Inlining ML with ML

Fu Yong Quah

Imperial College London

27 June 2018

Outline

Table of Contents

OCaml

- Functional programming language (like haskell, F#).
- Automatic memory management with garbage collection.
- Used for soft real-time systems, theorem provers and compilers.

Function Inlining

In principle, inlining is dead simple: just replace the call of a function by an instance of its body (Jones & Marlow 2002)

Before inlining `f`:

```
let f items y =
  List.map
    (fun x -> x + y)
    items
;;

f [ 1; 4; ] 37
```

After inlining `f`:

```
let f items y = ....

let _res =
  let items = [ 1; 4 ] in
  let y = 37 in
  List.map (fun x -> x + y)
    items
in
_res
```

Why Inline Functions

Function inlining can make programs faster ☺

- Remove function call overhead
- Remove value allocations, hence reducing garbage collection
- Remove call-by-pointer overhead
- Reduce code size via dead-code elimination
- Expose further optimisation opportunities

Why NOT TO Inline Functions

But ... function inlining can also degrade program performance ☹

- Significantly increase compilation time
- Hinder further optimisations
- Increase cache pressure
- Increased pressure for register allocator

Table of Contents

Objective

The objective of the project is to learn a policy that makes intelligent *static* inlining decisions, such that they make programs faster.

(An inlining policy decides whether to inline a function at a function call site.)

Problem Statement

Given a set of training benchmarks $\mathcal{B}_{train}$, learn an inlining policy $\pi : \mathcal{F} \rightarrow \mathcal{A}$.

where $\mathcal{A} = \{Inline, Apply\}$ and $\mathcal{F}$ is a function call site.

The learnt policy will be evaluated on the a set of unseen test benchmarks $\mathcal{B}_{test}$.

Table of Contents

What do we start with?

What do we start with?

- A set of training benchmarks $\mathcal{B}_{train}$

What do we want?

Generate data of benchmarks with different sets of inlining decisions and their corresponding execution times.

Representing Inlining Decisions

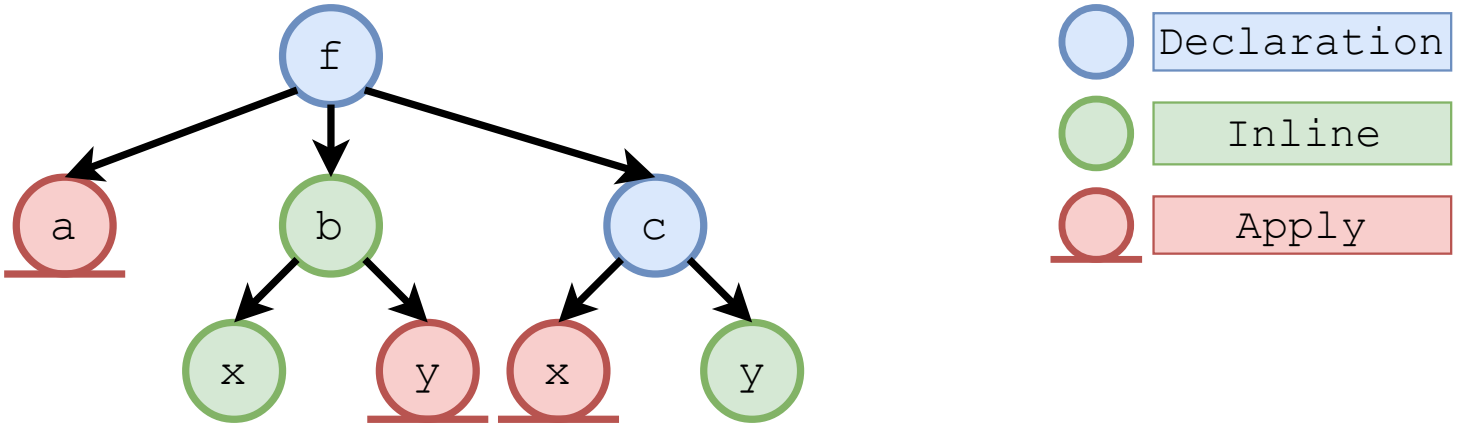
We require a data structure that:

- Maintains the hierarchical nature of inlining decisions
- Allows function declaration inside function call sites
- Allows arbitrary function calls at call sites

Inlining Tree

Inlining Tree

A tree-based representation of function declarations and inlining decisions in a program.

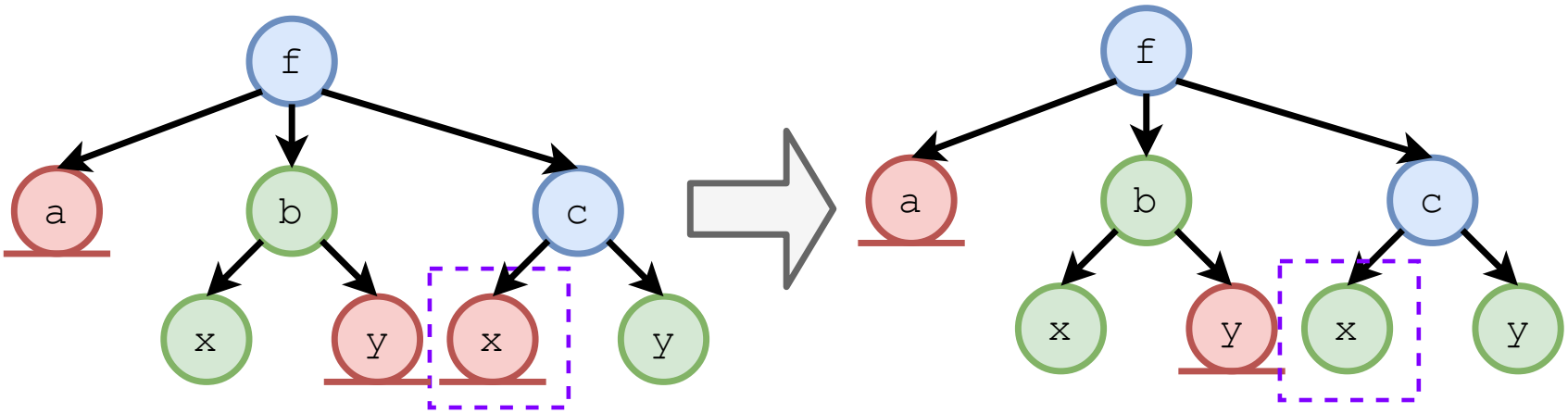


Iterative Compilation

For every benchmark, we generate pairs of inlining trees and execution times as training data.

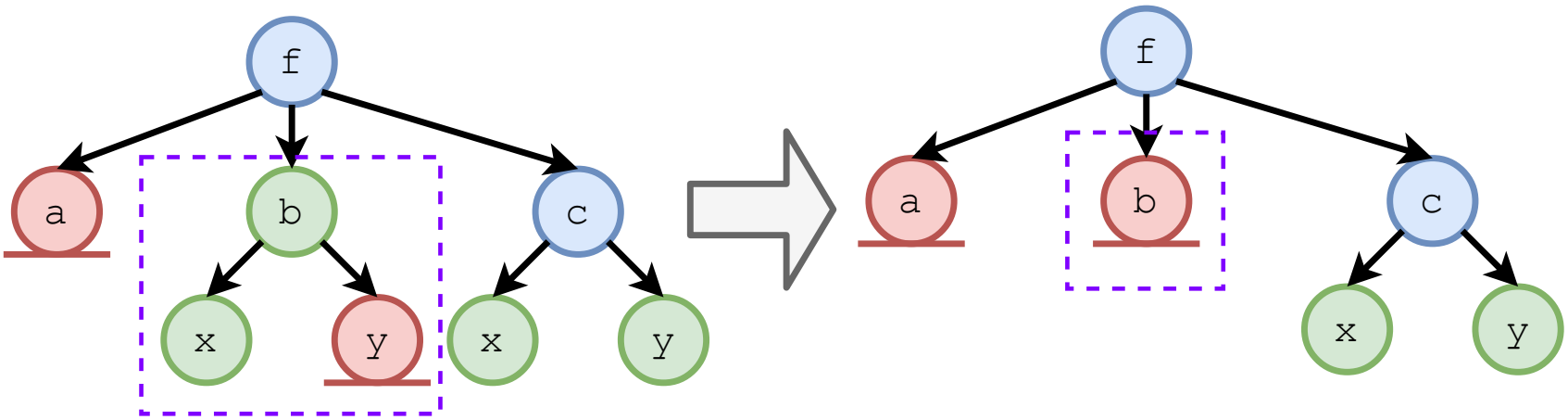
- Compile the same benchmark multiple times with different inlining decisions.
- Systematically explore the space of inlining trees using *simulated annealing* with tree perturbations.

Perturbation 1 : Flip a Node



Flip an inlining decisions from *Inline* to *Apply*, or vice versa.

Perturbation 2 : Backtrack an Inline Node



Flip a non-leaf *Inline* Node to an *Apply Node*

Table of Contents

What do we have now?

What do we have now?

- N executions of each benchmark
- Each execution has its own set of inlining decisions and execution time. The inlining decisions can be represented as $d : \mathcal{F} \rightarrow \mathcal{A}$
- Each benchmark has a function $\mathcal{C}(s)$ that yields the set of function call sites beneath inlining a function call.

What do we have now?

What do we have now?

- N executions of each benchmark
- Each execution has its own set of inlining decisions and execution time. The inlining decisions can be represented as $d : \mathcal{F} \to \mathcal{A}$
- Each benchmark has a function $\mathcal{C}(s)$ that yields the set of function call sites beneath inlining a function call.

What do we want?

Assign numerical rewards to inlining versus applying at a call site.

Tree-Value Formulation

How do we compute the value of an inlining tree?

Tree-Value Formulation

How do we compute the value of an inlining tree?

Use a linear combination of rewards across the entire tree!

Tree-Value Formulation

How do we compute the value of an inlining tree?

Use a linear combination of rewards across the entire tree!

Inlining (Sub-)Tree Value

Given d that records all inlining decisions taken in a compilation:

$$V_d(s) = \begin{cases} \mathcal{R}_{apply}(s) & d(s) = Apply \\ \mathcal{R}_{inline}(s) & d(s) = Inline \end{cases}$$

where s is a function call site

Tree-Value Formulation

How do we compute the value of an inlining tree?

Use a linear combination of rewards across the entire tree!

Inlining (Sub-)Tree Value

Given d that records all inlining decisions taken in a compilation:

$$V_d(s) = \begin{cases} \mathcal{R}_{apply}(s) & d(s) = Apply \\ \mathcal{R}_{inline}(s) + \gamma \sum_{s' \in \mathcal{C}(s)} V_d(s') & d(s) = Inline \end{cases}$$

where s is a function call site, $0 <= \gamma <= 1$ is a decay factor

Predictive Modelling via Demonstrations

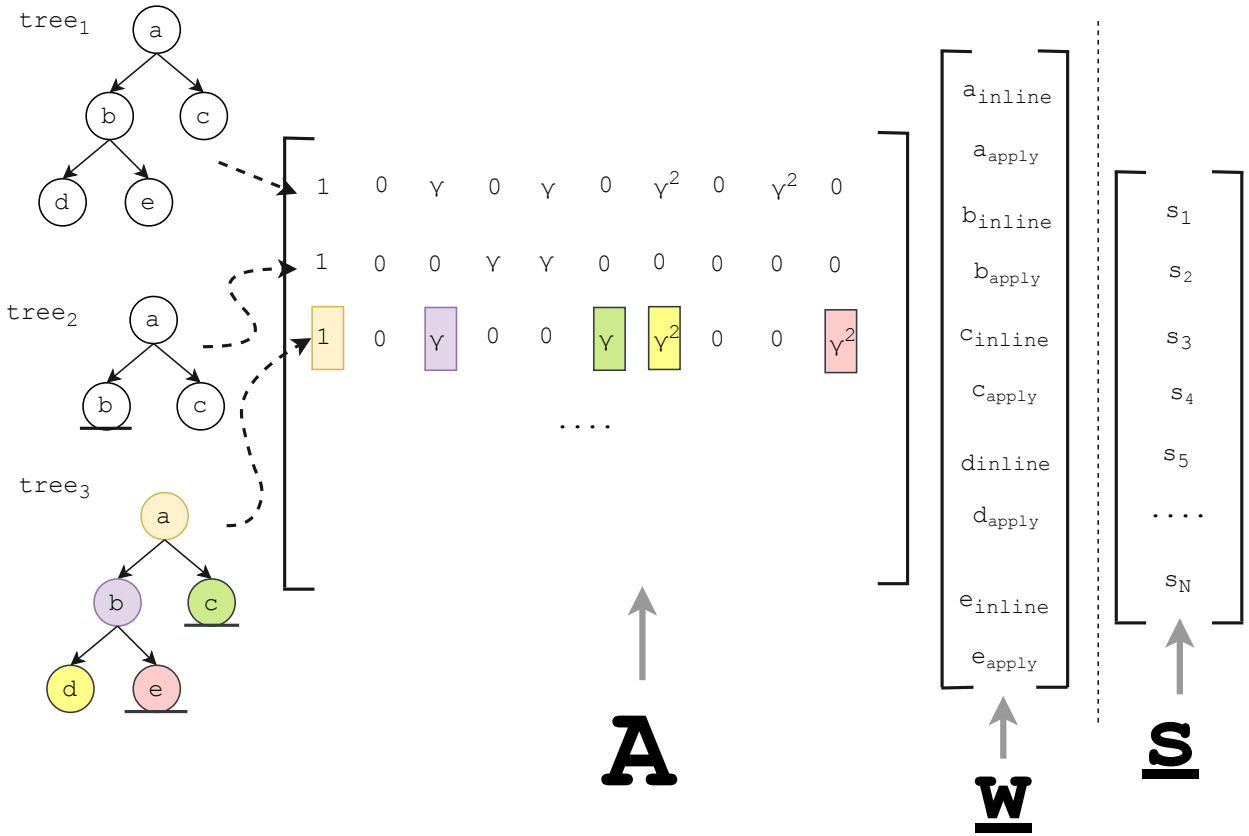
We can estimate $\mathcal{R}_{apply}$ and $\mathcal{R}_{inline}$ by using executions' data of a given program.

Predictive Modelling via Demonstrations

We can estimate $\mathcal{R}_{apply}$ and $\mathcal{R}_{inline}$ by using executions' data of a given program.

1. Use the acheived speedup over baseline as a target inlining tree value.
2. Express every execution of the program as a linear combination of call sites decision rewards throughout the inlining tree.
3. Solve $\mathcal{R}_{apply}$ and $\mathcal{R}_{inline}$ of all call sites by minimising Mean Squared Error (MSE).

Predictive Modelling via Demonstrations (cont.)



Express $\mathcal{R}_{apply}$ and $\mathcal{R}_{inline}$ of all call sites as a vector **w**.
Express speedup of all executions as a vector **s**.

Predictive Modelling via Demonstrations (cont.)

The prediction MSE can be expressed as $\frac{1}{N}||A\mathbf{w} - \mathbf{s}||_2$

Predictive Modelling via Demonstrations (cont.)

The prediction MSE can be expressed as $\frac{1}{N}||A\mathbf{w} - \mathbf{s}||_2$

Problem: We have found experimentally that
$rank(A) < min(|\mathcal{F}|, N)$

Predictive Modelling via Demonstrations (cont.)

The prediction MSE can be expressed as $\frac{1}{N}||A\mathbf{w} - \mathbf{s}||_2$

Problem: We have found experimentally that
$rank(A) < min(|\mathcal{F}|, N)$

Solution: Add regularisation to minimisation objective. We have applied L2 and L1 regularisation.

Optimal Sub-Tree Values

We define the optimal sub-tree inlining value, V^*_{inline} as the inlining tree value when we take optimal inlining decisions recursively beneath inlining a function call.

Optimal Sub-Tree Values

We define the optimal sub-tree inlining value, V^*_{inline} as the inlining tree value when we take optimal inlining decisions recursively beneath inlining a function call.

Optimal Inlining (Sub-)Tree Value

$$V^*_{inline}(s) = \mathcal{R}_{inline}(s) + \gamma \sum_{s' \in \mathcal{C}(s)} \left(max(\mathcal{R}_{apply}(s'), V^*_{inline}(s')) \right)$$

Optimal Sub-Tree Values

We define the optimal sub-tree inlining value, V^*_{inline} as the inlining tree value when we take optimal inlining decisions recursively beneath inlining a function call.

Optimal Inlining (Sub-)Tree Value

$$V^*_{inline}(s) = \mathcal{R}_{inline}(s) + \gamma \sum_{s' \in \mathcal{C}(s)} \left(max(\mathcal{R}_{apply}(s'), V^*_{inline}(s')) \right)$$

Better to inline if $V^*_{inline}(s) > \mathcal{R}_{apply}$, better to apply otherwise.

Table of Contents

What do we have now?

What do we have now:

- A set of function call sites $\mathcal{F}$
- Foreach call site, their apply rewards $\mathcal{R}_{apply} \in \mathbb{R}$ and their optimal sub-tree value $V^*_{inline} \in \mathbb{R}$

What do we have now?

What do we have now:

- A set of function call sites $\mathcal{F}$
- Foreach call site, their apply rewards $\mathcal{R}_{apply} \in \mathbb{R}$ and their optimal sub-tree value $V^*_{inline} \in \mathbb{R}$

What do we want?

Train a policy $\pi : \mathcal{F} \to \mathcal{A}$ to make inlining decisions at arbitrary function call sites.

Labeling Call Sites - Inline, Apply or Don't Know?

We can label the dataset of call sites using an uncertainty threshold τ and their reward assignments.

$$f(v;\tau) = \begin{cases} 0 & |x| < \tau \\ x & otherwise \end{cases}$$

$$y(v;\tau) = \begin{cases} DontKnow & f(V^*_{inline}(v);\tau) = f(\mathcal{R}_{apply}(v);\tau) \\ Inline & f(V^*_{inline}(v);\tau) > f(\mathcal{R}_{apply}(v);\tau) \\ Apply & f(V^*_{inline}(v);\tau) < f(\mathcal{R}_{apply}(v);\tau) \end{cases}$$

Training an Inlining Policy

The process for training an inlining policy:

Training an Inlining Policy

The process for training an inlining policy:

1. Choose a reward assignment (L1 or L2)
2. Choose an uncertainty threshold τ

Training an Inlining Policy

The process for training an inlining policy:

1. Choose a reward assignment (L1 or L2)
2. Choose an uncertainty threshold τ
3. Run feature extractions throughout dataset of call sites
4. Label dataset of call sites with target inlining decisions

Training an Inlining Policy

The process for training an inlining policy:

1. Choose a reward assignment (L1 or L2)
2. Choose an uncertainty threshold τ
3. Run feature extractions throughout dataset of call sites
4. Label dataset of call sites with target inlining decisions
5. Train a classification model with supervised learning
6. Evaluate the inlining policy on test benchmarks

Training an Inlining Policy

The process for training an inlining policy:

1. Choose a reward assignment (L1 or L2)
2. Choose an uncertainty threshold τ
3. Run feature extractions throughout dataset of call sites
4. Label dataset of call sites with target inlining decisions
5. Train a classification model with supervised learning
6. Evaluate the inlining policy on test benchmarks

Problem: It is unclear how exactly to choose reward assignment or τ. Training on all reward assignments yields many sets of inlining policies.

Decreasing Decisions Variance with Averaging

Instead of using one labelling model, why not use all models?

Decreasing Decisions Variance with Averaging

Instead of using one labelling model, why not use all models?

Sum up the probability distribution over output actions from every policy!

Decreasing Decisions Variance with Averaging

Instead of using one labelling model, why not use all models?

Sum up the probability distribution over output actions from every policy!

$$P_{averaging}(\hat{y} = y|\mathbf{x}) = \frac{1}{Z}\sum_{i \in P} P_i(\hat{y} = y|\mathbf{x}) \tag{4}$$

$$\pi_{averaging} = argmax_{a \in \mathcal{A}} P_{averaging}(\hat{y} = a|\mathbf{x}) \tag{5}$$

Decreasing Decisions Variance with Averaging

Instead of using one labelling model, why not use all models?

Sum up the probability distribution over output actions from every policy!

$$P_{averaging}(\hat{y} = y|\mathbf{x}) = \frac{1}{Z}\sum_{i \in P} P_i(\hat{y} = y|\mathbf{x}) \tag{4}$$

$$\pi_{averaging} = argmax_{a \in \mathcal{A}} P_{averaging}(\hat{y} = a|\mathbf{x}) \tag{5}$$

Actions will have high probability only if it they are overwhelmingly preferred amongst policies. Variance of decisions is much lower.

Table of Contents

Single-Compilation Policy Results

To evalute inlining policy in static scenarios, we simply compile the program with our inlining policy and record the obtained speedup.

This answers the project's objective, that is to develop a static inlining policy.

Single-Compilation Policy Results (cont.)

benchmark	Averaging Policy	Best Uni-Model Policy
qr-decomposition	32.756%	28.865%
sieve	18.194%	13.281%
naive-multilayer	17.693%	20.191%
valet-lwt	7.545%	8.314%
nullable-array	7.230%	8.250%
nucleic	6.444%	6.473%
boyer	5.332%	4.766%
list-benchmark	3.248%	6.069%
chameneos-redux-th	2.102%	1.830%
levinson-durbin	1.751%	-0.197%
format-benchmark	1.450%	6.351%
chameneos-redux-lwt	1.011%	-1.310%
chameneos-redux-evtchn	0.912%	-1.673%
sauvola	0.718%	1.086%
vector-functor	-0.006%	-0.474%
lu-decomposition	-0.239%	0.994%
valet-async	-4.681%	2.889%
chameneos-redux-async	-5.826%	-6.548%
fyq-rev-list (hand-crafted)	-737.989%	0.254%

Out of 18 test cases, averaging policy has median speedup of 1.93%. It observed an improvement in 12, less than 1% difference in 4 and pathological degradation in 2.

Table of Contents

Conclusion

Key research contributions of the project:

- Formally proven stable call site identification (not presented)
- Representing inlining decisions with inlining trees
- Call-site reward assignment
- Learning a policy from call site values

Conclusion

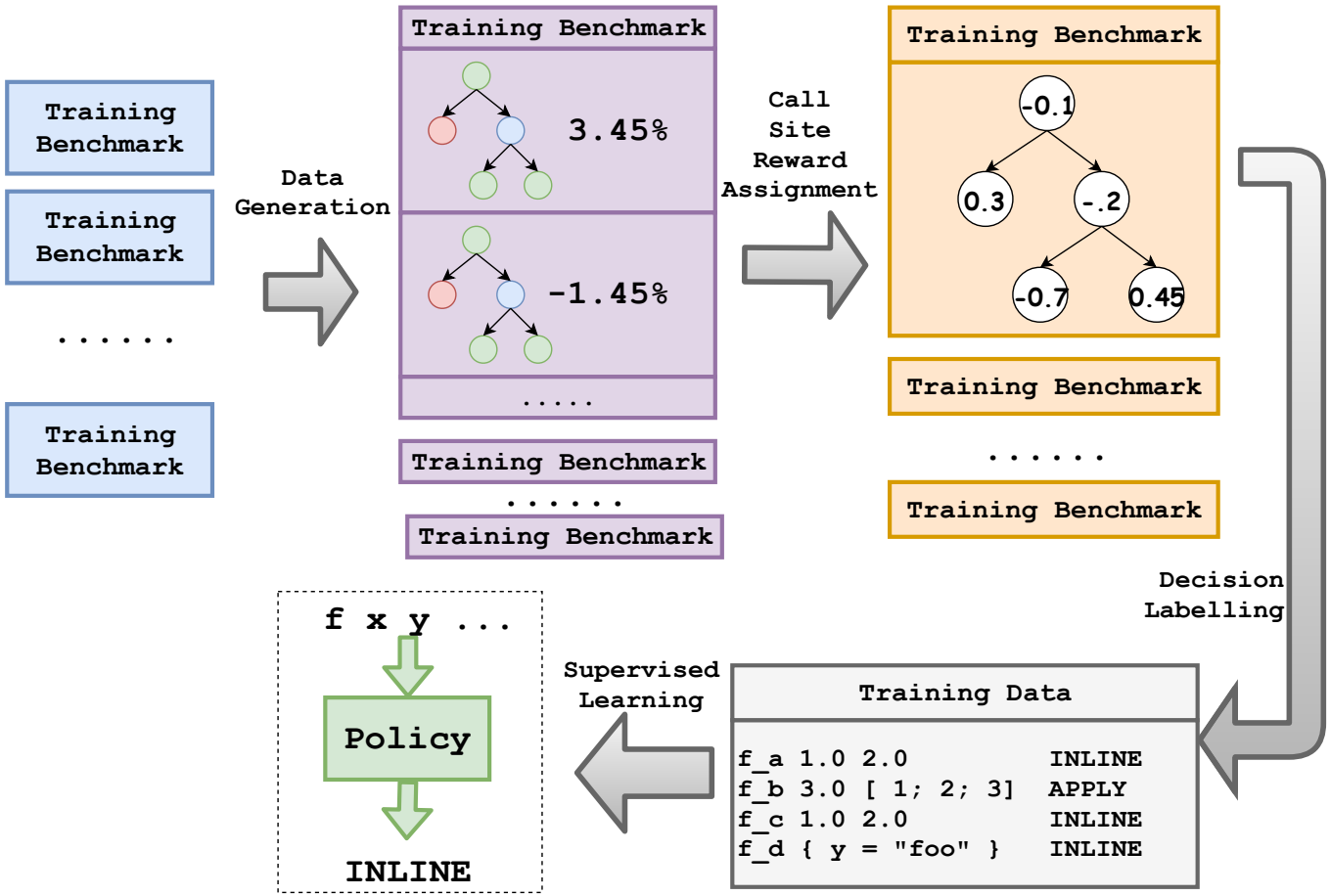
Key research contributions of the project:

- Formally proven stable call site identification (not presented)
- Representing inlining decisions with inlining trees
- Call-site reward assignment
- Learning a policy from call site values

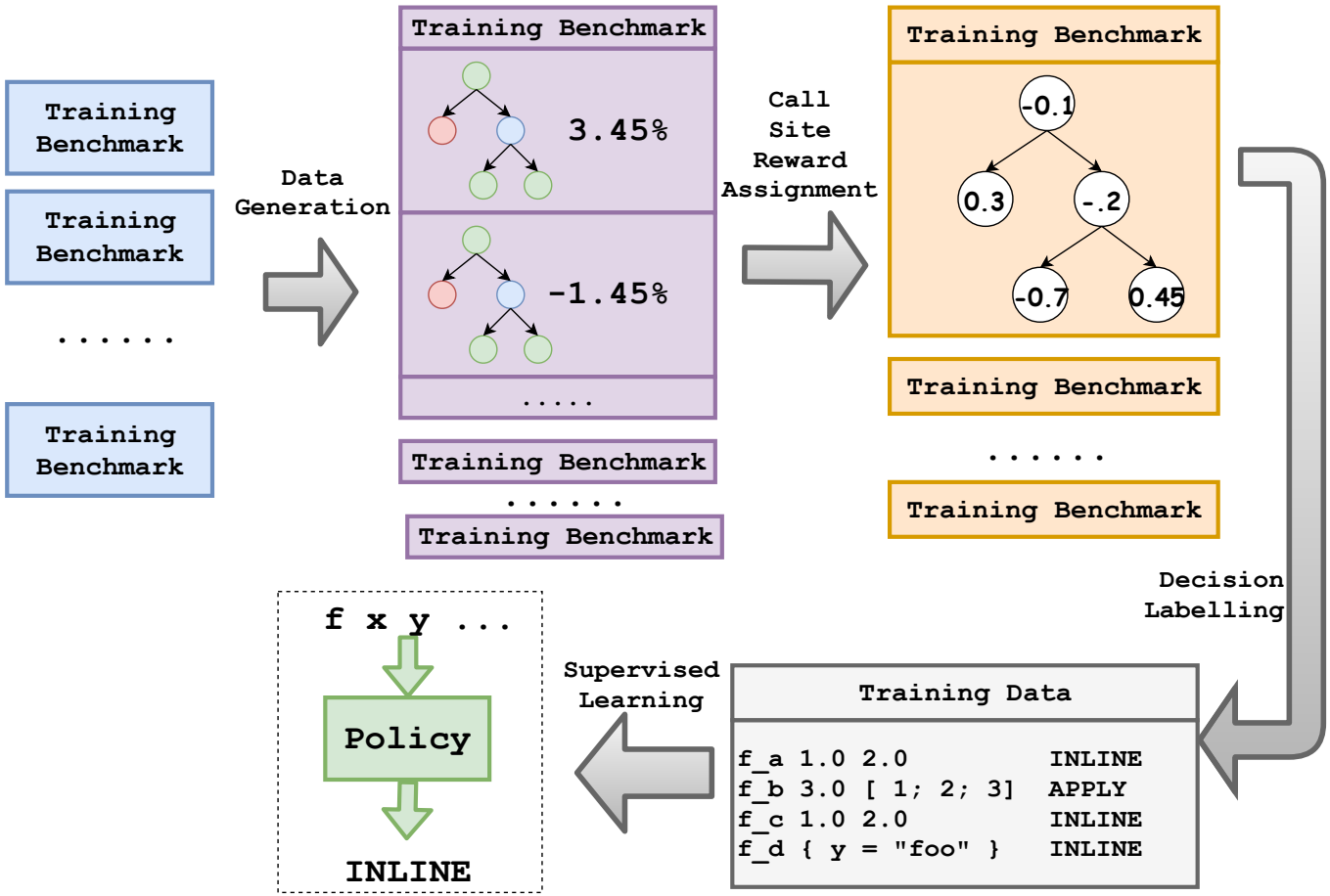
Future work of the project:

- upstream to the OCaml compiler
- use a more diverse set of training benchmarks
- fine-tune feature selection
- thorough evaluation on real-world benchmarks

Conclusion (cont.)



Conclusion (cont.)



Questions?

References I

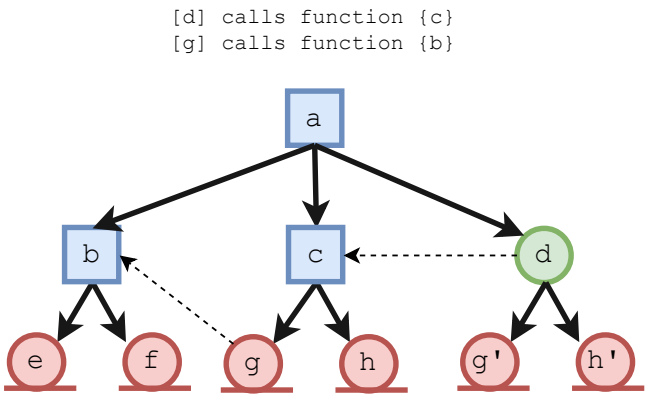
Jones, S. P. & Marlow, S. (2002), 'Secrets of the glasgow haskell compiler inliner', *Journal of Functional Programming* **12**(4-5), 393–434.

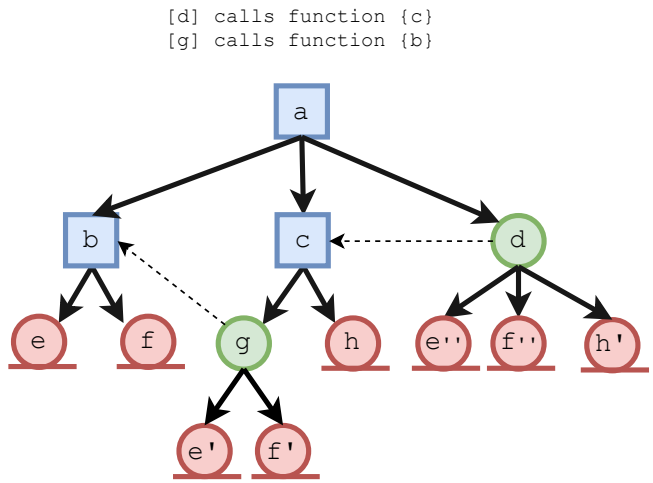
Type Declaration of Inlining Tree

```
type node =
  | Declaration of declaration
  | Apply_inlined_function of inlined_function
  | Apply_non_inlined_function of non_inlined_function
and declaration =
  { declared  : Function_metadata.t;
    children  : node list;
  }
and non_inlined_function =
  { applied   : Function_metadata.t;
    apply_id  : Apply_id.t;  (* refers to [id] in path labelling *)
  }
and inlined_function =
  { applied   : Function_metadata.t;
    offset    : Apply_id.t;  (* refers to [id] in path labelling *)
    children  : node list;
  }
```

Expanded Tree

Problem: The raw inlining tree does not have stable children. $\mathcal{C}(s)$ is not independent of inlining decisions of the declarations.





Expanded Tree (cont.)

Solution: artificially *expand* inlining decisions from the function declarations.

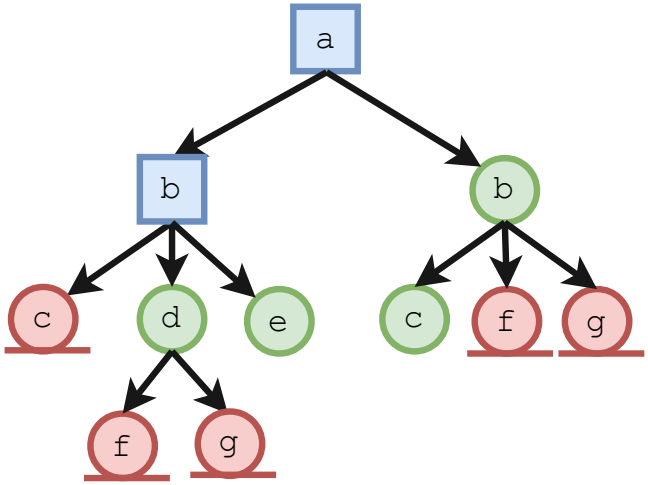


Figure 1: Original Inlining Tree

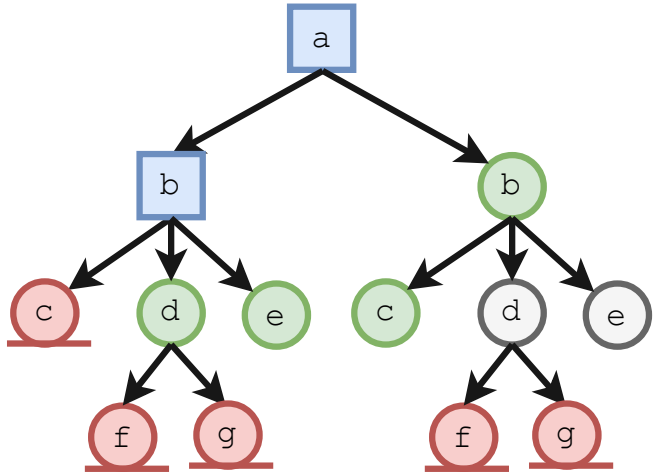
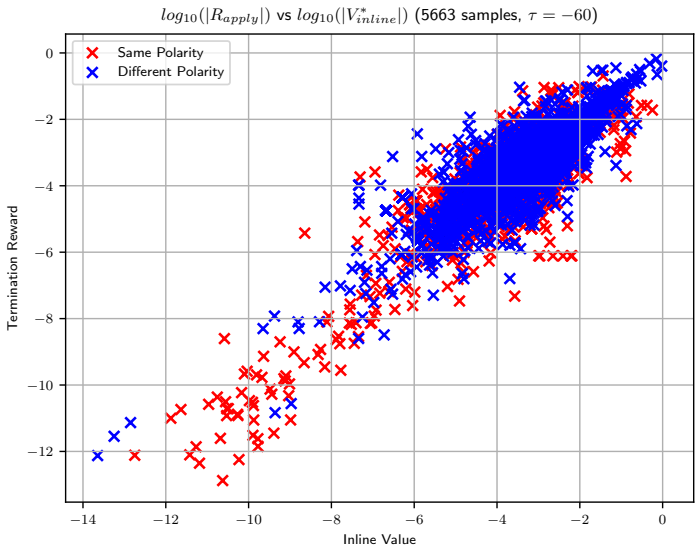
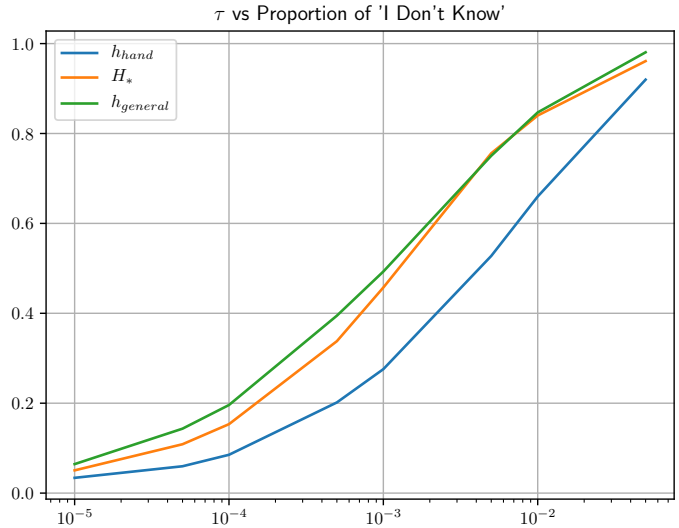


Figure 2: Expanded Inlining Tree

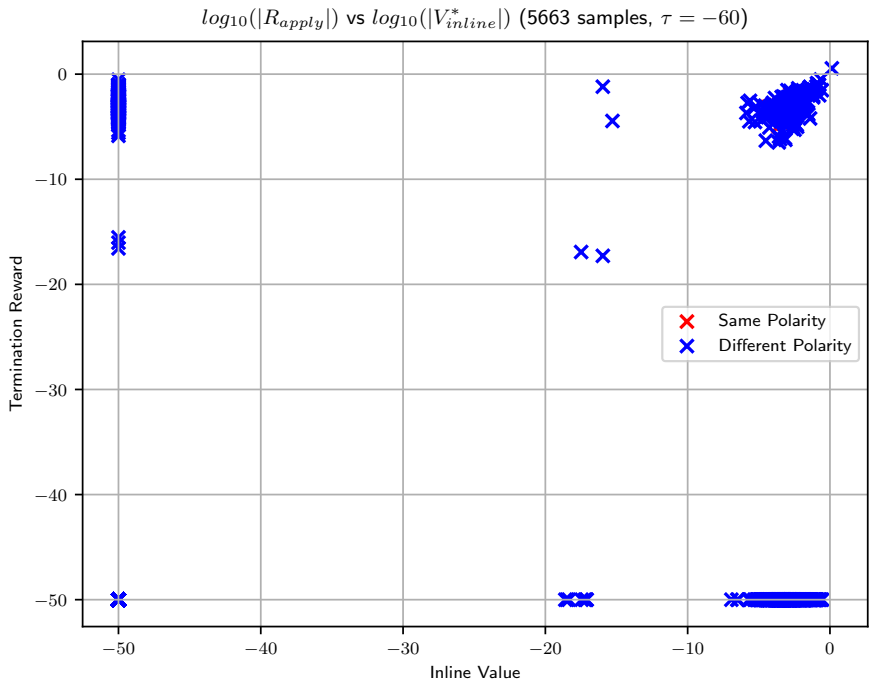
Selecting τ for L2 Rewards





No clear cut-off-point for τ. Proportion of "I Don't Know" examples increases logarithmically w.r.t λ. Try $\lambda = \{0.0005, 0.001, 0.005, 0.01, ..., \}$ in magnitude steps.

Selecting τ for L1 Rewards



Sparse rewards in L1. Select τ such that small values due to numerical errors are zero-ed out.

Is It Random? - Test Performance Relative to Random

We rank the averaging policy against 303 executions of benchmarks that inlines randomly.

- 10 out of 18 greater than median
- 1 out of 18 performs roughly close to the median
- 7 out of 18 performs worse (Both pathalogical cases falls in this category)
- Median Percentile across benchmark: 0.619
- Mean Percentile: 0.610
- Percentile variance: 0.06

The distribution of percentile over benchmark does not look normal, so we did not run t-test.

Is It Random? - Test Performance Relative to Random (cont.)

Benchmark	Speedup	Averaging Rank	δ_{median}	Min	25-th	Median	75-th	Max
nucleic	6.444%	0.983	5.749%	-5.228%	-0.751%	0.694%	2.173%	7.848%
sieve	18.194%	0.941	11.462%	-2.606%	0.781%	6.731%	16.461%	22.173%
qr-decomposition	32.756%	0.914	5.721%	-17.466%	2.434%	27.035%	31.437%	35.441%
levinson-durbin	1.751%	0.898	1.007%	-1.748%	-0.084%	0.744%	1.510%	2.919%
naive-multilayer	17.693%	0.888	12.101%	-9.886%	-2.611%	5.592%	9.392%	22.365%
boyer	5.332%	0.799	3.147%	-1.411%	1.407%	2.185%	4.548%	17.764%
chameneos-redux-evtchn	0.912%	0.729	0.556%	-4.409%	-0.586%	0.355%	0.993%	3.086%
vector-functor	-0.006%	0.686	0.582%	-1.438%	-0.961%	-0.588%	0.022%	0.247%
chameneos-redux-th	2.102%	0.650	0.356%	-2.125%	1.027%	1.746%	2.377%	4.196%
valet-lwt	7.545%	0.587	1.598%	-4.037%	0.821%	5.948%	8.533%	9.590%
chameneos-redux-lwt	1.011%	0.502	0.007%	-16.668%	-1.842%	1.004%	3.044%	7.615%
nullable-array	7.230%	0.482	-0.160%	-1.684%	4.676%	7.391%	10.273%	17.165%
sauvola	0.718%	0.403	-0.123%	-5.801%	0.464%	0.842%	1.070%	1.705%
valet-async	-4.681%	0.389	-4.098%	-8.850%	-5.875%	-0.582%	1.150%	3.680%
lu-decomposition	-0.239%	0.389	-0.306%	-8.228%	-0.876%	0.067%	0.745%	2.324%
list-benchmark	3.248%	0.376	-0.431%	0.049%	2.729%	3.679%	4.513%	7.317%
format-benchmark	1.450%	0.228	-0.853%	-3.716%	1.494%	2.303%	3.307%	6.565%
chameneos-redux-async	-5.826%	0.135	-2.945%	-13.883%	-4.734%	-2.880%	-1.031%	1.968%

Is It Random? - Training Performance Relative to Random

Benchmark	Speedup	Averaging Percentile	δ_{median}	Min	25-th	Median	75-th	Max
bdd	4.175%	1.000	6.151%	-10.772%	-5.377%	-1.976%	-0.966%	3.711%
floats-in-functor	50.864%	1.000	51.070%	-0.981%	-0.556%	-0.206%	31.888%	48.978%
lexifi	2.715%	0.937	2.760%	-7.108%	-1.185%	-0.045%	1.101%	4.267%
quicksort	0.853%	0.888	0.885%	-1.396%	-0.588%	-0.033%	0.474%	1.558%
kahan-sum	7.925%	0.842	5.526%	-6.096%	1.213%	2.399%	5.684%	19.014%
fft	1.992%	0.809	0.389%	-0.498%	1.286%	1.603%	1.898%	2.595%
almabench	0.692%	0.795	0.840%	-5.926%	-0.944%	-0.148%	0.551%	2.489%
lens	53.631%	0.789	54.411%	-3.026%	-1.236%	-0.780%	53.605%	54.090%
kb	3.439%	0.789	2.244%	-6.252%	-0.423%	1.195%	2.993%	8.425%
hamming	0.790%	0.726	1.196%	-3.551%	-1.457%	-0.406%	0.928%	4.497%
sequence-cps	-0.001%	0.442	-0.002%	-0.245%	-0.007%	0.000%	0.003%	0.007%
sequence	-0.541%	0.399	-0.521%	-8.474%	-2.197%	-0.020%	2.007%	12.317%

- 10 out of 12 greater than median
- 1 of the 2 cases has unnoticable improvement
- 2 training cases performs strictly better than random

Features

- **Approximation Features** - the number of different kinds of value approximation the compiler can perform in the IR.
- **Declaration Features** - features of function declaration disregarding the function body, such as number of parameters or whether it is a functor.
- **Argument Features** - one-hot encoding of the types of arguments passed to the function.
- **Environment Features** - information about surrounding context of function, such as inlining depth.
- **Flambda Heuristics Features** - features used by the default inlining heuristic.
- **Callee body features** - frequency of certain structures in the IR. They have nothing directly to do with inlining.

Multiple-Compilation Policy Results

To evaluate the efficiency of the policies in a scenario where the user can compile multiple copies:

1. Compile the same program using 9 different policies[1]
2. Benchmark the 9 generated binaries
3. Select the one that performs best

[1] 9 was chosen via greedy allocation based on $\mathcal{B}_{train}$.

Multiple-Compilation Policy Results

To evaluate the efficiency of the policies in a scenario where the user can compile multiple copies:

1. Compile the same program using 9 different policies[1]
2. Benchmark the 9 generated binaries
3. Select the one that performs best

This is compared against a grid search of command line arguments.

[1] 9 was chosen via greedy allocation based on $\mathcal{B}_{train}$.

Multiple-Compilation Policy Results (cont.)

benchmark	Grid Search	ML-based Inlining
qr-decomposition	2.526%	**34.960%**
naive-multilayer	7.859%	**19.102%**
sieve	0.084%	**17.397%**
nullable-array	9.497%	**13.120%**
nucleic	1.062%	**9.202%**
format-benchmark	5.060%	**6.902%**
chameneos-redux-th	3.509%	**6.902%**
chameneos-redux-lwt	3.924%	**6.303%**
chameneos-redux-evtchn	1.002%	**2.480%**
valet-lwt	8.476%	8.630%
valet-async	1.852%	2.597%
sauvola	1.060%	1.214%
chameneos-redux-async	1.071%	1.191%
lu-decomposition	0.000%	0.611%
vector-functor	0.013%	0.405%
boyer	**8.639%**	7.074%
list-benchmark	**6.593%**	5.459%
levinson-durbin	**5.140%**	2.871%

Out of 18 test cases:

- Performs better in 9
- Performs similarly in 6
- Performs worse in 3