Static-link optimization

João Paulo Pizani Flor Liewe Thomas van Binsbergen

Wednesday, 27th March, 2013

1 Introduction

The third practical assignment of the master's course "Compiler Construction", given on the third academic period of 2012/2013 involved the generation and optimization of assembly code targeting a model stack machine, the Simple Stack Machine.

More specifically, we were given an implementation of a simple imperative programming language, and asked to perform an optimization (caching) that would avoid having to traverse a chain of *static links* everytime a global variable is to be accessed.

The imperative source language we had to compile allows for nested function definitions, and the *static link* is an element in the *stack frame* (or *incarnation record*) of a function that points to the stack frame of the immediately surrounding lexical scope. Here is an example of a program in the source language:

```
function main () {
    print f(2, 3) ;
}

function f(x, y) {
    var z;
    function g(a, b) {
        var c;
        function h (p, q) {
            var s;
            s = 6 + x;
            return x + y + z + a + c + p + s;
        }
        c = 5;
        return h (x, b);
    }
    z = 4;
    return g (x, z);
}
```

In this example, the deepest level of nesting (the `h` function) has 3 "outer" scopes which it can access. In the original implementation, the access to the `x` variable would require traversing the static-link chain upto the level of `f`, which uses 3 instructions.

The optimization which we implemented basically created – for each function stack frame – a "cache" that contains references to the frames of all scopes outside of it. We have taken measures to ensure that this "cache" is only created and used in cases in which it's useful: in particular, we don't use caching neither for variables in the immediate enclosing scope, nor for variables which are reference less than 2 times in the body of a function.

2 Attributes and data structures employed

In order to construct the static-link cache for each incarnation record, as well as to use the cache while getting and setting the values of variables, we needed to introduce three new attributes to the attribute grammar in `CodeGeneration.ag`.

The first attribute that we introduced – and that takes care of fullfilling subtasks 1, 2 and 3 of the assignment – is an inherited attribute that maps each identifier in the program to the scope in which it is declared. A scope identifier is just an integer from 0 to n, where 0 means that an identifier is in the local scope.

```
{
type ScopeId = Int  -- 0 upto n, where 0 is the innermost (local) scope
type SEnv    = [(Ident, ScopeId)]

identUsedEnough :: Ident -> VarRefs -> Bool
identUsedEnough i refs = M.findWithDefault 0 i refs >= minUsages
    where minUsages = 2
}

attr Decl Decls Stmt Stmts Exp Exps
    inh senv :: {SEnv}
    syn copy :: self

sem Prog
    | TopLevelDecls  ds.senv = [(i, 0) | VarDecl i <- @ds.copy]

sem Decl
    | FunDecl
        loc.allsenv =
            [(i, a+1) | (i, a) <- @lhs.senv] ++  -- parents
            [(i, 0)   | (i, _) <- @loc.params] ++  -- local parameters
            [(i, 0)   | Decl (VarDecl i) <- @b.copy]  -- local vardecls
        loc.senv = filter (flip identUsedEnough @b.colrefs . fst) @loc.allsenv
        b.senv   = @loc.allsenv
```

As can be seen in the UUAG code excerpt above, the `senv` attribute is "initialized" in the top-level declarations node by associating all declarations in the global level with scope level 0. Then, at each function declaration, the function's locally-declared identifiers are inserted with scope level 0, while the identifiers carried from the parent have their scope level incremented.

This `senv` attribute is then used in the functions that generate the assembly responsible for writing the static-link cache, but these functions will be discussed in section 3.

In order to attack subtask 4 of the assignment (*not cacheing* the links to variables used *only once* in the body of a function), we introduced a pair of attributes: `colrefs` and `varrefs`, respectively synthesized and inherited.

```
{
type VarRefs = M.Map Ident Int  -- variables referenced, with their refcount
}

attr Decl Decls Stmt Stmts Exp Exps
    syn colrefs use {M.unionWith (+)} {M.empty} :: {VarRefs}

attr Stmt Stmts Exp Exps
    inh varrefs :: {VarRefs}

sem Stmt | Decl lhs.colrefs = M.empty

sem Decl
    | FunDecl  b.varrefs = @b.colrefs

sem Exp
    | Var  lhs.colrefs = M.fromList [(@x, 1)]
```

Both attributes contain a mapping of all identifiers in a function body to the number of times they are referenced in the body. If an identifier is not present at all in the map, it is assumed that it is referenced *0 times* in the body of the function.

The idea is that attribute `colrefs` is propagated up the tree, from the expressions up until the function declaration node. Then we pass we down again as the `varrefs` inherited attribute. The `varrefs` attribute will then be used to determine whether the link to a function stack frame should or should not be included in the static-link cache.

3 Changes in the generated code

To comply with all the requirements established in the assignment, we needed to perform 4 changes in the way in which code is generated, incorporating the attributes previously discussed in section 2.

3.1 Change #1: Shifting offsets

The attribute `offsets`, present in the original grammar, carries a mapping from each identifier to the offset inside the stack frame in which it is declared. This attribute had to be extended, so that all offsets were incremented by n, where n is the number of elements in the static-link cache of the function.

```
sem Decl
```

```
    | VarDecl  lhs.offset = @lhs.offset + 1
    | FunDecl
        b.offset =
            let unique = nub $ filter ((> 1) . snd) @loc.senv
            in 1 + length unique

sem Stmt
    | Block  lhs.offset = @lhs.offset
```

3.2 Change #2: Inserting assembly code to write the cache

Having in hand the mapping from each identifier to the scope level in which it is declared, the next step is to effectively generate the assembly code which writes the static-link cache to the stack frame of a function.

```
storecache :: SEnv -> CodeS
storecache senv = storecache_ (filter (> 1) scopes)
    where scopes = nub (map snd senv)

storecache_ :: [ScopeId] -> CodeS
storecache_ []         = id
storecache_ (i:is) = ldl (-2) . lda (-2) . storecache__ is

storecache__ :: [ScopeId] -> CodeS
storecache__ []     = id
storecache__ (i:is) = lds 0 . lda (-2) . storecache__ is
```

3.3 Change #3: Accessing variables (possibly) using the cache

In the original code, there were two functions (`get` and `set`) that were used, respectively, the obtain the value of a variable and to assign to it. These functions were very similar, thus they were refactored into a single parameterized function, called `access`. Then we proceeded to change `access`: Depending on the number of times that a variable is used in the function body, the access can be done using the static-link cache or by traversing the whole static-link chain.

```
get :: VarRefs -> Ident -> Syms -> CodeS
get = access ldl lda

set :: VarRefs -> Ident -> Syms -> CodeS
set = access stl sta

type OffsetInst = Int -> CodeS

access :: OffsetInst -> OffsetInst -> VarRefs -> Ident -> Syms -> CodeS
access close_i far_i refs x (local : globals) = case lookup x (vars local) of
    Just (V offset) -> close_i offset
    Nothing         ->
        if identUsedEnough x refs
            then accessCache  far_i 1 x globals
            else ldl (-2) . accessGlobal far_i x globals
```

```
accessCache :: OffsetInst -> Int -> Ident -> Syms -> CodeS
accessCache _    _     x []           = error ("unknown variable: " ++ x)
accessCache inst level x (env : envs) = case lookup x (vars env) of
    Just (V offset) -> ldl slcacheidx . inst offset
    Nothing         -> accessCache inst (level+1) x envs
    where slcacheidx = if level == 1 then -2 else (level-1)

accessGlobal :: OffsetInst -> Ident -> Syms -> CodeS
accessGlobal inst x []           = error ("unknown variable: " ++ x)
accessGlobal inst x (env : envs) = case lookup x (vars env) of
    Nothing         -> lda (- 2) . accessGlobal inst x envs
    Just (V offset) -> lda offset
```

3.4 Change #4: Returning