DRAFT Stable AST Labeling for Function Inlining

FU YONG QUAH, Imperial College London
DAVID THOMAS, Imperial College London

In this work, we present an algorithm that consistently label nodes in an AST. This labeling algorithm is resilient to transformations performed to the AST, most notably, function inlining [2]. Informally, our algorithm gurantees that the labels assigned to nodes in the AST are independent of the set of transformations performed on the AST. The algorithm asymptotically consumes constant memory and time for every label generated, meaning that integration into existing compilers will not impose significant compile-time overhead. We have applied this algorithm to label function call sites to study function-inlining via iterative compilation [5] in the OCaml programming language [7].

CCS Concepts: •**Software and its engineering → Compilers;**

Additional Key Words and Phrases: Language Processors, Naming, Abstract Syntax Tree

ACM Reference format:
Fu Yong Quah and David Thomas. 2018. *DRAFT* Stable AST Labeling for Function Inlining. *ACM Trans. Program. Lang. Syst.* 9, 4, Article 39 (March 2018), 7 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Name generation for nodes in AST and tree-based intermediate representations (IR) is commonly used in compilers and language processors. A common practice is to label nodes and generate variable names with ad-hoc unique identifiers, such as the `gensym` function as described by [1].

```
int gensym() {
  static int i = 0;
  return i++;
}
```

With the increasing interest in machine learning for compiler optimisation [10], we have witness the increase in compilation of multiple of copies of the same program whilst varying compilation decisions, such as loop-unrolling factor and function inlining. This results in multiple instances of semantically equivalent AST instances that have undergone different sets of manipulations. A well-constructed labelling algorithm will allow a programmer or optimiser to identify the similarities between the different instances of the ASTs. This is not trivial in the presence of mutations due to function inlining.

Stability (formally defined later) ensures that the label of a node is independent of the set of transformations taken throughout the program. The `gensym` function does not satisfy the stability property, as demonstrated by a simple example in figure 1. Taking different inlining decisions on

function call `f(2)` results in different labels for `g(3)` (which was obtained from inlining `f(3)` in both cases).

```
int g() { ... }
int f(int x) {
   return g(x); // [1]
}
int main() {
  int a = g(2)   // [2]
        + f(3);  // [3]
  printf("%d\n", a); // [4]
}
```

```
int g() { ... }
int f(int x) {
   return g(x); // [1]
}
int main() {
  int a = g(2)   // [4] (inline f(2))
        + g(3);  // [5] (inline f(3))
  printf("%d\n", a); // [4]
}
```

```
int g() { ... }
int f(int x) {
  return g(x); // [1]
}
int main() {
 int a = f(2)   // [2] (don't inline f(2))
       + g(3);  // [4] (inline f(3))
  printf("%d\n", a); // [4]
}
```

Fig. 1. The numbers in square brackets in comments refers to the label of the call site generated via `gensym`. The centre and right-most column shows the modified code after different inlining decisions are taken from the code snippet on the left-most column.

An obvious naive solution is to copy the identifiers from the transitive closure of the declaration to function call site, supposedly preserving the stability of the labels. This implementation, however, fails to provide uniqueness across the siblings of a node, as depicted in figure 2. Such conflict can arise from code of the form $f(x) = g(x) + g(x + 23)$, where both application of function g are inlined.

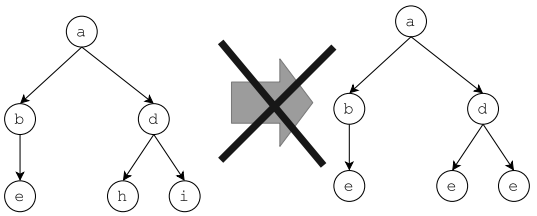

Fig. 2. Replacing nodes e and nodes f with the contents of node b results in a label conflict.

There has been previous work on generating unique identifiers and avoiding the name-capture problem [1, 4] in function inlining. LLVM [6] amends variable names by adding a suffix to register names when performing function inlining. The OCaml compiler [7] uses an approach similar to `gensym` to generate unique names for Flambda, a tree-based IR. The Glasgow Haskell Compiler (GHC) [3] takes an interesting approach towards variable renamining on function inlining in the GHC Core Pass, a tree-based IR, by using trees to generate new variable names [4]. However, based on our knowledge, there is no previous work on generating consistent labels for ASTs and tree-based IRs resilient towards function inlining.

The algorithm and notation described in the text that follows works for both ASTs and tree-based IRs. For simplicity, we will simply refer to all trees as ASTs.

2 PROBLEM DEFINITION

ASTs that require labeling will be constructed in two phases: (1) *Initialization* - consistent labels are not required for nodes at this stage. ASTs can be arbitrarily constructed. The key requirement is that all instances of ASTs must be initialised similarly regardless of later inlining decisions. (2) *Manipulation* - ASTs can be modified with two following operations, which maintains semantic equality in program behaviour:

- Node removal $remove(tree : T, u : V) \rightarrow T$ - Nodes in the AST can be removed without any restriction. Removal does not affect how labels are assigned in other nodes in the AST. This is useful when dead-code elimination is carried out simultaneously with function inlining.

- Inlining - $inline(tree : T, d : V, a : V) \rightarrow T$ is defined as an operation that replaces function application node a with the transitive closure of declaration d, but excluding declaration d itself.

The initialisation and fixation phase correspond to parsing a source file of an AST prior to any compilation passes. The stable manipulation phase corresponds to compilation and optimization passes, where the optimizer is free to make differing inlining decisions for different compilation runs.

The goal of the algorithm is to be able to reference nodes within an AST using labels. We approach this problem via two stages, first, construct labels that are only locally-unique with several useful properties; secondly, construct globally unique application node labels from locally-unique labels.

3 LOCAL APPLICATION LABELS

We define several properties that are useful for local application node labels in ASTs:

Property 1. ***Mutation unrolling*** *where a is the merge destination of some source node d, $\exists(f, g).\forall(v, v') \in M(d, a).g(id(v')) = f(g(id(a)), g(id(v)))$. The identifier yields information about the total amount of merge that was required to arrive at a node.*

Property 2. ***Stability*** $\forall u \in V.\forall v \in C(u)$. *$id(v)$ is independent of $\{id(v') \mid v' \in C(u) \setminus \{v\}\}$ and its descendants. This ensures that the identifier of a node independent of unrelated merge operations.*

Property 3. ***Local uniqueness*** *After an arbitrary number of function inlining, $\forall d \in V.\forall v \in C(d).\forall v' \in C(d) \setminus \{v\}.id(v) \neq id(v')$.*

It is trivial show that node removal will retain all of the above properties on a correctly labelled tree. Function inlining is the main transformation that the algorithm requires special care studying.

We first define a simple AST type definition that consists of only function declarations and applications. In our AST type definition, declarations can be nested within other declaration, allowing high-order functions. We use an integer type as an arbitrary simple stamp.

```
data V  = Apply Id | Decl Id [V]
data Id = (Int, [Id])

id (Apply x) = x
id (Decl x _) = x
children (Apply children) = []
children (Decl _ xs) = xs
update_id (Apply id) id' =
  Apply id'
update_id (Decl id xs) id' =
  Decl id' xs

stamp (x, _) = x
parents (_, p) = p
```

$V = Apply\ \mathcal{I}\ +\ Decl\ (\mathcal{I}\ \times\ [V])$
$\mathcal{I} = (\mathbb{Z} \times [\mathcal{I}])$

$id : V \rightarrow \mathcal{I}$

$children : V \rightarrow [V]$

$update\_id : V \times \mathcal{I} \rightarrow V$

$stamp : \mathcal{I} \rightarrow \mathbb{Z}$
$parents : \mathcal{I} \rightarrow [\mathcal{I}]$

Note that the label, $\mathcal{I}$ is not a single numerical value or a list, but rather, a non-trivial recursive data structure. The algorithm comprises of two main functions: (1) *LabelInline* generates identifiers for children of a declaration node upon function inlining and (2) *LabelUnroll* turns identifiers to a list of stamps. *LabelUnroll* will also be used to prove the necessary properties above. The main idea of *LabelInline* is to use the identifier of apply nodes recursively reference the identifiers of function

applications that was inlined to unveil the said apply node, whereas *LabelUnroll* is executed by recursively walking the identifier data structure. *LabelInline* and *LabelUnroll* in algorithms 1 and 2, while a haskell implementation is depicted in figure 3. Both algorithms are conceptually very simple - the main challenge is proving their conformance with the above properties and ensuring that they have realistic memory footprints.

ALGORITHM 1: Creates a set of nodes with rewritten node identifiers upon function inlining.

Input: Function declaration, d and call site a
Output: $[V]$ a list of nodes with rewritten identifiers.
Function `Inline`(d, a)**:**

```
acc ← []
foreach child in children(d) do
    id_child ← id(child)
    parents' ← id(a) : parents(id_child)
    id'_child ← (stamp(id_child), parents')
    acc ← concat(acc, [update_id(child, id'_child)])
end
return acc
```

ALGORITHM 2: Turns an identifier $\mathcal{I}$ to a list of stamps. This function is semantically equivalent to `concat(concat_map(Unroll, parents(id)), [stamp(id)])`

Input: Node label $\mathcal{I}$
Output: $[\mathbb{Z}]$ a list of stamps corresponding to the identifiers specified above.
Function `Unroll`(id)**:**

```
acc ← [stamp(id)]
foreach parent in reverse(parents(id)) do
    acc ← concat(Unroll(parent), acc)
end
return acc
```

```
inline d a =
  map (\child ->
      let ps = parents (id child)
          ps' = id child : ps
          id' = (stamp (id child), ps')
      in update_id child id')
    (children d)
```

```
unroll id = unroll' [] id
where unroll' init id =
  foldr (\p acc -> unroll' acc id)
    (id stamp : init) (parents id)
```

Fig. 3. Haskell implementations of the algorithm

A alternative naive implementation of this idea is to simply have $\mathcal{I}$ represented as its unrolled form, namely $O(n)$ time for generating identifiers and $O(1)$[1] time for accessing the identifiers. This remove the need of separate *LabelInline* and *LabelUnroll* functions. The downside of this method is that after all transformations, the worst-case total memory required to store all identifiers is quadratic with respect to the number of inline operation on the original AST. A proof can be

[1]Realistically, any attempt to use the identifier will result in a linear-time operation. eg: equality checks. The only real exceptions are pointer comparisons.

trivially be considering the size of identifiers when inlining a function N times, ie: `["a", "a/a", "a/a/a", "a/a/a/a", ...]`.

Proposition 3.1. *LabelInline satisfies property 1*

Proof. Consider $(v, v') \in M(u, a)$ for some (u, a). Now let $f = concat$ and $g = Unroll$

$$
\begin{aligned}
& g(id(v')) = f(concat\_map(g,\ parents(v')),\ [stamp(v')]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v'))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ g(id(v)))
\end{aligned}
$$

In the base case, ie: a freshly initialised AST, $parents(\phi) = \emptyset \leftrightarrow LabelUnroll(\phi) = [stamp(\phi)] = concat(concat\_map(LabelUnroll, parents(\phi)),\ [stamp(\phi)])$.

□

Corollary 3.2. *LabelUnroll will never return an empty sequence.*

Proposition 3.3. *LabelUnroll will always terminate for all well-defined identifiers.*

Proof. $LabelUnroll(x)$ will not terminate if and only if the the $parents(x)$ has a cyclic connection back to x. Hence, it suffice to prove that there is not cycles in the parents relation. This can be proven by induction. Assume there are no cycles in the directed graph defined by edges $\bigcup_{u \in V} \{\ (u, p) \mid p \in parents(u)\ \}$, that is the edges define a Directed Acyclic Graph (DAG). Identifiers generated as a consequence of inline operations add new nodes to the existing identifier's graph, maintaining the graph's acyclic-ness. Since the base case is an identifier with no outgoing edges, hence a DAG, the proposition is inductively proven. □

Proposition 3.4. *LabelInline satisfies property 2.*

Proof. In the proof of proposition 3.3, it has been shown that the $\mathcal{I}$ data structure forms a DAG. The DAG formed by nodes curated via *LabelInline* points strictly towards parents. Hence, the $\forall \phi.LabelUnroll(\phi)$ is dependent only on the itself and parent identifiers and hence, independent of its sibling nodes and descendants. □

Proposition 3.5. *LabelInline satisfies property 3.*

Proof. First, we define identifier equality as $\forall x \in \mathcal{I}.\forall y \in \mathcal{I}.x = y \leftrightarrow LabelUnroll(x) = LabelUnroll(y)$, with the latter equality following conventional list equality semantics. The proposition can then be proven inductively.

Consider two identifiers $id(a_x)$ and $id(a_y)$ with common prefixes. This denotes that they originated from a common sequence of inline operations. The inductive assumption states that $prefix(id(a_x)) = prefix(id(a_y)) \implies suffix(id(a_x)) \neq suffix(id(a_y))$, where for some suffix and prefix functions such that $LabelUnroll(id(a)) = concat(prefix(id(a)),\ suffix(id(a)))$.

Consider the case where $|Unroll(id(a_x))| = |Unroll(id(a_y))|$. By inlining some function declaration u into call site a_x, some $v_x^{(i)}$ is unveiled. Since $|Unroll(id(v_x^{(i)}))| < |Unroll(id(a_y))|$, therefore $id(a'_y) \neq id(v_x^{(i)})$. Now by merging u into a_y, $v_y^{(i)}$ is obtained where $\forall i.|v_x^{(i)}| = |v_y^{(i)}|$

For $Unroll(id(v_x^{(i)})) = Unroll(id(v_y^{(i)}))$, it is necessary for $suffix(id(a_x)) = suffix(id(a_y))$. The requirement contradicts with the initial assumption, and hence, the assumed argument is unsatisfiable. By contradiction, it is shown that if the uniqueness constraint hold $\forall a \in V$. □

As all the propositions and corollary presented above were shown inductively, a very useful result follows: It is possible to run this inline for an arbitrary number of times whilst retaining the properties in propositions 3.1, 3.3, 3.4 and 3.5. In other words, this labelling algorithm can be used across multiple inlining passes.

It is straightforward to observe that creating a single new identifier with *LabelInline* uses $O(1)$ memory, based on algorithm 1. Consequently, the memory requirement for storing the entire tree is $O(N_{Inline} \times E_v[|C(v)|])$. This is consistent with the earlier argument that such structural sharing will perform better than a naive string concatenation algorithm.

4 GLOBAL APPLICATION LABELS

Property 3 guarantees only local uniqueness between siblings. If the AST does not permit high-order functions (or does not inline first class functions), then the local labels suffice. However, for ASTs with high-order functions, the local application labels are not globally unique. A simple conflict example can be obtained by inlining a function declaration d that contains a nested function declaration d', in two distinct function call sites. Application nodes beneath both instances of d' will have similar local labels.

A globally unique label can be constructed by as a path-based identifier, namely $\mathcal{I}_{global} = [\mathcal{I}] \times \mathcal{I}$. The list of identifier contains the local labels of function declarations of nodes from the root of the AST leading to the node, whereas the latter is the local label of the node itself. A uniqueness proof for $\mathcal{I}_{global}$ can be constructed by drawing comparisons to file names in file-system trees.

5 EVALUATION

We have used the above methodology for labeling function applicaiton nodes in an AST to override compiler inlining decisions in the OCaml compiler, such that it takes customized inlining decisions based on the user input. Our customisation supports arbitrarily nested function declarations and applications. We speculate that this methodology can be used to for labelling in other forms of inlining, such as template inlining in C++ or type-class inlining in Haskell [9].

To measure the asymptotic behaviour of the algorithm, we have benchmarked this algorithm by recursively inlining a self-call within a function declaration in an AST with 100 nodes and a branching factor of 3[2]. We compared the proposed algorithm (using raw pointers and shared pointers) against the naive implementation described above to illustrate the importance of structual sharing. Both benchmarks are written in C++. The relative execution time and memory shown in tables 1 and 2 are obtained by dividing by the case with only a single inline operation. The memory usage is measured with valgrind [8] and the execution time with the bash time utility. The

[2]Source code for this benchmark is available at https://github.com/fyquah95/stable-tree-labeling

Inline Ops	*Raw Pointers*	*Shared Pointers*	*Naive*
1	1.000	1.000	1.000
10	1.011	1.019	1.016
100	1.111	1.203	1.715
1000	2.074	2.966	63.492
10000	12.759	22.532	6164.270
50000	57.100	103.724	153866.345

Table 1. Relative Memory Usage

Inline Ops	*Raw Pointers*	*Shared Pointers*	*Naive*
1	1	1	1
10	1	1	1
100	1	1	1
1000	1	1	2
10000	2	3	73
50000	6	12	1469

Table 2. Relative Execution Time

tables show that execution time and memory usage scales linearly with respect to in the proposed algorithm, whereas a naive algorithm scales quadratically.

REFERENCES

[1] Lennart Augustsson, Mikael Rittri, and Dan Synek. 1994. Functional Pearl: On generating unique names. *Journal of Functional Programming* 4, 1 (1994), 117fi?!123. https://doi.org/10.1017/S0956796800000988

[2] John Cocke. 1970. Programming languages and their compilers: Preliminary notes. (1970).

[3] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Vol. 93.

[4] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming* 12, 4-5 (2002), 393–434.

[5] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. 2003. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing* 24, 1 (2003), 43–67.

[6] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[7] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2017. *The OCaml system release 4.06: Documentation and user's manual.* Ph.D. Dissertation. Inria.

[8] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science* 89, 2 (2003), 44–66.

[9] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space, In Haskell workshop. https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/

[10] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* (2018).