12 points

Course Project Deliverable 3

CS 4620
assigned: 10/21/2022
due: 11/4/2022 at 5pm

The course project provides a context for you to study concepts related to compilation more deeply. This is the 3rd 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 3 : Problem Statement

You extended the front end with SIP support in deliverables 1 and 2. In this deliverable, you are to extend the semantic analysis phase with SIP support. More specifically, you will extend the:

Weeding pass updating the assignability checking to handle SIP constructs;

Type system adding new subtypes of `TipType` to define the new SIP types; and

Type constraints adding new methods to build type constraints from SIP AST nodes.

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. In this case you may need to update some of the existing TIP unit tests.

You are to provide CATCH2 unit tests that thoroughly exercises the functionality of your frontend. You can build on the existing unit tests available for the frontend. 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/gencov.sh` script and then loading the `tipc/coverage.out/index.html` file in a browser and exploring your coverage.

Deliverable 3 : 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.

CS 4620
assigned: 10/21/2022
due: 11/4/2022 at 5pm

12 points

Course Project Deliverable 3

You are expected to extend and modify, as necessary, the implementation of the `tipc` semantic analyses, to extend semantic unit test suites with new test files beginning with the `SIP` character sequence, and to provide a brief description of your work. The description of your work should in a file `SOLUTION-D3.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. 4, 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 3 : Scope, Strategy, and Suggestions

To give you a sense of the scope of work for deliverable 3, not including our test code, our solution is comprised of:

- 2 new header and cpp files
- 207 new lines of cpp source code
- 95 new lines of comments

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 for it across the semantic analyses. You may find that some of the support for TIP needs to be updated, e.g., does it make sense in SIP that conditional expressions in `if-then-else` have integer type.

A key part of this deliverable is to extend `tipc/src/semantic/types/constraints/TypeConstraintVisitor.cpp` to generate type constraints for the new AST nodes that you introduced to support SIP. In order to do this we suggest that you begin by writing the type rules for each new SIP construct as header comments for the appropriate visitor methods. The existing methods in the visitor have examples for TIP constructs. This will help you clarify and debug exactly what you want to implement. It's the reason we have so many new comment lines in our solution.

12 points

Course Project Deliverable 3

CS 4620
assigned: 10/21/2022
due: 11/4/2022 at 5pm

There are new types introduced in SIP so you will need to appropriately extend the representation of types in the compiler. This is done by creating new subtypes of `tipc/src/semantic/types/concrete/TipType.h`. You will also need to update other parts of the type implementation that depend on these types. For instance, `TipTypeVisitor.h` and any of its subtypes and any other code that explicitly tests for types, e.g., look for `dynamic_cast<...>` where the `...` involves a subtype of `TipType`. This will be reminiscent of the the work you did in deliverable 2 to extend the AST, so put what you learned there into practice here.

The unit testing for the semantic analysis in tipc is pretty well structured and you can use it as a model for your tests. You don't need to invent a new way of structuring tests, e.g., no new helpers are needed, and can just build your tests as variations on what exist for TIP unit tests. Much of the unit testing is straightforward, but the testing of type constraint generation is a bit tricky so I'll describe it in more detail.

The tests in `tipc/test/unit/semantic/types/constraints/TypeConstraintCollectTest.cpp` follow a regular structure. There is a `runtest` function that accepts a stringstream encoding the program and a vector of strings describing the expected results. The comment in the first test in this file is important to understand when setting up the expected results:

```
/*
 * For all of the test cases the expected results should be written with the following in mind:
 *   @l:c  lines and columns are numbered from 1 and 0, respectively
 *   ( )   non-trivial expressions are parenthesized
 *   order the order of constraints is determined by the post-order AST visit
 *   space there is no spacing between operators and their operands
 *   id    identifier expressions are mapped to their declaration
 *
 * Note that spacing is calculated immediately after the R"( in the program string literal.
 */
```

Once you have a solid understanding of type constraints, you will be able to walk through a few of these tests and confirm the expected results – recall "[[" and "]]" denote a type variable for a given AST node. It is a bit trickier to come up with the expected results. I suggest writing very small tests, like the one for "main" in the existing tests. This will make it much easier for you to define the expected results.

Here are some typical problems encountered when defining expected results.

Getting the line and column numbers right Numbering starts immediately after the "R"(" in the definition of the program literal. The column for an expression is the first column in which the characters of the expression appear. Numbering for identifiers is mapped to the line and column for its declaration not its appearance as an expression.

Ordering This is easy to get wrong because you have to imagine the AST and the post-order visit. One good way to address this is to use the logging in the code base to help out. I discuss this below.

Spacing The print routines for types use the AST print routines, so those define the conventions for text within the "[[" and "]]".

When running tipc you can use the `--verbose` option to dump logging information. Logging within the type checker is particularly plentiful. For this assignment what is most relevant are the lines with the string "Generating constraint". A standard workflow for us was the following run within `tipc/build` on file `foo.tip`:

```
./src/tipc --log=foo.log foo.tip
grep "Generating type constraint" foo.log
```

This will print the type constraints that were generated by the compiler. Note that for this to be useful for your project you need logging commands for the type constraint generation for SIP constructs. Fortunately, this logging is implemented in the `ConstraintCollector::handle` method so you don't need to to anything in this deliverable to get these log messages.