Design and Implementation of Simple Inliner in SSLANG

Eric Feng, Emily Sillars
Columbia University
{ef2648,ems2331}@columbia.edu

ABSTRACT

Inlining describes the process of replacing an ocurrence of a name with its associated value. It is a key optimization that reveals further optimizing transformations during compilation for producing an efficient executable. Simon Peyton Jones and Simon Marlow's seminal work in *Secrets of the Glasgow Haskell Compiler Inliner* detail key insights and algorithms in developing a production inliner for GHC [3]. In this report, we describe our work in understanding, designing, and implementing a similar but more simplified version of their inliner for SSLANG. In particular, our inliner currently supports two of the three inlining phases detailed in the paper: *PreInlineUnconditionally* and *PostInlineUnconditionally*. We detail lessons learned and the future work necessary in producing a more capable and intelligent inliner.

1 REFERENCE INLINER

We model the design of our inliner based on the inlining phase of the *simplifier* in GHC. In this section, we describe key takeaways from *Secrets of the Glasgow Haskell Compiler Inliner* to give the background of our design and implementation.

1.1 Overview

Consider a definition `x = E`; inlining is the process of substituting the occurrence of `x` with E. Jones and Marlow identify three distinct transformations which they describe as inlining [3]:

> *Inlining itself* replaces an occurrence of a let-bound variable by (a copy of) the right-hand side of its definition
> *Dead code elimination* discards bindings that are no longer used; this usually occurs when all occurrences of a variable have been inlined.
> $\beta - reduction$ rewrites a lambda application `(\x->E) A` as `let {x = A} in E`

In GHC, inlining belongs to a larger transformation process known as the *simplifier*. This is because inlining often reveals opportunities for further code transformations that the remaining optimizers in the simplifier carry out [2]. Simplification alternates between a two-step process involving an *occurence analyzer* and the actual *simplifier* until either no transformations occur or the number of iterations exceeds 4. The overarching architecture of the GHC simplifier thus looks as follows [3]:

```
while something-happened && iterations < 4
do
    perform occurrence analysis
    simplify the result
end
```

1.2 Occurrence Analyzer

The occurrence analyzer is run before the simplifier and is a bottom-up pass of the program that annotates each binder with an indication of how it occurs. It informs the simplifier on how to approach the inlining decision of a variable. For a let expression like `let x = E in B`, we analyze how often x occurs in B and the context of its occurrence. Based on this information, we place the binder in one of the following categories:

- `LoopBreaker`: The binder is part of a group of mutually recursive definitions. We've decided never to inline this partiuclar member in order to prevent non-termination of the simplifier. A more elegant solution involving solving for strongly connected components and breaking links as detailed by Jones and Marlow remain future work [3].
- `Dead`: The binder is not used. In this case, we may discard the binder entirely and eliminate the dead code.
- `OnceSafe`: The binder occurs once and is not inside a lambda. It is safe for us to inline unconditionally.
- `MultiSafe`: The binder occurs at most once in distinct case branches, none of which are inside lambda expressions. It is safe for us to inline unconditionally.
- `OnceUnsafe`: The binder occurs once but is inside a lambda. Inlining will not duplicate code but may duplicate work
- `MultiUnSafe`: The binder may occur many times, including inside lambdas.

When we say that a binder is *safe* to be inlined, we mean that we can be assured of no code duplication or work duplication. To see why a binder that is marked `OnceUnsafe` (that is, a binder that occurs once but is inside a lambda) may give rise to work duplication, consider the following example [3]:

```
let x = foo 100
    f = \y -> x * y
in ...(f 3 ) .. (f 4) ...
```

Where foo is a computationally expensive function. If we inline `foo 100` in `f`, `foo 100` will be called every time `f` is. In this case, we would rather just calculate `foo 100` once rather than inline it and have it called each time `f` is.

1.3 Simplifier and the Three-Phase Inlining Strategy

At its core, the simplifier takes in a "substitution", a set of in-scope variables, an "in-expression", and a "context," and returns an "out-expression".

```
simplExpr :: Subst -> InScopeSet -> InExpr -> Context
↪ -> OutExpr
```

Where `InExpr` and `OutExpr` signify the in-expression and out-expression respectively. An in-expression is simply an expression that has not yet been processed by the simplifier (in other words, has not yet undergone consideration for being inlined). An out expression is an expression that has already been processed.

Before we dive into greater detail on the specifics of each argument, let us develop some intuition for the overall inlining strategy described in the paper. When the simplifier meets an in-expression, it decides whether to inline `x` in `B` at three distinct phases. Consider again the expression:

```
let x = E in B
```

- `PreInlineUnconditionally`: If `x` is marked `OnceSafe` during occurrence analysis, then it will inline `x` unconditionally in B. In this case, the simplifier does not attempt to simplify the binder's right-hand side, `E`, at the site of the definition. The substitution is extended by binding the following substitution: `x -> E` and we discard the binding `x` completely. We then simplify B using the extended substitution.
- `PostInlineUnconditionally`: If `PreInlineUnconditionally` failed on `x`, the simplifier tries to first simplify its right-hand side `E` to produce `E'`. Then, it will decide again whether to inline `x` unconditionally in B. It does so if:
  - x is not exported,
  - x is not a loop breaker, and
  - `E'` is just a literal, variable or trivial constructor application (all of which guarantee that work nor code will be duplicated).

  If `x` and `E'` pass the described criteria, then the substitution is extended with `x -> E'` and we again discard the binding `x` before simplifying B using the extended substitution.
- `CallSiteInline`: If the above two phases failed, the simplifier adds `x` to the in-scope and then proceeds to process `B`. (In this case, the substitution is not extended). Later, when the simplifier encounters an occurrence of `x`, it decides whether to inline `x` based on information stored in the in-scope set and a set of complex heuristics.

While it may seem that `PreInlineUnconditionally` and `PostInlineUnconditionally` could be combined into a single phase, they cannot due two issues: reliance on state and exponential blowup. It can be helpful to think of `PreInlineUnconditionally` as a substitution set *writer*, and `PostInlineUnconditionally` as a substitution set *reader-writer*. `PreInlineUnconditionally` does not need to read from the substitution set; it relies purely on static occurrence information to decide whether to write to the substituion set, never needing to even peek at the right hand side (RHS) of a let binding. `PostInlineUnconditionally`, on the other hand, needs more information to decide whether to write. The occurrence info of the binder is not enough, so it must simplify the RHS of a let-binding before making its decision. Simplifying requires actually inlining occurrences of variables, which means `PostInlineUnconditionally` relies on reading the substitution set accumulated from past recursive steps. It runs *after* inlining inside the RHS takes place, hence the name *post inline*. If we try to combine these two distinct phases, we either must *always* process the right hand side (RHS) of a let binding which leads to exponential blow up (we process the RHS once at the let binding and then again at each occurrence site), or *never* process the RHS, which leads to incorrect inlining (serious problems arise when we use `E` instead of `E'` to make decisions). For these reasons, the two phases must remain distinct.

1.3.1 *Substitution.*

The substitution that the aforementioned `SimplExpr` takes as an argument informs us of how we can substitute a binder with its expression, thereby instantiating inlining behavior.

```
type Subst = Map InVar SubstRng
data SubstRng = DoneEx OutExpr
              | SuspEx InExpr Subst
```

A substitution is a map whose domain consists of *in-variables* and whose range is either a (`DoneEx OutExpr`) or (`SuspEx InExpr Subst`). A `SuspEx` is used by `PreInlineUnconditionally`. This is because, for `PreInlineUnconditionally`, we do not examine the binder's right-hand side at all. Therefore, `E` remains unsimplified and an in-expression. The in-expression is paired with the existing substitution. We keep the substitution here because because it now holds the unprocessed right hand side of our binder `x`. As we will see, when we meet the occurrence of `x` in `B`, we use the substitution map to retrieve `E` for inlining. Once we replace `x` with `E`, we will at that point finally process the in-expression `E`. On the other hand, a `DoneEx` is used by `PostInlineUnconditionally` — recall in our previous section that in `PostInlineUnconditionally`, the expression on the binder `x`'s right-hand side is simplified to produce `E'`. Thus, `E'` is effectively a (simplified) out-expression. For this reason, we do not need to pair it with the existing substitution as with `SuspEx`.

1.3.2 *In-Scope Set.*

When a binder is absent from the current substitution set, the simplifier uses information from the in-scope set to help it decide whether to inline it. The in-scope set holds information about the binder's associated value (or "RHS" in the case of let-bindings).

```
type InScopeSet = Map OutVar Definition
data Definition = Unkown
                | BoundTo OutExpr OccInfo
                | NotAmong [DataCon]
```

`Unkown` means the binder is bound inside a lambda or appears in a case pattern. `BoundTo` means the binder comes from a let-expression; it holds a copy of the binder's RHS as an `OutVar` as well as its occurrence information. `NotAmong` records negative information about a binder's value within local settings for case arm elimination. For example,

```
case x of
  Red ->
  Blue ->
  Green ->
  DEFAULT -> ... case x of
                   Blue -> ...
                   Pink -> ...
                   DEFAULT -> ...
```

Inside the second case expression, x can be stored inside the in-scope set as `NotAmong [Red, Blue, Green]` which allows us to eliminate the `Blue` arm in the second case expression.

1.4 Example

Let's run the simplifier on an example SSLANG program.

```
main cin cout =
  let r = 25
  let q = r + 25
  r + q
```

First the occurrence analyzer runs, which finds r to be `MultiUnsafe` (since it appears more than once), and q to be `OnceSafe` (since it appears once, not inside a lambda).

`PreInlineUnconditionally` fails for r, but `PostInlineUnconditionally` passes, so r gets added to the substitution set. `PreInlineUnconditionally` passes for q, so it's added to the substitution set.

When the simplifier encounters the expression r+q, it recurses on the arguments to the addition operator, looking up r and q in the substitution set and inlining accordingly. The final output:

```
main cin cout =
  25 + (25 + 25)
```

For a more detailed step-by-step walkthrough of the simplifier, see the appendix.

1.5 Name Capture

"It is well known that any transformation-based compiler must be concerned about *name capture*" [3]. In the case of inliner transformations, name capture refers to the problem of correctly inlining with shadowed variables. Specifically, the rule is within lambda abstraction λx.M, one can only replace y with E provided that x is not free inside E. Otherwise, any x inside of λx.M must be renamed before inlining of y can occur. Consider the following example:

```
let y = a+b in
  let a = 7 in
    y+a
```

Here, the a in the first let is shadowed by the redefinition of a in the second let. We can rewrite this example using β-expansion:

```
(\y -> (\a -> y+a) 7)) (a+b)
```

We cannot replace y with a+b inside λa.y+a because a is free inside a+b. (Free refers to when a variable is not associated with a lambda [1], and a+b does not contain any lambda expression at all.)

The solution then is to rename all instances of a inside λa.y+a:

```
(\y -> (\s796 -> s796+a) 7)) (a+b)
```

which using β-reduction becomes:

```
let y = a+b in
  let s796 = 7 in
    y+s796
```

which performing *inlining itself* followed by *dead code elimination* yields:

```
let s796 = 7 in
(a+b) + s796
```

Jones and Marlow outline a fairly complex strategy for eliminating shadowed variables dubbed the "rapier". Since eliminating shadowed variables will be handled by a separate compiler pass, we will not cover the details here. Instead, we will assume moving forward that all names encountered by our simplifier are unique.

2 DESIGN AND IMPLEMENTATION

2.1 Overview

We added the inlining process `simplifyProgram` as a compiler pass `transformation` in `IR.hs`. Just like the design of the simplifier as shown in *Secrets of the Glasgow Haskell Compiler Inliner*, our inliner is split into two parts: the occurrence analyzer and the inliner itself. However, our inliner currently only supports the first two inlining conditions — namely `PreInlineUnconditionally` and `PostInlineUnconditionally`. At the moment, `CallSiteInline` and the heuristics involved in conditional inlining remain future work. Moreover, the simplifier currently only runs once (in the GHC, it is run multiple times until there are no more changes or the number of runs exceeds 4 as shown in section 1). We intend to add support for running multiple occurrences of the inliner in the future.

2.2 Occurrence Analyzer

The occurrence analyzer is the first procedure we run in `SimplifyProgram`. The categories we use to annotate binders are the same as those described in the paper. The actual GHC annotates binders in a similar, but more sophisticated fashion. They take into consideration of mutually recursive definitions which we do not currently support. Moreover, we do not currently support inlining match statements. We hold the occurrence information as a global map with keys `varId` and values `OccInfo`.

```
  data OccInfo = Dead
               | LoopBreaker
               | OnceSafe
               | MultiSafe
               | OnceUnsafe
               | MultiUnsafe
               | Never
```

In addition to what was described by Jones and Marlow, we have a temporary category `Never`. Since we are only supporting `PreInlineUnconditionally` and `PostInlineConditionally` currently, we mark binders that do not fall into either under a catch-all to denote those binders which would typically be considered under `CallSiteInline` but for the moment we will not consider for inlining.

The default value of a binder being added to the occurence analyzer is `Dead`. If we meet the binder again, and the binder is not within a lambda or a branch of a match statement, it is annotated `OnceSafe`. Otherwise, we annotate the binder as `Never`.

We begin the occurrence analyzer by running `swallowArgs` on the program definitions. It leverages unfoldLambda to unroll top-level functions' curried lambdas. The semantics of the program is preserved when we do this since a top-level function with $x - 1$ arguments that returns a lambda has the same type as that of a top-level function with x arguments that returns a non-lambda a expression. The occurrence information of the top-level function name and arguments are then added to the map. Next, we turn our attention to the body of the top-level function and annotate the binders in a top-down fashion. Since we unroll the initial lambdas, the root of the tree becomes the first non-lambda expression we meet. The implementation details depending on the specific IR node are as follows:

- **Let**: We map over the binders and their respective right-hand sides. The occurrence information of both is captured. We then recurse on the body of the let expression.
- **Variables**: The occurrence information of the variable is updated.
- **Lambdas**: First, we record that we are entering a layer of lambda. This information is necessary because the heuristics involved in terms of inlining expressions that are not literals or variables within lambdas are complicated and are handled by **CallSiteInline**[3]. Even though we do not handle **CallSiteInline** at the moment, we use this information to let our inliner know to not try and inline non-literal/variable expressions belonging in lambdas. If the lambda has a named binder, we also add its occurrence information. We then recurse on the body of the lambda before recording that we are exiting a layer of lambda.
- **Application**: We recurse on both the left-hand side and the right-hand side.
- **Match**: We record that we are entering a match expression in the same way as lambdas. The occurrence information of the scrutinee is added. We then recurse on the arms of the match expression.
- **Primitives**: The occurrence information of each argument to the primitive operation is recursively added.
- All other IR nodes: The expression is returned with no information added to the map.

2.3 Inliner

After running the occurrence analyzer, we feed its results into the inliner via calls to the `simplExpr` function. Our type signature mirrors the 1990 GHC's simplExpr function, except that the in-scope set and context are placeholder string types for the time being.

```
simplExpr :: Subst -> InScopeSet -> InExpr -> Context
↪  -> SimplFn OutExpr
```

As we were coding the inlining portion, we realized that emulating the layered substitution map described in the paper only introduced needless complexity. Since another compiler pass would ensure the `simplifyProgram` pass would only ever encounter unique binders (no shadowing), we had no need for a layered substitution map! We switched to using a flat substitution map, calling it subst and storing it as a field in our simplifier environment state monad; we plan to remove the unused Subst argument passed around in future work, as well as change the type of subst from `M.Map InVar SubstRng` to `M.Map InVar (I.Expr I.Type)`.

Given a top level definition, we simplify each expression using `simplExpr` in a top down fashion. Like the occurrence analyzer, the implementation details vary depending on the IR node.

The most interesting case of `simplExpr` naturally occurs when we meet let expression nodes. In this case, we map over all binders and their respective right hand sides looking for the binders' occurrence information in the global map that the occurrence analyzer filled. If the key does not exist (wild cards), then we recursively simplify its right hand side and return the wildcard with its simplified right hand side. If the variable is annotated as **Dead**, we remove the binding altogether (dead code elimination). If the binder is annotated as **OnceSafe**, then we perform **PreInlineUnconditionally**. We insert the binder with **SuspEx** `rhs` to the substitution map and remove the binder at the site of declaration. If the binder is in the global map but is not **Dead** or **OnceSafe**, we attempt **PostInlineUnconditionally**. In this case, we recursively simplify the expression of the binders' right hand side. If the simplified right hand side is a literal or variable, we insert the pair into the substitution and remove the binding. If it is neither, then we would default to the criteria fit for **CallSiteInline**. Since we do not currently handle this situation, we just return the binder and its right hand side, unmodified.

Eventually, recursive calls to `simplExpr` will reach lone variable nodes. This means that we have arrived at the occurrence site of a binder. On matching with a variable node, we perform a look up of the variable in the substitution map. If the variable exists in the map as **SuspEx**, we know the value associated with this variable in the map is unprocessed. Therefore we recurse on the value held by **SuspEx**, and then inline it. If the variable exists in the map as **DoneEx**, we know the variable's associated value has already been processed and we inline it right away. If the variable does not exist in the substitution map, it belongs in the category of **CallSiteInline** and we return the variable unchanged since we do not support it yet.

For all other forms of IR nodes, we recursively simplify the given expression. At the lowest level, in the catch all case (e.g literals), we return the expression unchanged.

3 FUTURE WORK

Our draft PR contains four working inlining examples in the regression-tests/tests folder. These test cases cover some simple **OnceSafe** and **MultiUnsafe** examples. A number of features remain to be added, namely:

- More extensive test cases on current functionality, especially tests covering lambda expressions for **OnceUnsafe** cases
- A final edge case for **PostInlineUnconditionally**, which addresses "trivial" constructor applications (referred to as the trivial-constructor-argument-invariant in the paper)
- Handling mutually recursive definitions and recursive data types (the **LoopBreaker** category)
- Extending the Occurrence Analyzer to identify categories other than **OnceUnsafe**
- Extending the Simplifier to handle **CallSiteInline**
- another thing

Jones and Marlow spend a substantial amount of time describing their algorthim for identifying LoopBreakers; this makes us optimistic that handling recursive data types and mutually recursive lets will be a trivial (if tedious) extension to the ocurrence analyzer. We expect implementing **CallSiteInline** to take up the bulk of the remaining work.

4 CONCLUSION

We present a simple inliner for SSLANG based on the Three Phase Inliner in *Secrets of the Glasgow Haskell Compiler Inliner*. Of the three phases, our inliner currently only supports

`PreInlineUnconditionally` and **`PostInlineUnconditionally`**. As the naming implies, these phases involve evaluating binders which are always worth inlining. In other words, inlining expressions where we are guaranteed no duplicate code or duplicate work will be done. The more complex **`CallSiteInline`** phase, which involves evaluating whether it is worth it to inline a particular expression based on its surrounding "context," remain future work. We intend to work on **`CallSiteInline`** next semester.

REFERENCES

[1] Susan B. Horwitz. 2022. Lambda Calculus (Part I). https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html

[2] Alexis King. 2022. Tweag: GHC Simplifier basics. https://www.youtube.com/watch?v=m_HX4hyOuog

[3] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12 (07 2002), 393–433. https://doi.org/10.1017/S0956796802004331

APPENDIX

(Continued on the next page in single column format)

Inlining Example Step by Step

```
main cin cout =
  let r = 25
  let q = r + 25
  r + q
```

```
(0) Occurrence Analyser runs, and determines
r is MultiUnsafe (because it occurs more than once)
q is OnceSafe


(1) Simplify (let r = 25 in ...)
simpleExpr {} {} (let r = 25 in ...) “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {}
InScopeSet: {}
InExpr: (let r = 25 in ...)
Context: “context”

The InExpr is a Let IR node, so let’s look up the binder’s occurrence information. . .
r is marked MultiUnsafe, so it FAILS preInlineUnconditionally.
What about postInlineUnconditionally? Let’s evaluate the RHS of binder r. . .
E’ <- simplExpr 25
Since 25 is just a literal, simplExpr will just return the value.
Then E’ is just 25, a literal, so r PASSES postInlineUnconditionally.
Let’s add r to our substitution set with value (DoneEx 25 {}), and remove the binder r.
Now that we have dealt with the binders, we recurse on the body of the let IR node!


(2) Simplify (let q = r + 25 in ...)
simpleExpr {(“r”, DoneEx 25 {})} {} (let q = r + 25 in ...) “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {(“r”, DoneEx 25 {})}
InScopeSet: {}
InExpr: (let q = r + 25 in ...)
Context: “context”

The InExpr is a Let IR node, so let’s look up the binder’s occurrence information. . .
q is marked OnceSafe, so it PASSES preInlineUnconditionally.
Let’s add q to our substitution set with value (SuspEx (r + 25) {}), and remove the binder q.
Now that we have dealt with the binders, we recurse on the body of the let IR node!


(3) Simplify r + q
simpleExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} (r + q) “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
InScopeSet: {}
InExpr: (r + q)
Context: “context”

The InExpr is a Prim IR node. Since the Prim IR node in the SSLANG compiler contains a list of arguments,
let’s map simpleExpr over each argument, then return a Prim IR node with processed arguments.
arg0 <- simplExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} r “context”
arg1 <- simplExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} q “context”
```

```
Based on recursive call 3.1, we know arg0 is 25
Based on recursive call 3.2, we know arg1 is (25 + 25)

So we return Prim [25, (25 + 25)]


(3.1) Simplify r
simpleExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} r “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
InScopeSet: {}
InExpr: r
Context: “context”

The InExpr is a Var IR node, so let’s look up the VarId in the substitution set:
sub  <- Lookup “r” {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
Based on our substitution set, sub is DoneEx 25 {}.
DoneEx means the expression 25 has already been processed, so let’s return literal 25.


(3.2) Simplify q
simpleExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} r “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
InScopeSet: {}
InExpr: q
Context: “context”

The InExpr is a Var IR node, so let’s look up the VarId in the substitution set:
sub  <- Lookup “q” {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
Based on our substitution set, sub is SuspEx (r + 25) {}.
SuspEx means the expression (r + 25) has not yet been processed, so let’s recurse on it.


(3.3) Simplify r + 25
simpleExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} (r + 25) “context”
simpleExpr :: Subst -> InScopeSet -> InExpr -> Context -> OutExpr
Subst: {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})}
InScopeSet: {}
InExpr: (r + 25)
Context: “context”

The InExpr is a Prim IR node. Since the Prim IR node in the SSLANG compiler contains a list of arguments,
let’s map simpleExpr over each argument, then return a Prim IR node with processed arguments.
arg0 <- simplExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} r “context”
arg1 <- simplExpr {(“q”, SuspEx (r + 25) {}), (“r”, DoneEx 25 {})} {} 25 “context”

Based on recursive call 3.4 (which is identical to 3.1), we know arg0 is 25
Since 25 is just a literal, simplExpr will just return the value (arg1 will get literal 25).

So we return Prim [25, 25]

Result after inlining:
```

```
main cin cout =
  25 + (25 + 25)
```