12 points

Course Project Deliverable 4

CS 4620
assigned: 11/4/2022
due: 11/18/2022 at 5pm

The course project provides a context for you to study concepts related to compilation more deeply. This is the 4th of 5 project deliverables:

Deliverable 1 Extend the parser to support SIP

Deliverable 2 Extend the front end to support SIP

Deliverable 3 Extend the semantic analyses to support SIP

Deliverable 4 Extend the code generator to support SIP

Deliverable 5 Extend the optimizer for SIP

The project is solved in teams of two, unless you want to do it by yourself.

Deliverable 4 : Problem Statement

In this deliverable, you are to extend the code generator with support for SIP expressions and statements. Most of your work will involve:

`codegen` **routines** modifying existing one's and implementing new one's in `tipc/src/codegen/CodeGenFunctions.cpp`;

add a `-dt` **commandline option** this option will disable type inference so that you can compile programs that do not type check because of limitations in our inference algorithm.

testing adding new system tests with SIP features.

As with prior deliverables, you must not introduce any errors in processing TIP programs, i.e., programs written without any of the SIP extensions, unless you find illegal TIP programs become legal SIP programs due to the extensions.

Unlike prior deliverables, testing the code generator is best done using system tests. You can build on the existing system tests and the `tipc/test/system/run.sh` script. Any new test files that you wish to include should be included the directory `tipc/test/system/siptests`; you will need to create this directory. There is already a portion of the `run.sh` script which will compile and run every `.tip` file in the `selftests` directory. You can simply extend that code to run the files in the `siptests` directory as well.

In this deliverable, you are expected to provide testing that covers every line of the new functionality you implement. You can determine this by running the `tipc/bin/runtests.sh` script and then the `tipc/bin/gencov.sh` script and then loading the `tipc/coverage.out/index.html` file in a browser and exploring your coverage.

12 points

Course Project Deliverable 4

CS 4620
assigned: 11/4/2022
due: 11/18/2022 at 5pm

Deliverable 4 : Accessing the Project, Solution Expectations and Grading

Work on this deliverable is a continuation of your prior deliverable. Just keep working in the same repository that your team has been using.

You are expected to extend and modify, as necessary, the implementation of the `tipc` code generator, add the new command line option, extend the system test suite with new test files for SIP features, and to provide a brief description of your work. The description of your work should in a file `SOLUTION-D4.md` that you add to the root directory of your project. You should briefly describe any tricky aspects of your solution, any design alternatives you considered and how you decided among them, and your approach to testing and achieving high test coverage.

We will access the latest project commit prior to 5pm Eastern Time on Nov. 18, 2022 to assess your solution. You may continue your work on the project after that time, but it will not be considered for grading this deliverable. Grading will be based on the following rubric:

Quality of Solution (7pts) The completeness and correctness of your solution when considering both the SIP extensions and the preservation of TIP support.

Quality of Testing (4pts) The completeness of testing of your solution.

Quality of Description (1pt) The completeness and quality of the description of your solution.

Deliverable 4 : Scope, Strategy, and Suggestions

To give you a sense of the scope of work for deliverable 4, our solution is comprised of 250 new lines of cpp source code, 130 lines of comments, and 38 new tests comprised of 700 lines of TIP code.

As with previous deliverables, taking an incremental approach will be helpful here. Start with one cohesive group of SIP features, e.g., booleans, and build out the support.

You will want to study existing code generation routines to understand the use of the LLVM APIs. One of the key APIs that you will learn to use is the `IRBuilder`. The main part of this API is inherited from this class `IRBuilderBase` and is documented here `https://llvm.org/doxygen/classllvm_1_1IRBuilderBase.html`. When you scroll down you will find a list of all of the methods you see used in the `tipc` code generator.

You have already started to become familiar with the LLVM bitcode. The language reference is a good guide `https://llvm.org/docs/LangRef.html`. There is a nice writeup on the `getelementptr` instruction available here `https://llvm.org/docs/GetElementPtr.html`. Most of all use the `tipc` code generator as a starting point.

CS 4620
assigned: 11/4/2022
due: 11/18/2022 at 5pm

12 points

Course Project Deliverable 4

You will realize that the `tipc` code generator does not really consume the inferred types. Instead it works based on the assumption that any program that reaches code generation will be type correct. This allows it to use a uniform representation of data `int64` and then to cast the data to the appropriate type at the point where it is consumed by an expression. For example, when you encounter an expression like `*x` the value of `x` is loaded as an `int64`, but in the dereference expressiosn it is cast using `inttoptr` to the right type so that memory can be accessed through it. You can adopt a similar approach for your code generation.

In SIP array index expressions are bounds checked. In essence when you generate code for an expression like `a[i]` you need to first check that `0 <= i and i < #a`. If not then the program should run the equivalent of `error i` - which will print an error message with the out of bonds index `i` and terminate program execution.

One interesting design challenge in this deliverable is how to encode the length of an array. You will need this for `#a`. I encourage you to think about design issues, like this issue and the code generation templates for all new constructs. This will be especially helpful for array expressions and you should challenge your thinking by exploring how your design will handle examples like `[[1,2],[3,4]]` before you start coding. Of course you should start with something smaller like `[1]`.

In prior deliverables you learned the value of small tests. It is even more important for the code generator. This is because a bug in the code generator might not be revealed until you execute the generated program. So there is a level of indirection in this debugging that is very challenging. For this reason, I suggest start with the smallest programs possible as test cases.

Do not just compile to binary, but instead run `tipc` with the `-asm -do` options and you will be able to look at the generated `.ll` files to see if they match your expectations. You can then run `llc`, the LLVM bitcode compiler, on the `.ll` file to check whether it is legal bitcode, e.g., type correct, etc. When you generate the binary directly detailed error messages like the one's from `llc` are not available. Instead you might get the dreaded `Invalid record (Producer: 'LLVM10.0.0' Reader: 'LLVM 10.0.0')` error message which is the generic "something is wrong in this bitcode file" message. Trust me you want to go through the `.ll` to `llc` flow initially to ensure you are generating legal bitcode.

Note that some of the examples provided in the `siptests` archive, like `folds.tip`, will fail to type check. The program is type correct, but we just need a more powerful version of inference. When you have the `-dt` option working, you can compile such programs and run them.