Scribe: Hongzhou Liu and Junhwan Kim Lecturer: Andrew Myers

1 Semantics of recursive types, part 2

There are two basic approaches to recursive types: iso-recursive and equi-recursive. The essential difference between them is captured in their response to a simple question: What is the relation between the type $\mu X.\tau$ and its one-step unfolding $\tau\{\mu X.\tau/x\}$?

So far we have seen the iso-recursive approach, which takes a recursive type and its unfolding as different, but isomorphic. In a system with iso-recursive types, we introduce, for each recursive type $\mu X.\tau$, a pair of functions unfold and fold that *witness the isomorphism* by mapping values back and forth between the two types:

$$\mu X.\tau \underset{\text{unfold}}{\overset{\text{fold}}{\cong}} \tau\{\mu X.\tau/X\}$$

The ML and C languages use this approach, but by implicitly introducing a fold or unfold along with other syntactic constructs. In ML, every datatype definition implicitly introduces a recursive type. Each use of a constructor to build a value of a datatype implicitly includes a fold, and each constructor appearing in a pattern match implicitly forces an unfold. For C, the * and & also signal an unfold and fold, respectively.

	ML	C
fold	constr	&
unfold	match	*

The other approach is what Pierce calls *equi-recursive* types: the two type expressions are definitionally equal—interchangeable in all contexts, because they stand for the same infinite tree.

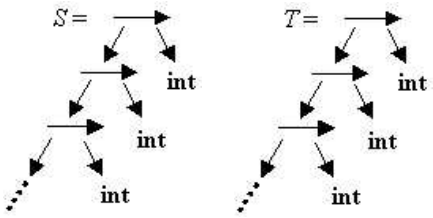
$$\vdash \mu X.\tau \cong \tau\{\mu X.\tau/X\} \tag{1}$$

Modula-3 uses this approach. In this language, a recursive type and its unfolding are substitutable. The fold and unfold operators are automatically supplied whenever needed.

For example, in a system with equi-recursive types, these two types are equivalent:

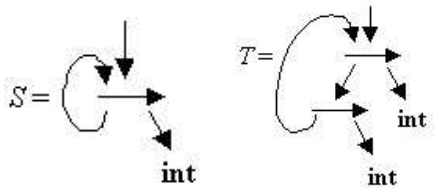
$$\mu S.S \to \mathsf{int} \cong \mu T.(T \to \mathsf{int}) \to \mathsf{int} \tag{2}$$

Since they correspond to the same infinite syntactic trees:



2 Structural equivalence

Now we have to solve this problem: In a system with equi-recursive types, when are two recursive types equivalent? As we have stated above, two types are equivalent if their infinite unfoldings are identical. Another way to think of the infinite trees above is as graphs containing loops. The μ constructor tells us where back edges are.



According to this intuition, we need a graph algorithm. The idea of this algorithm is to see if all (finite or infinite) paths in each syntactic tree are possible in the other. The algorithm goes as follow:

- Recursively walk both trees
- On back edges, record that the two nodes were reached simultaneously (and are presumably equivalent)
- If comparing two nodes and they are already recorded as equivalent, return

To implement this in a formal way, we introduce a new context E to the type equivalence system; its purpose is to record assertions about equivalent types.

$E \vdash \tau_1 \cong \tau_2$. E is defined inductively:

$$E = \emptyset \quad | \quad E, \tau_1 \cong \tau_2 \tag{3}$$

The context E records type expressions assumed to be equivalent. Two types are equivalent if $\emptyset \vdash \tau_1 \cong \tau_2$.

$$\frac{}{E \vdash \tau \cong \tau} \quad \frac{\tau_2 \cong \tau_1 \in E}{E \vdash \tau_1 \cong \tau_2} \quad \frac{E \vdash \tau_2 \cong \tau_1}{E \vdash \tau_1 \cong \tau_2} \quad \frac{E \vdash \tau_1 \cong \tau_2 \quad E \vdash \tau_2 \cong \tau_3}{E \vdash \tau_1 \cong \tau_3}$$

$$\frac{E \vdash \tau_1 \cong \tau_1' \quad E \vdash \tau_2 \cong \tau_2'}{E \vdash \tau_1 \times \tau_2 \cong \tau_1' \times \tau_2'} \quad \frac{E \vdash \tau_1 \cong \tau_1' \quad E \vdash \tau_2 \cong \tau_2'}{E \vdash \tau_1 + \tau_2 \cong \tau_1' + \tau_2'} \quad \frac{E \vdash \tau_1 \cong \tau_1' \quad E \vdash \tau_2 \cong \tau_2'}{E \vdash \tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'}$$

$$\frac{E, \mu X.\tau_1 \cong \tau_2 \vdash \tau_1\{\mu X.\tau_1/X\} \cong \tau_2}{E \vdash \mu X.\tau_1 \cong \tau_2} \ (\mu \text{ equiv})$$

Using (μ equiv), we can derive the following useful rule:

$$\frac{E, \mu X.\tau_1 \cong \mu Y.\tau_2 \vdash \tau_1\{\mu X.\tau_1/X\} \cong \tau_2\{\mu Y.\tau_2/Y\}}{E \vdash \mu X.\tau_1 \cong \mu Y.\tau_2}$$

Using these rules, we can now show the equivalence of the two types given earlier. Let $\tau_1 \equiv \mu S.S \to \mathsf{int}$, $\tau_2 \equiv \mu T.(T \to \mathsf{int}) \to \mathsf{int}$,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\tau_1 \cong \tau_2, \tau_1 \cong \tau_2 \to \mathsf{int} \vdash \tau_1 \cong \tau_2} \quad \dfrac{}{\vdash \mathsf{int} \cong \mathsf{int}}}{\tau_1 \cong \tau_2, \tau_1 \cong \tau_2 \to \mathsf{int} \vdash \tau_1 \to \mathsf{int} \cong \tau_2 \to \mathsf{int}}}{\tau_1 \cong \tau_2 \vdash \tau_1 \cong \tau_2 \to \mathsf{int}} \quad \dfrac{}{\tau_1 \cong \tau_2 \vdash \mathsf{int} \cong \mathsf{int}}}{\tau_1 \cong \tau_2 \vdash \tau_1 \to \mathsf{int} \cong (\tau_2 \to \mathsf{int}) \to \mathsf{int}}}{\emptyset \vdash \mu S.S \to \mathsf{int} \cong \mu T.(T \to \mathsf{int}) \to \mathsf{int}}$$

3 Modeling $\mu X.\tau$

Currently, our semantics model τ as a domain $\mathcal{T}[\![\tau]\!]$. We would like to model the domain for the recursive type, $\mu X.\tau$. Recursive types introduce names for types. Thus we need a type environment χ to capture recursive types: $\chi \in \textbf{TypeVar} \rightharpoonup \textbf{Domain}$. In our definition, $\mathcal{T}[\![\tau]\!]\chi$ gives the domain corresponding to type τ, given type variables whose meanning is defined by χ.
Here are the inductively defined rules for interpreting $\mathcal{T}[\![\tau]\!]\chi$:

$$\begin{aligned}
\mathcal{T}[\![X]\!]\chi &= \chi(X) \\
\mathcal{T}[\![\tau_1 + \tau_2]\!]\chi &= \mathcal{T}[\![\tau_1]\!]\chi + \mathcal{T}[\![\tau_2]\!]\chi \\
\mathcal{T}[\![\tau_1 \times \tau_2]\!]\chi &= \mathcal{T}[\![\tau_1]\!]\chi \times \mathcal{T}[\![\tau_2]\!]\chi \\
\mathcal{T}[\![\tau_1 \to \tau_2]\!]\chi &= \mathcal{T}[\![\tau_1]\!]\chi \to (\mathcal{T}[\![\tau_2]\!]\chi)_\perp \\
\mathcal{T}[\![\mu X.\tau]\!]\chi &= \mu D.\mathcal{T}[\![\tau]\!]\chi[X \mapsto D]
\end{aligned}$$

In these rules, we introduced a constructor for recursive domains, which is given by $\mu D.\mathcal{F}(D)$, where functor $\mathcal{F}$ maps one domain into another domain, $\mathcal{F}$: Domain $\to$ Domain. If $D = \mu X.\mathcal{F}(X)$, then $\mathcal{F}(D)$ produces a domain related to D by continuous functions *up* and *down* that are inverses of one another.

$$\mu D.\mathcal{F}(D) \quad \underset{\underset{down}{\longrightarrow}}{\overset{\overset{up}{\longleftarrow}}{\cong}} \quad \mathcal{F}(\mu D.\mathcal{F}(D))$$

That is, $up_{\mu X.\tau} \in \mathcal{T}[\![\tau\{\mu X.\tau/X\}]\!] \to \mathcal{T}[\![\mu X.\tau]\!]$ and $down_{\mu X.\tau} \in \mathcal{T}[\![\mu X.\tau]\!] \to \mathcal{T}[\![\tau\{\mu X.\tau/X\}]\!]$.
Having *up* and *down* in hand, we can write out denotational semantics for the fold and unfold expressions:

$$\begin{aligned}
\mathcal{C}[\![\Gamma \vdash \mathsf{fold}_{\mu X.\tau} e : \mu X.\tau]\!]\rho &= up_{\mu X.\tau}\mathcal{C}[\![\Gamma \vdash e : \tau\{\mu X.\tau/X\}]\!]\rho \\
&\in \mathcal{T}[\![\mu X.\tau]\!]\phi \\
\mathcal{C}[\![\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu X.\tau/X\}]\!]\rho &= down_{\mu X.\tau}\mathcal{C}[\![\Gamma \vdash e : \tau\{\mu X.\tau/X\}]\!]\rho \\
&\in \mathcal{T}[\![\tau\{\mu X.\tau/X\}]\!]
\end{aligned}$$

4 Type interpretation and compiler construction

The type interpretation function $\mathcal{T}[\![\tau]\!]\chi$ can be read as a somewhat forbidding statement about how to construct complex domains that underlie the semantic model for a language. A less mathematical interpretation is that they explain a nice way to write a compiler for a language with recursive types. In building a compiler, we have two basic choices for how to represent types internally. One is to represent them as the abstract syntax tree corresponding to the type expression. This approach works but can be somewhat clumsy, because the type expression may name type variables. In order to make sense of the type expression, it is necessary to always keep track of the associated naming context that binds the type variables. Tests for type equivalence can be expensive and may need to be done repeatedly if the language supports structural equivalence.

Another implementation choice is to interpret these types and convert them into *type objects*. The advantage of the latter approach is that the type objects can be simpler; they need not record certain information such as the actual name of the type variable introduced in a recursive type, for example. In addition, the compiler can be designed to interpret all equivalent recursive types as precisely the same type object, reducing the problem of testing type equivalence to one of testing pointer equality.

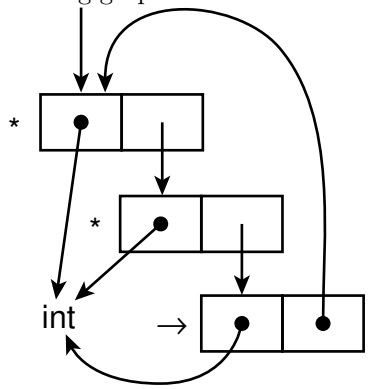
For example, consider the following recursive type declaration: a Modula-like language:

$$\mathsf{typePoint} = \mathsf{int} * \mathsf{int} * (\mathsf{int} \to \mathsf{Point})$$

Using the recursive type constructor, this can also be written as:

$$\mathsf{Point} = \mu P \;.\; \mathsf{int} * \mathsf{int} * (\mathsf{int} \to P)$$

Recalling that we can interpret the taking of a fixed point (in this case, by the domain constructor μ) as the creation of a cyclical graph, we see that the action of the type interpretation function $\mathcal{T}$ on this type can be thought of as constructing the following graph:



Note that here there is only one int type, and therefore only one object representing it. Type checking can often be implemented more efficiently if all type expressions are first interpreted to construct graphs in this manner.

5 Solving domain equations

5.1 Domain equations v.s. BNF

For domain equations that only use the $+$, $\times$, and $\lfloor\cdot\rfloor$ domain constructors, we can write a definition of the domain in BNF style. For example,

$$\begin{array}{lcl} D & \cong & (D+1)_\perp \qquad \text{(Domain equation)} \\ D & ::= & \perp \mid \lfloor u \rfloor \mid \lfloor D \rfloor \qquad \text{(BNF)} \end{array}$$

This correspondance suggests that the inductive set construction can be reused here to interpet domain equations. Recall that the inductive construction gives the least fixed point solution to a rule operator. In this solution, all the elements have derivations of finite height.

There are other solutions to the fixed point of a rule operator; in particular, there is also a greatest fixed point, which is the largest set that is both complete and consistent with respect to the rules. The greatest fixed point corresponds to all elements with either finite *or infinite* derivations.

5.2 Solving equations

How do we solve equations? Our previous recipe for finding fixed points to set constructions was inductive. Given a rule that defines the set of elements of a set based on other elements, we defined a rule function from sets to sets whose fixed point is a solution, then took the union of a series of sets that were approximations to the fixed point. Is that all we need to solve domain equations?

For some functions, the inductive construction suffices, but the problem is that inductive construction doesn't always give us a CPO. Consider the equation $N \cong \mathbb{U} + N_\perp$. The inductive construction gives us elements

$$in_1(u), in_2(in_1(u)), in_2(in_2(in_1(u))), \ldots, 0, 1, 2, \ldots, in_2(\perp), in_2(in_2(\perp)), in_2(in_2(in_2(\perp))), \ldots 0_\perp, 1_\perp, 2_\perp, \ldots$$

Unfortunately, this is not a CPO, because it contains the infinite chain $0_\perp \sqsubseteq 1_\perp \sqsubseteq 2_\perp \sqsubseteq \ldots$ which has no upper bound.

For this particular example, the coinductive (greatest fixed point) construction does give us a CPO.

The coinductive construction does not handle another problem that arises with domain equations that involve the function space constructor: *cardinality*. Consider the domain equation $D \cong \mathbb{B}^D \cong \mathcal{P}(D)$. Cantor's diagonal argument tells us that there is no isomorphism between D and $\mathcal{P}(D)$, so there is no solution to this equation. Thus, we cannot expect to solve all the domain equations; clearly the inductive construction will not produce a solution since there is none.

We can extend the previous solution technique to find solutions over domains, which of course differ from sets in introducing an ordering among their elements. To solve a domain equation, we need the following:

1. An ordering on domains themselves: $D \trianglelefteq E$. The relation $\trianglelefteq$ is usually not a partial order, because it's not anti-symmetric. We expect that there are many domains that are isomorphic but whose elements happen to be labeled differently.

2. A least domain. For the least domain, we have two natural choices, depending on whether we want the solution domain to be pointed or not. If we are trying to find a solution among all domains, the empty domain $\mathbf{0}$ is the least domain. Among pointed domains, it is $\mathbf{0}_\perp \cong \mathbb{U}$.

Our goal is to construct a series of approximations to the solution domain, with the same form as earlier:

$$\mathbf{0} \trianglelefteq \mathcal{F}(\mathbf{0}) \trianglelefteq \mathcal{F}^2(\mathbf{0}) \trianglelefteq ...$$

The solution domain will, as before, be a least upper bound (within equivalence) on all of the $\mathcal{F}^n(\mathbf{0})$: $\mu D.\mathcal{F}(D) = \bigsqcup \mathcal{F}^n(\mathbf{0})$. For the same reasons as for fixed points of functions *within* a CPO, to guarantee a fixed point, $\mathcal{F}$ should also be continuous and monotonic:

$$D \trianglelefteq E \Rightarrow \mathcal{F}(D) \trianglelefteq \mathcal{F}(E)$$

$$\mathcal{F}(\bigsqcup_i (D_i)) \cong \bigsqcup_i \mathcal{F}(D_i)$$

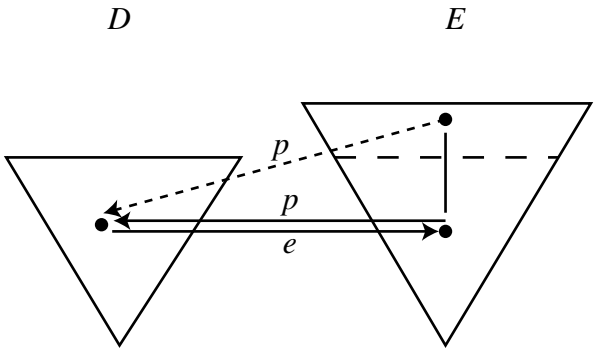
As we have stated above, we cannot expect to solve all domain equations, but we can always find solutions for some equations. One important class of domains over which we can find solutions is the *Scott domains*, which are isomorphic to the *information systems* described in Winskel Ch. 12. For any domain equation $D \cong \mathcal{F}(D)$, we can find the least solution if $\mathcal{F}$ is built up from discrete CPOs using only the domain constructors $\cdot + \cdot, \cdot \times \cdot, \cdot \to \cdot_\perp$.

The key idea is to construct an ordering on domains that captures the idea that one domain contains another domain as substructure, so that each step along the chain $\mathcal{F}^n(\mathbf{0})$ grows the domain towards the solution. We define the ordering, so that $D \trianglelefteq E$ if there exists an *embedding-projection* pair between D and E.

An embedding-projection pair between two domains D and E is a pair of continuous functions (e, p) such that:

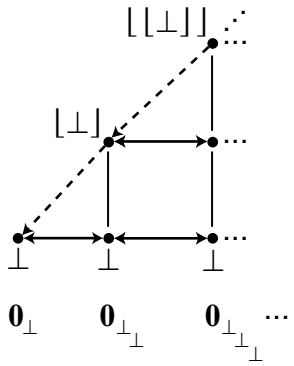
1. $e \in D \to E$
2. $p \in E \to D$
3. $p \circ e = id_D = \lambda x \in D \,.\, x \quad (\forall x \in D . p(e(x)) = x)$
4. $e \circ p \sqsubseteq id_E = \lambda x \in E \,.\, x \quad (\forall x \in E . e(p(x)) \sqsubseteq x)$

The relationship between e and p is shown in this figure:



Note that because of condition 3, every element of D is mapped by e into an element of E that is mapped back into the same element of D by p. Therefore, e maps the elements of D onto an image that is in E and isomorphic to D. Any elements of E that are not in the image of e must be projected to an element of D whose embedding is an approximation of ($\sqsubseteq$) the element of E. The mapping that is not part of the isomorphism is depicted in the figure by a dotted line.

Let us illustrate this process by an example. Consider the simple domain equations $D \cong D_\perp$, corresponding to the domain $\mu D.D_\perp$. The inductive set construction fails to produce a solution to this equation, just as in the earlier example, because it does not result in a CPO. Starting from the least pointed domain $\mathbf{0}_\perp$, we apply the functor $\mathcal{F}(D) = D_\perp$ repeatedly to obtain a series of domains as shown here:

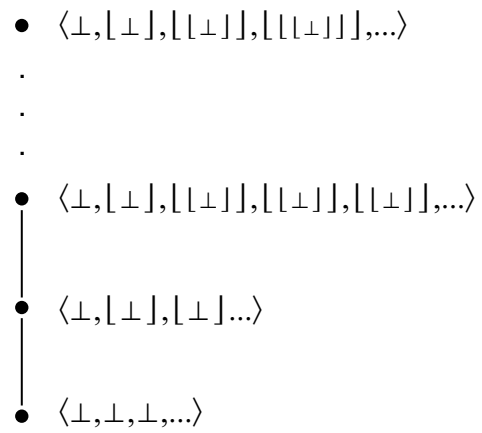


Consider a sequence of pointed domains $D_0, D_1, \ldots$ ordered through embedding-projection pairs (e_i, p_i). For the solution of domain equations, we are working with $D_i = \mathcal{F}^i(\mathbf{0}_\perp)$. We can construct a limit domain that is a least upper bound to all the domains in the sequence, as the set of *commuting tuples*. These are all the infinitely long tuples such that each tuple element comes from one of the approximating domains and successive tuple elements are related by projections:

$$D = \{\langle d_0, d_1, d_2, \ldots\rangle \mid \forall n \;.\; d_n \in \mathcal{F}^n(\mathbf{0}_\perp) \wedge d_n = p_n(d_{n+1})$$

Note that here p_n is the projection part of the embedding-projection pair that maps between D_n and D_{n+1}.

In our example equation, the set of such tuples consists of finite (compact) elements that extend "horizontally" across the diagram above, plus one infinite element corresponding to the diagonal of the diagram:



It is possible to show that this domain is a LUB of the approximating domains (Winskel effectively shows it, although one has to map all the proofs regarding information systems to the isomorphic domains.)

It is also possible to find solutions to domain equations that are not pointed domains. For example, the equation $D = 1 + D$ naturally produces a domain isomorphic to the natural numbers. Non-pointed domains that otherwise possess the properties of a Scott domain are called *predomains*. They can be solved for by finding embedding-projection pairs such that $p \circ e = id_D$ and $e_\perp \circ p^* \sqsubseteq id_{E_\perp}$, and starting from the initial domain $\mathbf{0}$ rather than from $\mathbf{0}_\perp$.

One important part of the solution process that we haven't discussed is how to demonstrate that various functors are, in fact, continuous, and how to find an embedding-projection pair between $\mathcal{F}^n(\mathbf{0})$ and $\mathcal{F}^{n+1}(\mathbf{0})$. These goals are most easily achieved through the information systems approach described by Winskel.