Introduction to “CSP theory” track

Andrzej Filinski
DIKU

Extreme Multiprogramming 1
November 19, 2013

Motivation: writing (correct) concurrent programs is **hard**

- **Factoid:** development costs of a major new airplane model (A380, B787) run in the billions of dollars. Typical breakdown:
  - Materials and mechanical engineering: 20%
  - Avionics (= flight control and navigation software) *design* and *implementation*: 30%
  - Avionics *verification* and *certification*: 50%
- We need all the help we can get from a sensible programming framework!
- CSP is one of Tony Hoare's many great contributions to CS.
  - PCSD'ers: find and read "The emperor's old clothes", Hoare's Turing Award lecture, CACM 1981.
- This course: develop *a sense* of formal reasoning about concurrent processes.

"CSP theory" track of XMP

- Approx. 6 lectures, covering mainly chapters 1–4 of Tony Hoare's *Communicating Sequential Processes* (free book).
  - Supplementary reading: Bill Roscoe's *Theory and Practice of Concurrency* (also freely available).
    - A bit more formal, but also mathematically heavier.
- **Goal**: familiarity with main concepts of CSP:
  - notation/syntax,
  - semantics, and
  - reasoning principles.
- **Non-goal** (for this course): understanding the mathematical foundations of the formalism.
  - Hoare worked very hard so that you won't have to
  - ... but you'll still need to put in a fair bit of effort.
- There will be several "paper" exercises, complementing the programming ones.

Concurrency $\neq$ parallelism

Often confused in informal usage, but good reason to distinguish:

- **Parallelism** / multi*processing*: focus on *computation*.
  - Goal: reduce wall-clock time to obtain result.
  - Inherently involves multiple computation units.
- **Concurrency** / multi*programming*: focus on *communication*.
  - Goal: organize program/system by logical activities.
  - May well be implemented on single processor by time slicing.
- Naturally some overlap:
  - A parallel algorithm *may* be expressed using concurrency primitives.
  - A concurrent program *may* run faster on parallel hardware.

  But neither concept presupposes the other.
- Our focus is on concurrent programming; parallel algorithms and parallel hardware are topics for other courses.

Approaches to concurrency

- Shared-state-oriented
  - Essentially traditional, sequential computation model, extended with low-level thread primitives/library.
  - Threads access shared data, protected by locks, signals, ...
  - Seemingly small conceptual up-front cost, but obscures inherent complexity (nondeterminism, deadlocks, ...)
  - Very involved to reason formally about.
- **Message-oriented**
  - Concurrent organization is main structuring principle.
  - No implicitly shared data; communication is by explicit message exchange *only*.
  - Requires some mental adjustment, much like step from imperative to functional programming.
  - But considerable payoff: formal reasoning *much* simplified.
  - Bonus: scales easily to physically distributed systems.

Perspectives

- CSP is not a concrete programming language, but a general programming model:
  - Can be extended to a complete language (e.g. Occam)
  - Or embedded into an existing one (e.g. Java CSP library)
    - Requires some programmer discipline to reap full reasoning benefits.
- Actually, CSP is even more than that:
  - A description tool for concurrent systems at higher levels of organization than concrete code.
  - An algebraic framework for reasoning about program equivalence.
  - A conceptual vocabulary for (human) communication about concurrency.
  - A baseline for many other concurrency formalisms (π-calculus, join-calculus, etc.): "like CSP, except ..."

Processes and events

- A **process** is an autonomous, "black box" unit of behavior; may interact with its environment by participating in **events**.
  - Running example: *vending machine*. Main events are accepting coins and dispensing products, with various refinements (additional events, complex/buggy behaviors, ...)
  - The *customer* can be modeled as another process.
- An event is a conceptually *atomic* action, but may require active participation (or at least acceptance) from multiple processes.
  - A successful coin insertion requires both that the customer is willing to insert coin and that machine's coin slot is open.
  - Successfully dispensing a product includes that the customer retrieves it!
  - Other events may be unilateral (e.g., machine making a noise).

Alphabets

- The set of events a process P may *conceptually* participate in is called its **alphabet**, written αP.
  - **Ex:** alphabet of vending machine: $\{coin, coke, sprite, noise\}$.
  - **Ex:** alphabet of customer: $\{coin, coke, sprite, drink, talk, \ldots\}$
- Roughly like a *type* in most programming languages. *Prescriptive*, not merely *descriptive*.
  - Combining processes with incompatible alphabets may be statically disallowed ("type error").
  - The *meaning* (observable behavior) of a composition may depend crucially on the declared alphabets of two processes!
- For now, alphabets are just unstructured sets; later they will be organized into communications over named **channels**.

Basic process syntax

- **Convention:** let x, y, z range over individual events, and A, B, C over *sets* of events.
- Grammar of processes:

$$P \quad ::= \quad STOP \mid x \to P \mid \cdots$$

- $STOP$ ("deadlock") is the completely inactive process: refuses to participate in any events in its alphabet.
- $x \to P$ (prefixing, "x then P") is the process that first engages in (only) x and then behaves like P.
  - **Ex:** $VM = coin \to noise \to coke \to STOP$

Choice

- *Simple* choice: process that may engage in one of several events, then continue in different ways:
  - $P ::= \cdots \mid (x_1 \to P_1 \mid \cdots \mid x_n \to P_n)$
  - Requires all x_i distinct, all P_i have same alphabet. .
  - **Ex:** $VMC = coin \to (coke \to STOP \mid sprite \to STOP)$.
  - Note: environment (customer) participates in selection.

- Later: *general* choice:
  - $P ::= \cdots \mid P_1 \Box P_2$, for $\alpha P_1 = \alpha P_2$.
  - Behaves either like P_1 or P_2, once the choice is made.
  - Chooses "intelligently", based on first events in P_1 and P_2
  - Does *not* require that initial events in P_1 and P_2 disjoint.
    - $\Rightarrow$ introduces *nondeterminism* if there is overlap.

Equational laws

- CSP is not only a language, but a *process algebra*.
- Several syntactically different terms may have exactly the same *meaning*. **Examples:**
  - Arithmetic: terms represent numbers, laws include $(x + y) + z = x + (y + z)$.
  - Functional programming: terms (of functional type) represent [partial] functions, laws include $(h \circ g) \circ f = h \circ (g \circ f)$.
  - CSP: terms represent processes, laws include $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$.
- CSP comes with a very powerful collection of laws for proving equivalence of various processes.
- XMP course: focus on *using* the laws. (Semantics and Types: techniques for *proving* such laws.)

Recursion

- Used mainly for expressing *loops*, rather than true recursion.
  - Like *tail recursion* in functional programming.
- Let X range over process *names*. Then processes may be defined by a system of mutually recursive definitions:

$$X_1 \triangleq P_1, \ldots, X_n \triangleq P_n$$

  where each P_i may use $X_1, ..., X_n$ as additional processes.
  - Caution: CSP book writes just $=$ for such definitions as well
  - **Ex:** $VMC \triangleq coin \to (coke \to VMC \mid sprite \to VMC)$.
  - Like defining top-level recursive functions in ML or Haskell
- Alternative, equivalent syntax: $P ::= \cdots \mid \mu X.\, P$.
  - **Ex:** $VMC = \mu X.\, coin \to (coke \to X \mid sprite \to X)$.

Concurrent composition

- $P ::= \cdots \mid P_1 \| P_2, \qquad \alpha P = \alpha P_1 \cup \alpha P_2.$
- P can engage in event x when:
  - $x \in \alpha P_1$, $x \in \alpha P_2$, and both P_1 and P_2 can engage in x, or
  - $x \in \alpha P_1$, $x \notin \alpha P_2$, and P_1 can engage in x, or
  - $x \notin \alpha P_1$, $x \in \alpha P_2$, and P_2 can engage in x.
- **Ex:** consider definitions:

$$\begin{aligned} VMC &\triangleq coin \to noise \to (coke \to VMC \mid sprite \to VMC) \\ CUST &\triangleq coin \to coke \to drink \to CUST \end{aligned}$$

  Then $VMC \| CUST = \mu X.coin \to noise \to coke \to drink \to X$.
- **Note:** If we had taken $\alpha CUST = \{..., noise, ...\}$ (customer can hear and potentially react to noise), we would get,

  $VMC \| CUST = coin \to STOP$ (deadlock!)

Concealment

- Remember: C ranges over sets of events.
- $P ::= \cdots \mid P \setminus C$
  - $\alpha(P \setminus C) = (\alpha P) - C \quad (A - B = \{x \mid x \in A \wedge x \notin B\})$
  - If P wants to engage in event $x \in C$, it will happen silently and asynchronously (more nondeterminism!).
  - If P wants to engage in event $x \notin C$, must synchronize with environment as usual.
- Common idiom: $(P_1 \parallel P_2) \setminus \{x\}$
  - Allows P_1 and P_2 to synchronize internally on event x, but hides this interaction from environment ("private channel").
- **Ex:** $(\mathit{VMC} \| \mathit{CUST}) \setminus \{\mathit{coin}, \mathit{noise}\} = \mu X.\, \mathit{coke} \rightarrow \mathit{drink} \rightarrow X.$
  - Environment can observe coke dispensing and drinking, but not the coin deposit or the noise.

Communication

- Specialize general theory by partitioning events into *channels*: sets of events of same kind, but still differing in attributes.
- **Ex:** multiple coins and bottle sizes
  $VM2 \triangleq (coin.10 \to coke.\frac{1}{2} \to VM2 \mid coin.20 \to coke.1 \to VM2)$
  $\alpha VM2 = \{coin.10, coin.20, coke.\frac{1}{2}, coke.1\}$
  (I.e., coke is 20 kr per ℓ.)
- *coin* and *coke* are channel *names*, and the numbers are *values* transmitted over the channels.
- We write αc for the alphabet of the channel c. Here, $\alpha coin = \{10, 20\}$ and $\alpha coke = \{\frac{1}{2}, 1\}$.
- Could then express the process (or its generalization to arbitrary amounts) more concisely as:

$$VM2 \triangleq coin?v \to coke!(\frac{1}{20} \cdot v) \to VM2$$

Communication, more formally

- Let c range over channel names and v over variable names. Also let E be a syntactic class of simple expressions:

$$E ::= n \mid v \mid E_1 + E_2 \mid \cdots \qquad (n \text{ ranges over numerals})$$

- We then introduce *output* and *input* operations:

$$P ::= \cdots \mid c!E \to P_1 \mid c?v \to P_2(v)$$

  where the variable v may occur inside expressions of P_2.
  - Binding vs. assignment.

- These are conceptually abbreviations for prefixing and *infinitary* choice:

$$\begin{aligned} c!E \to P &= c.n \to P, \quad \text{where } n \text{ is the value of } E \\ c?v \to P(v) &= (c.0 \to P(0) \mid c.1 \to P(1) \mid \cdots) \end{aligned}$$

Process-local state

- A recursive process definition can also have *parameters*, to maintain variable values across iterations:

$$X_{v_1,\ldots,v_n} \triangleq \cdots \to X_{E_1,\ldots,E_n}$$

(Nominally, an infinite *family* of process definitions.)

- **Ex:** stateless process:
  $DOUBLE \triangleq in?v \to out!(v+v) \to DOUBLE$
  - For each n received on in, send $2 \cdot n$ on out
- **Ex:** stateful process:
  $ACCUM \triangleq ACC_0$
  $ACC_a \triangleq in?v \to out!(a+v) \to ACC_{a+v}$.
  - For each n received on in, add to running total a, and also report that total on out.
- (When CSP embedded in imperative language with loops, local state is usually kept in ordinary, assignable variables.)

Communication networks

- Convention: channels are always unidirectional links between exactly two processes, that agree on the alphabet of the channel.
- Can build large process networks out of simple components by parallel composition, concealment, and channel renaming (not discussed above).
- Typical "Lego bricks":
  - $DELTA \triangleq in?v \rightarrow out_1!v \rightarrow out_2!v \rightarrow DELTA$
  - $DEMUX \triangleq in?v \rightarrow (out_1!v \rightarrow DEMUX \mid out_2!v \rightarrow DEMUX)$
  - $ZIP \triangleq in_1?v_1 \rightarrow in_2?v_2 \rightarrow out!(v_1, v_2) \rightarrow ZIP$
  - $MUX \triangleq (in_1?v \rightarrow out!v \rightarrow MUX \mid in_2?v \rightarrow out!v \rightarrow MUX)$
  - $NATS \triangleq FROM_0$, $FROM_v \triangleq out!v \rightarrow FROM_{v+1}$
  - $\vdots$

Overview of CSP book

1. Processes
2. Concurrency
3. Nondeterminism
4. Communication
5. (Sequential processes)
6. (Shared resources)
7. (Discussion)

Logical progression, but means that some concepts only introduced quite late in "theory track"; will probably see them in "programming track" first.

- For next time: read Chapter 1, and try to understand all the examples. You may skip the "implementation" sections. Allocate **at least** a couple of hours.