexed by identifier ID, and a set as an array of Booleans. To allow efficient iteration, a linked list should be maintained recording the unique IDs present in the map/set; this also allows efficient clearing.

The only issue is that initialisation is $O(|I|)$; this is problematic because Algorithm 1 and Algorithm 1b create a large number of sets. Fortunately, it is possible to reduce the number of sets to a constant with some small changes:

(A) In Algorithm 1, instead of creating a new set in each invocation of *multi_source_DFS*, we can reuse a single set across all invocations and clear the set between calls.

(B) In Algorithm 1b, we change the values in *EP* from sets to lists; the challenge is to keep the check $v \notin EP[g]$ efficient. Observe that the pairs pushed onto the stack by lines 4–7 are grouped by variable, and that the **while** loop only pushes pairs with the same variable as the most recently popped pair. Hence all pairs with a given variable will be popped from the stack consecutively. So we can use a single set (*traversed_funcs*) to track the visited functions for the current variable: in line 12, we also add $g$ to *traversed_funcs*; and in line 9, if $v$ is different from the variable of the previously-popped pair (or if this is the first iteration), we clear *traversed_funcs* and add the functions where $v$ is a referenced non-local. Then the check $v \notin EP[g]$ can be replaced with $g \notin$ *traversed_funcs*.

(C) The preprocessing step also requires small changes. The natural way to compute the reverse function reference graph is to use a map from each function to a set of functions that reference it, which is problematic. But we can use lists instead of sets, and efficiently exclude duplicate references using a single set as follows: when processing a function $f$, deduplicate the functions it references by adding them to the set, then for each $g$ in the set add $f$ as a function that references $g$; then clear the set. Likewise, the sets of functions where each variable is a referenced non-local can be represented using lists, and duplicate functions can be efficiently excluded using a single set by deduplicating the referenced non-local variables when processing each function.

With these changes the number of maps and sets created by Algorithm 1 and Algorithm 1b is a constant, so the total initialisation time is $O(|I|)$. The time complexity of the clear operations is amortized by the insertions. The proof of Theorem 4.8 covers the rest of Algorithm 1b, so its overall time complexity is linear in the size of the output program.

# 5 Specification

## 5.1 Fischbach–Hannan lifting specification

To prove that our solution produces a valid lifting, we turn to the Fischbach–Hannan specification.

Space precludes a discussion of that specification, so we refer the reader to [6, §3.2]; we also include the rules in Figure 4 for comparison with our Lean translation. Briefly, the judgement $\Gamma \triangleright e : \tau \Rightarrow e'$ states that the expression $e$ can be lifted to the expression $e'$ of type $\tau$, where $\Gamma$ is a context mapping identifiers to types. $\Sigma$ is a predefined signature mapping each constant to its type.

Proving that our solution satisfies the Fischbach–Hannan specification is desirable because then the theorems [6, Theorem 2 (Type Correctness)] and [6, Theorem 4 (Operational Correctness)] apply to our lifting. However, we also wish to prove that our solution gives the minimal extraneous parameters within the class of liftings that we are interested in. So we give a specification for "Johnsson-style" lifting, and prove that this is a restriction of the Fischbach–Hannan specification (Theorem 6.1); then we prove that the solution satisfies our specification for Johnsson-style lifting.

## 5.2 Johnsson-style lifting

In Figure 1 we give a specification of "Johnsson-style" lifting, where each function is assigned a set of extraneous parameters and corresponding extraneous arguments are inserted at each occurrence of the function's name. This specification is slightly more general than Johnsson's algorithm:

- We leave the choice of extraneous parameters unspecified.
- We allow function names to be inserted as extraneous parameters. This is unnecessary when all functions are to be block-floated to the top-level scope, because in that case they will all be in scope; but allowing it simplifies the specification and permits selective lifting. However, we do not allow function names declared in the same `letrec` expression to

$$\frac{\Lambda(x) = \theta}{\Lambda \rhd x \Rightarrow x \mathbin{@} \theta_1 \mathbin{@} \cdots \mathbin{@} \theta_n}\ \text{(var)} \qquad \frac{}{\Lambda \rhd c \Rightarrow c}\ \text{(const)}$$

$$\frac{\Lambda\{y : \emptyset\} \rhd e \Rightarrow e' \qquad y \notin \mathrm{dom}(\Lambda) \qquad \theta \subseteq \mathrm{dom}(\Lambda)}{\Lambda \rhd {}^{\theta}\lambda y.e \Rightarrow (\lambda\theta_1.\cdots.\lambda\theta_n.\lambda y.e') \mathbin{@} \theta_1 \mathbin{@} \cdots \mathbin{@} \theta_n}\ \text{(abs)}$$

$$\frac{\Lambda \rhd e_1 \Rightarrow e_1' \qquad \Lambda \rhd e_2 \Rightarrow e_2'}{\Lambda \rhd e_1 \mathbin{@} e_2 \Rightarrow e_1' \mathbin{@} e_2'}\ \text{(app)}$$

$$\frac{\begin{array}{c}\Lambda^* = \Lambda\{\overline{f_i : ep(e_i)}\} \\ \Lambda^* \rhd f_i = e_i \Rightarrow f_i = e_i' \qquad \Lambda^* \rhd e \Rightarrow e' \qquad f_i \notin \mathrm{dom}(\Lambda) \\ \forall x \in ep(e_j),\ x \in \{\overline{f_i}\} \supset \neg\,\mathrm{abs}(\{\overline{f_i : e_i}\}(x)) \qquad i, j \in \{1..n\}\end{array}}{\Lambda \rhd \texttt{letrec}\ \overline{f_i = e_i}\ \texttt{in}\ e \Rightarrow \texttt{letrec}\ \overline{f_i = e_i'}\ \texttt{in}\ e'}\ \text{(letrec)}$$

$$\frac{\Lambda \rhd e \Rightarrow e' \qquad \neg\,\mathrm{abs}(e)}{\Lambda \rhd f = e \Rightarrow f = e'}\ \text{(var-binding)}$$

$$\frac{\Lambda\{y : \emptyset\} \rhd e \Rightarrow e' \qquad y \notin \mathrm{dom}(\Lambda) \qquad \theta \subseteq \mathrm{dom}(\Lambda)}{\Lambda \rhd f = {}^{\theta}\lambda y.e \Rightarrow f = \lambda\theta_1.\cdots.\lambda\theta_n.\lambda y.e'}\ \text{(fun-binding)}$$

$$ep({}^{\theta}\lambda y.e) = \theta \qquad ep(e) = \emptyset$$

**Figure 1: Johnsson-style lifting**

be inserted as extraneous parameters, because that can lead to infinite types.

- Johnsson's algorithm does not support anonymous functions; instead, it is assumed that each anonymous function $\lambda\ldots$ is replaced with the equivalent expression $\texttt{letrec}\ f = \lambda\ldots\ \texttt{in}\ f$ where $f$ is a fresh identifier. We have chosen to support anonymous functions directly; their extraneous arguments are inserted at the occurrence of the anonymous function, producing an equivalent result.
- We exclude the block-floating step (following [6]).

Our specification loosely resembles the two-phase specification in [6, Fig. 5 and Fig. 6]. That specification is similar to Johnsson's algorithm, but it does not support variable bindings and it allows anonymous functions (however it does not eliminate their free variables).

The judgement $\Lambda \rhd e \Rightarrow e'$ given in Figure 1 states that the input expression $e$ is lifted to the expression $e'$. Each function in the input expression is required to be annotated with an ordered set $\theta$ specifying the chosen extraneous parameters, notated ${}^{\theta}\lambda y.e$ (based on the notation from [6]). For multi-parameter functions, only the first $\lambda$ should have an annotation; the rest should be implicitly $\emptyset$ (but we do not enforce this). $\Lambda$ maps function names to their annotated extraneous parameters; variables are mapped to $\emptyset$.

We use the symbol $\supset$ to denote logical implication, and the notation $\mathrm{abs}(e)$ denotes that $e$ is an abstraction. In the rule (letrec), the premise $\forall x \in ep(e_j), \ldots$ ensures that function names declared in the current `letrec` expression are not used as extraneous parameters. (If $f_i$ are assumed to be distinct, that premise is equivalent to $ep(e_i) \cap \{f_j \mid \mathrm{abs}(e_j)\} = \emptyset$.)

$$\frac{\Psi[x] \in \{\mathrm{G}, \mathrm{L}\}}{\Psi \rhd SC(x)}\ \text{(var)} \qquad \frac{}{\Psi \rhd SC(c)}\ \text{(const)}$$

$$\frac{\begin{array}{c}\Psi' = \{y : \mathrm{N} \text{ if } \Psi[y] = \mathrm{L} \text{ else } \Psi[y] \mid y \in \mathrm{dom}(\Psi)\} \\ \Psi^* = \Psi'\{\overline{x_i : \mathrm{L}}\} \qquad \Psi^* \rhd SC(e) \qquad \neg\,\mathrm{abs}(e) \qquad i \in \{1..n\}\end{array}}{\Psi \rhd SC(\lambda x_1.\cdots.\lambda x_n.e)}\ \text{(abs)}$$

$$\frac{\Psi \rhd SC(e_1) \qquad \Psi \rhd SC(e_2)}{\Psi \rhd SC(e_1 \mathbin{@} e_2)}\ \text{(app)}$$

$$\frac{\begin{array}{c}\Psi^* = \Psi\{\overline{f_i : \mathrm{G} \text{ if } \mathrm{abs}(e_i) \text{ else } \mathrm{L}}\} \\ \Psi^* \rhd SC(e_i) \qquad \Psi^* \rhd SC(e) \qquad i \in \{1..n\}\end{array}}{\Psi \rhd SC(\texttt{letrec}\ \overline{f_i = e_i}\ \texttt{in}\ e)}\ \text{(letrec)}$$

$$\frac{\begin{array}{c}\Psi^* = \{\overline{f_i : \mathrm{G}}\} \\ \Psi^* \rhd SC(e_i) \qquad \Psi^* \rhd SC(e) \qquad i \in \{1..n\}\end{array}}{\rhd SC(\texttt{letrec}\ \overline{f_i = e_i}\ \texttt{in}\ e)}\ \text{(program)}$$

**Figure 2: Complete lifting**

## 5.3 Complete lifting

We are interested in block-floating all functions (including anonymous functions) to the top-level scope. For the resulting program to be valid, the lifted program must not contain any non-local variable references prior to block-floating. This is equivalent to the definition of a supercombinator from [19] except that free function references are allowed (because after block-floating all functions will be in scope).

We refer to a lifting where the lifted program $e'$ has this property as a "complete lifting", formalised by the judgement $\rhd SC(e')$ given in Figure 2 (the name *SC* is a reference to the closely-related term "supercombinator"). $\Psi$ maps each identifier to one of $\{\mathrm{G}, \mathrm{L}, \mathrm{N}\}$ denoting a global variable, local variable, or non-local variable respectively. Function names are mapped to G because after block-floating they will be global.

The rule (program) specifies whether an entire program (represented as a `letrec` expression) is a complete lifting; it is almost the same as the rule (letrec) but it interprets variable bindings as global variables.

# 6 Theorems

This section consists of proofs that together show the solution gives the minimal extraneous parameters for a complete Johnsson-style lifting. We have chosen to do some of the proofs in Lean, while others as traditional "pen and paper" proofs. This choice is driven by practical considerations—some obvious proofs can be difficult to formalise in theorem provers due to the level of detail required.

Theorem 6.1. *(Johnsson-style lifting satisfies the Fischbach–Hannan specification). Let $e$ be an expression annotated with extraneous parameters. Let $\Gamma = \{\}$ and $\Lambda = \{\}$. If $\Gamma \vdash e : \tau$ and $\Lambda \rhd e \Rightarrow e'$ then $\Gamma \rhd e : \tau \Rightarrow e'$.*

Proof. We have proven this in Lean. This theorem is similar to [6, Theorem 5], but it is more general: it applies to any extraneous parameters permitted by our specification of Johnsson-style lifting, as opposed to the particular extraneous parameters chosen by the algorithm from [6]. Our Lean proof uses the ideas from the natural-language proof in [6] ("corresponding contexts" and "closed contexts"). □

Proposition 6.2. *(Well-nesting.) Let $e$ be a closed expression with no shadowing. For all functions $f$ and $g$ and for all identifiers $x$, if $g$ references $f$ and $x$ is in scope at the definition of $f$, then $x$ is also in scope at the reference to $f$ in $g$ (and resolves to the same binding).*

Proposition 6.3. *(Scope extension.) For all functions $f$ and for all identifiers $x$ and $y$,[7] if $y$ is referenced directly by $f$, and $x$ is in scope at that reference, and $f$ is not the declaring function of the binding of $x$, then $x$ is also in scope at the definition of $f$ (and resolves to the same binding).*

Theorem 6.4. *(The extraneous parameters in the solution are in scope). Let $e$ be a closed expression with no shadowing and let $EP$ be the solution. For all functions $f$ and for all identifiers $v \in EP[f]$, $v$ is in scope at the definition of $f$ (and is bound to a non-global variable).*

Proof. We proceed by induction. Consider an identifier $v$ which is in $EP[f]$ by condition (1); so $v$ is a referenced non-local variable of $f$. By Proposition 6.3 (with $x = v$ and $y = v$), $v$ is in scope at the definition of $f$. Now consider an identifier $v$ which is in $EP[g]$ by condition (2); so $v \in EP[f]$, $g$ references $f$, and $g$ is not the declaring function of $v$. By the inductive hypothesis, $v$ is in scope at the definition of $f$. By Proposition 6.2, $v$ is in scope at the reference to $f$ in $g$. By Proposition 6.3 (with $y = f$ and $f = g$), $v$ is in scope at the definition of $g$. □

*Definition 6.5.* The extraneous parameter scoping premises of the Johnsson-style lifting specification are the premise $\theta \subseteq \mathrm{dom}(\Lambda)$ in the rules (abs) and (fun-binding) and the premise $\forall\, x \in ep(e_j), \ldots$ in the rule (letrec).

Corollary 6.6. *Let $e$ be a closed expression with no shadowing. Then the solution satisfies the extraneous parameter scoping premises.*

Proof. The premise $\theta \subseteq \mathrm{dom}(\Lambda)$ in the rules (abs) and (fun-binding) asserts that the extraneous parameters of a function are in scope at the function's definition, so it is satisfied. The premise $\forall\, x \in ep(e_j), \ldots$ in the rule (letrec) is satisfied because the extraneous parameters in the solution are all variables, not function names. □

Theorem 6.7. *(Sufficient conditions for Johnsson-style lifting). Let $e$ be a closed expression with no shadowing, annotated with extraneous parameters that satisfy the extraneous parameter scoping premises. Then there exists an expression $e'$ such that $\{\} \rhd e \Rightarrow e'$.*

[7] $y$ is unimportant, it only serves to establish a location inside $f$.

Proof. We have proven this in Lean, by constructing $e'$ and using the premises of the theorem to prove the corresponding premises of the Johnsson-style lifting specification. □

Corollary 6.8. *The solution satisfies Johnsson-style lifting.*

Theorem 6.9. *(The solution satisfies SC). Let $e^s$ be a closed expression with no shadowing, annotated with the extraneous parameters given by the solution. If $\{\} \rhd e^s \Rightarrow e'$ then $\rhd SC(e')$.*

Proof. Observe that $\mathrm{dom}(\Psi) = \mathrm{dom}(\Lambda)$, since extraneous parameters are only inserted if they are already in scope. By Theorem 6.4, the inserted extraneous parameters only shadow non-global variables; so the resolution of global variables and of functions is the same in $e^s$ and $e'$.

To prove $\rhd SC(e')$, we need to prove $\Psi[x] \in \{\mathrm{G}, \mathrm{L}\}$ for each identifier occurrence $x$ in $e'$. We have proven in Lean that $e'$ is closed; so $x \in \mathrm{dom}(\Psi)$, and we need only prove $\Psi[x] \neq \mathrm{N}$. Assume for contradiction that there exists an identifier occurrence $x$ such that $\Psi[x] = \mathrm{N}$.

If $x$ is not an extraneous argument, then $\Psi[x] = \mathrm{N}$ implies $x$ is enclosed by some function $f$ with $x \notin EP[f]$. This contradicts condition (1) from the definition of the solution.

Otherwise $x$ is an extraneous argument of some function $f$, and $\Psi[x] = \mathrm{N}$ implies the reference to $f$ is enclosed by some function $g$ with $x \notin EP[g]$. If $g$ is not the declaring function of $x$ (in $e^s$), we have a contradiction from condition (2). Otherwise $g$ is the declaring function of $x$; and by Proposition 6.2 (and Theorem 6.4), the binding of $x$ is in scope at the reference to $f$ in $g$, contradicting $\Psi[x] = \mathrm{N}$. □

Theorem 6.10. *(The solution is minimal). Let $e$ be a closed expression with no shadowing, annotated with the extraneous parameters given in a mapping $A$. Suppose $\{\} \rhd e \Rightarrow e'$. Let $EP$ be the solution. If $\rhd SC(e')$ then $EP[f] \subseteq A[f]$ for all functions $f$.*

Proof. We show $v \in EP[f]$ implies $v \in A[f]$ by induction over $EP$. Observe that if an identifier occurrence in $e$ resolves to a non-global variable, then the corresponding occurrence in $e'$ must also resolve to a non-global variable (since the inserted extraneous parameters only create parameter bindings), and recall that $\mathrm{dom}(\Psi) = \mathrm{dom}(\Lambda)$.

Consider $v \in EP[f]$ by condition (1); so $v$ is a referenced non-local variable of $f$ in $e$. By the observation, $v$ must resolve to a non-global variable in $e'$, so $\Psi[v] = \mathrm{L}$ (from $\rhd SC(e')$). The declaring function of $v$ is not $f$ in $e$, so $v$ must be in $A[f]$.

Now consider $v \in EP[g]$ by condition (2); so $v \in EP[f]$, $g$ references $f$, and $g$ is not the declaring function of $v$. By induction, $v \in A[f]$. Since $g$ references $f$ and $v \in A[f]$, $v$ will be inserted as an extraneous argument at the reference to $f$ inside $g$ in $e'$. By Theorem 6.4, $v$ is in scope at the definition of $g$ and resolves to a non-global variable in $e$, and hence also in $e'$. So $\Psi[v] = \mathrm{L}$. The declaring function of $v$ is not $g$ in $e$, so $v$ must be in $A[g]$. □

Corollary 6.11. *The solution gives the minimal extraneous parameters for a complete Johnsson-style lifting.*

## 7 Lean implementation

To allow us to machine-check some of our proofs we translated the specifications to Lean 4, which is both an interactive theorem prover based on the Calculus of Inductive Constructions and a purely functional programming language influenced by Haskell [4]. The source code of the translations, some of the proofs, an implementation of Algorithm 1, and examples is published at [13]; archival artifacts are available at [12]. In this section we show and explain some extracts from the Lean definitions.

The definition of `Expr` maps to $e$ directly:

```
/-- Expressions.
    e ::= c | x | λx.e | e₁ @ e₂ | letrec f̄ᵢ = eᵢ in e  -/
inductive Expr where
  | const : String → Expr
  | var : String → Expr
  | abs : String → Expr → Expr
  | app : Expr → Expr → Expr
  | letrec : List (String × Expr) → Expr → Expr
deriving BEq, Repr
```

`Ty` maps to the version of $\tau$ without singleton types, and $\texttt{Ty}_\texttt{s}$ maps to the extended version of $\tau$ that uses singleton types. The separation allows us to explicitly allow or forbid singleton types. As mentioned earlier, we restrict the expression $e$ in singleton types to be an identifier, and we are careful to disallow "bare" singleton types in $\texttt{Ty}_\texttt{s}$.

```
/-- Types.  τ ::= ι | τ → τ  -/
inductive Ty where
  | base : String → Ty
  | arrow : Ty → Ty → Ty
deriving DecidableEq, Repr

/-- Types, extended using singleton types.
    τ ::= ι | τ → τ | {e}_τ → τ  -/
inductive Tyₛ where
  | base : String → Tyₛ
  | arrow : Tyₛ → Tyₛ → Tyₛ
  | singarrow : String → Tyₛ → Tyₛ → Tyₛ
deriving DecidableEq, Repr
```

We use `deriving DecidableEq` for `Ty` to allow our type checker to test whether elements of `Ty` are equal, and likewise for $\texttt{Ty}_\texttt{s}$.

The name `Ty` was chosen to avoid clashing with Lean's `Type` keyword. For the same reason we later use `S` instead of $\Sigma$ for the mapping from pre-defined constants to their types.

Additional definitions are given in Figure 3.

`Sig` is the type of $\Sigma$, and `Ctx` is the type of $\Gamma$ (with a variant $\texttt{Ctx}_\texttt{s}$ that allows singleton types). We use `Lean.AssocList` to represent these mappings; lookup is not computationally efficient, but that is not a concern for proofs. Note that `Lean.AssocList` keeps duplicate keys when entries are inserted using `insertNew` (but not when using `insert`). Operations such as `find?` return the most recent value inserted for a key, but older values are visible to operations that iterate over the entries such as `any`. The presence of older values is not an issue for us, because we only use operations such as `find?` that ignore older values. We have chosen to keep older values, to make scope shadowing visible when mappings are displayed for debugging, and because `insertNew` is easier to reason about than `insert` (which contains a conditional).

The operations supported by `Lean.AssocList` are not comprehensive, so we extend it with some basic operations (not shown here): `keys`, `insertManyNew`, and `[]?`. We also define the aliases `dom` and `extend` to match the terminology used in the specification.

The premises of the form $\Gamma^* = \Gamma\{\ldots\}$ are difficult to translate to Lean, because `let` is awkward to use within type declarations. Instead, we inline the right-hand side in the places where $\Gamma^*$ is used. The function `extend'` helps make these inlined occurrences concise.

We use Lean's `Vector` type to ensure that lists such as `fs` and `τs` have the same length (`List.zip` silently discards excess elements).

We use `List` to represent sets in `FV` because it is most convenient, although it is not computationally efficient. We do not need to erase duplicate elements because Lean's $\subseteq$ operator ignores duplicates.

Figure 4 shows the Lean translation of the Fischbach–Hannan lifting specification.

The inductive type `LiftJudgement S Γ e τ e'` represents the lifting judgement $\Gamma \rhd e : \tau \Rightarrow e'$ with $\Sigma = \texttt{S}$. We also declare a Lean syntax notation ("mixfix operator") to match the formal notation; the Lean notation differs in that it explicitly mentions `S`, to allow different examples to use different pre-defined constant mappings.

In Lean terminology, the argument `S` of `LiftJudgement` is a *parameter* and is the same across the uses of `LiftJudgement` within the declaration; the remaining arguments are *indices* and vary across the uses [1].

The `notation` declaration requires some explanation. Lean macros are automatically *hygienic*, meaning identifiers are resolved in the context of the macro declaration rather than its use [11]. This is generally a good thing because it prevents unexpected identifier capture, but in our case it creates problems with ordering: if the notation is declared before the declaration of `LiftJudgement`, then the identifier `LiftJudgement` cannot be resolved; and if the notation is declared after `LiftJudgement`, then the notation is not available for use inside the declaration of `LiftJudgement`.

The workaround we have used is to declare the notation before `LiftJudgement`, but use `$(Lean.mkIdent `LiftJudgement)` rather than `LiftJudgement` to delay the resolution of the identifier (effectively disabling macro hygiene).

We have tested our algorithm implementation on examples from the literature and on a large suite of generated test cases; we checked that the output of our implementation satisfies the Johnsson-style lifting specification, the translation of the Fischbach–Hannan specification, and the specification of complete lifting.

## 8 Conclusion

We have formally specified "Johnsson-style" lambda lifting, including the requirements for a "complete" lifting where all functions are converted to top-level functions. We proved our specification satisfies the existing Fischbach–Hannan lifting specification (allowing us to apply their correctness results). We described an algorithm for computing extraneous parameters for lambda lifting, and proved it produces the minimal extraneous parameters that satisfy our specification. The worst-case time complexity of our algorithm is $O(n^2)$ like the fastest existing algorithm; further, in the output-sensitive complexity model, the time complexity of our algorithm is linear in the size of the output—an improvement over existing algorithms.

```
import Lean.Data.AssocList
import LambdaLifting.AssocListExtras
open Lean (AssocList)

/- [The definitions of Expr, Ty, and Tyₛ are given inline. They should be inserted here to produce complete source code.] -/

/-- Converts a `Ty` to a `Tyₛ`. -/
def Ty.toTyₛ : Ty → Tyₛ
  | .base s => .base s
  | .arrow τ₁ τ₂ => .arrow τ₁.toTyₛ τ₂.toTyₛ

/-- The type of pre-defined signatures (Σ) mapping each constant to its `Ty`. -/
abbrev Sig := AssocList String Ty

/-- Typing context (Γ) mapping identifiers to types. -/
abbrev Ctx := AssocList String Ty
abbrev Ctxₛ := AssocList String Tyₛ

/-- Safe alternative to `List.zip` that ensures the arguments have the same length. -/
def zip' {α β} {n : Nat} (as : Vector α n) (bs : Vector β n) : List (α × β) := (as.zip bs).toList

abbrev dom := @AssocList.keys /- Domain (identifiers) of a context. -/
abbrev _root_.Lean.AssocList.extend := @AssocList.insertManyNew /- Extend a context. -/

/-- Convenience wrapper for `extend` for use in `letrec_rule`. -/
def _root_.Lean.AssocList.extend' {α β} {n : Nat} (m : AssocList α β) (as : Vector α n) (bs : Vector β n) :=
  m.extend (zip' as bs)

/-- Set of identifiers appearing as singleton types in τ. -/
-- For simplicity, we don't erase duplicates.
def FV : Tyₛ → List String
  | .base _ => []
  | .arrow τ₁ τ₂ => FV τ₁ ++ FV τ₂
  | .singarrow x τ₁ τ₂ => [x] ++ FV τ₁ ++ FV τ₂
```

**Figure 3: Additional basic definitions**

```
notation:26 "(" S ", " Γ ")" " ⊳ " e " : " τ " ⇒ " e':26 =>
  $(Lean.mkIdent `LiftJudgement) S Γ e τ e'

/-- Parameter lifting judgement, notated as `(S, Γ) ⊳ e : τ ⇒ e'`. -/
inductive LiftJudgement (S : Sig) : Ctx_s → Expr → Ty_s → Expr → Prop where

  | var_rule (Γ : Ctx_s) (x : String) (τ : Ty_s) :
      Γ[x]? = some τ →
      (S, Γ) ⊳ .var x : τ ⇒ .var x
```

$$\frac{\Gamma(x) = \tau}{\Gamma \rhd x : \tau \Rightarrow x}\ \text{(var)}$$

```
  | const_rule (Γ : Ctx_s) (c : String) (τ : Ty) :
      S[c]? = some τ →
      (S, Γ) ⊳ .const c : τ.toTy_s ⇒ .const c
```

$$\frac{\Sigma(c) = \tau}{\Gamma \rhd c : \tau \Rightarrow c}\ \text{(const)}$$

```
  | abs_rule (Γ : Ctx_s) (y : String) (e : Expr) (τ₁ τ : Ty_s) (e' : Expr) :
      (S, Γ.extend [(y, τ₁)]) ⊳ e : τ ⇒ e' →
      y ∉ dom Γ →
      FV (τ₁.arrow τ) ⊆ dom Γ →
      (S, Γ) ⊳ .abs y e : τ₁.arrow τ ⇒ .abs y e'
```

$$\frac{\Gamma\{y : \tau_1\} \rhd e : \tau \Rightarrow e' \qquad y \notin \mathrm{dom}(\Gamma) \qquad FV(\tau_1 \to \tau) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \rhd \lambda y.e : \tau_1 \to \tau \Rightarrow \lambda y.e'}\ \text{(abs)}$$

```
  | lift_abs_rule (Γ : Ctx_s)
      (y : String) (e : Expr) (τ : Ty_s)
      (z : String) (e' : Expr)
      (x : String) (τ₁ : Ty_s) :
      Γ[x]? = some τ₁ →
      (S, Γ) ⊳ .abs y e : τ ⇒ .abs z e' →
      (S, Γ) ⊳ .abs y e : .singarrow x τ₁ τ ⇒ .abs x (.abs z e')
```

$$\frac{\Gamma(x) = \tau_1 \qquad \Gamma \rhd \lambda y.e : \tau \Rightarrow \lambda z.e'}{\Gamma \rhd \lambda y.e : \{x\}_{\tau_1} \to \tau \Rightarrow \lambda x.\lambda z.e'}\ \text{(lift-abs)}$$

```
  | app_rule (Γ : Ctx_s) (e₁ e₂ : Expr) (τ₁ τ : Ty_s) (e₁' e₂' : Expr) :
      (S, Γ) ⊳ e₁ : τ₁.arrow τ ⇒ e₁' →
      (S, Γ) ⊳ e₂ : τ₁ ⇒ e₂' →
      (S, Γ) ⊳ e₁.app e₂ : τ ⇒ e₁'.app e₂'
```

$$\frac{\Gamma \rhd e_1 : \tau_1 \to \tau \Rightarrow e_1' \qquad \Gamma \rhd e_2 : \tau_1 \Rightarrow e_2'}{\Gamma \rhd e_1 \mathbin{@} e_2 : \tau \Rightarrow e_1' \mathbin{@} e_2'}\ \text{(app)}$$

```
  | lift_app_rule (Γ : Ctx_s) (e : Expr) (τ : Ty_s)
      (x : String) (τ₁ : Ty_s) (e' : Expr) :
      (S, Γ) ⊳ e : .singarrow x τ₁ τ ⇒ e' →
      Γ[x]? = some τ₁ →
      (S, Γ) ⊳ e : τ ⇒ e'.app (.var x)
```

$$\frac{\Gamma \rhd e : \{x\}_{\tau_1} \to \tau \Rightarrow e' \qquad \Gamma(x) = \tau_1}{\Gamma \rhd e : \tau \Rightarrow e' \mathbin{@} x}\ \text{(lift-app)}$$

```
  | letrec_rule (Γ : Ctx_s) (n : Nat)
      (fs : Vector String n) (es : Vector Expr n)
      (τs : Vector Ty_s n)
      (e : Expr) (τ : Ty_s)
      (es' : Vector Expr n)
      (e' : Expr) :
    (∀ i, (_ : i < n) →
      (S, /-Γ*-/ Γ.extend' fs τs) ⊳ es[i] : τs[i] ⇒ es'[i]) →
    (S, /-Γ*-/ Γ.extend' fs τs) ⊳ e : τ ⇒ e' →
    (∀ f_i ∈ fs, f_i ∉ dom Γ) →
    FV τ ⊆ dom Γ →
    (S, Γ) ⊳ .letrec (zip' fs es) e : τ ⇒ .letrec (zip' fs es') e'
```

$$\frac{\Gamma^* = \Gamma\{\overline{f_i : \tau_i}\} \qquad \Gamma^* \rhd e_i : \tau_i \Rightarrow e_i' \qquad \Gamma^* \rhd e : \tau \Rightarrow e' \qquad f_i \notin \mathrm{dom}(\Gamma) \qquad FV(\tau) \subseteq \mathrm{dom}(\Gamma) \qquad i \in \{1..n\}}{\Gamma \rhd \mathtt{letrec}\ \overline{f_i = e_i}\ \mathtt{in}\ e : \tau \Rightarrow \mathtt{letrec}\ \overline{f_i = e_i'}\ \mathtt{in}\ e'}\ \text{(letrec)}$$

**Figure 4: Lean translation of the Fischbach–Hannan lifting specification.**

## Acknowledgments

Tom Levy would like to thank Bill Rogers for supervising earlier related work and Marco Morazán for validating and reviewing this work. This paper would not have been written without their support and encouragement.

We wish to thank the anonymous reviewers for their valuable comments, and the School of Computing & Mathematical Sciences of the University of Waikato for providing funding for conference travel.

## References

[1] David Thrane Christiansen. 2025. Functional Programming in Lean. https://docs.lean-lang.org/functional_programming_in_lean/Programming-with-Dependent-Types/Indexed-Families/ Section "Indexed Families".

[2] Alonzo Church. 1941. *The Calculi of Lambda-Conversion.* Princeton University Press, USA.

[3] Olivier Danvy and Ulrik P. Schultz. 2002. Lambda-Lifting in Quadratic Time. In *Functional and Logic Programming*, Zhenjiang Hu and Mario Rodríguez-Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–151. doi:10.1007/3-540-45788-7_8

[4] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:10.1007/978-3-030-79876-5_37

[5] Arthur Evans. 1968. PAL—a language designed for teaching programming linguistics. In *Proceedings of the 1968 23rd ACM National Conference (ACM '68)*. Association for Computing Machinery, New York, NY, USA, 395–403. doi:10.1145/800186.810604

[6] Adam Fischbach and John Hannan. 2003. Specification and correctness of lambda lifting. *Journal of Functional Programming* 13, 3 (2003), 509–543. doi:10.1017/S0956796802004604

[7] Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. arXiv:1910.11717 [cs.PL] https://arxiv.org/abs/1910.11717

[8] R. J. M. Hughes. 1982. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) *(LFP '82)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/800068.802129

[9] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203. doi:10.1007/3-540-15975-4_37

[10] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. doi:10.1093/comjnl/6.4.308

[11] The Lean Developers. 2025. The Lean Language Reference. https://lean-lang.org/doc/reference/4.23.0/Notations-and-Macros/Macros/ Section "Macros". Lean version 4.23.0.

[12] Tom Levy and Steve Reeves. 2026. Lambda Lifting in Lean – source code artifact. Zenodo. doi:10.5281/zenodo.18349369

[13] Tom Levy and Steve Reeves. 2026. Lambda Lifting in Lean – source code repository. https://gitlab.com/tom93/lambda-lifting-lean/-/tree/FormaliSE-2026 (mirrors: https://github.com/tom93/lambda-lifting-lean/tree/FormaliSE-2026 and https://gitlab.com/stever00/lambda-lifting-lean).

[14] John McCarthy. 1978. History of LISP. *SIGPLAN Not.* 13, 8 (Aug. 1978), 217–223. doi:10.1145/960118.808387

[15] Marco T. Morazán and Barbara Mucha. 2006. Improved Graph-Based Lambda Lifting. In *Software Engineering Research and Practice* (Las Vegas, Nevada, USA). CSREA Press, 896–902.

[16] Marco T. Morazán and Ulrik P. Schultz. 2008. Optimal Lambda Lifting in Quadratic Time. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–56. doi:10.1007/978-3-540-85373-2_3

[17] Joel Moses. 1970. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bull.* 15 (July 1970), 13–27. doi:10.1145/1093410.1093411

[18] Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages.* Ph. D. Dissertation. Princeton University, NJ. https://www.cs.princeton.edu/techreports/2020/006.pdf

[19] Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall, Inc., USA.

[20] Rob Pike. 2012. Go at Google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 5–6. doi:10.1145/2384716.2384720 (https://go.dev/talks/2012/splash.article).

[21] Brian Randell and Lawford J. Russell. 1964. *Algol-60 Implementation.* Academic Press, Inc., USA.

[22] Guy L. Steele, Jr. and Gerald Jay Sussman. 1978. *The Art of the Interpreter or, The Modularity Complex.* Technical Report. Massachusetts Institute of Technology, USA.

[23] Gerald Jay Sussman and Guy L. Steele, Jr. 1998. SCHEME: An Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation* 11, 4 (01 Dec 1998), 405–439. doi:10.1023/A:1010035624696

[24] D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. doi:10.1007/978-3-642-40447-4_1

[25] Joseph Weizenbaum. 1968. The FUNARG problem explained. (1968). Unpublished memorandum.