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

Corky Cartwright

February 1, 2023

Key Intuitions of Domain Theory

- Computation is incremental not monolithic
- Slogan: general computation is successive approximation–typically in response to successive demand for more information from the context that is applying the function).
  In simple computations, only the standard output stream is repeatedly demanded until EOF (end-of-file) datum is encountered. Think Of a function that returns the infinite stream of primes in ascending order.

Key Mathematical Concepts

- A *partial order* (**po**) is a set S with a *reflexive*, *transitive*, *anti-symmetric* binary relation $\leq$. See Wikipedia for a complete definition.
- A *chain* in a **po** is a countable *totally ordered* set $c_0 \sqsubseteq c_1 \sqsubseteq c_2 \ldots \sqsubseteq c_k \sqsubseteq \ldots$. 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 $\perp$, 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.

Mathematical Details

- 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. It is comparatively unimportant since we will always construct domains using a finitary basis.)

More Mathematical Details

- All (incrementally) computable functions *f* mapping domain **A** into domain **B** are:
  - *monotonic*: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ [information preserving]
  - *continuous*: given a chain $C = \{c_i \mid i \in \mathrm{N}\}$, $f(\sqcup C) = \sqcup \{f(c) \mid c \in C\}$
- All computable functions are continuous, but the converse is false. A continuous function may not be computable as shown on the next slide.
- A function *f* mapping domain A into domain B is *strict* iff *f* maps ⊥ to ⊥. A function *f* mapping $A_1 \times \ldots \times A_n$ into B is *strict* iff $f(x_1, \ldots, x_n) = \bot$ when any input is ⊥.
- In practical programming languages, all primitive and library functions *f* are *strict* with the exception of control operators like *if-then-else*, *and*, and *or* which are typically not classified as primitive functions [values, constant expressions]. (In the preceding statement, the set of "practical programming languages" *excludes* Haskell.

More Mathematical Details

- 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 including $\perp$. If the argument can be a tree, then infinite trees can be constructed. Function domains are obviously not flat.
- **Lemma**: the only non-strict functions mapping flat domain $\mathbf{A}$ into flat domain $\mathbf{B}$ are constant.

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 $\perp$. If data structures can contain $\perp$ (unevaluated elements), then **D** contains non-trivial chains.
- Non-trivial function domains **A** $\longrightarrow$ **B** (where **A** and **B** are domains, $|\mathbf{A}| > 1$ and $|\mathbf{B}| > 0$) obviously are not flat since approximation is defined point-wise on functions and functions can diverge or 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 with lazy constructors are *not* flat, because lazy arguments can be $\perp$.. If the lazy argument has recursive type including a lazy constructor, then infinite trees can be constructed.

azy Binary Trees: a Domain with Infinite Elements

he domain $\mathbb{BT}\langle\mathbb{T}\rangle$ of lazy binary trees with leaf type $\mathbb{T}$ has only binary constructor that we ill denote < ·,·> (sometimes called **`cons`**) and a set of leaf elements $\mathbb{T}$. Structurally typed nguages like the ML family require leaf values to be wrapped in a unary constructor ometimes called **`leaf`**] which avoids pathologies in domains like $\mathbb{BT}\langle\mathbb{BT}\langle\mathbb{T}\rangle\rangle$. (Is a subtree a af or a branch?) For the sake of simplicity, we will use the Boolean domain $\mathbb{B}$ (a flat domain) our leaf type and dispense with the separate **`leaf`** constructor.

ike all Scott-domains, the domain of lazy binary trees $\mathbb{BT}\langle\mathbb{B}\rangle$ has a least element ⊥ rresponding to divergence. It also contains all of the elements of $\mathbb{B}$ as raw leaves.

he simplest element of $\mathbb{BT}\langle\mathbb{B}\rangle$ that is not a leaf or ⊥ is <⊥, ⊥>, although <⊥,**`true`**>, ⊥,**`false`**>, <**`true`**,⊥>, <**`false`**,⊥>, <**`true`**,**`true`**>, <**`true`**,**`false`**>, <**`false`**,**`true`**>, and **`alse`**,**`false`**> are contenders for that designation.

he approximation ordering ≤ on $\mathbb{BT}\langle\mathbb{B}\rangle$ is defined as follows.

$\perp \sqsubseteq x$, for all $x \in \mathbb{BT}\langle\mathbb{B}\rangle$.

$<x,y> \sqsubseteq <x',y'>$ iff $x \sqsubseteq x'$ and $y \sqsubseteq y'$.

nce ⊥ can be embedded anywhere inside an element of $\mathbb{BT}\langle\mathbb{B}\rangle$, elements of $\mathbb{BT}\langle\mathbb{B}\rangle$ can be finite trees (LUBs of infinite ascending chains of finite trees). There are an uncountably any elements of $\mathbb{BT}\langle\mathbb{B}\rangle$ but only countably many are computable (Why?). The real numbers ve exactly the same property. (The floating point numbers are not the reals. All conventional oating point number systems are **finite** subsets of the rational numbers!) The computable real mbers are called the *constructive* reals.

Some Domain Examples

- Flat domains like N, Z, arrays of flat domains.
- Strict function spaces ($\mathbf{A} \rightharpoonup \mathbf{B}$) on flat domains **A** and **B**. Call-by-value evaluation of function arguments forces strictness. Note: the notation $\rightharpoonup$ is non-standard. Unfortunately, no standard notation for the strict function space construction has yet developed
- Strict function spaces ($\mathbf{A} \rightharpoonup \mathbf{B}$) mapping a domain **A** into domain **B.**
- Non-strict function spaces (call-by-name!) $\mathbf{D} \rightarrow \mathbf{D}$ and $(\mathbf{D} \rightarrow \mathbf{D})_\perp$.
  The non-strict functions in Jam (if we stipulate call-by-name) do *not* form the simple space $\mathbf{D} \rightarrow \mathbf{D}$, but rather $(\mathbf{D} \rightarrow \mathbf{D})_\perp$ which adds a distinct $\perp$ element that is not a function! The reduction semantics required to support $\mathbf{D} \rightarrow \mathbf{D}$ is wasteful because evaluating such a function must separate those that are not $\perp$ somewhere from those that are $\perp$ everywhere. The evaluation of the latter obviously never terminates.
- Lazy binary trees of booleans
- Lazy abstract syntax trees (infinite programs!)
- Continuous functions from domain **A** into domain **B**, denoted $\mathbf{A} \rightarrow \mathbf{B}$
- What if domain $\mathbf{A}^+$ contains **A** and domain **B** contains $\mathbf{B}^-$?
- What is relationship between $\mathbf{A} \rightarrow \mathbf{B}$ and $\mathbf{A}^+ \rightarrow \mathbf{B}^-$? The latter is a subset of the former.
- The continuous function domain constructor $\rightarrow$ 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 *to support 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 ultimately must rely on the syntactic magic of the Y combinator (invented by the creators of the λ-calculus) — which can be explained by the sophisticated mathematical machinery [pioneered by Dana Scott] sketched in this lecture. This theory was developed much later than the λ-calculus [1970].