Lecture 13, Apr 9, 2024

State Machines

- The *state machine* is a model of the dynamic behaviour of a system, represented by:
  - *States*: physical or logical states of the program, e.g. different ranges of variable values, program flow locations, etc that hold a certain meaning
  - *Transitions*: pre-define paths between states, triggered by specific actions or conditions
- State machines are an abstract but functionally equivalent representation of an underlying program or hardware
  - They can be used as a design pattern
  - Can be used as a diagnostic tool for existing code, e.g. automatic state minimization/optimization, consistency checking, etc
- A *finite state machine* (FSM) is a state machine with a finite set of states and allowable transitions between states, possibly with input required for each transition
  - FSMs can have *nondeterminism*, where the same input can lead to multiple different transition
  - We usually assume that our systems are fully deterministic (*deterministic finite automata*, DFAs)
- Each state also has a *payload* or output, i.e. the action taken by the program when that state is reached
- We can specify a state machine by the following:
  - List of states
  - List of outputs for each state (sharing indices with the state list)
  - List of transitions, as tuples of (source state, destination state)
  - List of inputs for each transition (sharing indices with the transitions list)
- *Dead ends* in the state machine are states that have no transitions out
  - These can be problematic
- Implementation of states and transitions can be explicit, but can also be implicitly defined using program flow itself
  - Explicit encoding derives directly from the mathematical representation; easy to modify, difficult to debug manually, has transition search overhead, hard to optimize by compiler
  - Implicit encoding is much more readable by a human; difficult to modify, easy to debug by a program, and has faster transitions
- State machine *minimization* is the process of removing *redundant* states
  - States are redundant if they have the same output and transition to the same set of states given the same input
  - We can search for redundant states, merge them, and then check states again and repeat
  - Given a state machine, we might want to rename/encode the states and write it in a mathematical representation, which makes the redundant states more obvious
    * Eliminate one of the redundant states and change all references to the eliminated state
    * Go through the list of transitions and eliminate duplicate transitions resulting from the change
  - To further simplify the state machine we may introduce variables
    * These variables are essentially state machines of their own
    * We control the transitions in this new state machine via the transitions of the original state machine; when we take transitions, we may choose to modify the variable, i.e. cause a transition in the variable's state machine
  - We can keep moving more of the payload onto transitions; eventually we reach a system with only a single state, with every action being a transition (input conditioned on a large number of variables)
    * This has diminishing returns
  - Since state machines are easy for code to understand, many automated systems exist for state machine minimization
    * These systems are capable of optimizing as much as we want, so we need to specify the correct level of optimization
- Note that state machine minimization does not necessarily get us faster code; the smaller state diagram can lead to faster execution, or less complexity and code size, etc