Compiler Construction

Mini Project

C Static-link Optimisation

The aim of this mini project is to implement a small optimisation in a supplied code generator.

Provided with the project distribution is a compiler `impc` that consumes programs written in an simple untyped imperative language (to be further discussed below) and produces code for the Simple Stack Machine. In particular, the source language is block-structured and has support for nested function declarations; in the generated code access to global variables is implemented by means of traversing a chain of static links. In this project we will optimise one aspect of the usage of these static links.

Syntax

Starting from a countable infinite set of identifier symbols, a set of constants (including the integers and the boolean constants `false` and `true`), and a set of binary operators,

$$\begin{array}{rcll} f, x & \in & \mathbf{Ident} & \text{identifiers} \\ c & \in & \mathbf{Const} & \text{constants} \\ \oplus & \in & \mathbf{Op} & \text{binary operators,} \end{array}$$

the syntax of the language under consideration is comprised from declarations, statements, and expressions,

$$\begin{array}{rcll} d & \in & \mathbf{Decl} & \text{declarations} \\ S & \in & \mathbf{Stmt} & \text{statements} \\ e & \in & \mathbf{Exp} & \text{expressions} \end{array}$$

defined by

$$\begin{array}{rcl} d & ::= & \mathbf{var}\ x;\ \mid\ \mathbf{function}\ f(x_1, \cdots, x_n)\{S_1 \cdots S_n\} \\ S & ::= & ;\ \mid\ d\ \mid\ x = e;\ \mid\ f(e_1, \cdots, e_2);\ \mid\ \mathbf{print}\ e;\ \mid\ \mathbf{return}\ e; \\ & \mid & \mathbf{if}\ (e)\ S_1\ \mathbf{else}\ S_2\ \mid\ \{S_1 \cdots S_n\} \\ e & ::= & c\ \mid\ x\ \mid\ f(e_1, \cdots, e_n)\ \mid\ e_1 \oplus e_2. \end{array}$$

Programs,

$$P \in \mathbf{Prog} \qquad \text{programs,}$$

are constructed from a list of top-level declarations,

$$P ::= d_1 \cdots d_n,$$

one of which should introduce a nullary function *main*, which then behaves as the entry point of the program.

Semantics

We will not provide a formal semantics for the language as its behaviour is completely standard and, apart from being untyped, consistent with the procedural parts of modern object-oriented languages such as Java. Moreover, the meaning of programs can easily be observed by compiling them into SSM-code and running the resulting code with the SSM-simulator.

Static-link Usage

In the generated code local identifiers are accessed via the mark pointer. In the code for a function body, global identifiers (i.e., identifiers that are neither formal parameters of the function nor bound by a variable declaration within the function body) are accessed by first following a chain of static links and then using the mark pointer thereby obtained.

Consider, for example, the following program, which uses Euclid's algorithm to compute the greatest common divisor of 42 and 56 while maintaining a global variable *count* which keeps track of the number of times the nested function *loop* is called:

```
var count;

function gcd(x, y){
    function loop (){
        count = count + 1;

        if (y ≡ 0) ;
        else if (x > y) { x = x − y; loop(); }
            else        { y = y − x; loop(); }
    }

    if (x ≡ 0) return y; else { loop(); return x; }
}

function main(){
    var r; r = gcd(42, 56); print count; print r;
    }
```

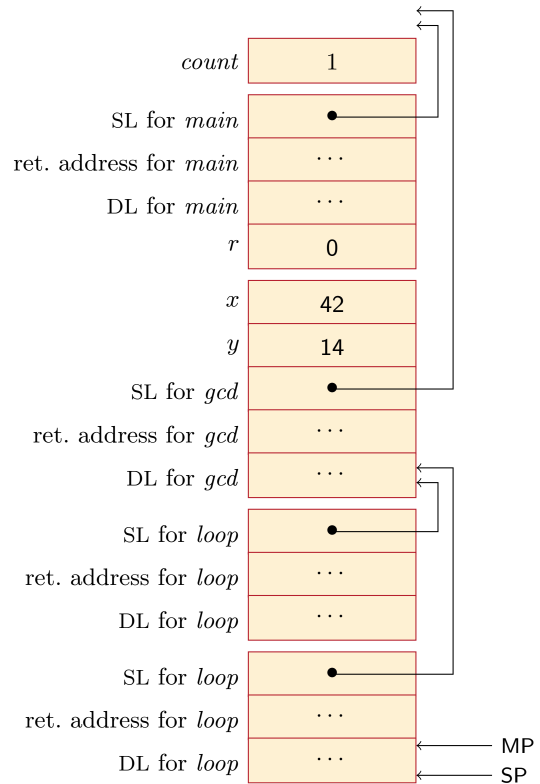
In the corresponding SSM-code, i.e., the code generated by the compiler from the project distribution, the code for loading the value of *count* from within the body of *loop* reads

```
ldl  -2
lda -2
lda 1.
```

Similarly, the code for assigning to *count* from within *loop* reads

```
ldl  -2
lda -2
sta 1.
```

That is, first the static link for *loop* (positioned at offset -2 relative to the location pointed at by the mark pointer) is pushed onto the stack and then this static link, targetting the address the mark pointer is pointing at during a particular invocation of *gcd*, is followed to obtain the address at which the mark pointer was pointing when execution of the program started. Finally, at offset 1 relative to this address, the memory cell containing the value for *count* is found. (The inset shows the layout of the stack just before the body of *loop* is run for the second time.)



Note that the same static-link traversal is thus performed twice: once for loading the value of *count* and once for assigning to *count*. In general, for subsequent accessses of any global variables declared at the same lexical level, identical static-link traversals are performed. A possible optimisation would be to perform such traversals once before the code of a function's body is executed and then to "cache" the end points of the traversals as local variables.

For example, assuming that the end point of the traversal for *count* is precomputed and stored at offset 1 relative to the stack location pointed at by the mark pointer, the code for loading *count* will, under this optimisation, read

```
ldl 1
lda 1.
```

What to Do?

Adapt the provided code generator so that, for each function, *all static-link traversals that are guaranteed to be performed more than once from within the body of the function* are performed before the code for the body is executed. Bind the thus obtained end points to "hidden" local variables and use them as demonstrated in the example above.

Do not pretraverse more static links than needed. For example, for the program shown above, from within the body of *main*, the address relative to which the global variable *count* can be accessed does not have to be stored as a hidden local variable as it is already readily available at offset -2 relative to the address the mark pointer is pointing at during the execution of *main*.

Plan of Attack

The following is a possible approach to implementing the required optimisation incrementally:

1. First, extend the code generator so that, for each function body, pretraversals are performed for all surrounding lexial levels, without paying attention to whether the obtained end points are actually needed from within the body.

2. Next, further adapt the generator so that the pretraversal for the closest surrounding lexical level is not cached. As mentioned, this end point of this traversal is already available at offset -2.

3. Then, make sure that from the remaining end points only the ones that are are guaranteed to be used from within the function body are precomputed.

4. Finally, exclude from precomputation those end points that are used only once from within the function body, for there is nothing to gain from caching these. (Even worse, the extra indirection would only make addressing relative to these end points more expensive.)

For the More Ambitious

In the rare case of very deeply nested functions, one could already use the cached traversal end point for the second closest surrounding level when looking up the end point for the third closest surrounding level etc. In determining whether an end point qualifies for precomputation, such additional uses should then be taken into account as well.

Submitting

The source code of your implementation should be handed in according to the submission instructions on the website of this course.

Submit the source code of your implementation (including both UUAG sources and Haskell sources *not* generated by the UUAG system).

Include in your submission a number of example programs that illustrate the capabilities of your implementation.