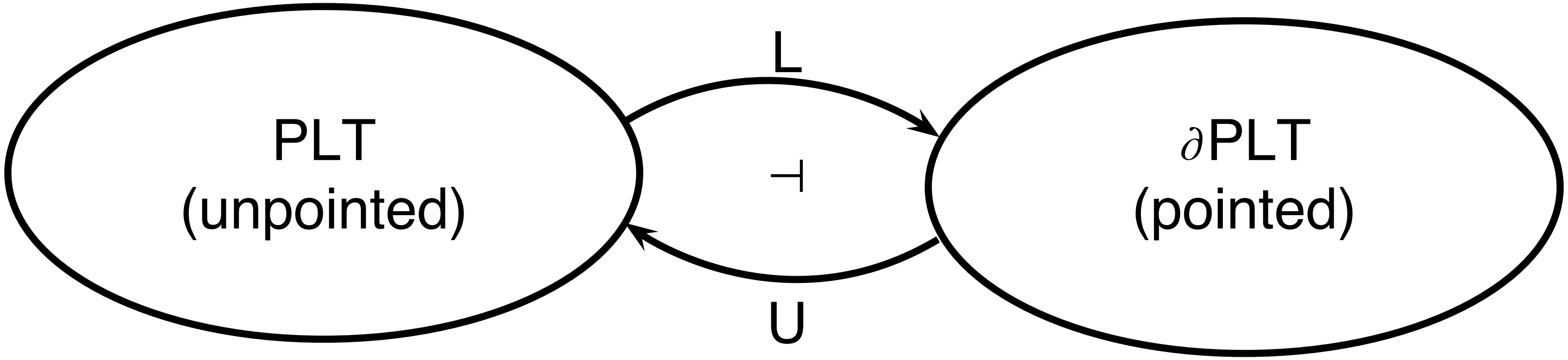
Effective, Formalized Domain Theory in Coq

Robert Dockins

`http://rwd.rdockins.name/domains/`

Summary of results



$\times$	Product
$+$	Disjoint Sum
$\Rightarrow$	Function Space
$\wp$	Powerdomains
disc(X)	Finite discrete domains
UL	Lifting monad

$\otimes$	Smash Product
$\oplus$	Coalesced Sum
$\multimap$	Strict Function Space
$\wp_{\bot}$	Powerdomains
flat(X)	Countable flat domains
µF	Recursive Domains
LU	Lifting comonad

Pointed = having a least element, ⊥, representing nontermination
Unpointed = not necessarily having a least element

What is novel?

- Theory of *algebraic domains* formalized in Coq

  Previous efforts formalize only CPOs in Coq and
  lack some standard constructions, like powerdomains

- *Fully constructive* presentation of profinite domains

  The library and examples are developed in the
  constructive metalogic of Coq using no axioms

- Two category setup (PLT/∂PLT) differs from textbook domain theory; provides some advantages

The competition: in Coq

Benton, Kennedy, Varming. "Some Domain Theory and Denotational Semantics in Coq." TPHOLS 2009.

Constructive, CPOs with *some* of the usual constructions, including recursive CPOs.

They report difficulty defining $\otimes$, and do not define powerdomains.

Coinductive ε-streams used to define lifted domains.

Their examples implicitly use axiom K via the `dependent destruction` tactic; functional extensionality is also assumed.

The competition: in Isabelle/HOL

Brian Huffman. "A Purely Definitional Universal Domain." TPHOLS 2009.

Formalized profinite domains in Isabelle/HOL, based on the construction of a universal domain. Now integrated into HOLCF.

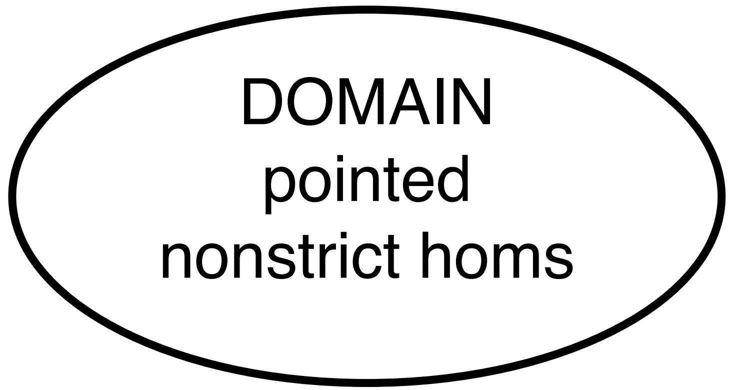
HOL is a classical logic with strong choice principles.

Different proof strategy: Huffman defines a particular universal domain and uses it to get other domains of interest.

I instead directly define the category PLT and take colimits to build recursive domains.

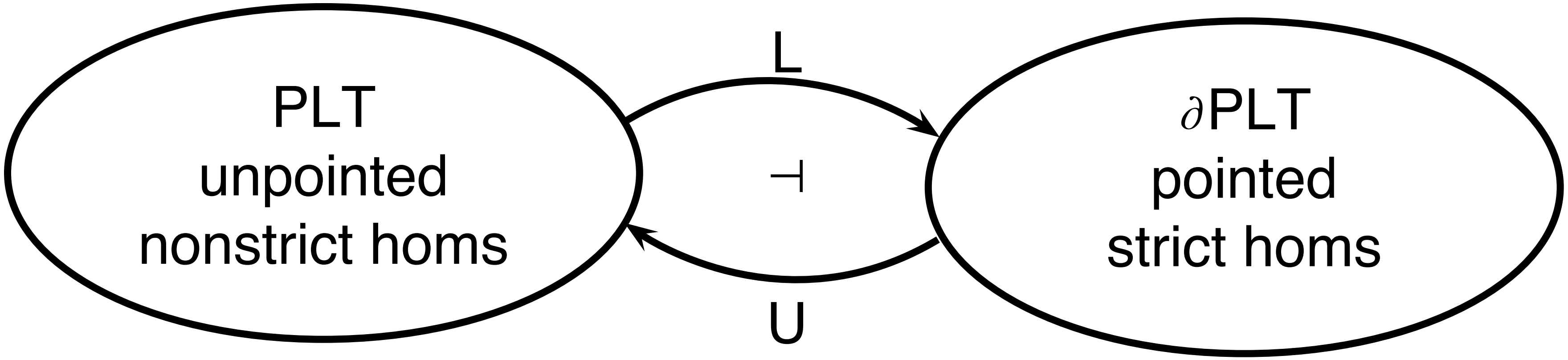
Why two categories?

Textbook presentations work with *pointed* domains and *nonstrict* continuous functions.



- This allows a general fixpoint operator.
- But, this category has strange properties, e.g. no coproducts.

Why two categories?



- Accurate semantics for normalizing calculi in PLT
- Can handle “unboxed” types and total functions
- Both categories have coproducts
- PLT is cartesian closed / ∂PLT is symmetric monoidal closed
- UL is a monad of recursion / LU is a comonad of laziness

General recursion

Problem 1: there is no fixpoint operator in PLT that applies to all domains.

Suppose A,B:PLT and let f:A → (B ⇒ B) be a PLT-hom. We cannot apply Kleene's fixpoint theorem because B might not have a least element.

General recursion

Problem 1: there is no fixpoint operator in PLT that applies to all domains.

Suppose A,B:PLT and let $f{:}A \to (B \Rightarrow B)$ be a PLT-hom. We cannot apply Kleene's fixpoint theorem beacuse B might not have a least element.

Problem 2: the fixpoint operator in ∂PLT is trivial.

Suppose A,B:∂PLT and let $f{:}A \to (B \multimap B)$ be a ∂PLT-hom. The least fixpoint of f exists; but it is always $\bot$ because f represents a strict function.

General recursion

Fact: a domain A:PLT has a least element iff A ≅ U(B) for some B:∂PLT.

Solution: fixpoints in PLT, but only for ∂PLT objects.

Let A:PLT, B:∂PLT, $f : A \to (U(B) \Rightarrow U(B))$.

Then we can construct $\mu f : A \to U(B)$, the least fixpoint of f. In particular, μf is the least hom such that:

$$\mu f = apply \circ < f, \mu f >$$

Why constructive domains?

- Philosophy: a theory of computation ought to have a constructive foundation.

- Challenge: every “useful” mathematical theory has some constructive counterpart which may be discovered if we expend sufficient effort.

- Pragmatics: a Coq library relying on no axioms can be used in any axiomatic extension of Coq, even, say, anticlassical extensions, or ones that refute axiom K (HoTT). The basic ideas should also be quite portable to other type theories.

Enumerable sets

Let <A, ≈> be a setoid.

$$\text{eset } A \;\equiv\; \mathbb{N} \rightarrow \text{option } A$$

$$x \in S \;\equiv\; \exists n\ y.\ S\ n = \text{Some } y \wedge x \approx y$$

$$\text{empty} : \text{eset } A$$
$$\text{single} : A \rightarrow \text{eset } A$$
$$\text{union} : \text{eset } A\ (\text{eset } A) \rightarrow \text{eset } A$$
$$\text{image} : (A \rightarrow B) \rightarrow \text{eset } A \rightarrow \text{eset } B$$
$$\text{union2} : \text{eset } A \rightarrow \text{eset } A \rightarrow \text{eset } A$$
$$\text{intersect2} : \text{eset } A \rightarrow \text{eset } A \rightarrow \text{eset } A\ ^*$$

$$\text{esubset } P\ S : \text{semidec } P \rightarrow \Sigma T,\ x \in T \longleftrightarrow x \in S \wedge P\ x$$
$$\text{indefinite\_description } S : (\exists x,\ x \in S) \rightarrow (\Sigma x,\ x \in S)$$

*when A has a decidable setoid equality

PLT = Effective Plotkin Orders

PLT objecsts are *effective Plotkin orders*, which have:

- an enumerable set of elements
- a decidable preorder relation
- certain closure properties based on minimal upper bounds.

PLT morphisms are *enumerable approximable relations:*

- must be enumerable as a set of pairs
- same as approx. relations as for Scott information systems

PLT ≅ effective profinite domains via ideal completions.

See the paper for definitions and details...

An Example

CBV λ-calculus with booleans and fixpoints
soundness and adequacy

http://rwd.rdockins.name/domains/html/st_lam_fix.html

Ongoing and Future Work

Ongoing work: continuous domains

Continuous domains are more general than algebraic domains.

Every continuous DCPO arises as a retract of an algebraic DCPO.

The category of retracts of PLT (∂PLT) is a well-behaved cartesian (monoidal) closed category called cPLT (c∂PLT).

The objects of cPLT can be defined as <A, r> where r: A→A is an idempotent PLT hom. This lets us reuse many constructions from PLT with very little work; likewise for c∂PLT.

Still to do: bilimits and powerdomains (no problems expected).

Ongoing work: exact real computation

A continuous (but not algebraic!) domain for exact real computation can be defined via a basis consisting of closed intervals with rational endpoints.

IR = Real Interval Domain

Domains for real numbers lets us define semantics for languages that deal with constructive real computations as first-class entities.

Future work: invariant relations

Recursively-defined domains are difficult to reason about.

Pitts' *invariant relations* allow one to define useful logical relations and induction/coinduction principles on recursive domains.

Benton et al. showed how these can be useful in formal proofs, e.g., for defining logical relations on untyped λ-calculi.

Unfortunately the definitions I want to develop this theory cause universe inconsistencies I don't understand...

☹

Perhaps universe polymorphism will save the day.

Future work: polymorphism

The domain-theoretic semantics of polymorphism requires sophisticated category-theory (indexed category theory and/or Grothendieck fibrations).

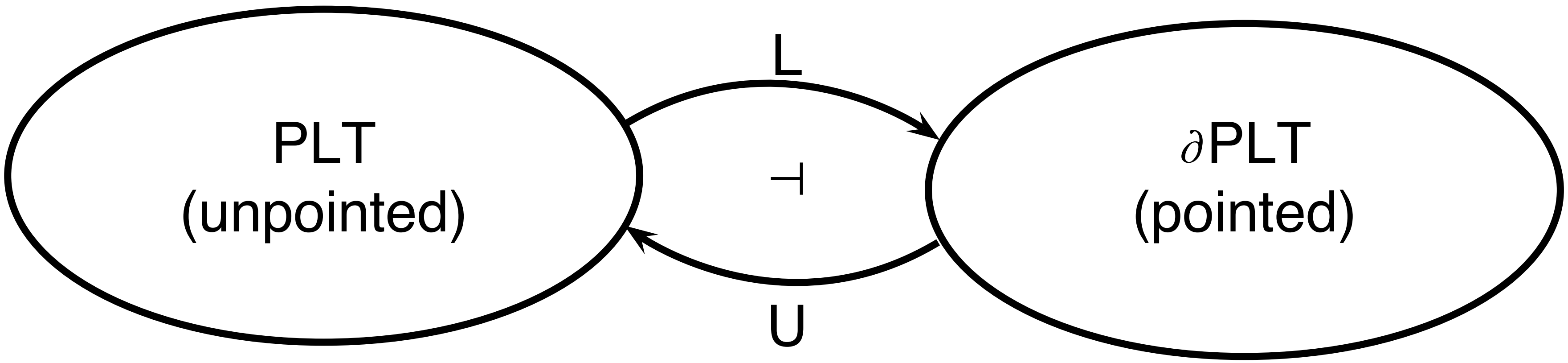
I would like to formalize a suitable category of domains satisfying parametricity properties, like those in R.E. Møgelberg's PhD thesis.

The feasibility of formalizing this work is (to me) an open question.

Thank you!

`http://rwd.rdockins.name/domains/`

Lifting and adjointness



The forgetful functor U from ∂PLT to PLT is left adjoint to the lifting functor L from PLT to ∂PLT. This adjunction is monoidal.

We thus get a monad UL in PLT: a monad of general recursion.

Likewise, we get a comonad LU in ∂PLT: a comonad of lazyness.

By passing through this adjunction, we can import constructions in one category into the other, at the cost of some “extra” bottoms.

Recursive domains

We get recursive domains from any continuous functor in ∂PLT:

- Lazy Binary Trees: $T \cong L(disc(1) + (U(T) \times U(T)))$
- Strict Binary Trees: $S \cong flat(1) \oplus (S \otimes S)$
- Untyped eager lambdas: $D \cong (D \multimap D)$
- Untyped CBV lambdas: $E \cong LU(E \multimap E)$
- Untyped CBN lambdas: $F \cong L(U(F) \Rightarrow U(F)) \cong LU(LU(F) \multimap F)$

All the operators provided (sums, products, functions, powerdomains, L, U) are continuous, and so can be used to build recursive domains.

Worked examples

The formal development contains worked examples that prove soundness and adequacy (WRT standard big-step operational semantics) for the following systems:

- Simply-typed, normalizing SKI calculus with booleans
- Simply-typed, CBN SKI+Y calculus with booleans
- Simply-typed, normalizing, named λ-calculus with booleans
- Simply-typed, CBN, named, λ-calculus with booleans and fixpoints

Future work: semantics of core Haskell

One of the original goals for starting down this path.

I'd like a semantics of core Haskell that smoothly accounts for even some of the tricky corners, especially how strict and nonstrict computation interact:

- The `seq` primitive and strictness annotations
- Unboxed types and unboxed functions
- Why is (a,b,c) not isomorphic to (a,(b,c))?
- Why does the function space have an "extra" bottom?

A core calculus with two base kinds (one for pointed and one for unpointed types) representing PLT and ∂PLT provides some answers: I'm still working out the details.