DSEL calc

Jens Emil Fink Højriis

March 8, 2024

Listing 1: ./AST.hpp

```
#pragma once
#include <memory>
#include "op.hpp"
#include "state.hpp"
#include "term.hpp"
namespace calculator {
    class symbol_table_t;
    class expr_t;
    class var_t : public term_t {
        size_t _id;
        double& get_value(state_t& s);
        var_t(size_t id)
            : _id(id) {}
    public:
        var_t(const var_t&) = default;
        var_t(var_t&&) = default;
        var_t& operator=(const var_t&) = default;
        var_t& operator=(var_t&&) = default;

        double operator()(state_t&);
        double operator()(state_t&, double new_value);
        friend class symbol_table_t;
    };

    class const_t : public term_t {
        double _value;
    public:
        const_t(double c)
            : _value(c) {}
        double operator()(state_t&) { return _value; }
    };

    class assign_t : public term_t {
        var_t _var;
        std::shared_ptr<term_t> term;
    public:
        assign_t(var_t var, std::shared_ptr<term_t> term)
            : _var(std::move(var)), term(std::move(term)) {}
        double operator()(state_t&);
    };

    class unary_t : public term_t {
        std::shared_ptr<term_t> term;
        op_t _op;
    public:
        unary_t(std::shared_ptr<term_t> term, op_t op)
            : term(std::move(term)), _op(op) {}
        double operator()(state_t& s);
    };
```

```
50
51      class binary_t : public term_t {
52          std::shared_ptr<term_t> _term1;
53          std::shared_ptr<term_t> _term2;
54          op_t _op;
55      public:
56          binary_t(std::shared_ptr<term_t> lhs, std::shared_ptr<term_t> rhs, op_t op)
57              : _term1(std::move(lhs)), _term2(std::move(rhs)), _op(op) {}
58          double operator()(state_t&);
59      };
60  }
```

Listing 2: ./expr.hpp

```
1   #pragma once
2   #include "op.hpp"
3   #include "term.hpp"
4   #include "symbol_table.hpp"
5   #include "AST.hpp"
6   #include <vector>
7   #include <memory>
8   #include <string>
9
10  namespace calculator {
11      class expr_t {
12      public:
13          std::shared_ptr<term_t> term;
14          expr_t(const var_t& var)
15              : term(std::make_shared<var_t>(var)) {};
16          expr_t(std::shared_ptr<term_t> term)
17              : term(std::move(term)) { }
18          expr_t(const expr_t&) = default;
19          expr_t(expr_t&&) = default;
20          expr_t& operator=(const expr_t&) = default;
21          expr_t& operator=(expr_t&&) = default;
22          expr_t(double c);
23          double operator()(state_t&);
24      };
25
26      expr_t operator-(expr_t e);
27      expr_t operator+(expr_t e);
28
29      expr_t operator+(expr_t lhs, expr_t rhs);
30      expr_t operator-(expr_t lhs, expr_t rhs);
31      expr_t operator*(expr_t lhs, expr_t rhs);
32      expr_t operator/(expr_t lhs, expr_t rhs);
33
34      expr_t operator<<=(expr_t v, expr_t e);
35      expr_t operator+=(expr_t v, expr_t e);
36      expr_t operator-=(expr_t v, expr_t e);
37      expr_t operator*=(expr_t v, expr_t e);
38      expr_t operator/=(expr_t v, expr_t e);
39  }
```

Listing 3: ./op.hpp

```
1   #pragma once
2   namespace calculator {
3       enum class op_t {
4           plus,
5           minus,
6           mul,
```

```
        div,
        assign
    };
};
```

Listing 4: ./state.hpp

```
#pragma once
#include <vector>
using state_t = std::vector<double>;
```

Listing 5: ./symbol_table.hpp

```
#pragma once
#include "op.hpp"
#include "AST.hpp"
#include "state.hpp"
#include <vector>
#include <string>


namespace calculator {
    class symbol_table_t;

    class symbol_table_t {
        std::vector<std::string> names; ///< stores the variable names
        std::vector<double> initial;    ///< stores the initial values of variables
    public:
        /// Creates a variable with given name and initial value
        [[nodiscard]] var_t var(std::string name, double init = 0) {
            auto res = names.size();
            names.push_back(std::move(name));
            initial.push_back(init);
            return var_t(res);
        }
        /// Creates a system state initialized with initial variable values
        [[nodiscard]] state_t state() const { return { initial }; }
    };
}
```

Listing 6: ./term.hpp

```
#pragma once
#include "state.hpp"
namespace calculator {
    class term_t {
    public:
        virtual ~term_t() noexcept = default;
        virtual double operator()(state_t&) = 0;
    };
}
```

Listing 7: ./calculator.cpp

```
#include "expr.hpp"
#include <stdexcept>
#include <memory>
namespace calculator {
    double expr_t::operator()(state_t& s) {
        return (*term)(s);
    }

```

```
9       double& var_t::get_value(state_t& s) {
10          if (_id < s.size()) {
11              return s[_id];
12          } else {
13              throw std::exception("var id is out of range");
14          }
15      }
16
17      double var_t::operator()(state_t& s) {
18          return get_value(s);
19      }
20
21      double var_t::operator()(state_t& s, double new_value) {
22          return get_value(s) = new_value;
23      }
24
25      double assign_t::operator()(state_t& s) {
26          auto value = (*term)(s);
27          return _var(s, value);
28      }
29
30      double unary_t::operator()(state_t& s) {
31          auto value = (*term)(s);
32          switch (_op) {
33          case op_t::plus:
34              return value;
35          case op_t::minus:
36              return -value;
37          default:
38              throw std::logic_error("Invalid operation");
39          }
40      }
41
42      double binary_t::operator()(state_t& s) {
43          auto lhs = (*_term1)(s);
44          auto rhs = (*_term2)(s);
45          switch (_op) {
46          case op_t::plus:
47              return lhs + rhs;
48          case op_t::minus:
49              return lhs - rhs;
50          case op_t::div:
51              if (rhs == 0) {
52                  throw std::logic_error("division by zero");
53              }
54              return lhs / rhs;
55          case op_t::mul:
56              return lhs * rhs;
57          default:
58              throw std::logic_error("Invalid operation");
59          }
60      }
61
62      expr_t operator+(expr_t lhs, expr_t rhs) {
63          return expr_t(std::make_shared<binary_t>(std::move(lhs.term), std::move(rhs.term),
    ↪op_t::plus));
64      }
65
66      expr_t operator-(expr_t lhs, expr_t rhs) {
67          return expr_t(std::make_shared<binary_t>(std::move(lhs.term), std::move(rhs.term),
    ↪op_t::minus));
```

```
    }

    expr_t operator*(expr_t lhs, expr_t rhs) {
        return expr_t(std::make_shared<binary_t>(std::move(lhs.term), std::move(rhs.term),
↪op_t::mul));
    }

    expr_t operator/(expr_t lhs, expr_t rhs) {
        return expr_t(std::make_shared<binary_t>(std::move(lhs.term), std::move(rhs.term),
↪op_t::div));
    }

    expr_t operator-(expr_t e) {
        return expr_t(std::make_shared<unary_t>(std::move(e.term), op_t::minus));
    }

    expr_t operator+(expr_t e) {
        return expr_t(std::make_shared<unary_t>(std::move(e.term), op_t::plus));
    }

    expr_t operator<<=(expr_t lhs, expr_t rhs) {
        auto var = std::dynamic_pointer_cast<var_t>(std::move(lhs.term));
        if (var == nullptr) {
            throw std::logic_error("assignment destination must be a variable expression");
        }
        return expr_t(std::make_shared<assign_t>(*var, std::move(rhs.term)));
    }

    expr_t operator+=(expr_t v, expr_t e) {
        return v <<= v + std::move(e);
    }

    expr_t operator-=(expr_t v, expr_t e) {
        return v <<= v - std::move(e);
    }

    expr_t operator*=(expr_t v, expr_t e) {
        return v <<= v * std::move(e);
    }

    expr_t operator/=(expr_t v, expr_t e) {
        return v <<= v / std::move(e);
    }
}
```

Listing 8: ./calculator_test.cpp

```
//#include "expr.hpp"
#include "expr.hpp"
#include <doctest/doctest.h>

#include <sstream>

TEST_CASE("Calculate expressions lazily") {
    auto sys = calculator::symbol_table_t{}; // create a symbol table for variables
    auto a = sys.var("a", 2); // create a variable with name "a" and initial value of 2
    auto b = sys.var("b", 3); // create a variable with name "b" and initial value of 3
    auto c = sys.var("c");    // create a variable with name "c" and default-initialize with 0
    auto state = sys.state(); // create a system state initialized with variable initial values
    auto os = std::ostringstream(); // string stream to output to

    SUBCASE("Reading the value of a variable from state") {
```

```
16          CHECK(a(state) == 2);
17          CHECK(b(state) == 3);
18          CHECK(c(state) == 0);
19      }
20      SUBCASE("Unary operations") {
21          CHECK((+a)(state) == 2);
22          CHECK((-b)(state) == -3);
23          CHECK((-c)(state) == 0);
24      }
25      SUBCASE("Addition and subtraction") {
26          CHECK((a + b)(state) == 5);
27          CHECK((a - b)(state) == -1);
28          // the state should not have changed:
29          CHECK(a(state) == 2);
30          CHECK(b(state) == 3);
31          CHECK(c(state) == 0);
32      }
33      SUBCASE("Assignment expression evaluation") {
34          CHECK(c(state) == 0);
35          CHECK((c <<= b - a)(state) == 1);
36          CHECK(c(state) == 1);
37
38          // TODO: implement multiplication
39          CHECK((c += b - a * c)(state) == 2);
40          CHECK(c(state) == 2);
41          CHECK((c += b - a * c)(state) == 1);
42          CHECK(c(state) == 1);
43
44
45          // TODO: implement other assignments: +=, -=, *=, /=
46          CHECK_THROWS_MESSAGE((c - a += b - c), "assignment destination must be a variable   ↙
  ↪expression");
47
48      }
49      SUBCASE("Parenthesis") {
50          CHECK((a - (b - c))(state) == -1);
51          CHECK((a - (b - a))(state) == 1);
52      }
53
54      // TODO: implement multiplication and division
55      SUBCASE("Evaluation of multiplication and division")
56      {
57          CHECK((a * b)(state) == 6);
58          CHECK((a / b)(state) == 2. / 3);
59          CHECK_THROWS_MESSAGE((a / c)(state), "division by zero");
60      }
61      SUBCASE("Mixed addition and multiplication")
62      {
63          CHECK((a + a * b)(state) == 8);
64          CHECK((a - b / a)(state) == 0.5);
65      }
66
67
68      // TODO: implement support for constant expressions
69      SUBCASE("Constant expressions")
70      {
71          CHECK((7 + a)(state) == 9);
72          CHECK((a - 7)(state) == -5);
73      }
74      SUBCASE("Store expression and evaluate lazily")
75      {
```

```
        auto expr = (a + b) * c;
        auto c_4 = c <<= 4;
        CHECK(expr(state) == 0);
        CHECK(c_4(state) == 4);
        CHECK(expr(state) == 20);
    }
}
```

Listing 9: ./CMakeLists.txt

```
   # CMakeList.txt : CMake project for session-1, include source and define
# project specific logic here.
#

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
set(BUILD_SHARED_LIBS OFF)

include(sanitizers.cmake)
include(doctest.cmake)

# Add source to this project's executable.
add_executable(calculator_test "calculator_test.cpp" "calculator.cpp" "calculator.hpp")
target_link_libraries(calculator_test PRIVATE doctest::doctest_with_main)

enable_testing()
add_test(NAME calculator_test COMMAND calculator_test)

# TODO: Add tests and install targets if needed.
```

Listing 10: ./README.md

```
# DSEL Calculator

## Purpose:
1) Revisit the [operator
↪overloading](https://people.cs.aau.dk/~marius/Teaching/SP2024/classes.html#/3/22), exercise
↪PImpl and dynamic polymorphism.
2) Introduce a small Domain Specific Embedded Language (DSEL) modeling a lazy calculator.


## Suggested Workflow
0) Before changing anything, run `calculator_test` first and see if the project works,
   then create/uncomment a test case in [calculator_test.cpp](calculator_test.cpp),
   (see a brief introduction to *doctest* in **Unit Testing** section below).
1) Implement multiplication and division operators by following the examples of addition and
↪subtraction in [calculator.hpp](calculator.hpp).
2) **Refactor** the `expr_t` code into a hierarchy of possible Abstract Syntax Tree (AST)
↪**term**s (independent of expression operators), e.g.:

![class diagram](calculator.png)

3) Refactor the adapter **struct expr_t** which wraps **term**s and builds the corresponding
↪Abstract Syntax Tree through operator overloading.
4) Implement support for constant expressions (values like `5` and `3.14`).
5) Implement support for printing (use unit testing!), e.g.:
```cpp
   std::cout << printer{sys, a+b} << std::endl; // prints "a+b"
```
```

```
23
24 ## Questions for Reflections
25 1) Compare the signatures of binary operators in this project with [std::valarray operator
   ↪overloads](https://en.cppreference.com/w/cpp/numeric/valarray/operator_arith3):
26    do they take arguments by value or by reference? return by value or by reference?
27    Are there a fundamental differences in semantics (meaning/implementation)?
28 2) Did you use inheritance or aggregation or both? where? why?
29 3) Did you use smart pointers? which? where? why?
30 4) Can we change ‘unique_ptr‘ into ‘shared_ptr‘ or vice-a-versa? why?
31 5) Which definitions can we safely hide into separate **cpp** file without disturbing the test
   ↪cases?
32 6) Did you like unit testing? Would you use it in your future projects?
33
34 ## Unit Testing
35 This exercise introduces [doctest](https://github.com/doctest/doctest) unit testing framework.
36
37 It is a very simple, yet rich and powerful unit testing framework packed just in one header file.
38
39 For now, here is a list of the most important concepts:
40
41 1) Test case starts with a ‘TEST_CASE‘ macro taking a name as a parameter.
42 2) The body contains statements just like any C/C++ function.
43 3) The body should include one or more test statements:
44    * ‘REQUIRE‘: test if the condition is true and abort the test case otherwise.
45    * ‘CHECK‘: test if condition is true and proceed regardless.
46    * ‘CHECK_THROWS_MESSAGE‘: test that the expression produces an exception with a given message.
47 4) The body may contain a nested ‘SUBCASE‘ which continue the test case,
48    except each ‘SUBCASE‘ is independent of its siblings, i.e. the shared test prefix is repeated
49    for each ‘SUBCASE‘ branch as if there was no other ‘SUBCASE‘.
50 ‘‘‘cpp
51 TEST_CASE("The name or even better: the purpose of this test") {
52     // body of the test
53     auto os = std::ostringstream{};
54     REQUIRE(2+3 == 5); // cannot proceed if arithmetics fail
55     os << "hello";
56     CHECK(os.str() == "hello"); // you get what you put in
57     CHECK_THROWS_MESSAGE(fn("naughty argument"), "error: someone is being naughty");
58     SUBCASE("Extra functionality") {
59         os << " world!"; // continue putting-in
60         CHECK(os.str() == "hello world!");
61     }
62     SUBCASE("Alternative scenario") {
63         os << " to you too"; // continue putting-in as if previous SUBCASE never happened
64         CHECK(os.str() == "hello to you too");
65     }
66 }
67 ‘‘‘
68 More details can be found in [doctest
   ↪tutorial](https://github.com/doctest/doctest/blob/master/doc/markdown/tutorial.md).
```