1 Logistics

The first part of the course will borrow heavily from Milner's book *Communicating and Mobile Systems: The Calculus*, ISBN: 978-0521658690.

We will also be drawing from Winskel's book entitled *The Formal Semantics of Programming Languages*, ISBN: 978-0262731034.

The course number has changed from CS 7110 to CS 6112. This should be handled automatically in studentcenter.

2 Introduction

It would not be an CS/ECE course if we did not mention Moore's Law. It has enabled the current advances in cloud and mobile computing. However, as the number of available transistors rises exponentially with time, we are rapidly outstripping our ability to perform useful serial computation in a power-efficient manner. As a result, we need to think about doing computation in a distributed, i.e. concurrent, manner.

This has the added implication that Von Neumann architectures are becoming more and more difficult to work with.

2.1 Topics

We will be covering a few different languages, namely Hoare's CSP and Milner's CCS, which essentially is CSP with the addition of concurrency. Eventually we will work our way up to π-Calculus, which is a powerful tool for expressing the kinds of programs we are interested in.

THe first part of class, we will offer a brief history of the ideas regarding concurrency. The aforementioned π-Calculus, for example, treats systems as a sea of interacting processes. It is most-often used in theory as opposed to practice, as it is well-known and understood by academics.

A related idea is bisimulations, which–loosely–is the idea of a pair of processes being able to simulate each other's states. This implies that the processes are equivalent. Bisimulation is constructed with co-inductance.

The second part of the class, we will cover the current state-of-the-art in concurrency languages.

2.2 Projects

Part of the requirements for this class is that the students undertake a group project, either individually or in groups of two to three. Some project ideas include: a literature survey or hacking on concurrency tools, but any concurrency-related project approved by the instructor is acceptable.

3 A Sequential Model of Computation

Before addressing concurrency, we will first start by examining a sequential model of computation using the language IMP, or the IMPerative Language. IMP can be thought of as a stripped-down version of the C langauge we are familiar with.

To describe the language, we will be using Backus-Naur Form (BNF), a discussion of which can be found on Wikipedia.

3.1 IMP

We will divide the BNF of IMP into 3 different syntatic classes: a for arithmetic expressions, b for boolean expressions, and c for commands.

$$a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2$$

a here is made up of x, which represents a variable, n, which is any integer, the addition operation, and the multiplication operation.

$$b ::= true \mid false \mid a_1 < a_2$$

b contains the true/false primitives as well as the inequality operation.

$$c ::= \text{skip} \mid x := a \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$$

c allows for a skip (or NOP), assignment statement, sequential composition with the semi-colon, if statements, and while loops.

Some quick examples follow. An infinite loop of nothing:

$$\text{while true do skip}$$

NOP, then set x to the arithmetic expression a:

$$\text{skip}; x := a$$

3.2 Transitions

We'll first introduce a new syntax which describes the configuration of a process. We will use tuples of the current state σ, and the current command c. σ is essentially a map from all the variables used in the process to their integer values.

Thus, to represent an execution of a program, we would write the following:

$$< \sigma, c > \rightarrow^* < \sigma', \text{skip} >$$

We start at state σ with the command c. We then do many transitions, represented by the $\rightarrow^*$, and end up in the final configuration where we have some final state σ' and the command is skip–in this context skip implies the termination of the program.

To examine a single transition from one configuration to another, we use the following syntax:

$$< \sigma, c > \rightarrow < \sigma', c' >$$

The primes denote that we have transitioned to a new state σ' and the command has changed to c'–as we've just executed something and have moved onto the next command.

3.3 Rules

In order to formalize the transition system, we require some rules with which to govern the system. In order to ensure clarity, we provide a short discussion of the syntax.

Rules have the following structure:

$$\frac{\text{Assumption}}{\text{Rule}}\text{Rule Name}$$

The Rule Name is written to the right and serves no other purpose but as a unique identifier for referencing the rules in proofs or discussion. It may be omitted if desired. The Rule itself is the concept which we are concerned with. In this context, it could be a transition as described earlier in Section 3.2. The Assumption is just that, the assumption(s) under which the rule is true. Multiple assumptions are separated by whitespace.

Note that in this syntax, we can also have axioms, i.e. rules without assumptions. They are denoted as follows:

$$\frac{}{\text{Rule}}\text{Example Axiom}$$

We can also chain these structures together, building what's known as a *proof tree*, which begins with a rule and ends at axioms. Such a structure would look something like this:

$$\frac{\overline{R1}\ \overline{R2}}{R0}\text{Rule to Prove}$$

Note that both rules $R1$ and $R2$ happen to be axioms, but if they were not, one could build a proof tree which fanned out until the leaves were all axioms.

Now we can example the rules for IMP, starting with the arithmetic expression evaluation rule.

$$\frac{< \sigma, a > \rightarrow_a < \sigma, a' >}{< \sigma, x := a > \rightarrow_c < \sigma, x := a' >}$$

This rule states that assuming we can evaluate or simplify the expression a to a new expression a', we can replace an a in an assignment command with a' as in the rule below. One thing to note before we continue is the subscripts a and c on the transition arrows. They denote that we are operating either on the arithmetic expression a or the command, which in this case is $x := a$.

Now, to actually perform the assignment, we'll need to update the state, σ. We can write an axiom for this:

$$\frac{}{< \sigma, x := n > \rightarrow_c < \sigma[x \rightarrow n], \text{skip} >}$$

This rule essentially is the description of how the assignment statement $x := n$ is performed. We set the variable x in the state σ to the integer value n, and we terminate the command–it becomes a skip.

We'd also be able to like to sequentially compose commands. The simple rule for this is the following axiom, which states that if we have a skip followed by a command c, we can safely drop the skip. Here it is formally:

$$\frac{}{< \sigma, \text{skip}; c > \rightarrow_c < \sigma, c >}$$

The more complicated case is the sequential composition $c_1; c_2$, which we represent here:

$$\frac{< \sigma, c_1 > \rightarrow_c < \sigma', c_1' >}{< \sigma, c_1; c_2 > \rightarrow_c < \sigma', c_1'; c_2 >}$$

In plain English, what this says is as follows: if command c_1 can make forward progress to c_1', potentially modifying the state to go from σ to σ', then we can safely say that in the sequential composition $c_1; c_2$, c_1 can safely make progress in that context as well in a similar fashion.

We'll provide one more example of a while loop below:

$$\frac{}{< \sigma, \text{while } b \text{ do } c > \rightarrow_c < \sigma, \text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip} >}$$

Essentially this formalizes a recursive construction of a while loop. Each iteration is broken out into an if statement which either does the command c and calls while again, or terminates if b is false.

4 Concurrency

It behooves us to clarify the difference between the words *parallel* and *concurrent* at this point, as they are different. *Parallel* tasks truly execute simultaneously, whereas *concurrent* tasks are non-deterministically interleaved but may or may not not be executing at the same time.

To describe this in the syntax we have just been discussion, what if we extended our syntactic class for commands, c, with parallel composition–expressed as $c_1 \parallel c_2$–as follows?

$$c ::= \text{skip} \mid x := a \mid c_1; c_2 \mid c_1 \parallel c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$$

We can then add the rules for parallel composition to our ruleset.

$$\frac{< \sigma, c_1 > \rightarrow_c < \sigma', c_1' >}{< \sigma, c_1 \parallel c_2 > \rightarrow_c < \sigma', c_1' \parallel c_2 >}$$

$$\frac{< \sigma, c_2 > \rightarrow_c < \sigma', c_2' >}{< \sigma, c_1 \parallel c_2 > \rightarrow_c < \sigma', c_1 \parallel c_2' >}$$

Much like our sequential operation rules before, all these rules state is that if a command c is allowed to progress on its own and modify the state σ, it is also allowed to progress in a parallel composition.

However, what is interesting is that depending on whether c_1 or c_2 makes progress first and modifies the state σ, we can get non-deterministic behavior as the sub-commands of c_1 and c_2 can be interleaved non-deterministically.

5 Communication

In the IMP language, there is no notion of explicit communication. In order to communicate between processes, we are forced to use shared variables in the state σ.

Hoare's CSP extends the system we have just described with communication channels.

5.1 CSP

We define a set of channels to range over the Greek letters, i.e. $\alpha, \beta, \gamma \in \text{Channels}$.

We now would like to add some channel-related syntax.

$\alpha!a$ sends the evaluated arithmetic expression a over the channel α.

$\alpha?x$ receives a value from the channel α and stores it in the variable x.

$c \backslash \alpha$, which means that the channel α only exists in the context of command c. In other words, α is in the namespace of the command c.

In the following example, the first α is different than the second and third α's, which are the same.

$$(c \backslash \alpha) \parallel \alpha!n \parallel \alpha?n$$

Now to formally declare CSP in BNF. We'll start with the familiar syntatic classes: arithmetic expressions–a, boolean expressions–b, commands–c, and we'll now add guarded commands–g, and channel actions–λ.

$$a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2$$

$$b ::= true | false | a_1 < a_2$$

$$c ::= \text{skip} \mid x := a \mid c_1; c_2 \mid \text{if } g \text{ fi} \mid \text{do } g \text{ od} \mid c_1 \parallel c_2$$

$$g ::= b \rightarrow c \mid b \wedge \alpha?x \rightarrow c \mid b \wedge \alpha!a \rightarrow c \mid g_1 [\!] g_2$$

For the expression $b \rightarrow c$ denotes that the boolean b guards the execution of the command c. In other words, c will not execute unless b evaluates to true. This is very similar to Dijkstra's guarded commands.

The next two expressions denote guarded commands as well, but with the addition that the boolean expression b must evaluate to true *and* a channel receive or send must occur, respectively.

The final $g_1 [\!] g_2$ represents a non-deterministic choice between guarded commands g_1 and g_2. If both are capable of firing, choose one non-deterministically. If only one is capable of firing, choose that one.

$$\lambda ::= \epsilon \mid \alpha!n \mid \alpha?n$$

Channel actions can comprise nothing–ϵ, a send, or a receive.

A transition involving a channel action is then written as follows–note that an if it's an empty channel action, or when $\lambda = \epsilon$, the λ can be omitted above the arrow:

$$< \sigma, c > \xrightarrow{\lambda} < \sigma, c' >$$

We'll need some rules for channels now, such as how to update the state, σ, during a channel receive action.

$$\frac{}{< \sigma, \alpha?x > \xrightarrow{\alpha?n} < \sigma[x \mapsto n], \text{skip} >}$$

And very quickly, let's look at one final rule, where we synchronize a parallel composition with a channel action:

$$\frac{< \sigma, c_1 > \xrightarrow{\alpha!n} < \sigma, c_1' > \quad < \sigma, c_2 > \xrightarrow{\alpha?n} < \sigma', c_2' >}{< \sigma, c_1 \parallel c_2 > \rightarrow < \sigma', c_1' \parallel c_2' >}$$