Hashing Modulo Alpha-Equivalence

Krzysztof Maziarz
Tom Ellis
Alan Lawrence
Andrew Fitzgibbon
Simon Peyton Jones
Microsoft Research, Cambridge, UK

Abstract

In many applications one wants to identify identical sub-
trees of a program syntax tree. This identification should
ideally be robust to alpha-renaming of the program, but no
existing technique has been shown to achieve this with good
efficiency (better than $O(n^2)$ in expression size). We present
a new, asymptotically efficient way to hash modulo alpha-
equivalence. A key insight of our method is to use a weak
(commutative) hash combiner at exactly one point in the
construction, which admits an algorithm with $O(n \log^2 n)$
time complexity. We prove that the use of the commutative
combiner nevertheless yields a strong hash with low collision
probability. Numerical benchmarks attest to the properties
of the method.

1 Introduction

Consider the following program fragment

$$(a + (v+7)) \times (v+7)$$

A standard common subexpression elimination (CSE) trans-
formation can rewrite this to

$$\text{let } w = v+7 \text{ in } (a + w) \times w$$

which can be computed more efficiently. However, CSE is
not entirely straightforward. Consider

$$(a + (\text{let } x = \exp(z) \text{ in } x+7)) \times
(\text{let } y = \exp(z) \text{ in } y+7)$$

We might hope that CSE would spot that the two let-bound
terms are $\alpha$-equivalent, and transform to

$$\text{let } w = \text{let } x = \exp(z) \text{ in } x+7 \text{ in } (a + w) \times w$$

We would like to similarly spot the equivalence of the two
lambda terms in

$$\text{foo } (\lambda x. x+7) \ (\lambda y. y+7)$$

and transform to

$$\text{let } h = \lambda x. x+7 \text{ in } \text{foo } h \ h$$

So, we want to find all pairs, or more precisely all equivalence
classes, of $\alpha$-equivalent subexpressions of a given program.
Since the program may be large, we would like to generate
the $\alpha$-equivalence classes in reasonable time. We would ide-
ally want a hash function that can be computed for every
node in a single pass over the tree, which would generate
equivalence classes in the cost of a single sort. Somewhat sur-
prisingly, the CSE literature barely mentions the challenge
of hashing modulo $\alpha$-equivalence, nor does the wider litera-
ture on hashing of program fragments. (One might wonder
whether switching to de Bruijn indexing would solve the
problem, but it does not, as we show in Section 2.4.)

Another challenge is that in typical compilers the pro-
gram is subjected to thousands of rewrites, each of which
transforms the program locally. Ideally, we would like an
incremental CSE algorithm, so that we can continuously
monitor sharing, for example for register pressure sensitive
optimization algorithms.

In this paper we address these challenges, making the
following contributions:

1. We present an algorithm that identifies all equivalence
classes of subexpressions of an expression, respecting $\alpha$
equivalence (Section 3). The algorithm is developed in
two steps. The first defines an $e$-summary at each node;
this step is invertible, allowing the original expression
(modulo $\alpha$) to be reconstructed (Section 4.2). In the
second step we develop a more efficient representation
for $e$-summaries, optimised for the task of producing a
hash code for the $e$-summary (Section 5). This two-step
approach makes the correctness argument easy (Section
3.2).

2. We show that the algorithm runs in sub-quadratic time
(Section 6.1), and is compositional, so that it can readily
be made incremental (Section 6.3).

3. A key step in making the algorithm efficient is to use
a weak hash combiner (exclusive-or) when computing
the hash of a finite map (Section 5.2). At first glance,
that weak combiner threatens the good properties of
hashing. However, we compute the theoretical collision
probability for our hash function, showing it can be
upper-bounded, with the bound decreasing exponen-
tially with the size of the hashing space (Section 6.2).

4. Our proof also lays down several lemmas about compo-
sitional hashing functions, which we believe will prove
useful for analyses beyond the one done in this work.

We give an empirical evaluation of our approach in Section 7,
and discuss related work in Section 8.
2 The problem we address

Many applications require the processing of source code, ranging from classical tools such as compilers and static analysis methods, to recent breakthroughs in code generation using deep learning [2, 3]. While the exact code representation varies depending on the downstream task, one widely used example is an Abstract Syntax Tree (AST), which represents computational expressions using a tree structure.

Subtrees of such a tree-structured expression — referred to as subexpressions — are useful, because they often correspond to semantically meaningful parts of the program, such as functions. However, many such subexpressions may turn out to be equivalent: for example, two functions in a large codebase may be named differently but perform the same computation.

Quickly identifying equivalent subexpressions in a large program is important for a number of tasks. Examples include CSE, as mentioned above; structure sharing to save memory, by representing all occurrences of the same subexpression by a pointer to a single shared tree; or pre-processing for machine learning, where subexpression equivalence can be used as an additional feature, for example by turning an AST into a graph with equality links [2].

2.1 What does “equivalent” mean?

Different downstream tasks may differ in what subexpressions they consider “equivalent”. For example, here are four candidates:

- **Syntactic equivalence** means that two subexpressions are equivalent if they are identical trees; the same shape, with the same nodes, and the same variable names.
- **α-equivalence** is like syntactic equivalence but is insensitive to renaming of bound variables. For example, the expression $(\lambda x. x+1)$ is equivalent to $(\lambda p. p+1)$ (by α-renaming the lambda-bound variable), but not equivalent to $(\lambda q. q+2)$, because the free variables differ.
- **Graph equivalence** goes beyond α-equivalence by treating a let expression as a mere textual description of a graph. So $(\text{let } x = 1 \text{ in } \text{let } y = 2 \text{ in } x+y)$ is equivalent to $(\text{let } y = 2 \text{ in } \text{let } x = 1 \text{ in } x+y)$, and to $(1+2)$, because all three describe the same underlying graph.
- **Semantic equivalence** says that two subexpressions are equivalent if they evaluate to the same value, regardless of the values of their free variables. For example $(3+x+4)$ is equivalent to $(x+7)$ and $(7+x)$ among many others.

The difficulty of deciding equivalence ranges from trivial (syntactic equivalence) to undecidable (semantic equivalence). In this paper we focus on α-equivalence. We specifically do not want to go as far as graph equivalence, because let-expressions express operational choices about object lifetimes and evaluation order, and so graph equivalence is too strong for the downstream tasks that we are interested in. Of course, graph equivalence might be just right for other applications, and it would be interesting to adapt the ideas presented here, but we leave exploring that to future work.

2.2 Baseline: purely syntactic equivalence

Purely syntactic equivalence is easy, and perfect for structure sharing, but not for much else. For example, it is inadequate for CSE, and other tasks, via two primary failure modes: false negatives and false positives.

- **False negatives: sensitivity to arbitrary variable names.** Consider this expression:

\[
\text{map } (\lambda y. y+1) \ (\text{map } (\lambda x. x+1) \ vs)
\]

The two lambda-expressions are not syntactically identical, but they are α-equivalent, and perform the same computation in the same way. Similarly, consider

\[
\text{foo } (\text{let } \text{bar} = x+1 \text{ in } \text{bar*}y)
\]

\[
(\text{let } \text{pub} = x+1 \text{ in } \text{pub*y})
\]

Here we would like to CSE the two arguments to foo, even though they use different binders internally; in short, we seek equality modulo α-equivalence.

- **False positives: name overloading.** Consider the syntactically repeated subexpression $x+2$ in this example:

\[
\text{foo } (\text{let } \text{bar} = x+1 \text{ in } x+2)
\]

\[
(\text{let } \text{pub} = x+1 \text{ in } x+2)
\]

The two subexpressions $x+2$ are unrelated, but they are syntactically identical. If the goal is structure sharing this is fine; indeed we might want to share the two $x+2$ subexpressions, to save memory. But sharing the two would be wrong for CSE and similar tasks. For example, it would be clearly wrong to transform the above expression into

\[
\text{let } \text{tmp} = x+2
\]

\[
(\text{let } \text{bar} = \text{tmp} \text{ in } \text{foo } (\text{let } \text{x} = \text{bar} \text{ in } \text{tmp}) \ (\text{let } \text{x} = \text{pub} \text{ in } \text{tmp})
\]

The second problem can readily be addressed, by preprocessing the expression so that every binding site binds a distinct variable name. This step takes time linear in the expression size $n$; or, more precisely, if we take account of the $O(\log n)$ time to look up a bound variable in the environment, $O(n \log n)$. We assume this preprocessing in all algorithms below.

2.3 Hashing for syntactic equivalence

A standard approach to determining subexpression equivalence is to use some form of hashing. We compute a fixed-size hash code for each node in the tree, and use these hash codes to insert every node into a hash table.

Hashing gives an simple, direct, and compositional implementation for syntactic equivalence: the hash for a node is computed by hashing the node constructor with the hashes of the children. When used for structure sharing (to save memory), this is often called hash-consing. When you are just about to allocate a new node, first compute its hash code, then look that up in the hash table, to see if that node already exists. If so, use it; if not, allocate it and add it to the hash
table. In effect, we simply memoise the node constructor functions.

But this simple approach fails for \(\alpha\)-equivalence, because \((\langle x. x+1 \rangle)\) and \((\langle y. y+1 \rangle)\) have different hash codes. How can we fix up the hashing approach to account for \(\alpha\)-equivalence?

### 2.4 De Bruijn indexing

One well-known way to become insensitive to \(\alpha\)-renaming is to use a nameless representation, typically \textit{de Bruijn indexing}. Lambdas have no binder; and each occurrence of a bound variable is replaced by a number that counts how many intervening lambdas separate that occurrence from its binding lambda. For example, the expression \((\langle x. y. x+y \rangle 7)\) is represented in de Bruijn form by \((\langle . . . \rangle \%1+\%0+7)\), where we use \% to represent a variable occurrence with de Bruijn index \(i\).

After switching to de Bruijn indexing, can we use vanilla hashing to determine equivalence, and thus solve the \(\alpha\)-equivalence challenge? Sadly, no: using de Bruijn indexing remains vulnerable to both false positives and false negatives:

- **False negatives.** Consider
  \[
  \begin{align*}
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ h \ (\langle y. h \rangle) \\
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ h \\
  \end{align*}
  \]
  
  The two subexpressions \((\langle x. x+t \rangle)\) are certainly CSE-equivalent, and we could profitably transform the expression to
  \[
  \begin{align*}
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ (\langle y. h \rangle) \\
  \end{align*}
  \]

  But with de Bruijn indexing, the two expressions look different:
  \[
  \begin{align*}
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ h \\
  \end{align*}
  \]

  Notice how the two occurrences of \(t\) have become \%1 and \%2 respectively.

- **False positives.** Consider
  \[
  \begin{align*}
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ (\langle y. y*(x+1) \rangle) \\
  \end{align*}
  \]

  With de Bruijn indexing this looks like
  \[
  \begin{align*}
  \text{let } h &= \langle x. x+t \rangle \text{ in } f o o\ (\langle . . . \rangle \%1*(\%0+1)) \\
  \end{align*}
  \]

  This has two occurrences of \((\langle . . . \rangle \%1*(\%0+1))\), which might be great for structure sharing, but is wrong for CSE.

  Note that, unlike with simple syntactic equivalence (§2.3), these false positives cannot be eliminated by giving every binder a unique name (§2.2) – with de Bruijn there are no binders!

Moreover, using de Bruijn indexing as the internal representation of an expression in a compiler incurs serious costs of its own, because terms need to be repeatedly traversed as they are substituted under other binders, to adjust their de Bruijn indices. We know of no systematic, quantitative comparison of the engineering tradeoff between de Bruijn and named representations in a substantial application (e.g. a compiler), perhaps because the choice has such pervasive effects that implementors are typically forced to make one choice or the other, and stick to it. A decent attempt was made in [11], with the conclusion that the costs of a de Bruijn representation exceed the benefits.

### 2.5 Locally nameless

The false negatives and false positives of de Bruijn indexing are a serious problem for CSE-like purposes. However they can be avoided by using the “locally nameless” representation [4, 10, 16]. The idea is simple: the hash of an expression is defined to be the hash of the de-Bruijn-ised representation of the subexpression taken in isolation. For example, the hash of \((\langle f x (\langle y. x+y \rangle) \rangle)\) would be the hash of \((\langle f x (\langle . . . \rangle \%1) \rangle)\). In this expression the free variables \(f\) and \(x\) remain unchanged, but the locally-bound variable \(y\) has been de-Bruijn-ised. The expression to be hashed has a mixture of de Bruijn indices and named free variables.

The hash of an application \((e_1 e_2)\) can be obtained by combining the hash of \(e_1\) and \(e_2\); but the hash of \((\langle x. e \rangle)\) cannot be obtained from the hash of \(e\). The hash of \(e\) incorporates the hash of each occurrence of \(x\) in \(e\); but the hash of \((\langle x. e \rangle)\) must instead incorporate the hash of \%1 at each of those occurrences. We cannot do this compositionally; instead we must first replace \(x\) by \%1 in \(e\), and then take the hash of \(that\).

This algorithm correctly does hashing modulo \(\alpha\)-equivalence, but it comes with a cost in asymptotic complexity: as we pass each lambda we must re-hash the entire body. We give the complete code in the supplementary material.

In practice, the locally-nameless scheme works sufficiently well in practice that it is used in Epigram [10] and the LEAN theorem prover [5]. However, it suffers from the same complexity issues as de Bruijn. Other things being equal, we would prefer to avoid compiler technology that has asymptotic complexity holes fundamentally built in, as well as the other index-shuffling costs imposed by a de-Bruijn (including locally nameless) representation [11]. Our contribution is to show how to use a nameful representation (avoiding that index shuffling), and still get compositional hashing with asymptotically-good complexity.

### 3 The key ideas

In this section we describe the key ideas of our approach. Before doing so, it is helpful to articulate our goal more precisely.

**Goal.** Given an expression \(e\), in which every binding site binds a distinct variable name (§2.2), identify all equivalence classes of subexpressions of \(e\), where two subexpressions are equivalent if and only if they are \(\alpha\)-equivalent. We wish to achieve this goal in a way that is:

- **Compositional.** If we have already done the computation for \(e_1\) and \(e_2\), computing the result for \((e_1 e_2)\) should not require work linear in the sizes of \(e_1, e_2\).
• **Efficient.** We consider this to mean that finding all equivalent subexpressions should be sub-quadratic in the size of the expression.

Compositionality helps with efficiency, because combining two smaller expressions \( e_1 \) and \( e_2 \) into a bigger one involves only combining the results of the subexpression. But, crucially for our applications, compositionality also allow hashing to be incremental: if we have already performed the equality-discovery task for a large expression, and we make a small rewrite in that expression, we would like to efficiently recompute the results by examining only parts of the expression that have changed. We give an analysis of incrementality in Section 6.3.

As to complexity, we consider an algorithm that is quadratic in expression size to be too expensive. Linear (constant work at each node) would be ideal; in this paper we achieve log-linear (generally \( O(n \log n) \)), with here \( k = 2 \), which we consider acceptable.

### 3.1 The challenge of compositionality

Given an application \( e_1 \ e_2 \), a compositional algorithm will somehow process \( e_1 \) and \( e_2 \) separately, and combine those results to get the results for \( (e_1 \ e_2) \). Let us call "the result of processing an expression" the *e-summary* for that expression. You could imagine attaching the e-summary for \( e \) to the tree node for \( e \), and computing the e-summary for \( (e_1 \ e_2) \) from the e-summaries for \( e_1 \) and \( e_2 \). The idea is that an e-summary precisely identifies the equivalence class: two nodes are \( \alpha \)-equivalent iff their e-summaries are identical.

The difficulty with this approach is that the expressions \( (x+2) \) and \( (y+2) \) are different, and must have different e-summaries, but \( (\alpha \cdot x+2) \) and \( (\gamma \cdot y+2) \) are \( \alpha \)-equivalent, and must have the same e-summary. So an e-summary cannot be a simple numeric hash code, because there is no way to get the hash-code for, say, \( x+2 \) from the hash-code of \( x+2 \).

So we need a richer e-summary. At one extreme, an expression could of course be its own e-summary! That would be fast to compute (a no-op), and compositional, but the task of finding equivalence classes would be absurdly expensive: doing a full \( \alpha \)-respecting equality comparison between every pair of expressions. We seek something in between the two. We will define an e-summary that is not fixed-size (like a hash code), but from which we can rapidly compute a hash code.

### 3.2 Overview of our approach

A general overview of our approach is as follows.

**Step 1 (Section 4).** We define a particular e-summary, with the following properties:

- The e-summary for an expression can be computed in a compositional way (Section 4.6). The cost of computing it for all subexpressions is quadratic in expression size, a problem we fix later.
- The e-summary for \( e \) can be converted back to an expression \( e' \) which is \( \alpha \)-equivalent to \( e \) (Section 4.2). That is, e-summaries lose no information, except the names of the bound variables.
- Two e-summaries are equal if and only if the expressions from whence they came are \( \alpha \)-equivalent.

**Step 2 (Section 5).** At this stage it may appear that not much has been gained. An expression \( e \) and its e-summary are inter-convertible, and comparing e-summaries is not much faster than comparing the corresponding expressions. But e-summaries enjoy a crucial extra property: unlike expressions, we can easily represent an e-summary in a hashed form that is much more compact than that of §4, and enjoys \( O(1) \) comparison time.

The two-step approach makes the algorithm easier to reason about. Step 1 loses no information, and hence cannot give rise to false positives. Step 2 is just regular hashing and, like any hash, can suffer collisions and hence false positives; but in §6.2 we show that the probability of collision is bounded and remains inversely exponential in the number of bits in the hash code.

### 4 Step 1: a compositional e-summary

In this section we give the details of our e-summary. Although it does not yet give a way to efficiently find equal subexpressions, it lays the groundwork for Step 2 (§5). Splitting the development in two in this way allows much easier reasoning about correctness.

For the sake of concreteness, we use Haskell to express our algorithm, its data types and its functions.

#### 4.1 Preliminaries

First, we need a datatype for representing the expression:

```haskell
type Name = String

data Expression = Var Name |
                  Lam Name Expression |
                  App Expression Expression
```

We simply use `String` for variable names; a production implementation might use a more efficient representation. As specified in Section 3, we assume that the variable names are all unique. This requirement is easy to satisfy by renaming the variables during preprocessing.

This language is very minimal, but it is enough to demonstrate the \( \alpha \)-equivalence challenge, and it can readily be extended to handle richer binding constructs (let, case, etc), as well as constants, infix function application, and so on.

#### 4.2 The basic e-summary

Recall from Section 3.1 that adding a lambda at the root of an expression must transform distinct e-summaries (for, say, \( x+2 \) and \( y+2 \)) into the same one (for \( \lambda x. x+2 \)). To account for
Hashing Modulo Alpha-Equivalence

Figure 1. (a) Input expression, with names at Lam and Var nodes. (b-f) E-Summaries for subexpressions, names only in the VarMap. This diagram depicts an $O(n^2)$ algorithm ($\S$4.6); then the “smaller subtree” ($\S$4.8) and “xor” ($\S$5.2) modifications reduce complexity for hash computation to $O(n(\log n)^2)$.

this, we define the e-summary of an expression $e$ to be a pair of:

- The structure, or shape, of $e$ ($\S$4.3). The structure of $e$ completely describes $e$ apart from its free variables. Imagine every free variable being replaced by $<$hole$>$, so (add x y) has same structure as (add x x).

- The free-variable map of $e$ ($\S$4.4). The variable map of $e$ is a list of $e$’s free variables, each with a tree of positions in $e$ where it occurs (we define positions in $\S$4.5).

Therefore, an e-summary is a pair of a structure and a free-variable map:

```
data ESummary = ESummary Structure VarMap
```

We will elaborate each of these types in the following sections. Our basic algorithm has the following signature:

```
summariseExpr :: Expression $\rightarrow$ ESummary
```

The function summariseExpr converts an Expression into its ESummary. A key correctness property is that we can reconstruct an Expression from an ESummary, up to the names of bound variables. That is, it is possible to implement

```
rebuild :: ESummary $\rightarrow$ Expression
```

so that rebuild (summariseExpr $e$) is $\alpha$-equivalent to $e$. Therefore an ESummary loses no necessary information. Indeed others have suggested using a representation in which a lambda contains a list of the occurrences of its bound variable as the primary representation of lambda terms [1, 9, 12].

4.3 Expression structure

The structure of an expression $e$ expresses the shape of $e$, ignoring the identity of its free variables.

```
data Structure = SVar  -- Anonymous
  | SLam (Maybe PosTree) Structure
  | SApp Structure Structure
```

Variables are replaced by an anonymous SVar. A lambda does not name its bound variable; instead it lists the positions at which that bound variable occurs in its body. Of course an actual list, of the form $\langle L, LLRL, RRL \rangle$ would be alarmingly inefficient, instead position trees (type PosTree below) are used, as described in $\S$4.5.

We will build values of type Structure using ”smart constructors”.

```
mkSVar :: Structure
mkSLam :: Maybe PosTree $\rightarrow$ Structure $\rightarrow$ Structure
mkSApp :: Structure $\rightarrow$ Structure $\rightarrow$ Structure
```

You can think of these as simply renamings of the underlying data constructors (e.g. $\text{mkSVar} = \text{SVar}$), but in $\S$5.1 we will exploit the flexibility of being able to redefine $\text{mkSVar}$.

4.4 Free variables map

The free-variable map of an expression maps each free variable of $e$ to the positions at which that variable occurs. It supports the following operations:

```
emptyVM :: VarMap
singletonVM :: Name $\rightarrow$ PosTree $\rightarrow$ VarMap
extendVM :: Name $\rightarrow$ PosTree $\rightarrow$ VarMap $\rightarrow$ VarMap
removeFromVM :: Name $\rightarrow$ VarMap $\rightarrow$ (VarMap, Maybe PosTree)
  -- Removes one item from the map, returning what
  -- the variable mapped to, or Nothing if it
  -- was not in the map
toListVM :: VarMap $\rightarrow$ [(Name, PosTree)]
```

One possible implementation of VarMap is to use Haskell’s Data.Map library:

```
type VarMap = Map Name PosTree
```

We will introduce a few more operations on VarMap as we go along; all can be implemented straightforwardly using standard libraries.

4.5 Position trees

A value of type PosTree identifies a set of one or more SVar nodes inside a Structure. A PosTree is a skeleton tree, with the same structure as the expression, reaching only the leaves of the expression that are occurrences of one particular variable:

```
data PosTree = PThere
```

We will introduce a few more operations on VarMap as we go along; all can be implemented straightforwardly using standard libraries.
After designing the auxiliary data structures, we may instantiate our algorithm as follows:

```haskell
| data ESummary = ESummary Structure VarMap
|
| summariseExpr :: Expression -> ESummary
| summariseExpr (Var v) = ESummary (mkSVar (singletonVM v mkPTHere))
|
| summariseExpr (Lam x e) = ESummary (mkSLam x_pos str_body) vm_e
|   where
|     ESummary str_body vm_body = summariseExpr e (vm_e, x_pos) = removeFromVM x vm_body
|
| summariseExpr (App e1 e2) = ESummary (mkSApp str1 str2) (merge vm1 vm2)
|   where
|     ESummary str1 vm1 = summariseExpr e1
|     ESummary str2 vm2 = summariseExpr e2
|     merge =
|       mergeVM mkPTLeftOnly mkPTRightOnly mkPTBoth
|
| mergeVM :: (PosTree -> PosTree) -- Left only
|   -> (PosTree -> PosTree) -- Right only
|   -> (PosTree -> PosTree -> PosTree) -- Both
|   -> VarMap -> VarMap -> VarMap
```

Most of the work is done in `App` nodes, where we need to combine variable maps from the node’s children. To that end, we use a new function `mergeVM`, which combines the position trees from the children’s maps. The three argument functions to `mergeVM` say what to do if only the left map has the variable in its domain, only the right map does, or both. In the call to `mergeVM` we simply use the constructors from `PosTree` for these three cases.

The time complexity of this version of our algorithm is quadratic, because at each `App` node the `mergeVM` operator must touch every element of the domain of the mapping, taking time proportional to the number of free variables of the expression. In §4.8, we discuss the key optimization needed to bring the complexity down to log-linear. However, we first prove that the conversion to `ESummary` is reversible by designing the `rebuild` function.

### 4.7 Rebuilding

The `rebuild` function (Section 4.2) is easy to write:

```haskell
rebuild :: ESummary -> Expression
rebuild (ESummary v) = Var (findSingletonVM v)
rebuild (ESummary (Var s)) = Lam x (rebuild (ESummary s (extendVM s x p)))
  where x = ...
rebuild (ESummary (SApp s1 s2)) = App (rebuild (ESummary s1 vm1)) (rebuild (ESummary s2 vm2))
  where m1 = mapMaybeVM pickl vm m2 = mapMaybeVM pickr vm
```

In this function we use two new functions over `VarMap`:

- `findSingletonVM` expects its argument to be a singleton map, and returns the unique name from its domain. (It should map this Name to `PTHere`.) The function fails if the map is not a singleton, but that should not happen if the `ESummary` is well-formed.

- `mapMaybeVM` applies a function to every element of the domain of the map; if the function returns `Nothing` that element is deleted.

In the `SLam` case we have to invent a fresh variable name, since the original name is not recorded, and hence the returned expression is only $\alpha$-equivalent to the original, not identical.

Why do we go to the trouble of defining `rebuild`, which is not even part of the original problem specification? We define `rebuild because its existence guarantees that our e-summary is not information-losing`, and that in turn guarantees that the hash-code for an e-summary will have few collisions (assuming it is a strong hash). This is important. For example, consider a degenerate, information-losing e-summary that

---

1 As an alternative, it would be easy to record that name in the `Structure`, to recover precisely the original expression, rather than just an $\alpha$-equivalent one. If we did so, this name should not participate in calculation of the hash values described in §5.
When processing an App node, the algorithm from §4.6 uses mergeVM which transforms every element of its range, thereby taking time proportional to the number of free variables of the expression. For very unbalanced trees this might be quadratically expensive. In this section we modify the algorithm so that it only transforms the smaller map, leaving the other unchanged. The more unbalanced the tree, the less traversal we do; the worst case becomes a balanced tree, and that has only $O(n \log n)$ complexity.

First, we augment the Structure datatype with a Bool flag in SApp that records which child has more free variables:

```haskell
data Structure
    = SVar
    | SVar (Maybe PosTree) Structure
    | SApp Bool Structure Structure
      -- True if the left expr has more free vars
      -- False if the right expr has more free vars
```

Now, the key App case of summariseExpr becomes

```haskell
summariseExpr (App e1 e2) = ESummary str vm
  where
    ESummary str1 vm1 = summariseExpr e1
    ESummary str2 vm2 = summariseExpr e2
    str = mkSApp left_bigger str1 str2
    tag = structureTag str
    vm = foldr add_kv big_vm (toListVM small_vm)
    left_bigger = vm1 `isBiggerThanVM` vm2
    (big_vm, small_vm) = if left_bigger
      then (vm1, vm2)
      else (vm2, vm1)
    add_kv :: (Name, PosTree) -> VarMap -> VarMap
    add_kv (v, p) vm
      = alterVM (\mp -> mkPTJoin tag mp p) v vm
    alterVM :: (Maybe PosTree -> PosTree)
      -> Name -> VarMap -> VarMap
      -- Alter the value to which the key is mapped

As you can see from the definition of vm, we convert the smaller map to a list of key-value pairs using toListVM, add them one at a time to the larger map using add_kv. The new function alterVM alters the mapping at one key; the argument function allows the caller to behave differently depending on whether or not the key was in the map beforehand. But what is this mysterious tag and the new mkPTJoin operation on position trees?\(^2\)

\(^2\) Readers who feel there must be a more mathematically elegant way to do this might enjoy Appendix C, but the way described here is simple and fast.

First, structureTag extracts from a Structure some kind of “tag” (an integer, say)

```haskell
type StructureTag = Int
structureTag :: Structure -> StructureTag
```

This function must satisfy one simple property: *a structure must have a different tag to the tag of any of its sub-structures.* We abstract away the exact implementation of structureTag, but one simple possibility is to have it return the depth of the Structure, which can be computed and stored at the point when a Structure is constructed.

Next, here is the new definition of PosTree:

```haskell
data PosTree
    = PHere
    | PTJoin StructureTag
    | PTHere
    | PLam (Maybe PosTree) Structure
    | SApp Bool Structure Structure
      -- Child from bigger map
      -- Child from smaller map
```

As you can see from add_kv, we make a tagged PTJoin for every variable in the smaller map, but variables that appear only in the larger map are left untouched. The tag allows rebuild to invert this combining operation, in a unique way, determined by whether or not each item is tagged with the tag for this structure:

```haskell
rebuild (ESummary str@(SApp left_bigger s1 s2) vm)
  = App (rebuild (ESummary s1 vm1))
    (rebuild (ESummary s2 vm2))
  where
    tag = structureTag str
    small_m = mapMaybeVM upd_small vm
    big_m = mapMaybeVM upd_big vm
    (vm1,vm2) = if left_bigger
      then (big_m,small_m)
      else (small_m,big_m)
    upd_small :: PosTree -> Maybe PosTree
    upd_small (PTJoin ptag mpt pt)
      | ptag == tag = Just pt
      | otherwise = Nothing
    upd_big :: PosTree -> Maybe PosTree
    upd_big (PTJoin ptag mpt pt)
      | ptag == tag = mpt
      | otherwise = Just pt
```

The left_bigger flag in SApp tells whether the bigger map came from the left or right argument. The tag in PTJoin tells whether it belongs to the SApp under consideration, or belongs to one deeper in the structure.

Note that in this version of summariseExpr, the amount of work done in an App node is proportional to the size of the smaller variable map from the node’s children.

5 Step 2: hashing an e-summary

To obtain an integer hash value that can be used for downstream tasks, we will use the following functions

```haskell
hashStructure :: Structure -> HashCode
hashVM :: VarMap -> HashCode
hashESummary :: ESummary -> HashCode
```
Our aim is for all of these functions to work in $O(1)$ time. Conversion from an e-summary to HashCode is information-losing, and non-invertible.

To implement hashESummary, we may simply do:

$$
\text{hashESummary (ESummary str map)} = \\
\text{hash (hashStructure str, hashVM map)}
$$

We deal with hashStructure in Section 5.1, and hashVM in Section 5.2.

## 5.1 Hashing structures

hashStructure can easily be implemented by computing the hash at construction time, and storing it in the Structure object itself. That would be enough to achieve the complexity bound we desire. But there is an even more attractive possibility: since hashStructure is the only function we will need for structures, we can represent a structure simply by its hash code, dispensing entirely with the tree, thus:

```
type Structure = HashCode
```

```
-- "Constructors" combine hash values
mkSVar :: Structure
mkSLam :: Maybe PosTree -> Structure -> Structure
mkSApp :: Structure -> Structure -> Structure

hashStructure :: Structure -> HashCode
hashStructure s = s
```

The “constructors” of the tree are implemented by $O(1)$ hash combinators, and hashStructure becomes the identity function. We can apply precisely the same reasoning to PosTree, and represent a value of type PosTree by its HashCode.

Identifying each Structure and PosTree with its HashCode, obviously has constant factor much lower than representing structures and positions as trees: instead, we only manipulate hash codes. In exchange, we will no longer be able to write rebuild. But rebuild is not used in the final implementation; its only purpose is that its existence shows the correctness of the algorithm. By thinking first in terms of the non-information-losing data structure, and then thinking of efficient representations of those structures, we can get both an easy correctness argument and an efficient implementation.

## 5.2 Hashing variable maps

Hashing variable maps is a little more tricky. It would be prohibitively (indeed asymptotically) slow to compute the hash of the variable map afresh at each node. Instead, as for structures, we would like to compute the hash of a node’s variable map using the hashes of its children. Doing so is far from trivial. We might try to pair a map with its hash, thus:

```
data VarMap = VM (Map Name PosTree) HashCode
```

and try to compute the hash for $(f \text{ } \text{vm} \text{ } \text{args})$, where $f$ is a function that returns a new VarMap, from the hash of $\text{vm}$ and $f$’s other arguments $\text{args}$. But consider removeVM: how can we start with the hash of a map, and compute the hash of a map from which a particular entry has been removed?

Our key idea is this: we define the hash of a variable map as the XOR, written $\oplus$, of the hashes of its entries, where an entry is a (variable,position-tree) pair $(a, p)$. This definition has big advantages:

- Since $\oplus$ is commutative and associative, it does not matter in which order we consider its entries.
- We can compute the hash of removing $(a, p)$ from a map $m$ by simply XORing $m$’s hash with the hash of $(a, p)$, since $(a \oplus b) \oplus a = b$

More generally, we could use any operator $\oplus$ that is associative, commutative, and invertible. The trouble is that XOR is a cryptographically weak hash combiner, so using it to combine hashes in this way looks suspicious—won’t we get lots of unwanted collisions? Fortunately, these fears are unfounded: we prove in Section 6.2 that in our algorithm the use of XOR does not lead to excess hash collisions.

Computing hashes is now rather easy. The algorithm of §4.8 needs only singletonVM, alterVM, and removeFromVM:

```
entryHash :: Name -> PosTree -> VarMapHash
entryHash key pos = hash (key, pos)

singletonVM :: Name -> PosTree -> VarMap
singletonVM key pos = FM (Map.singleton key pos)

alterVM :: (Maybe PosTree -> PosTree)
\rightarrow Name \rightarrow VarMap \rightarrow VarMap
alterVM f key (VM entries old_hash)
| Just old_pt <- lookupVM entries key
, let new_pt = f (Just old_pt)
= VM (Map.insert key new_pt entries)
(old_hash @ entryHash key old_pt
@ entryHash key new_pt)
| otherwise
, let new_pt = f Nothing
= VM (Map.insert key pt entries)
(old_hash @ entryHash key new_pt)

removeFromVM :: Name -> VarMap
\rightarrow (VarMap, Maybe PosTree)
-- Deletes a Name from the VarMap, returning
-- its current PosTree, or Nothing if it was not in the map
removeFromVM key map@(VM entries existingHash)
| Just pt <- Map.lookup key entries
= (VM (key \map\{\text{Map}\}.\text{delete}\} entries)
(existingHash @ entryHash key pt), Just pt)
| otherwise
= (map, Nothing)
```

## 6 Analysis

In this section, we formally analyze the time complexity of our final algorithm, and then upper-bound the probability of
obtaining incorrect results due to hash collisions. Throughout this section we use $|e|$ to denote the number of nodes in expression $e$.

### 6.1 Time complexity

We start by bounding the amount of work done in $\text{App}$ nodes, and then derive the time complexity for the entire algorithm. We present first a formal proof, and then an intuitive argument which may help the reader to see why the formal proof works.

**Lemma 6.1.** Let $e$ be an expression. The total number of $\text{extendVM}$ and $\text{removeFromVM}$ operations performed in $\text{App}$ nodes by $\text{summariseExpr}$ ran on $e$ is $O(|e| \log |e|)$.

**Proof.** Denote the number of operations in question as $O_{\text{App}}(e)$. Let us define

$$T(n) = \max_{|e|: |e| \leq n} O_{\text{App}}(e).$$

We will prove that $T(n)$ is $O(n \log n)$, which will conclude the proof of the lemma.

First, consider a single $\text{App}$ node $v$ in $e$ with children $v_1, v_2$. The number of map operations performed in $v$ is $O(\min(m_1, m_2))$, where $m_i = |\text{map}_i|$ is the size of the free variables map from $v_i$. Since a free variables map only contains variables that are used in a given subtree, its size is bounded by the number of nodes in the subtree - i.e. $|\text{map}_i| \leq |v_i|$, and therefore $\min(|\text{map}_1|, |\text{map}_2|) \leq \min(|v_1|, |v_2|)$.

From the analysis above we get that for $n > 1$

$$T(n) \leq \max_{1 \leq a \leq n-1} (T(a) + (n - 1 - a) + C \cdot \min(a, n - 1 - a)),$$

where $a$ and $n - 1 - a$ correspond to $|v_1|$ and $|v_2|$, respectively, and $C$ is a constant resulting from the use of $O$ notation. Due to symmetry, we can rewrite Equation 1 as

$$T(n) \leq \max_{1 \leq a \leq \frac{n}{2}} (T(a) + T(n - 1 - a) + Ca).$$

Now, we will prove inductively that $T(n) \leq Cn \log_2 n$. The base case of $n = 1$ holds, since $T(1) = 0$ as an expression consisting of a single node cannot have any $\text{App}$ nodes. Then

$$T(n) \leq \max_{1 \leq a \leq \frac{n}{2}} (T(a) + T(n - 1 - a) + Ca)$$

Using $T(n - a) \leq T(n - a + 1)$:

$$\leq \max_{1 \leq a \leq \frac{n}{2}} (T(a) + T(n - a) + Ca)$$

Using max $e \leq \max e$ when $S \subseteq S'$:

$$\leq \max_{1 \leq a \leq \frac{n}{2}} (T(a) + T(n - a) + Ca)$$

By inductive hypothesis on $T(a)$ and $T(n - a)$:

$$\leq \max_{1 \leq a \leq \frac{n}{2}} (C \log_2 a + (n - a) \log_2 (n - a) + Ca)$$

Using $\log_2 a + 1 = \log_2 2a$:

$$= \max_{1 \leq a \leq \frac{n}{2}} (a(\log_2 a + 1) + (n - a) \log_2 (n - a))$$

And $2a \leq n$:

$$\leq \max_{1 \leq a \leq \frac{n}{2}} (a \log_2 n + (n - a) \log_2 n)$$

Expression under max does not depend on $a$:

$$= C \cdot n \log_2 n$$

An intuitive alternative to the above derivation may prove illuminating: in an $\text{App}$ node, we do work proportional to the smaller subtree. Let’s imagine that we are touching every node in that subtree to mark the amount of work done. We need to compute the total number of touches. Now flip this around: how many times could a fixed node $v$ be touched? If we follow a path $v_1, v_2, ..., v_k$ from $v$ to the root, any $\text{App}$ node $v_i$ on such path could “trigger” $v$ being touched, but only if $v_{i-1}$ was the smaller child of $v_i$. Therefore, if we follow the path $v_1, ..., v_k$, any time we see a node that triggered touching $v$, the current subtree size at least doubles. Therefore $v$ could only have been touched $\log n$ times. There are $n$ nodes $v$, so total touches can’t exceed $n \log n$.

**Lemma 6.2.** Let $e$ be an expression. The total number of map operations (i.e. $\text{extendVM}$, $\text{removeFromVM}$ and $\text{singletonVM}$) performed by $\text{summariseExpr}$ ran on $e$ is $O(|e| \log^2 |e|)$.

**Proof.** We perform exactly one map operation per every $\text{Var}$ and $\text{Lam}$ node, while the total number of map operations performed in $\text{App}$ nodes is bounded due to Lemma 6.1.

**Theorem 6.3.** Let $e$ be an expression. The total running time of the summariseExpr algorithm ran on $e$ is $O(|e| \log^2 |e|)$.

**Proof.** The $\text{extendVM}$, $\text{removeFromVM}$ and $\text{singletonVM}$ operations are dominant, so it is sufficient to bound the time spent in these operations.

From Lemma 6.2, we get that the total number of these operations is $O(|e| \log |e|)$. Since we implement the map as a balanced binary search tree, addition and removal take time logarithmic in terms of the size of the map (which never exceeds $|e|$), while $\text{singletonVM}$ takes constant time.

### 6.2 Proof that our hashing function is strong

In this section, we show that $\text{summariseExpr}$ composed with $\text{hashESummary}$ is a strong hashing function. We assume that we have access to a source of true randomness (e.g. a stream of random bits), which can be used to instantiate randomly chosen hash combinators. Under this assumption, we prove that it is possible to choose all hashing functions and hash combinators in a randomized way, such that the probability of hash collisions is low.
Definition 6.4. We will call a function \( f : A \rightarrow B \) random if every value of \( f(a) \) for \( a \in A \) was chosen uniformly over \( B \), and independently from all the other values \( f(a') \) for \( a' \neq a \).

Note that a random function \( f \) according to Definition 6.4 is one that was chosen randomly, but when \( f \) is called for a fixed argument, its value is deterministic.

In practice, it may not be possible to obtain true randomness, or one may prefer to fix the seed and make the hashing algorithm deterministic; nevertheless, our theoretical results indicate that there is no more reason to expect hash collisions than if we had used a strong combiner in §5.2.

Throughout this section we denote the hash width as \( b \in \mathbb{Z}_+ \), and \( \mathbb{H} = \{0,1\}^b \). By hash we denote calls to a generic hash function for primitive objects.

Lemma 6.5 (XOR hash combiner for sets). Given a random function \( f : A \rightarrow \mathbb{H} \), define the set hash \( h : 2^A \rightarrow \mathbb{H} \) by

\[
h(S) = \bigoplus_{s \in S} f(s)
\]

where \( \bigoplus \) denotes XOR-aggregation. Then

\[
\forall s_1,s_2 \subseteq A, s_1 \neq s_2, h(S_1) = h(S_2) = 1/2^b
\]

Proof. Fix \( S_1 \) and \( S_2 \); from the properties of XOR, we have

\[
\bigoplus_{s \in S_1} f(s) = \bigoplus_{s \in S_2} f(s) \iff \bigoplus_{s \in S_2} f(s) = 0
\]

where \( S_1 \cap S_2 \) is the symmetric difference of \( S_1 \) and \( S_2 \). As \( S_1 \cap S_2 \neq \emptyset \), we can take any \( x \in S_1 \cap S_2 \), and obtain

\[
\bigoplus_{s \in S_2 \cup S_1} f(s) = 0 \iff f(x) = \bigoplus_{s \in S_2 \cup S_1 \setminus \{x\}} f(s) \quad (3)
\]

Since the values of \( f \) are chosen independently, we may assume the value for \( x \) is drawn last, at which point the right side of Equation 3 is a constant. As \( f(x) \) is chosen uniformly, the probability of \( f(x) \) being equal to any constant is \( 1/2^b \). \( \square \)

Lemma 6.6. Let \( D \) be a datatype defined recursively (such as Structure or PosTree). It is possible to construct in a randomized way a compositional hashing scheme \( h : D \rightarrow \mathbb{H} \) (that is, compute the hash for \( d \in D \) in the constructor by calling a hash combiner on the hashes of children), so that

\[
\forall a,b \in D, a \neq b, p(h(a) = h(b)) \leq \frac{|a| + |b|}{2^b}
\]

where \( |d| \) is the number of constructor calls when building \( d \) (i.e. both those for “leaf” objects and “branch” combiners).

Proof. See Appendix A. \( \square \)

Theorem 6.7. Let \( E \) be the set of all Expression objects. It is possible to instantiate summariseExpr with hashing functions and combiners into \( \mathbb{H} \), chosen in a randomized way, so that

\[
\forall e_1,e_2 \in E, e_1 \neq e_2, p(h(e_1) = h(e_2)) \leq 5 \frac{|e_1| + |e_2|}{2^b}
\]

where \( h(e) = \text{hashESummary(summariseExpr(e))} \) and \( e_1 \neq e_2 \) means \( e_1 \) and \( e_2 \) are not \( \alpha \)-equivalent.

Proof. Since a full (hash-free) e-summary preserves all information relevant to \( \alpha \)-equivalence, the only way a collision can happen is when we convert pieces of an e-summary into hash values. We now consider all such places one by one.

First, we may get a collision when either hashing variable names, or auxiliary compositional objects (structures, position lists, and variable map entries that combine the position list with the variable name). As the total number of calls to hash combiners in each of the four categories does not exceed \( |e_1| + |e_2| \), from Lemma 6.6, the probability of collision in any of the four cases is bounded by \( 4 \frac{|e_1| + |e_2|}{2^b} \).

Moreover, collisions may arise due to XOR-aggregation of hashes when hashing variable maps. From Lemma 6.5, the probability of this event is bounded by \( \frac{1}{2^b} \).

Finally, the top-level call to a hash combiner (to combine hashes of structure and variable map) may produce a collision. The probability of this event is bounded by \( \frac{1}{2^b} \).

Summing up, we get

\[
p(h(e_1) = h(e_2)) \leq 4 \frac{|e_1| + |e_2|}{2^b} + 2 \leq 5 \frac{|e_1| + |e_2|}{2^b}
\]

\( \square \)

Theorem 6.8. Let \( E \) be the set of all Expression objects. It is possible to instantiate summariseExpr with hashing functions and combiners into \( \mathbb{H} \), chosen in a randomized way, so that for any \( e \in E \), summariseExpr recovers the correct set of equivalence classes of subexpression with probability at least \( 1 - 5|e|^3 \cdot 2^{-b} \).

Proof. There are \( \binom{|e|}{2} < \frac{1}{2}|e|^2 \) pairs of subexpressions; any single pair can cause a collision leading to summariseExpr returning an incorrect set of equivalence classes. Probability of a collision for a fixed pair of subexpressions is bounded by Theorem 6.7. \( \square \)

One practical consequence of Theorem 6.8 is that 128-bit hashes are enough even for very large-scale applications. Specifically, if \( b = 128 \) and we consider expressions up to a billion nodes, \(|e| \leq 10^6 \), then the probability of having at least one collision is bounded by approximately \( 5 \cdot 10^{27} \cdot 2^{-128} \leq 10^{-10} \). In Appendix B, we empirically verify that the observed collision rate is indeed consistent with theory.

6.3 Incrementality

One crucial property of our algorithm is compositionality: computing the hash of a subtree only requires the results from children, and there is no need to orchestrate anything across the entire expression, with the exception of ensuring all variable names are unique, which is an invariant that is easy to maintain. Therefore, the algorithm can be made incremental: if a subtree of node \( v \) in an expression \( e \) is modified, e-summaries (and therefore hashes) of most nodes will often stay unchanged.
More specifically, let us say that we have already computed all subtree hashes for an expression \( e \), and we modify a subtree under node \( v \). The only affected nodes are those lying on the path from \( v \) to root, and also those in the rewritten subtree (in particular, modifying the subtree might have required creating some fresh nodes). Recomputing e-summaries for the latter is unavoidable, and the cost of that depends on the specifics of a rewrite – for example, if the subtree of \( v \) has constant size, then the work done for the nodes in that subtree will be also constant. In this section, we will focus on the former, and try to bound the amount of work needed to recomput e-summaries for all nodes on the path from \( v \) to root.

Let \( v = u_k, u_{k-1}, \cdots, u_0 \) be the path from \( v \) to the root, where \( h \) is the depth of \( v \). Work done when recomputing the e-summary for \( u_i \) is in the worst case proportional to the size of the free variable map for \( u_i \). Note that any free variable that is used in the subtree of \( u_i \) is either bound as \( f \), then the work done in \( u_i \) is \( O(i + f) \); summing over all \( i \), we get a bound of \( O(h^2 + hf) \). Of course, the upper-bound of \( O(|e|(|e|)^2) \) still holds.

In summary, if a subtree of a node at depth \( h \) is rewritten, then recomputing all subtree hashes takes time \( O(\min(h^2 + hf, |e|(|e|)^2)) \). While in the worst case this is of the same order of magnitude as recomputing all hashes from scratch, if all variables are bound (\( f = 0 \)), and the tree is reasonably balanced, we obtain something much faster; in particular, if the tree is balanced (i.e. has height \( O(|e|) \), then recomputing after a rewrite takes time \( O(|e|)^2) \).

## 7 Empirical evaluation

In this section, we empirically evaluate the running time of our final algorithm. We consider two settings: synthetic, automatically generated lambda terms (§7.1), and several hand-picked realistic examples corresponding to commonly used machine learning models (§7.2).

In Figure 1, we list the hashing algorithms that we compare in this section. The first two, Structural and DeBruijn, are incorrect: they do not meet the specification of §3. Specifically they may equate distinct expressions (false positives), or fail to equate \( \alpha \)-equivalent ones (false negatives). We present these algorithms here to give a sense of the extra performance cost of hashing modulo \( \alpha \)-equivalence.

The LocallyNameless algorithm is the fastest one we know that meets the specification, while Ours is the algorithm presented in this paper.

We implemented all four in Haskell, over the following expression type

```haskell
data Expression h = Var h Name | Lam h Name Expression | App h Expression Expression
```

In each case the hashing algorithm simply annotates each node with a hash-value, yielding a result of type `Expression HashCode`. We did not model the cost of putting these hash-codes into a hash table and identifying equivalence classes, since these costs are the same in all cases. All code needed to reproduce our results is readily available at omitted for double-blind review. The implementation of our algorithm is optimized over the source code listed in this paper by the addition of strictness annotations and replacing two map operations with a single fused map operation in a couple of places. Similar optimisations were applied to the baseline algorithms. The garbage collector was disabled during timing. Constant factors may of course vary in other implementations, but we are mainly interested in their behaviour relative to each other.

### 7.1 Random expressions

In Figure 2a, we show time taken by the four algorithms to hash all subexpressions of randomly generated expressions of varying size. We generated two different families of random expressions.

- **Balanced trees.** Here we generated expressions that are roughly balanced trees, at each point generating a `Lam` or `App` node with equal probability. Each `Lam` node has a fresh binder, and at variable occurrences we choose one of the in-scope bound variables.
- **Wildly unbalanced trees** with very deeply nested lambdas. This case is not as unrealistic as it sounds: a realistic language will include `let` bindings, and deeply-nested stacks of `let` expressions are very common in practice, especially in machine-generated code.

The results reassure us that our algorithm meets the claimed complexity bounds – note the quadratic behaviour of LocallyNameless for unbalanced trees. Moreover, although there is a constant-factor cost compared to the non \( \alpha \)-respecting algorithms, the slowdown is much smaller than LocallyNameless.

### 7.2 Real-life examples

In Table 2, we show a similar evaluation as in the previous subsection, but using real expressions found in machine
Table 2. Empirical evaluation on hand-picked realistic expressions. Each measurement is in milliseconds to compute all subexpression hashes for each expression. Note that the algorithms marked with (*) produce an incorrect set of equivalence classes.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MNIST CNN</th>
<th>GMM</th>
<th>BERT 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural*</td>
<td>0.011 ms</td>
<td>0.027 ms</td>
<td>0.38 ms</td>
</tr>
<tr>
<td>DeBruijn*</td>
<td>0.035 ms</td>
<td>0.089 ms</td>
<td>1.70 ms</td>
</tr>
<tr>
<td>LocallyNameless</td>
<td>0.30 ms</td>
<td>2.00 ms</td>
<td>820.0 ms</td>
</tr>
<tr>
<td>Ours</td>
<td>0.14 ms</td>
<td>0.36 ms</td>
<td>3.6 ms</td>
</tr>
</tbody>
</table>

learning workflows: "MNIST CNN" is a convolution kernel from a deep neural network used in computer vision; "GMM" is the Gaussian Mixture Model benchmark from the ADBench suite [14], and "BERT" is a model from natural language processing. The latter has a parameter controlling the number of "layers", which controls the expression size due to loop unrolling. We see that on practical examples, our algorithm is only up to 4x slower than running de Brujin on the expression once, and much faster than the locally nameless baseline, while enjoying a better bound on the worst-case time complexity. In Figure 2b we show performance on BERT as the number of layers is varied.

8 Related work

We found surprisingly few papers about the problem of finding common subexpressions modulo \(\alpha\)-equivalence. Shao et al describe the impressive FLINT compiler, which uses de-Bruijn representation and aggressive hash-consing to achieve very compact type representations and constant-time equality comparison [13]. It is not clear how they deal with the false positives and false negatives we mention in §2.4. Murphy takes a similar approach in the TILT compiler [11]. Again, the goal is structure sharing and the mechanism is de Bruijn indexing, but he seeks to conceal the tiresome de-Bruijn index-shuffling (which is somewhat exposed in FLINT) behind an abstraction "curtain" that allows the client to use a simpler named interface. He mentions the problem of false negatives, and concludes that the overheads of his abstractions are too high.

Filliatre and Conchon describe a hash-consing library in OCaml, again with the goal of structure sharing [7]. But their focus is very different to ours: they are concerned about the API design for a hash-consing library, including issues such as when to clear out the hash table. Concerning \(\alpha\)-equivalence, they use de Bruijn indexing from the outset, without discussion.

The "locally nameless" representation [4, 10] has a long history, indeed Weirich et al. [16] observe that it "is mentioned in the conclusion of de Bruijn’s paper." They further note that "If we remove names from bound patterns (which are preserved only for error messages) the locally nameless representation interacts nicely with hash-consing, as all \(\alpha\)-equivalent terms have the same representation."

The idea of representing a lambda with a “map” of the occurrences of its bound variable, which we adopt for our e-summaries in §4, has been studied before (e.g. [1, 12]). Kennaway and Sleep describe another representation, director strings, in which information about occurrences is stored in the application nodes, rather than the lambdas [8]. McBride gives a very helpful overview of these approaches [9], but none of them addresses the question of compositional hashing.

Dietrich et al [6] discuss hashing source code abstract syntax trees (ASTs), with the goal of minimising rebuilds in a build system. If, for example, you change white-space layout, the timestamp of the file will change, but the AST (and its hash) will not. They do not consider \(\alpha\)-equivalence at all, and it seems likely that an \(\alpha\)-renamed program would indeed be considered different by their system.

There is a literature [15, 17] on detecting code clones, or plagiarism, which does typically hash ASTs, but there the goal is typically to generate many pairs of candidates (i.e. false positives are welcomed), so does not apply to our use cases.
References


A Proof of Lemma 6.6

For convenience, we first restate Lemma 6.6 below, and then outline the proof.

**Lemma 6.6.** Let $\mathcal{D}$ be a datatype defined recursively (such as Structure or PostTree). It is possible to construct in a randomized way a compositional hashing scheme $h : \mathcal{D} \to \mathbb{H}$ (that is, compute the hash for $d \in \mathcal{D}$ in the constructor by calling a hash combiner on the hashes of children), so that

$$\forall_{a,b \in \mathcal{D}, a \neq b} p(h(a) = h(b)) \leq \frac{|a| + |b|}{2^b}$$

where $|d|$ is the number of constructor calls when building $d$ (i.e. both those for “leaf” objects and “branch” combiners).

**Proof.** Given $d = \text{Con}(d_1, \cdots, d_k) \in \mathcal{D}$, we define

$$h(d) = f(|d|, \text{hash}(\text{Con}), h(d_1), \cdots, h(d_k))$$

where $f$ is a random hash combiner. That is, we combine the hashes of children and the constructor, and salt it with the size $|d|$ of the object $d$. As $f$ only accepts elements of $\mathbb{H}$ as arguments, here we silently assume $|d| < 2^b$. If that is not the case, then the bound to be proven is vacuous, and hence there is nothing to do.

We will now bound the probability of $h(a) = h(b)$ by induction on $\max(|a|, |b|)$.

It is easy to check that for $|a| = |b| = 1$ (i.e. both $a$ and $b$ are leaves) the inequality holds. Now, assume $\max(|a|, |b|) \geq 2$; without loss of generality $|a| \geq |b|$.

Denote $a = \text{Con}_a(a_1, \cdots), b = \text{Con}_b(b_1, \cdots)$. We now distinguish three cases.

**Case 1:** $|a| > |b|$. We can see that computing $h(a)$ and $h(b)$ involves exactly one call to $f$ of the form $f(|a|, *)$, while all the other calls are for $f(x, *)$ for $x < |a|$. Since the values for $f$ are drawn uniformly and independently, we can assume they are drawn in the order of increasing first argument to $f$ (breaking ties arbitrarily). When the value for $f(|a|, *)$ is being drawn, all other values can be considered constant, and so $p(h(a) = h(b)) = \frac{1}{2^{|a|}}$.

**Case 2:** $|a| = |b|$, $\text{Con}_a \neq \text{Con}_b$. Here $h(a) = h(b)$ requires that either $\text{hash}(\text{Con}_a) = \text{hash}(\text{Con}_b)$, or the top level calls to $f$ produce a collision. Similarly to the previous case, the probability of either of these events can be bounded by $\frac{1}{2^{|a|}}$, and we get $p(h(a) = h(b)) \leq \frac{1}{2^{|a|}}$.

**Case 3:** $|a| = |b|$, $\text{Con}_a = \text{Con}_b$. In this final case, the first two arguments to $f$ are the same, which means that a hash collision on children of $a$ and $b$ can possibly propagate upwards and imply a collision at the top level. Therefore, we will need to use the inductive hypothesis.

More specifically, $h(a) = h(b)$ can arise in two ways: either $h(a_i) = h(b_i)$ for all $i$, which of course implies $h(a) = h(b)$, or the two tuples of arguments to $f$ do not match, and the collision is produced with the two top-level calls to $f$.

For the former case, recall that $a \neq b$, and therefore $a_i \neq b_i$ for some $i$. Combining that with $h(a_i) = h(b_i)$, we can apply the inductive hypothesis; this is legal, since $\max(|a_i|, |b_i|) < \max(|a|, |b|)$. The probability of $h(a_i) = h(b_i)$ is therefore bounded by $\frac{|a_i| + |b_i|}{2^{2b}}$. The case when the top level calls to $f$ produce a collision can be analyzed as before, yielding probability of collision equal to $\frac{1}{2^{|a|}}$.

Summing up the probabilities from the two subcases, we get

$$p(h(a) = h(b)) \leq \frac{|a| + |b| + 1}{2^b} < \frac{|a| + |b|}{2^b}$$

$\Box$

B Empirical frequency of hash collisions

To experimentally verify the bound from Theorem 6.7, in this section we evaluate the empirical frequency of hash collisions. To do this, we first modified our algorithm to use 16-bit integers, as for 32-bit and above one needs an enormous number of trials to find a collision.

However, measuring the amount of collisions in a meaningful way is non-trivial: while we can check how often hashes of two random expression collide, it might be that more collisions arise in real applications; what is worse, there may be adversarial pairs of expressions specially crafted to make our hashes collide. Of course, if the hash function is fixed, there exist pairs of expressions that produce a hash collision. However, note that Theorem 6.7 assumes our hash function is not fixed: instead, the hash combinators that are used should be chosen randomly. In practical terms, our theorem states the following: if we instantiate our algorithm, and seed its hash combinators with a randomly chosen seed, then there is no way to consistently break it – while for a fixed seed one can laboriously find a collision, there is no pair of expressions that would collide reliably across many seeds. This is a much stronger claim than just stating that two random expressions rarely collide, and to verify it, one needs to take a malicious user, and try to construct adversarial examples that are more likely to collide than random ones.

Therefore, in this section we consider two ways to create a pair of expressions:

- **Random expressions.** Here we generate two random balanced expressions as in §7.1, and discard pairs that turn out to be alpha-equivalent.
- **Adversarial expressions.** As discussed above, we acted as an adversary, and designed a way to generate pairs of expressions that are more likely to produce collisions. We describe this in detail in Appendix B.1.

For both ways of generating pairs of expressions we varied the expression size between 128 and 4096, for each size drawing $10 \cdot 2^{16}$ pairs and hashing them looking for collisions. We then divided the resulting number of collisions by 10 to get an estimated number of collisions per $2^{16}$ samples. Note that
the resulting value for a perfect hash function would be 1 (in expectation). On the other hand, Theorem 6.7 upper-bounds the collision probability by \( \frac{\log \frac{1}{1 - n}}{2n} \), and since we are comparing pairs of expressions of the same size \( |e_1| = |e_2| = n \), we get an upper-bound of 10n collisions per \( 2^{16} \) samples.

In Figure 3, we plot the resulting number of collisions for the two ways of generating expression pairs, as well as a lower-bound (perfect hash function) and upper-bound (Theorem 6.7). For random expressions, our hash achieves a close-to-perfect number of collisions, which does not appear to grow with \( n \). On the other hand, adversarial expression pairs generate more collisions as \( n \) grows, but still two orders of magnitude less than the theoretical upper-bound. Note that we do not consider \( n > 4096 \), as for \( n = \frac{2^{16}}{10} \approx 6500 \) our upper-bound becomes vacuous: \( n \) is too close to \( 2^b \) to provide any guarantees on the frequency of collisions.

B.1 Generating adversarial expression pairs

Here we describe the procedure to generate adversarial pairs of expressions. This process is not specialized to our specific algorithm, and hence may work for other compositional hashing algorithms that act on tree-like objects.

The idea is as follows: we start with two small non-alpha-equivalent expressions with no free variables. Concretely, we choose

\[
\begin{align*}
e_1 &= \text{\texttt{\textbackslash x . \ App (Var x) (App (Var x) (Var x))}} \\
e_2 &= \text{\texttt{\textbackslash x . \ App (App (Var x) (Var x)) (Var x)}}
\end{align*}
\]

Then, until the right expression size is reached, we transform the expressions by wrapping both of them in either a \texttt{Lam} or an \texttt{App} node. In other words, we create a pair of highly unbalanced expressions (similarly to §7.1) which differ only at the very bottom. Intuitively, when hashing the resulting expressions, the (likely different) hashes of \( e_1 \) and \( e_2 \) will get repeatedly transformed \emph{in the same way} when the algorithm computes the hashes for larger subtrees. Crucially, if the hashes collide at some point, they will stay the same indefinitely, as the way \( e_1 \) and \( e_2 \) are extended upwards is the same.

Hence, a collision at the lower level will propagate to cause a collision at the top level, causing the collision probability to grow with expression size.

C An alternative to StructureTag

Our initial algorithm of §4.6 transforms PosTrees from both subtrees of an \texttt{App} node. Since this is prohibitively expensive, in §4.8 we show that it is enough to transform only \emph{one} of the subtrees, as long as we introduce an appropriately chosen StructureTag. In this section, we discuss an alternative to the StructureTag approach, which yields an algorithm with the same final time complexity.

Since this alternative is more complex, we refrain from mentioning it in the main body of the paper. Here we describe the key ideas involved, which the reader may find interesting.

To arrive at the alternative formulation, consider the following question: can we transform PosTrees from \emph{both} children, and still get good time complexity due to laziness? In other words, can we avoid actually tagging PosTrees in a given map, and instead lazily store the transformation to be applied?

Formally, consider a variable map after the optimizations of §5.1, i.e. with PosTrees simply being represented by hash-codes (elements of \( \mathbb{H} \)). Extending a PosTree with one of the markers (PTLeftOnly, PRightOnly and PTBoth) corresponds to transforming the hash-code in a fixed way (although dependent on the particular hash combiner); denote these transformations as \( f_L, f_R : \mathbb{H} \rightarrow \mathbb{H} \) and \( f_{\text{both}} : \mathbb{H}^2 \rightarrow \mathbb{H} \). To simplify the following analysis, we will now focus on \( f_L \) and \( f_R \); extending it to handle \( f_{\text{both}} \) is easy, as in given node it only needs to be called at most as many times as the size of the \emph{smaller} of the children’s variable maps.

If we focus on the set of values in the variable map, the problem is the following: we want to maintain a set of hash-codes, with the possibility of quickly applying either \( f_L \) or \( f_R \) on all the values. This hints at a lazy solution: maintain a transformation \( f : \mathbb{H} \rightarrow \mathbb{H} \) together with the set of hash-codes, with the meaning that \( f \) should be applied on all elements of the set. Applying \( f \) on all elements of such a lazy-transformation-augmented set is just a matter of setting \( \tilde{f} = f_0 \circ f \).

However, this is not so easy: when we look up an entry from the variable map, we need to pass the obtained value through the lazy transformation \( f \). Therefore, \( f \) has to be represented in a way such that it is possible to evaluate it in constant time, even though \( f \) may have been created out of a very long sequence of function compositions. Moreover, adding a new entry to the variable map requires passing the newly added value \( x \) through \( f^{-1} \), so that when the value is read out later and passed through \( f \) we recover \( f(f^{-1}(x)) = x \).
It remains to show an efficient representation of functions \( f : \mathbb{H} \to \mathbb{H} \), such that composing, evaluating, and inverting takes constant time.

To simplify this, we can notice that fast inversion is not strictly necessary if we always manipulate pairs of a function and its inverse i.e. \((f, f^{-1})\); note that composing \((f, f^{-1})\) with \((g, g^{-1})\) can be computed as \((f \circ g, g^{-1} \circ f^{-1})\). We still need to be able to invert our fundamental building blocks \((f_L, f_R)\), but since that happens only once before the algorithm commences, it does not have to be done in constant time.

While many possible representations that satisfy the aforementioned requirements exist, one natural choice are linear functions, i.e. functions of the form \( f(x) = a \cdot x + b \), where \( a, b \in \mathbb{H} \), and all operations are carried out modulo \(|\mathbb{H}|\). Indeed, a linear \( f \) can be represented with just a pair of \((a, b)\), evaluating it on \( x \in \mathbb{H} \) takes constant time, and a composition of two linear functions represented by \((a_f, b_f)\) and \((a_g, b_g)\) is \((a_f \cdot a_g, a_f \cdot b_g + b_f)\).

Using linear transformations on hash-codes poses a potential risk, as it could lead to collisions, especially if the choice of \( f_L \) and \( f_R \) is particularly unfortunate; because of that, using a StructureTag-based variant is preferable. However, we have also implemented the variant described in this section, and found that it in practice it also produces strong hashes.