Comp 411
Principles of Programming Languages
Lecture 10
The Semantics of Recursion

Corky Cartwright

February 12, 2021

Key Intuitions

- Computation is incremental not monolithic
- Slogan: general computation is successive approximation (typically in response to successive demand for more information). In simple computations, only the standard output stream is repeatedly demanded until EOF (end-of-file) datum is encountered.

Key Mathematical Concepts

- A *partial order* (**po**) is a set S with a *reflexive*, *transitive*, *anti-symmetric* binary relation ≤. See Wikipedia for a complete definition.
- A *chain* in a **po** is a countable *totally ordered* set $c_0 \leq c_1 \leq c_2 \ldots \leq c_k \leq \ldots$ See Wikipedia for the definition of a *countable set*, which may be empty.
- A **po** is *chain-complete* iff every chain has a least upper bound (LUB) in the **po**. Such a partial order is called a *complete partial order* (**cpo**). Since a chain can be empty, every **cpo** must have a least element, which we denote by the symbol ⊥, called "bottom". In the domain theory monograph, *directed sets* are used instead of chains; it is easy to prove the two notions are equivalent for domains with a countable basis (defined below). We are only interested in **cpo**s with countable bases (because we are computer scientists!).
- A subset S within a **po** is *consistent* iff it has an upper bound in the **po**.
- A **po** is *finitely consistent* if every finite subset has a LUB.
- A *finitary basis* is a countable **po** in which every finite consistent set has a LUB.

Key Mathematical Concepts

Semantic Domains II

- Given a finitary basis **B**, the (*Scott*) *domain determined by* **B** is the **cpo** created by adding LUBs for infinite chains in **B**. The elements of B are called the *finite* elements of this domain. The monograph contains an explicit construction of this domain using *ideals*. The intuition is simple: the generated domain simply adds an element for each infinite chain of finite elements that is *only* above all elements in the downward closure of the chain. Note that several different chains may have the same LUB.
- Given any subset S of a domain **D**, the downward closure S↓ of S is the set of all elements of **D** less than some element of S. Two chains are equivalent if their downward closures are identical.
- The *topologically finite* elements of the **cpo** determined by **B** are precisely the elements of **B**. (Don't worry about the definition of *topologically finite*; it is defined in the monograph.)

More Mathematical Details

All (incrementally) computable functions f mapping domain **A** into domain **B** are:

- *monotonic*: $x \leq y \Rightarrow f(x) \leq f(y)$
- *continuous*: given a chain $C = \{c_i \mid i \in \mathrm{N}\}$, $f(\sqcup C) = \sqcup \{f(c) \mid c \in C\}$

Note that a continuous function may not be computable. (Consider the *natural extension* [as defined below] of any conventional function mapping N into N that is not recursive.)

In practical programming languages, all primitive and library functions f are *strict*, *i.e.*, f maps $\perp$ to $\perp$. If f is n-ary ($n > 1$), f is *strict* iff $f(x_1,\ldots, x_n) = \perp$ if any input is $\perp$.

Note: the *if-then-else* construct is not classified as a primitive function because it is not strict!

Excluding function domains, the data domains supported by most programming languages are *flat*: every element $d \in \mathbf{D}$ except $\perp$ is *finite* and *maximal*. Some examples include integers, booleans, strings, structures, arrays of structures, *etc*. All conventional data values including finite trees, lists, and tables are flat because every conventional data constructor is *strict*; no embedded elements can be $\perp$. Consider some unary total function g on the natural numbers that is not recursive (computable). In domain theory, there is a simple function corresponding to g over the flat domain of natural numbers called the *natural extension* of g where $g(\perp) = \perp$. This function is monotonic and continuous but it is not computable. In languages supporting the lazy construction of objects (structures), the data domains corresponding to lazy constructions are *not* flat, because each lazy argument (subtree) in a construction can be an element of the domain designated for that argument. If the argument can be a tree, then infinite trees can be constructed. Function domains are obviously not flat.

More Mathematical Details cont.

- Excluding function domains, the data domains **D** supported by most programming languages are *flat*: every element $d \in$ **D** except $\perp$ is *finite* and *maximal*. Some examples include integers, booleans, strings, structures, arrays of structures, *etc*. All conventional data values including finite trees, lists, and tables are flat because every conventional data constructor is *strict*; no embedded elements can be $\perp$.
- Function domains are obviously not flat since approximation is defined point-wise on functions and functions can diverge on converge on particular inputs.
- Consider some unary total function ***g*** on the natural numbers that is *not* recursive (computable). In domain theory, there is a simple function corresponding to ***g*** over the flat domain of natural numbers called the *natural extension* ***g'*** of ***g*** where ***g'***$(\perp) = \perp$. This function is monotonic and continuous but it is not computable.
- In languages supporting the lazy construction of objects (structures), the data domains corresponding to lazy constructions are *not* flat, because each lazy argument (subtree) in a construction can be an element of the domain designated for that argument. If the argument can be a tree, then infinite trees can be constructed.

Some Domain Examples

- Flat domains like N, Z, arrays of flat domains.
- Strict function spaces ($\mathbf{A} \blacktriangleright \mathbf{B}$) on flat domains $\mathbf{A}$ and $\mathbf{B}$. Note: the notation $\blacktriangleright$ is non-standard; more typically some form of modified right arrow is used.
- Strict function spaces ($\mathbf{A} \blacktriangleright \mathbf{B}$) mapping a domain $\mathbf{A}$ into domain $\mathbf{B}$.
- Non-strict function spaces (call-by-name!) $\mathbf{D} \to \mathbf{D}$ and $(\mathbf{D} \to \mathbf{D})_\perp$.
  The non-strict functions in Jam (as we implement call-by-name) do *not* form the simple space $\mathbf{D} \to \mathbf{D}$, but rather $(\mathbf{D} \to \mathbf{D})_\perp$ which adds a distinct $\perp$ element that is not a function! The reduction semantics required to support $\mathbf{D} \to \mathbf{D}$ is wasteful because it must reduce inside lambda-abstractions because
- Lazy binary trees of booleans
- Lazy abstract syntax trees (infinite programs!)
- Continuous functions from domain $\mathbf{A}$ into domain $\mathbf{B}$, denoted $\mathbf{A} \to \mathbf{B}$
- What if domain $\mathbf{A}^+$ contains $\mathbf{A}$ and domain $\mathbf{B}$ contains $\mathbf{B}^-$?
- What is relationship between $\mathbf{A} \to \mathbf{B}$ and $\mathbf{A}^+ \to \mathbf{B}^-$? The latter is a subset of the former.
- The continuous function domain constructor $\to$ is co-variant in its second argument (the *co-domain*) and contra-variant in its first argument (the *domain*).

A Bigger Challenge

- In an earlier lecture, we posed and solved the minor challenge of how to modify our meta-interpreter to support the recursive generalization of **`let`**, which is particularly interesting if we only consider purely functional meta-interpreters.
- Lets reconsider that problem but impose more restrictions on how we are allowed to solve it. Assume that we want to write an interpreter for an extension of LC (or Jam as in Assignment 2) that includes recursive binding (e.g., **`letrec`**) that simply expands the input program into an equivalent program that eliminates all uses of **`letrec`**. We are not allowed to modify our interpreter for the original (unextended) language without a recursive binding construct (say functional Scheme without **`define`** and **`letrec`**)?
- Key problem: we must expand code with **`letrec`** as a binding construct into equivalent code that only has **`lambda`** available as a binding construct.
- No simple solution to this problem. We need to devise some syntactic magic (as did the creators of the lambda-calculs or develop some sophisticated mathematical machinery (as did Dana Scott).