CS3733-D01: Software Engineering

The Software Integration Project : (Individual)

Due: Sunday, April 15, 11:59pm (via turnin)

1 Project Description

Your company designs circuits for some of its products. The designers routinely propose optimizations to the circuits. Before accepting an optimized design, however, the company wants to know whether the optimized circuit has the same behavior (produces the same output) as the original circuit. The company therefore wants to build a software tool to determine whether two circuits have the same behavior.

The company has purchased an existing software package that provides efficient representations and implementations of boolean logic expressions and key operations on them. That package is written in C, while the company's circuit design tools are written in MzScheme. Your job is provide a MzScheme file that uses the C-based package to implement the comparator for combinational circuits (ie, circuits with and, or, and not gates, but no latches/memory elements). Your code must use the company's existing data structures for circuits.

This document describes the desired circuit comparator, the C package you must interface to, and how to extend MzScheme with code written in C. It also provides details of what to turn in for this assignment.

1.1 Goal/Purpose of This Assignment

Real software engineering often requires you to build one product out of code written in two different languages. This project introduces you to some of the potential pitfalls that arise when integrating code from two languages, including

- differences in type systems,
- differences in memory management strategies, and
- sharing data representations across languages.

These issues can arise (to different extents and in different forms) in *any* pair of languages that you try to integrate. This exercise is designed to make you confront these issues in a context with a lot of contrast between the two languages; this will prepare you to handle these issues for any given pair of languages.

2 The Desired Circuit Comparator

Circuits define functions from inputs to outputs. Two circuits are equivalent if they have the same input and output names and if the function computed for each output is the same in both circuits. The company's software represents circuits with objects. To implement the comparator, you need to extend the circuit class with a method *check-equiv* that accepts a circuit and determines whether the two circuits are equivalent. For example, given circuit objects C1 and C2 representing equivalent circuits, the following expression demonstrates an interaction with your code:

```
> (send C1 check-equiv C2)
#t
```

If the circuits are not equivalent, your code will raise various exceptions depending upon the reason why the circuits are not equivalent. The precise exceptions are described later in this document.

2.1 The (Provided) Classes

2.1.1 The Class Hierarchy

Figure 1 shows the class hierarchy for circuits. These classes require the initialization arguments and satisfy the interfaces shown in the following table. All classes implement the interfaces of their ancestor classes. The initialization

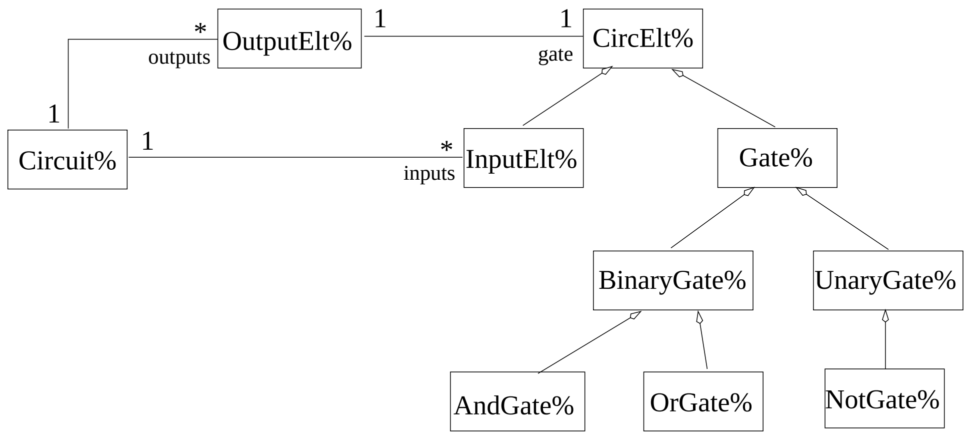
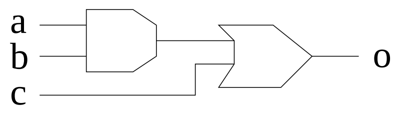


Figure 1: Class diagram for Circuits.

arguments are given in the order that the class expects to receive them. If a class is not listed, it requires the constraints on its closest ancestor.

Class	**Initialization Args**	**Interfaces**
Circuit%	list-of-InputElt, list-of-OutputElt	CircuitI
OutputElt%	string (name), CircElt	NamedEltI, CircOutputI
InputElt%	string (name)	NamedEltI
BinaryGate%	CircElt, CircElt (the gate inputs)	BinaryGateI
UnaryGate%	CircElt (the gate input)	UnaryGateI

As an example of how the hierarchy is used to capture a circuit, consider the following simple circuit and its representation in objects:



```
(define C1
  (let ([a (make-object InputElt% "a")]
        [b (make-object InputElt% "b")]
        [c (make-object InputElt% "c")])
    (let ([o (make-object OutputElt% "o"
               (make-object OrGate% (make-object AndGate% a b) c))])
      (make-object Circuit% (list a b c) (list o)))))
```

This circuit captures the expression "$(a \wedge b) \vee c$". Assume C2 captures circuit "$(a \vee c) \wedge (b \vee c)$", C3 captures circuit "$a \vee c$", and C4 captures "$a \wedge c$". Then your comparator should behave as follows (the exceptions are discussed in more detail later in the document):

- (*send C1 check-equiv C2*) returns #t
- (*send C1 check-equiv C3*) results in an InputMismatchExn containing mismatched input names (*list* "b")
- (*send C3 check-equiv C4*) results in a FunctionMismatchExn containing output name "o" and an assignment to *a* and *c* representing a counterexample. One possible counterexample would be (*list* (*make-var-assign* "a" 1) (*make-var-assign* "b" 0)); var-assigns are discussed in more detail later in the document.

2.1.2 The Interfaces

- **NamedEltI** : allows access to the name of an element

  - *get-name* : → *string*

- **CircOutputI** : allows access to the circuit element that drives an output
  - *get-gate* : → *CircElt*
- **BinaryGateI** : allows access to the inputs to a binary gate
  - *get-in1* : → *CircElt*
  - *get-in2* : → *CircElt*
- **UnaryGateI** : allows access to the input to a unary gate
  - *get-in* : → *CircElt*
- **CircuitI** : methods for accessing/processing circuit attributes
  - *get-inputs* : → *list-of-InputElt*
  - *get-outputs* : → *list-of-OutputElt*
  - *get-input-names* : → *list-of-string*
    Returns the names labelling the inputs objects
  - *get-output-names* : → *list-of-string*
    Returns the names labelling the output objects
  - *find-output-elt* : *string* → *OutputElt*
    Returns the OutputElt with the name given in the input string. Raises a NoMatchingOutput exception if no such output exists.

2.2 What Your Code Must Provide

Your comparator file must define extensions to the existing classes as needed to implement the new comparator interface for the Circuit% class. The new interface, **CircCompareI**, supports the following method:

- *check-equiv* : *Circuit* → *true*

  Given a circuit, return true if the given circuit and the current circuit are equivalent. If they are not equivalent, raise one of the following exceptions:

  - OutputMismatchExn : if the circuits do not agree on their output names
  - InputMismatchExn : if the circuits do not agree on their input names
  - FunctionMismatchExn : if the inputs and outputs agree, but the circuits disagree on the function computed by some output

Your code may define extensions to any of the classes at the leaves of the hierarchy shown in Figure 1. To be compatible with our test suite, the extended classes that you define should have the same names as the original classes. Your extensions may not add initialization arguments to the classes. You can assume that the company's code base defines the following data structures, classes and functions:

- The classes shown in Figure 1, satisfying the interfaces described in the table in Section 2.1.
- *make-var-assign : string* $\{0,1\}$ → *VarAssign*

  Consumes a string (for the name of a variable) and either 0 or 1 (for the value of a boolean variable). Returns an instance of the VarAssign data structure containing the input data.

- *NoMatchingOutputExn?* : *value* → *boolean*

  Indicates whether a value is a NoMatchingOutput exception.

- *raise-output-mismatch-exn* : *list-of-string* → *void*

  Consumes a list of output names that the circuits disagree on and raises an OutputMismatchExn exception with that information.

- *raise-input-mismatch-exn* : *list-of-string* → *void*

  Consumes a list of input names that the circuits disagree on and raises an InputMismatchExn exception with that information.

- *raise-function-mismatch-exn* : *string list-of-VarAssign* → *void*

  Consumes the name of an output and a list of assignments of values to input variables that demonstrates that two circuits compute different functions for the named output. Raises a FunctionMismatchExn exception with this information.

The web page provides a sample of the company's code base that you can use for testing your work. **You should not edit this file or duplicate its contents**, as we may test your code with a different implementation of the company's code base than what is on the web page. Your code should be in a separate file and should simply extend the existing code base.

3 The C-Based Propositional Logic Package

The C-based propositional logic package (called CUDD) provides a compact and canonical representation (called a DdNode) of boolean expressions and operations over those expressions. A canonical representation of boolean formulas represents equivalent formulas with the same data structure. In CUDD, logically equivalent expressions point to the same address, so you can test whether two expressions are logically equivalent by testing whether the DdNodes are the same (*i.e.* ddnode1 == ddnode2).

In order to maintain shared addresses for equivalent expressions, each expression/DdNode is associated with a manager data structure (the details of which are irrelevant to this project). Two expressions may be combined or compared only if they are associated with the same manager. You should therefore use only one manager structure per circuit comparison.

The package handles memory management using a technique called reference counting. A reference counting system maintains a count of how many references (essentially variables) refer to each datum. When a datum has a reference count of zero, the system reclaims its memory. Programmers are responsible for maintaining the reference counts; the system performs the actual malloc and free operations. Whenever you create a new expression, you must increment its reference count (using the function Cudd_Ref); whenever you would normally free an expression, you must decrement its reference count (using the function Cudd_RecursiveDeref which decrements the counts for an expression and its subexpressions).

For this assignment, you may use the following CUDD functions (and no others):

- Cudd_Init which creates a (new) DdManager*. Call this function as

  ```
  Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
  ```

  where CUDD_UNIQUE_SLOTS and CUDD_CACHE_SLOTS are provided by cudd.h, which your interface file must include.

- Cudd_bddNewVar : DdManager* → DdNode*

  Returns a new expression consisting of a single variable, registering that variable with the given Manager.

- Cudd_Not : DdNode* → DdNode*

  Returns the expression representing the negation of the given expression. Does not require the Manager.

- Cudd_bddOr : DdManager* DdNode* DdNode* → DdNode*

  Returns the expression representing the disjunction (or) of the two given expressions. Both input expressions must have been created with the given manager. The returned expression is also registered with the given manager.

- Cudd_ReadZero : DdManager* → DdNode*

  Returns the expression for false as defined in the given Manager.

- Cudd_RecursiveDeref : DdManager* DdNode* → void

  Decrements the reference count for the given expression in the given Manager.

- Cudd_Ref : DdNode* → void

  Increments the reference count for the given expression.

CUDD is provided as a set of object files. You therefore need to interact with this package via MzScheme's foreign function interface.

4 Extending MzScheme with C Code

At a high level, adding functions written in C to MzScheme requires writing and compiling some wrapper code (in C) that performs three tasks:

1. Creates new Scheme type tags for each type of data from the C code that will pass to the Scheme code. (Scheme tags each value with a type that is used to check whether primitives are properly applied at run-time. This step creates tags for the new kinds of values that the C code provides).
2. Defines wrappers for each C function that you wish to export to Scheme. These wrappers add, remove, and check the type tags on all values passed between the C and Scheme code.
3. Registers the new functions with Scheme, so that they become globally available with MzScheme.

The rest of this section describes the low-level steps required for these tasks. Create a single C file called adapt.c. All of the code described in the following list goes into this file. Figure 2 provides a sample adapt.c file showing the key steps.

1. Include header file escheme.h.
2. Declare new scheme type tags for each type of value that will pass between the C code and the Scheme code. Each declaration takes the form

   Scheme_Type new_scheme_type

   where new_scheme_type is a name you chose for the new type tag and Scheme_Type is provided in escheme.h.
3. Define a set of C-structs that package a C value with a scheme type tag. You should define one struct for each type tag you defined in the previous step. Define new C types for each of these structs.
4. For each C function that you wish to make available in Scheme, write a wrapper function to provides the interface between the Scheme and C functions. The wrapper should return a value of type Scheme_Object* (escheme.h provides this type). The wrapper should consume two arguments: an int (giving the argument count) and an array of Scheme_Objects (the arguments coming from Scheme). A sample wrapper header appears as follows:

   ```
   Scheme_Object* wrapper_func (int argCount , SchemeObject **argList)
   {
   <body of wrapper here>
   }
   ```

```
#include <escheme.h>

Scheme_Type scheme_frob_type ;

struct scheme_frob_struct
{
  Scheme_Type type ;
  frob*       data ;
} dummy_scheme_frob_struct ;

typedef struct scheme_frob_struct * scheme_frob ;

Scheme_Object* process_frob_wrap (int argCount , Scheme_Object **argList)
{
  scheme_frob a_frob ;
  frob        *new_frob ;
  scheme_frob new_scheme_frob ;

  a_frob = (scheme_frob) argList [ 0 ] ;

  /* insert error checking */

  new_frob = process_frob ( ( a_frob -> data ) ) ;

  /* write construct_scheme_frob, which packages new_frob in a scheme_frob struct */
  /* Use scheme_malloc (sizeof (dummy_scheme_frob_struct)) to allocate the memory */

  new_scheme_frob = construct_scheme_frob ( new_frob ) ;

  return ( (Scheme_Object*) new_scheme_frob ) ;
}

Scheme_Object* scheme_initialize (Scheme_Env  *env)
{
  scheme_frob_type = scheme_make_type ( "<scheme-frob>" ) ;

  scheme_add_global ("handle-frob",
                     scheme_make_prim_w_arity (process_frob_wrap,
                                               "handle-frob", 1, 1),
                     env) ;

  return scheme_void ;
}

Scheme_Object *scheme_reload(Scheme_Env *env) {
   return scheme_initialize(env);
}
```

Figure 2: A sample adapt.c file. This example extends Scheme with some C code that creates data of type frob and provides a function process-frob which consumes and returns a frob. Function process-frob-wrap is the wrapper function. We wish to create a Scheme function handle-frob that consumes a frob and calls process-frob via the wrapper.

The body of the wrapper must do four things:

(a) Extract the arguments from the argList. Each argument will be one of the structure types that you declared in step 3.

(b) Check that each argument has the desired type tag. If some argument does not, print an error message indicating which argument was incorrect (using printf) and exit (this supplies Scheme's required run-time checking that operators are only applied to values of the correct type).

(c) Call the corresponding C function. You may do any necessary pre- or post-processing of the C data here.

(d) Create a structure (from step 3) containing the returned value and its appropriate type tag. Cast that structure to Scheme_Object* and return it from the wrapper.

5. Write a function scheme_initialize that creates the names of the new Scheme functions and associates each name with one of the wrappers you wrote in the previous step. This function consumes an argument of type Scheme_Env (defined in escheme.h) and returns something of type Scheme_Object*. The body of the function creates the actual scheme type tags using function scheme_make_type and registers the wrapper functions using function scheme_add_global (both provided in escheme.h). Figure 2 shows how to use these functions.

6. Include the following definition in adapt.c:

```
Scheme_Object *scheme_reload(Scheme_Env *env) {
   return scheme_initialize(env);
}
```

4.1 Loading Extension Code into MzScheme

MzScheme provides a compiler that compiles C extensions into shared-object files that MzScheme/DrScheme can load. The directory `/home/kfisler/CS3733/Comparator/` on CCC provides an appropriate Makefile. Run make to create file adapt.so. Then use MzScheme/DrScheme command (*load-extension* `"adapt.so"`) to add your new functions to Scheme.

5 What to Turn In

Submit two files via turnin, under assignment name *comparator*:

- adapt.c, which contains the wrapper code described above. Do not use absolute pathnames when you include escheme.h and cudd.h, as we will be compiling these on CS rather than on CCC.
- comparator.ss, which contains the Scheme code you wrote to implement the comparator.

6 Grading Criteria

You will be graded on the following criteria:

- Whether your adapt.c file compiles and whether your comparator.ss file loads.
- Whether your class extensions satisfy the specified signatures and interfaces.
- How many features (multiple output circuits, correct exception raising, counter-example generation, etc) your code implements correctly.
- Whether your adapt.c file provides sufficient type checking.
- Whether you adequately address memory management issues.

- Whether you used only the allowable CUDD functions.
- Whether your code is well-designed, uses helper functions as appropriate, and is well-documented. We won't waste time trying to understand your code. If it doesn't run and isn't documented, you won't get much credit for it. (If it does run, it still needs to be well-documented).

7 Collaboration Policy

You may talk to other students about adapt.c and problems you encounter getting it to compile. **You may not share or borrow adapt.c code with or from other students. Your design, implementation, and work on the comparator file must be entirely your own.** You may, as always, talk to the course staff about the project.

8 Miscellaneous Notes

- escheme.h defines constants scheme_true and scheme_false, which you can use to return booleans from C functions to Scheme functions.
- You may install the CUDD package on your own computer (it runs under many Unix/Linux flavors). The source is available on CCC in the directory
  `/home/kfisler/CS3733/Comparator/CUDD/cudd-2.3.0.tar.gz`
- The MzScheme library "function.ss" defines quicksort, in case you need a sorting routine. See the helpdesk for details on the arguments that quicksort expects.