1 Introduction

This document provides more detail on optimizing the SimpleAST IR. The shrinking optimization (Section 3) is a required part of Project 3; the other two optimizations are optional and may be implemented for extra credit.

2 Analysis

Program optimization typically involves an analysis part and a transformation part, where the analysis is used to determine when it is sound to perform the transformation.

For the simple optimizations described in this document, we use two kinds of analysis information: a census of variable use and application counts, and simple dataflow information.

2.1 Variable Use and Application Counts

For each variable in the program, we track the number of use occurrences of the variable. This information is kept in a mutable reference in the variable’s representation and the SimpleVar structure provides a collection of functions for manipulating the value. The Census structure provided in the sample code computes the initial counts for variables.

To support the contractive inlining optimization (see Section 4 below), we also track application occurrences of variables. These are supported by the Census structure, but can be ignored when implementing the basic shrinking optimization.

2.2 Dataflow

We use a trivial dataflow analysis based on the lexical scoping of variables. If a variable is bound to a value in a let or function binding, then the value flows to the use occurrences of the variable. As we saw with binding analysis for type checking, we can use a finite map as an environment for tracking this flow information.\(^1\)

\(^1\)Note that because every bound variable in the SimpleAST is unique, a variable will never have a use occurrence outside its scope.
3 Shrinking

The shrinking (or contraction) optimization makes the program smaller (and hopefully more efficient) by performing a number of simple transformations. This optimization is particularly useful as a clean-up pass for other optimizations (and for the simplification pass), so we typically run it after each optimization phase.

A shrinking pass is implemented as a sequence of passes over the tree until we no longer have any changes. We need multiple passes because reductions done in one pass may enable further reductions in subsequent passes. Typically, however, shrinking converges to a fixed point in two or three passes.

In the remainder of this section, we describe the various shrinking transformations that are performed during a shrinking pass.

3.1 Let Floating

The purpose of let floating is to de-nest let bindings in a way that expands the scope of variables, which provides more opportunities for optimization. The basic let-floating transformations are

\[
\text{let } x = \text{let } y = \text{rhs } \text{in } \text{exp}_1 \text{ in } \text{exp}_2 \implies \text{let } y = \text{rhs } \text{in } \text{let } x = \text{exp}_1 \text{ in } \text{exp}_2
\]

\[
\text{let } x = \text{fun } f \text{ param } = \text{exp}_1 \text{ in } \text{exp}_2 \text{ in } \text{exp}_3 \implies \text{fun } f \text{ param } = \text{exp}_1 \text{ in } \text{let } x = \text{exp}_2 \text{ in } \text{exp}_3
\]

The best approach to handling let floating is to first do a let-floating pass prior to any shrinking passes. With care, this pass can be done in a single walk over the expression. In addition, it is useful to look for local let-floating opportunities when building the result expression for a let binding where the right-hand-side is an expression.

3.2 Dead Code Elimination

When we have an expression of the form \(\text{let } \overline{x} = \text{rhs } \text{in } \text{exp}\) and the variables in \(\overline{x}\) are unused (i.e., their use counts are zero), then we may be able to delete the \(\text{rhs}\) code. We need to be careful, however, to avoid deleting code that has side effects. You will need to implement functions for checking if SimpleAST expressions and right-hand-sides are pure or not. Such a function must be conservative, since it is, in general, undecidable if an expression has a side effect. The simplifying assumption is to treat function applications as impure, but most other code is pure (there are predicates provided in the Prim and Runtime structures for testing the purity of primitive operators and runtime-system functions).

3.3 Constant Folding

The sample code in the Contract structure defines the following representation of static values that a variable might be bound to:

\[
\text{datatype bind}
\begin{align*}
\ &= \text{Tuple of S.value list} \\
\ &= \text{Value of S.value} \\
\ &= \text{ConApp of DC.t * S.value}
\end{align*}
\]
The `Func` can be ignored, unless you are implementing contractive inlining (Section 4) for extra credit.

Constant folding works by looking up the binding information for a variable when it is used in a context where there is a potential for compile-time evaluation. For example, if we have

```plaintext
let x = {a, b, c} in
...
let y = #1(x) in
...
```

Then when processing the binding of `y`, we would look up `x` in the binding environment and get back `Tuple[a, b, c]`. This, we can evaluate the selection operation at compile time and bind `y` to `b` when processing its scope. Assuming that every occurrence of `y` is replaced by `b`, we can then remove the binding for `b` from the program.

Your optimizer should perform the following kinds of constant folding optimizations:

- Evaluation of primitive operators when applied to compile-time constants.
- Selection from known tuples.
- Conditional tests of known values.
- Case expressions of known arguments.
- Constant propagation; *i.e.*, if a variable is bound to a SimpleAST value, then we can replace occurrences of it with the value.

It is worth expanding on the compile-time evaluation of primitive operators. Integers in `LangF` are represented as 2's complement 63-bit numbers, which means that they must be in the range $-2^{62} \ldots 2^{62} - 1$. The sample code contains function `sNarrow` in the `Contract` module that will map a value outside this range to its 63-bit representation. You should use this function to make sure that the result of compile-time arithmetic is representable as a 63-bit number. You should also be careful not to try to evaluate expressions like `1/0` or `1%0`, since they are undefined (a runtime error will be generated when these are evaluated). Lastly, you should not try to evaluate the dereference operator, since doing so correctly requires a bit more analysis than we are doing.

### 3.4 Traversal Order and Bookkeeping

The structure for optimizing a let binding `let \pi = rhs in exp` follows a common form.

1. We check if the \pi are all unused (*a.k.a.* dead); if that is true and the `rhs` is pure, then we can eliminate the binding, decrement the use counts of any variables in the `rhs` and return the result of shrinking `exp`.
2. If the \pi are not unused, then we check to see if we can shrink the `rhs` (*e.g.*, by performing constant arithmetic). If that is possible, then we can bind the \pi to the resulting value(s), decrement the use counts of the `rhs` and return the result of shrinking `exp`.
3. Otherwise we record the binding of the $x_1, \ldots, x_n$ in the environment and proceed to shrink $\text{exp}$.

4. After shrinking $\text{exp}$ to some $\text{exp}'$, we check again for dead variables, since the shrinking of $\text{exp}$ might have eliminated the uses of the $\overline{x}$. If the variables are now dead (and the $\text{rhs}$ is pure), we return $\text{exp}'$. Otherwise, we return $\text{let} \overline{x} = \text{rhs in} \ \text{exp}'$.

Note that it is important to check for dead variables on both the downward and upward parts of the traversal in order to ensure quick convergence of the optimization.

4 Contractive Inlining

Function inlining is a transformation that applies a function at compile time. In general, inlining can cause the size of the program to grow, since a function may be inlined at multiple call sites, but there is the special case of functions that are only called once in the program. We call this restricted form of inlining contractive inlining (and the general form is called expansive inlining), since it shrinks the program.

Contractive inlining fits naturally into the shrinking pass. When we encounter the definition of a function $f$, where both the use and application counts of $f$ are equal to one, then we know that $f$ has only a single application occurrence. In that case, we enter the function binding into the binding environment and then process its scope.\(^2\) When we encounter an application occurrence of the function $f$ to values $\text{vs}$, and $f$ that maps to $\text{Func(xs, exp)}$ in the binding environment, then we want to replace the application with $\text{exp}$, where we have substituted the values $\text{vs}$ for the parameters $\text{xs}$. Fortunately, the binding environment that we use for constant folding is sufficient for this purpose.

There are a couple of technical complications that one should be aware of. First, if an argument to the application of $f$ is a variable (call it $x$), then we need to adjust its use and application counts, since it is going to be substituted for the corresponding parameter (call it $y$) in $\text{exp}$. The adjustments are as follows:

\[
\begin{align*}
\text{useCount}(x) &= \text{useCount}(x) + \text{useCount}(y) - 1 \\
\text{appCount}(x) &= \text{appCount}(x) + \text{appCount}(y)
\end{align*}
\]

(We subtract one from the use count because the use of $x$ in the application of $f$ is being eliminated).

The other issue is self-recursive functions. Consider the following SimpleAST code, for example:

```plaintext
fun f (x) = let t = IntAdd(x, 1) in f (t) in exp
```

Assuming that there are no occurrences of $f$ in $\text{exp}$, then $\text{useCount}(f) = \text{appCount}(f) = 1$ and we enter the information for $f$ into the environment when processing $\text{exp}$. When we finish processing $\text{exp}$, the application and use counts of $f$ will still be one, so we will know that the application of $f$ must be inside the body of $f$. With that information, we can delete the definition of $f$ from the program.

\(^2\)Note that we are not handling curried functions with this optimization.
5 Uncurrying

As we will see in the next project, curried function application is relatively expensive, since each stage of the application requires a function application and return. We can avoid this cost in many cases by replacing curried applications with applications of multiple arguments.

To illustrate the basic concepts of the uncurrying optimization, we consider the following code fragment:

fun f () (x) (y) = ... in
let a = f () 12 in
let a = g1 12 in
let b = f () 17 42 in
let h2 = f () in
let b = h3 42 in

In this example, the curried function f has two call sites; one that applies it to two arguments and one that applies it to three. Notice that there are no variables in the first parameter, a case that arises from type abstraction in the original source code. The first step is to replace the definition of f with an uncurried version, plus partially curried versions.

fun f3 (x, y) = ... in
fun f2 (x) (y) = f3 (x, y) in
fun f () (x) (y) = f3 (x, y) in

Having performed this transformation, we now have a version of f specialized to each of the possible number of arguments it might get. We use these to specialize the call sites in our example, which results in the final code:

fun f3 (x, y) = ... in
fun f2 (x) (y) = f3 (x, y) in
fun f () (x) (y) = f3 (x, y) in
let a = f2 (12)
let b = f3 (17, 42) in

Running a shrinking pass after uncurrying will remove any unused function definitions.

To implement uncurrying, we need to track those functions for which we have available specialized applications and then identify call sites where the staged applications are occurring. It is also necessary to correctly manage the census counts so that the subsequent shrinking pass will correctly clean up an unused function definitions.

One other issue to be aware of is the challenges of recursive functions. For example, if f from above is recursive, we might have something like the following:

fun f () (x) (y) = ... let g1 = f () in let g2 = g1 u in g2 v in ...

In this case, since the recursive call supplies all three arguments to f, we can replace the call with a
call to $f_3$ in the transformed program.

$$\text{fun } f_3 \ (x, \ y) = \ldots \ f_3 \ (u, \ v) \ldots$$

If, however, the recursive mentions of $f$ do not supply all of the arguments, things are more complicated.\(^3\) One solution would be include a copy of $f$ in the body of $f_3$, which could then be used for partial applications inside $f_3$.

$$\text{fun } f_3 \ (x, \ y, \ z) = \text{fun } f \ () \ (x) \ (y) = f_3(x, \ y) \ \text{in} \ldots$$

It would also be reasonable to just not try to optimize such situations, but if a curried function is recursive and all of its recursive calls supply all of the arguments, then we can replace the recursive calls.

6 Document History

**November 19, 2020** Added missing text about uncurrying for self-recursive functions.

**November 13, 2020** Added some discussion about constant folding to Section ??.

**November 11, 2020** Original version.

---

\(^3\)In part, this complication is because [LangF](#) does not support mutually recursive functions.