Extending SSLANG's Simplifier Pass

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

ABSTRACT

Static inlining and match arm elimination are both powerful optimization techniques that can improve the performance of a programming language. In our previous report, we outlined our process of designing and implementing an initial simple inliner for SSLANG. We spent a large portion of the previous semester emulating the GHC's inliner as described in Jones' and Marlowe's *Secrets of the Glasgow Haskell Compiler Inliner* [1]. This semester, after adding polish to and resolving some subtle bugs in our initial inliner, we depart from this reference implementation to specialize our optimizations for SSLANG. Our approach deviates from the reference implementation to cater to the unique characteristics of SSLANG which, unlike Haskell, includes side effects. Additionally, we incorporate match arm elimination into our overarching "Simplifier" pass, which previously only included the static inliner. By revealing and exploiting the connections between these techniques, we enable the removal of multiple layers of match statements and uncover additional opportunities for optimization.

1 OVERVIEW

To address the challenges posed by side effects in SSLANG, we implemented a function that discerns when side effects may occur within a given context. This feature helps prevent unwanted inlining in scenarios where preserving the order of operations is crucial to maintaining program correctness.

Additionally, we extend the inlining pass to handle the inlining of scrutinees of match statements in SSLANG. From doing so, we were able to introduce match arm elimination, a technique that removes redundant match arms during the inlining process. By analyzing the match expressions and associated patterns, we identify cases where certain match arms are guaranteed to be unreachable or have no effect on program behavior. We can then completely rewrite these redundant match expressions as let expressions. This optimization not only reduces code size but also eliminates unnecessary branching and scrutinee evaluation, leading to more efficient execution.

By repeatedly applying these techniques, we can unveil multiple layers of redundant match statements, exposing opportunities for further optimization. In this sense, the interplay between the inliner and match arm elimination process in the Simplifier provides a mutually beneficial relationship.

2 ADDRESSING SIDE EFFECTS

Our initial inliner implementation, despite passing all regression tests, incorrectly ignored SSLANG side effects! Consider the example program below.

E6901: Projects In Computer Science, ,
2023.

2.1 Motivating Example

```
// print single digit number; account for ascii offset
putd cout_ c =
  after 1, cout_ <- (c + 48)
  wait cout_

f r =
    r <- deref r + 1
    deref r

main cin cout =
    let r = new 0
    let x = f r  // 1
    let y = f r  // 2
    putd cout (x + y + y) // 1 + 2 + 2
```

Occurrence analysis would mark `r` as **`Multiunsafe`**, `x` as **`OnceSafe`**, and `y` as **`Multiunsafe`**. Then the simplifier's **`PreInlineUnconditionally`** phase goes ahead and inlines `x`, producing:

```
main (cin : _t46) (cout : (& Int32)) -> () =
  let r = new 0
  let y = f r
  putd cout (f r + y + y) // 2 + 1 + 1
```

After inlining, the semantics of the original program has changed; 4 is printed instead of 5! This is a serious flaw in our initial inliner's implementation. The problem stems from the right-hand side of `r`, `f r`. Let's take a step back for a moment.

In the Haskell compiler, if a binder `x` is marked **`Oncesafe`**, the Haskell simplifier can go ahead and inline `x` without looking at its right hand side. Whether `x` gets evaluated before or after the evaluation of `y` has no effect on program semantics, since all functions in Haskell are pure.

In contrast, the right hand side of the SSLANG binder `x` modifies is a function call that modifies a mutable object. The call to `f` here is impure, determining an exact evaluation order for `x` and `y`. Due to its reference types, the SSLANG simplifier cannot, in fact, unconditionally inline *anything*. It must *always* check the right-hand side of a binder before inlining.

2.2 isPure Predicate

We introduce the *isPure* predicate, which we use to inspect the right-hand side of binders marked **`Oncesafe`** before proceeding to inline. Any operation on a reference type we consider impure. We conservatively mark function applications as impure as well. While it is certainly possible that a function application might be pure, the analysis needed to determine whether a particular function is pure balloons quickly and is susceptible to cycles from mutually recursive functions. We choose to sidestep this complexity for now. Primitive

operations like arithmetic operations with only pure arguments (variables, literals) are considered pure. In the same vein, Data constructors with only pure arguments are also considered pure.

After enforcing use of the *isPure* predicate, the simplifier no longer inlines `x`, as it recognizes the right-hand side as an impure function application.

3 MATCH ELIMINATION

Inlining the scrutinee of a match expression exposes new opportunities for further inlining. Consider the following SSLANG program.

3.1 Motivating Example

```
type List
  Cons Int List
  Nil

putd cout_ c =
  after 1, cout_ <- (c + 48)
  wait cout_

main cin cout =
   let test = Cons 4 (Cons 3 (Cons 9 Nil))
   match test
       Cons _ (Cons x (Cons y _)) = putd cout (x+y)
       Cons _ g = putd cout 7
       _ = putd cout 6
```

The occurrence analyzer marks `test` as **`Oncesafe`**. Next, the simplifier examines the right-hand side of binder `test` with the *isPure* predicate. Since the right-hand side is a data constructor containing only other data constructors, literals, and variables, *isPure* returns true, and the Simplifier proceeds to inline `test`.

```
main cin cout =
   match Cons 4 (Cons 3 (Cons 9 Nil))
       Cons _ (Cons x (Cons y _)) = putd cout (x+y)
       Cons _ g = putd cout 7
       _ = putd cout 6
```

After inlining, the match statement inside main becomes redundant; a let expression suffices:

```
main cin cout =
   let x = 3
   let y = 9
   putd cout (x+y)
```

Eliminating the match expression not only reduces code size but also execution time. Evaluating scrutinees in SSLANG (with the exception of integer scrutinees) requires checking the tag of a boxed object, which is essentially a pointer dereference. Rewriting this match as a let expression eliminates an unnecessary read from the heap.

This example program is taken from inlining-ex-16.ssl, a new regression test we added to our PR after implementing match elimination. Once our updates to the SSLANG simplifier are merged into main, this program will indeed reduce to a nested let expression.

Notice that if the simplifier runs twice, this program would reduce even further to

```
main cin cout =
   putd cout (3+9)
```

Running the simplifier multiple times until a program remains unchanged is an exciting next step for SSLANG optimization!

4 FUTURE WORK

Plentiful opportunities exist in enhancing the simplifier pass further.

- Enhance Side Effect Analysis: While our approach incorporates a function to discern side effects in SSLANG, there is area for further improvements in accurately identifying and analyzing side effects. In particular, we can extend our *isPure* predicate by analyzing the bodies of functions where function application occurs.
- Enhance Static Inlining: The current inliner is still relatively conservative with consideration to code/work duplication. We only allow the inlining of binders in two situations: if the binder is considered **`OnceSafe`** (which are binders that occur once, not inside a lambda) or if binder occurs multiple times (**`MultiUnsafe`**), *and* the right-hand side of this binder reduces to a literal, variable or trivial data constructor application. Future work could be done to analyze the context of **`MultiUnsafe`** binders whose right-hand sides are non-trivial. The decision to inline could be made more fine grained in these situations by computing a cost function based on a binder's context, essentially deciding to inline on a case-by-case basis. This strategy is known as *CallSiteInline* in *Secrets of the Glasgow Haskell Compiler Inliner* [1] . However, additional considerations would need to be taken relative to the implementation in the GHC due to the existence of side effects in SSLANG.
- Incorporate further optimizations: Static inlining and match arm elimination are just two optimization techniques among many. Future work could explore the integration of the simplifier with other optimization strategies.
- Integration with other transformations: Future work could explore the repeated application of our simplifier module with the other IR transformations to reveal further optimizations found in other modules as well.

5 CONCLUSION

We present an extended simplifier for SSLANG which includes an enhanced approach to static inlining relative to the inliner in the previous simplifier and the introduction of match arm elimination. Our modified simplifier pass (in a PR ready for review here) introduces a function that is able to conservatively discern side effects. Additionally, we are now able to inline binders that are scrutinees of match statements, opening avenues for match-arm elimination. By applying inlining and match arm elimination repeatedly, multiple layers of match statements can be removed, leading to more concise and efficient code.

REFERENCES

[1] 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