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

Corky Cartwright
February 1, 2019
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 \( \leq \).

- 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 \( \bot \), 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 **cpos** with countable bases.

- 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 \textit{(Scott) domain determined by} $B$ is the \textbf{cpo} created by adding LUBs for infinite chains in $B$. The elements of $B$ are called the \textit{finite} elements of this domain. The monograph contains an explicit construction of this domain using \textit{ideals}. The intuition is simple: the generated domain simply adds an element for each infinite chain of finite elements that is \textit{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 $S\downarrow$ 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 \textit{topologically finite} elements of the \textbf{cpo} determined by $B$ are precisely the elements of $B$. (Don’t worry about the definition of \textit{topologically finite}; it is defined in the monograph.)
Key Mathematical Concepts

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 $C = \{c_i | i \in \mathbb{N}\}$, $f(\sqcup C) = \sqcup \{f(c) | c \in C\}$

Note that a continuous function may not be computable.

The domains, other than function domains, supported by most programming languages are **flat**: every element $d \in D$ except $\bot$ 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 $\bot$.

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 $g(\bot) = \bot$. 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
- Strict function spaces on flat domains
- Non-strict function spaces (call-by-name!)
- Lazy binary trees of boolean
- Lazy abstract syntax trees
- Functions from domain \( A \) into domain \( B \)
- What if domain \( A^+ \) contains \( A \) and \( B \) contains domain \( B^- \) ?
- What is relationship between \( A \rightarrow B \) and \( A^+ \rightarrow B^- \) ?
- Contra-variance vs. co-variance
A Bigger Challenge

- Assume that we want to write a meta-interpreter for LC in a purely functional language without a recursive binding construct (say functional Scheme without `define` and `letrec` [recursive binding as in Java methods])?
- Key problem: must expand `letrec` into `lambda`.
- No simple solution to this problem. We need to invoke syntactic magic or develop some sophisticated mathematical machinery.