Zach Cora & Austin Moore

P423 Compilers

Final Project

5/3/2016

Dead Code Elimination & Additional Language Features

For our final project, Zach and I chose to focus on the titular topics above. The reasons for this choice were:

1. The compiler felt somewhat empty when lacking some arithmetic features.
2. Dead code inflated program size proportionally to the point at which it was introduced.

Because of these reasons and the given time allotment, we felt that this was a good choice of development. It also allowed our final project to be modular; that is, new goals could be added in the form of other methods of dead code elimination or even more supported language syntax.  
 We began our project focusing on the removal of dead code. The primary focus at this point was the uncover-live pass. We chose to work in this pass because it already has the information about which variables are used in the future. Our logic was quite simple: if an instruction creates a value that is not in the live-after set, it is not used, and therefore the entire instruction can be removed. This works for any instruction without side effects. We discovered problems with vectors not being instantiated correctly, or perhaps modified correctly, when we tried to eliminate moves into a vector. Besides uncover-live, we also created a new pass that removes unnecessary consecutive projects/injects. For example, if a project is applied directly after an inject, both are removed. This lowers the amount of type checking that happens at runtime in exchange for dramatically shorter resulting code. We do this before flatten, as it is easiest to detect the consecutive projects/injects while it is not in a list form.

After our successful attempts at the elimination of useless code in our output programs, we shifted our attention to adding language support for several other syntax forms. Our primary goals were to have multiplication, division, cons cells, and lists. We began with the implementation of multiplication as it bore much resemblance to addition in the x86 assembly language.

Multiplication in x86 is quite straightforward. It utilizes the ‘imul’ operation, which multiplies a source datum by a destination register. The source can be an immediate, register, or offset location. Note that unlike add, imul requires the destination of the calculation to be a register. Because of the resemblance, this operation was quite easily carried through the compiler in a similar fashion. The main difference is that in patch-instructions we required the program to move the right hand side to a register and then perform the imul operation if the right hand side was not a register location.

Division, on the other hand, is not as simple. The ‘idiv’ operation has several caveats that must be taken care of when implementing signed division. The first of which is the necessity of the ‘cdq’ operation. This must occur before an idiv instruction, as it combines the rax and rdx registers. This is due to the behavior of the idiv instruction: the result answer is placed into rax and the remainder is stored in rdx. All of these complications exist alongside the requirement of the numerator being a positive number. In order to support divisions with negative numerators, our compiler requires the output program to check the value of the numerator against zero, and if it is – negq the numerator, apply the idiv instruction and then return to negq the resulting value. Of course, this division still expects integer input and output, so the result should be expected to truncate.

Our next focus came to be exponentiation. Our expt function only supports non-negative exponents, as our compiler deals singularly with integer values. Another possible way of handling negative exponents is to simply return zero. In a truncated-integer world, all bases with a negative exponent will be less than 1, and thus are zero. We decided to stick to the former – simply not supporting negative exponents. This case is different than the work we did to support the negative numerators in idiv because previously idiv simply returned an incorrect answer, which was a much more serious issue.

Now onto how exactly we generate expt code: because the more elegant and efficient way of calculating exponentiation requires float values, we chose to implement a recursive version using imul. Select-instructions sends a cmpq and an unconditional jump down through the passes, which checks the exponent and comapres with 0. If we have reached zero, we jump to the end. Otherwise, we multiply by the base (into the value 1 stored in the destination) and continue after subtracting one from the exponent. This straightforward recursive loop eventually retrieves the result.

While we are discussing the subtraction of one in the expt example, we can bring to light the sub1 and add1 functions which were simple to add and send through the compiler. These two functions use the ‘inc’ and ‘dec’ x86 instructions, and they pass through the compiler in a similar way to add. The majority of the work was adding them to the reserved list and creating a unique instruction set within select-instructions.

Finally, we come to the more intriguing portion of our project: implementation of support for cons and list functions just as in Racket. Interestingly, we decided to work in a somewhat backwards fashion for lists. First, we changed our runtime.c file to support two more Any-tag values: cons and null. These any-tag values allow the compiler to know when it is dealing with a cons cell or when a certain piece of data is representing the empty list. Additionally, we changed the print-any function to support printing cons cells and lists as near to Racket output as possible (we didn’t notice any differences).

In order to generate assembly for cons cells, we start out by type checking with a (Cons t1 t2) type. This allows car and cdr to get their respective types from the cons cell type. When we add injects and projects, we add injections and projections to the type Cons. This sets the any tag as 5. After that, but before flatten, we change all instances of cons to instances of vector with type (Vectorof Any). This allows the data to be stored as a vector, but interpreted as a cons cell. With cons cells stored as vectors, it was rather simple to turn (car c) into (vector-ref c 0), and (cdr c) into (vector-ref c 1). As for the empty list, we changed it into (hasType 0 Null), so we would store 0, but treat it as null, similarly to how we treat 0 as void. At this point the only thing we needed to add was the new type tags to select-instructions so that it could change the any tag correctly. The rest was resolved by previously working code over vectors, along with our changes to print\_any.

As a final implementation, we added support for lists and any combination of car and cdr (ex: caddr). Both of these we treat as a macro, expanding them into already supported syntax even before typecheck runs. For example, (list 1 2) expands into (cons 1 (cons 2 ())),while (caddr x) expands into (car (cdr (cdr x))). There lies a potential problem in doing this in that a user defined function fitting the aforementioned form could be overwritten by the macro expansion of car and cdr combined forms. Unfortunately, we recognized this late enough into the development process that we could not implement a fix.

Given more time, our next steps were to look at side effects other than vector-set! by means of set! and begin statements. Integrating begin statements seems as though it would have been trivial, but set! is where the trouble comes in. When a set! is used, all previously defined closures would have to be checked for instances of the variable that is being set!ed and changed to the correct value. Much like the fix for the macro expansion, we simply ran out of time to get to this interesting feature. Still, the experience we have garnered from the development of our compiler has been a fantastic learning experience that we will never forget.

Language Syntax Features

