CS522 - Programming Language Semantics

Recursion

Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign

We are already familiar with various facets of recursion (in languages that we encountered, in rewriting, in lambda-calculus, etc.). The following is an interesting observation that deserves some further investigation:

> *The simpler the programming or specification language paradigm, the simpler the treatment of recursion.*

Indeed, recursion is not even noticeable as a “special” feature in a rewrite engine (for example Maude) or in an imperative programming language (like C), but it requires special language support, including special syntax, for more complex languages (such as `ML`).

Recursion in Term Rewriting

Rewriting supports recursion naturally, without any particular technical or theoretical infrastructure. It is just us, humans, who understand a certain rewriting definition as "recursive"; the rewrite engine does not even need to be aware of that. Consider, for example, a rewriting "implementation" of the length operator on lists using Maude notation:

```
eq length(nil) = 0 .
eq length(M ; L) = 1 + length(L) .
```

From the perspective of rewriting, there is nothing special going on here. A term `length(L)` for some list `L` is iteratively rewritten using the rules above as well as the rules of integer numbers, until it is eventually reduced to a natural number. Even though we may perceive the rewriting definition of `length` as "recursive", the rewrite engine does nothing but *match-and-apply* rules until a normal form is reached, which we can interpret as the "value" obtained after "evaluating" the original expression.

However, note that the process of rewriting itself is recursive. Indeed, we can regard the process of rewriting a term t, say *rewrite(t)*, as a procedure defined recursively as follows:

```
rewrite(t)
1. find a rule $l \to r$ such that $l$ matches some subterm $u$ of $t$; if no such rule exists then return $t$;
2. let $\theta$ be a substitution with $\theta(l) = u$;
3. replace $u$ by $\theta(r)$ in $t$ and obtain a new term $t'$;
4. rewrite(t')
```

In order for rewriting definitions to be practical, one needs to ensure that they *terminate*. To achieve that, one typically needs to ensure that a term is always reduced to a *simpler* term. This reduces to showing that any instance of the right-hand side of a rule is somehow simpler than the corresponding instance of the left-hand side. What “simpler” means is dependent on the particular definition. In the case of `length`, simpler means the operator is applied on a “strictly smaller list” in the right-hand-side term. Similar well-founded techniques are needed to prove termination of recursive programs in any programming language; the point here is, again, that rewriting definitions do not need to treat “recursive” definitions any differently.

It is interesting to note that the concept of “simpler term” is a semantical one - the term does not have to be simpler as a tree or w.r.t. some other immediate syntactic criterion, as shown by the following rewriting definition of bubble-sort:

```
eq bubbleSort(L)
 = if process(L) == L then L else bubble(process(L)) fi .

eq process(N ; M ; L)
 = if N <= M then N ; process(M ; L)  else M ; process(N ; L) fi .
eq process(N) = N .
eq process(nil) = nil .
```

Here, the computation of `bubble(L)` eventually reduces to the computation of `bubble(L')`, where the latter is simpler than the former in the sense that `L'` is *more sorted* than `L`.

Note, however, that one can speculate the recursive nature of rewriting and define bubble sort by just one simple and elegant conditional rewriting rule:

```
ceq M ; N = N ; M if N <= M .
```

Now there is no explicit term in the right-hand-side of the rule that one can show “smaller” that a corresponding term in the left-hand-side. Nevertheless, one can show that any application of the rule above would decrease the “weight” of any list that it applies on.

Besides termination, *confluence* is another important factor that needs to be considered when designing “recursive” rewriting definitions. Confluence generalizes the Church-Rosser result (for variants of λ-calculus) to any rewrite system: a rewrite system is confluent if and only if for any two reductions of a term t to t_1 and t_2, there is some term t' such that t_1 and t_2 reduce to t'. A rewrite system that is both confluent and terminating is called *canonical*. Note that terms have *unique normal forms* when reduced using canonical rewrite systems. In our context, a recursive rewriting definitions that is a canonical rewrite system can and should be regarded as a *well-behaved definition*.

Consider, for example, the following (deliberatly more complicated than needed) definition of addition of natural numbers:

```
eq M + 0 = M .
eq 0 + N = N .
eq s(M) + N = s(M + N) .
eq M + s(N) = s(M + N) .
```

The above simulates a non-deterministic recursive definition. According to this definition, there are many ways to compute the sum of m and n, since each time there is no unique way to reduce a term containing `+` to a simpler one. One can relatively easily show that this definition is well-behaved, because each rule simplifies the term to reduce in some way (first two eliminate a `+` operator, while the other two push an `s` operator higher).

We can therefore see that canonical rewrite systems support in a simple and natural way arbitrarily complex recursive definitions.

Moreover, since rewrite rules can be applied in parallel when their applications do not overlap, canonical rewrite systems can also be regarded as potentially very efficient implementations of recursive definitions. In particular, divide-and-conquer recursive definitions can be regarded operationally as starting a new "thread" solving the subproblems; on a parallel implementation of a rewrite engine, one can achieve a computational speed close to as many times as processors available at no additional programming effort.

One should not underestimate the difficulty of showing rewrite systems canonical. Since rewriting is Turing-complete, not even termination is decidable. One can show that in general neither termination nor confluence of a rewrite system is decidable. To have a better feel for the difficulty of the confluence problem, consider the following problem.

Suppose that a group of children have a large collection of black and white balls and decide to play a game. They put a bunch of balls in a bag and then each of them, at any moment they wish and potentially in parallel, extracts two balls from the bag and puts one back as follows: if the two balls have the same color they put a black ball back in the bag, otherwise they put a white one. If the bag is declared as an associative binary operator __, then this game can be easily defined as a four rule ground rewrite system:

$$\bullet\ \bullet \to \bullet\ ,\ \circ\ \circ \to \bullet\ ,\ \bullet\ \circ \to \circ\ ,\ \circ\ \bullet \to \circ.$$

Using variables, one can define it as just one rule:

```
eq X Y = if X == Y then black else white fi .
```

Exercise 1 *Prove that the rewrite system above is canonical.*

Recursion in Simple Imperative Languages

Recall the implementation of the Hanoi tower problem in a simple C-like imperative language:

```
function h(x, y, z, n) {
  if n >= 1 then {
    h(x, z, y, n - 1) ;
    print(x) ;
    print(z) ;
    h(y, x, z, n - 1)
  }
}
```

In this recursive function, `h` refers to itself directly and "transparently". This is possible because of the default design conventions of the imperative language:

- all functions are declared at the top level of the program (more precisely at the begining);
- no nested function declarations allowed;
- function declarations are assumed visible anywhere in the program, including in the body of the functions themselves.

Therefore, languages can support recursion "naturally" if they are constrained by design appropriately. Nevertheless, many programmers may not be happy with such language constraints just for the sake of supporting recursion transparently.

Recursion in Higher Order Languages

Recursion can be supported in a more flexible (but also more intricate) way when we climb to higher order languages, where functions, as first-order volatile citizens of the program, start raising non-trivial scoping and visibility issues. For instance, in the recursive definition of factorial

```
letrec f n = if n eq 0 then 1 else n * f(n - 1)  in  f 7
```

one has to *explicitly* state, using `letrec` instead of `let`, that the expression defining `f` is not a usual, but a recursive binding. Any other trick of simulating recursion would be unnatural and statically untypeable, like the classical

```
let f n g = if n eq 0 then 1 else n * g (n - 1) g  in  f 7 f
```

Recursion in Untyped λ-Calculus

The latter (untypeable) definition of factorial, which did not have to use the explicit recursion construct `letrec`, was given in the style of untyped λ-calculus, by passing a function to itself as an argument. As we have already seen, this technique can be applied very generally in the context of untyped λ-calculus, taking advantage of the *fixed point theorem*:

> **Theorem.** If Y denotes $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, then $Y F \equiv_\beta F(Y F)$ for any λ-expression F.

A λ-expression Y with the property above, namely that $Y F \equiv_\beta F(Y F)$ for any λ-expression F, is called a *fixed point operator*. The particular Y above is also known as the *call-by-name* fixed point operator (for reasons that will become clear shortly). As expected, there can are many other fixed point operators, including the following:

Exercise 2 *Show that the following λ-expressions are also fixed point operators:*

- $(\lambda x.\lambda f.f(xxf))(\lambda x.\lambda f.f(xxf))$, *also known as the* Turing *fixed point operator;*
- $\lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$, *also known as the* call-by-value *fixed point operator.*

We have already discussed how the fixed-point operator Y in the fixed-point theorem can be used to prove equational properties about "recursive programs". For example, if we let F be the usual "factorial" λ-expression

$$\lambda f.\lambda n.\ \texttt{if}\ n\ \texttt{eq}\ 0\ \texttt{then}\ 1\ \texttt{else}\ n * f(\mathrm{pred}\ n)\ ,$$

where we assume `if-then-else` and pred , the latter taking any strictly positive number n into $n - 1$ and 0 into 0, as already defined syntactic sugar (note that `if-then-else` has no associated evaluation strategy.), then one can easily show by using the equational rules of λ-calculus properties like $(Y F)3 \equiv_{\beta} 6$.

But how does (recursive) *computation* actually take place in λ-calculus? Of course, our computational mechanism must be based on β-reduction here. We have no need, and no intuitive computational meaning, for the η rule; the latter makes sense only for expressing and proving *equivalence of programs*. However, one cannot blindly apply β-reduction whenever a possibility is found inside the current term, since this way one might miss oportunities of termination; although it is true that, by the Church-Rosser theorem, no β-reduction is an irreparable ”mistake“, nevertheless a persistent chain of "wrong" β-reductions might indefinitely continue a virtually terminating computation. For example, one can easily see that in the case of our factorial, the following can take place

$$(YF)3 \Rightarrow_\beta Y_F 3 \Rightarrow_\beta FY_F 3 \Rightarrow_\beta F(FY_F)3 \Rightarrow_\beta F(F(FY_F))3 \ldots,$$

where Y_F is the λ-expression $Y[f \leftarrow F]$.

Exercise 3 *If for any λ-expression F we let Y_F denote the λ-expression $(\lambda x.F(xx))(\lambda x.F(xx))$, show that $Y_F \Rightarrow_\beta FY_F$. The fixed-point theorem tells us that $YF \equiv_\beta F(YF)$ anyway; is it also true that $YF \Rightarrow_\beta F(YF)$? How about the other two fixed-point operators in Exercise 2?*

Thanks to the Church-Rosser theorem, we know that if a normal form exists then it is unique. In particular, 6 is the only normal form of $(YF)3$ despite the fact that the reduction of $(YF)3$ may not terminate. Needless to mention that, in practical implementations of programming languages, one would like to devise reduction or evaluation "strategies" that would lead to normal forms whenever they exists.

One simplistic way to achieve the desired computation would be by breadth-first search in the space of computations, until a β-normal form is eventually reached. Clearly, such a computation strategy would be unacceptably inefficient in practice. For that reason, strategies that are not based on search are desirable. Many reduction strategies have been investigated in the literature and implemented in various languages.

We next discuss one interesting reduction strategy, called *leftmost-outermost β-reduction*, also known as *call-by-name* or *normal-order*: each time, β-reduction is applied outside in and left-to right; this corresponds to β-reducing the first suitable subterm found by a prefix traversal of the term to reduce. The application of a β-reduction step can enable another reduction which occurs earlier in the prefix traversal, so each time a reduction is applied, the entire term to rewrite is assumed to be retraversed using the prefix strategy (better algorithms are used in practical implementations). Notice that argument expressions passed to λ-abstractions are not evaluated at “call time”, but instead at “need time”.

The first programming language providing a call-by-name evaluation strategy was Algol-60. An important variation of call-by-name, supported by several programming languages including most notoriusly Haskel, is *call-by-need*. Under call-by-need, λ-expressions are evaluated at most once: any subsequent use of a λ-expression just uses the previously calculated normal form. As we know from CS422, in the context of side effects call-by-name and call-by-need can lead to different normal forms.

Theorem. If a λ-expression has a β-normal form, then it can be reached by leftmost-outermost β-reductions.

An intuitive argument for the above result is the following: if one applies β-reductions as close to the top as possible, then one can prevent any unnecessary non-termination obtained by infinitely rewriting a term which is supposed to dissapear anyway by an upper β-reduction.

Exercise 4 *Modify your Maude implementation of λ-calculus (either the one based on the De Bruijn transformation or the one based on combinators) to support the call-by-name reduction strategy.*
Hint. *You only need to add appropriate rewriting strategies to certain operations in the signature.*

Another very important, if not the most important, reduction strategy is the one called *call-by-value*. Under call-by-value,

λ-expressions are always evaluated *before* they are passed to λ-abstractions. In order for call-by-value to work, one needs to inhibit reductions under certain language constructs, such as within the body of λ-abstractions and within the two choices of the conditional. Also, an appropriate fixed-point operator needs to be chosen.

Exercise 5 *Reduce* $(Y F)3$ *using a call-by-value strategy. Can you just use the call-by-need fixed-point operator? How about the other two fixed-point operators in Exercise 2?*

Mu Simply-Typed Lambda Calculus

Once one decides that types are a useful feature of one's language and one decides to declare and/or check them statically, the next important thing one should ask oneself is how such a decision affects recursive definitions.

Typed functional languages do not allow functions to pass themselves to themselves, so in order to support recursion, one needs to introduce a new language construct (`letrec`). The goal of this topic is to discuss in more depth the relationship between types and recursive definitions of functions. To focus on the essence of the problem, we start with the simplest typed language discussed so far, namely simply-typed λ-calculus.

Let us first understand why the presented fixed-point technique to support recursive definitions in untyped λ-calculus does not work in the context of types. Consider the fixed-point combinator Y suggested by the fixed-point theorem, $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. To make Y a well-typed λ-expression in simply typed λ-calculus, we need to come up with some types t_f, t_x^1 and t_x^2, such that the λ-expression $\lambda f\!:\!t_f.(\lambda x\!:\!t_x^1.f(xx))(\lambda x\!:\!t_x^2.f(xx))$ can be typed. The reason for that is the subexpression xx: there is no type that one can assign to x such that xx can be calculated a type.

Let us assume though, for the sake of the discussion, that fixed-point combinators Y existed in simply-typed λ-calculus and let us try to understand what their type should be. Recall that such combinators are intended to apply on λ-expressions F encoding the one-step "unrolling" of a recursive definition.

For example, the "unrolling" λ-expression associated to the definition of the (typed) factorial function of type nat → nat would be

$$F := \lambda\, f : \mathrm{nat} \to \mathrm{nat}. \lambda\, n : \mathrm{nat}.\ \texttt{if}\ \ n\ \ \texttt{eq}\ \ 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * f(\mathrm{pred}\, n)$$

having the type $(\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat} \to \mathrm{nat}$.

Any recursive function, of any type, has such a one-step “unrolling” λ-expression associated to it. For example, a function of type $\mathrm{nat} \to \mathrm{bool}$ returning the parity of a natural number is associated the following λ-expression of type $(\mathrm{nat} \to \mathrm{bool}) \to \mathrm{nat} \to \mathrm{bool}$:

$$F := \lambda\, f : \mathrm{nat} \to \mathrm{bool}. \lambda\, n : \mathrm{nat}.\ \texttt{if}\ \ n\ \ \texttt{eq}\ \ 0\ \ \texttt{then}\ \ \mathrm{true}\ \ \texttt{else}\ \ \texttt{not}\ \ (f(\mathrm{pred}\, n)).$$

It is interesting to note that recursion can mean more than just a recursive function. For example, in lazy functional programming it is customary to define “infinite” data structures, such as infinite lists or infinite trees, by defining their actual behavior when asked to produce values. For example, an infinite list of zeros should be able to produce a zero whenever asked for its first element (by passing it to car). The “unrolling” λ-expression for such a recursive data-structure would be

$$F := \lambda\, l : \texttt{list}\ \mathrm{nat}.\ \texttt{cons}\ 0\ l,$$

which has the type $\texttt{list}\ \mathrm{nat} \to \texttt{list}\ \mathrm{nat}$.

Finally, one may want to compactly define “error” values for any type t as recursive expressions referring just to themselves, having therefore associated the trivial “unrolling” λ-expression $F := \lambda\, x : t.x$ of type $t \to t$.

Consequently, for any type t, it is reasonable to assume that any recursively defined λ-expression E of type t has an “unrolling” λ-expression F_E of type $t \to t$. Since one expects that $E \equiv Y F_E$, one can intuitively claim that a fixed-point combinator Y, if it exists, must have the type $(t \to t) \to t$. More precisely, one would expect for each type t some corresponding fixed-point combinator Y_t of type $(t \to t) \to t$.

In fact, one can prove that there exist no such magic fixed-point combinators Y_t in the simply-typed λ-calculus language. It is relatively usual in computer science foundational works to enrich a certain formalism with desired features “by definition”, or “by axiom”, when those features cannot be explained or simulated by the existing framework. Since the concept of fixed-point seems to very naturally capture the essence of recursion, the solution that has been adopted by scientists to allow recursion in typed programming environments is to extend the language by bringing a Y_t for each type t “from the outside”.

Thus, for each type t we assume the existence of a new constant $Y_t : (t \to t) \to t$, called for simplicity the *fixed-point operator for type t*. Whether Y_t can or not be explained from more basic principles does not concern us here. Instead, we axiomatize its behavior by requiring that

$$Y_t F \equiv F(Y_t F) \text{ for any } \lambda\text{-expression } F \text{ of type } t \to t.$$

Alternatively and more compactly, noticing that the considered expression F always has the form $\lambda\, x\!:\!t.E$, we can define a new "recursive" binding operator μ to bind x in E and replace the functionality of Y_t as follows:

$$Y_t F = Y_t(\lambda\, x\!:\!t.E) = \mu x\!:\!t.E.$$

More precisely, we are going to have a new binding operator μ with the same syntax as λ and a new equational rule saying that

$$\mu x\!:\!t.E \equiv E[x \leftarrow (\mu x\!:\!t.E)],$$

with the intuition that arbitrarily many copies of $\mu x\!:\!t.E$ can be generated, typically by need, nesting the original expression in all places where x occures free. An even more intuitive equivalent form of this rule is

$$\mu x\!:\!t.E \equiv (\lambda\, x\!:\!t.E)(\mu x\!:\!t.E),$$

which says that $\mu x\!:\!t.E$ is a fixed point of the function

λ-abstracting the expression E in the variable x.

So far, we only justified the μ-extension of simply-typed λ-calculus by examples. A rigorously formulated reason for extending simply-typed λ-calculus is that it is not expressive enough as a model of computation: it can only encode very simple programs which do not have any recursive behavior. In particular, simply-typed λ-calculus is *decidable*; this follows from two facts:

- like for untyped λ-calculus, the Church-Rosser theorem holds;
- unlike for untyped λ-calculus, all computations eventually terminate, thus leading to the reach of unique normal forms.

Hence one can decide if two terms are provably equal by simply reducing them into their normal forms and then syntactically comparing the results.

Programming Language for Computable Functions

As one might expect, simply typed λ-calculus extended with the recursion operator μ has full computational power. We are going to present a language called PCF (abbreviating Programming language for Computable Functions), a μ-extension of simply typed λ-calculus with a particular but important signature (S, Σ) together with corresponding equations.

The BNF syntax for types and expressions in PCF is the following

$$\mathit{Var} ::= x \mid y \mid \ldots$$

$$\mathit{Type} ::= S \mid \mathit{Type} \to \mathit{Type}$$

$$\mathit{Exp} ::= \Sigma \mid \mathit{Var} \mid \mathit{Exp}\mathit{Exp} \mid \lambda\mathit{Var} : \mathit{Type}.\mathit{Exp} \mid \mu\mathit{Var} : \mathit{Type}.\mathit{Exp}.$$

where $S = \{\mathrm{nat}, \mathrm{bool}\}$ is the set of basic types. and $\Sigma {=} \{0 : \mathrm{nat}, \mathrm{succ} : \mathrm{nat} \to \mathrm{nat}, \mathrm{pred} : \mathrm{nat} \to \mathrm{nat}, \mathrm{true} : \mathrm{bool}, \mathrm{false} :$

$\mathrm{bool}, \mathrm{zero?} : \mathrm{nat} \to \mathrm{bool}\} \cup \{\mathrm{cond}_t : \mathit{bool} \to t \to t \to t \mid \forall t \in \mathit{Type}\}$ is the signature.

When discussing typing and equations, we build on top of the already defined (S, Σ)-simply-typed λ-calculus.

We only add one typing rule:

$$\frac{X, x\!:\!t \ \rhd \ E\!:\!t}{X \ \rhd \ \mu x\!:\!t.E\!:\!t} \qquad \text{for any type assignment } X, x\!:\!t, \ \lambda\text{-expression } E \text{ of type } t;$$

Exercise 6 *Type the following expressions:*

- $+ := \mu f\!:\!\mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}.\lambda x\!:\!\mathrm{nat}.\lambda y\!:\!\mathrm{nat}.$
  $\mathrm{cond}_{\mathrm{nat}}\ (\mathrm{zero?}\ x)\ y\ (\mathrm{succ}\ (f(\mathrm{pred}\ x)\ y)),$
- $* := \mu f\!:\!\mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}.\lambda x\!:\!\mathrm{nat}.\lambda y\!:\!\mathrm{nat}.$
  $\mathrm{cond}_{\mathrm{nat}}\ (\mathrm{zero?}\ x)\ 0\ (+\ (f(\mathrm{pred}\ x)\ y)\ y).$

We add the folowing equational rules, describing the desired behavior of the Σ-constants and of the recursion operator μ; by a language abuse, we let n denote succ succ ...succ 0 (n times):

$$\mathcal{E} \vdash (\forall\emptyset)\ \text{pred}\ 0 =_{\text{nat}} 0$$

$$\mathcal{E} \vdash (\forall X)\ \text{pred succ}\ n =_{\text{nat}} n$$

$$\mathcal{E} \vdash (\forall\emptyset)\ \text{zero?}\ 0 =_{\text{bool}} \text{true}$$

$$\mathcal{E} \vdash (\forall\emptyset)\ \text{zero? succ}\ n =_{\text{bool}} \text{false}$$

$$\mathcal{E} \vdash (\forall X)\ \text{cond}_t\ \text{true}\ E\ E' =_t E \qquad \text{if } X \triangleright E : t,\ X \triangleright E' : t$$

$$\mathcal{E} \vdash (\forall X)\ \text{cond}_t\ \text{false}\ \ E\ E' =_t E' \qquad \text{if } X \triangleright E : t,\ X \triangleright E' : t$$

$$(\xi_\mu) \qquad \frac{\mathcal{E} \vdash (\forall X, x:t)\ E =_t E'}{\mathcal{E} \vdash (\forall X)\ \mu x:t.E =_t \mu x:t.E'}$$

$$(\mu) \qquad \mathcal{E} \vdash (\forall X)\ \mu x:t.E =_t E[x \leftarrow (\mu x:t.E)]$$

$$(\mathsf{Con}_\sigma) \qquad \frac{\mathcal{E} \vdash (\forall X)\ E =_{\mathrm{nat}} E'}{\mathcal{E} \vdash (\forall X)\ \sigma E =_{\mathrm{nat}} \sigma E'} \quad \text{, where } \sigma \in \{\mathrm{succ}, \mathrm{pred}\}$$

$$(\mathsf{Con}_{\mathrm{zero?}}) \qquad \frac{\mathcal{E} \vdash (\forall X)\ E =_{\mathrm{nat}} E'}{\mathcal{E} \vdash (\forall X)\ \mathrm{zero?}\ E =_{\mathrm{bool}} \mathrm{zero?}\ E'}$$

$$(\mathsf{Con}_{\mathrm{cond}}) \qquad \frac{\mathcal{E} \vdash (\forall X)\ C =_{\mathrm{bool}} D,\ \mathcal{E} \vdash (\forall X)\ E =_t F,\ \mathcal{E} \vdash (\forall X)\ E' =_t F'}{\mathcal{E} \vdash (\forall X)\ \mathrm{cond}_t\ CEE' =_t \mathrm{cond}_t\ DFF'}$$

Exercise 7 *Derive the following identities:*

- $(\forall\emptyset)\ \ +\ \ (\mathrm{succ}\ 0)\ (\mathrm{succ}\ (\mathrm{succ}\ 0)) =_{\mathrm{nat}} \mathrm{succ}\ (\mathrm{succ}\ (\mathrm{succ}\ 0)),$
- $(\forall\emptyset)\ \mathrm{zero?}\ (\mathrm{succ}\ 0) =_{\mathrm{bool}} \mathrm{false}\ .$

We have already discussed (in a rather informal context) a possible computational semantics of untyped λ-calculus, based on leftmost-outermost β-reduction.

Now we shall give the PCF language two operational semantics, a transitional or small-step one, and a natural or big-step one.

Transitional semantics

The transitional semantics will be given as a *restricted* rewriting relation on λ-terms ("restricted" in the sense that it cannot be applied inside any arbitrary context). This relation will tell us how to *evaluate* programs (that is, terms) to *values* (that is, terms which are normal forms). But what are the values in our framework? They are just the usual natural and boolean values, plus the λ-abstractions. Here is the BNF description of values:

$$\mathit{Value} ::= 0 \mid \mathrm{succ}\ \mathit{Value} \mid \mathrm{true} \mid \mathrm{false}\ \mid \lambda\, \mathit{Var} : \mathit{Type}.\mathit{Exp}$$

The fact that we take *all* λ-abstractions as values, even though it might be possible to further evaluate their body, might seem strange at a first glance; however, this is natural if we consider the body of a λ-abstraction as just the text describing a function, which will be evaluated only if the function is to be applied (remember *closures* from functional languages). We shall soon come back to this discussion.

Like in any functional language, we have to make a choice between several evaluation strategies when we define semantics. We choose *call-by-name* below, since this is immediately related to the intuition of β-reduction, but one can easily adapt it to consider other evaluation startegies.

Exercise 8 *In a similar manner to the call-by-name transitional semantics below, give a transitional semantics to PCF following the call-by-value evaluation strategy.*

We define the *one-step-transition* binary relation → on λ-expressions as the smallest relation closed under the following rules, where we assume that all (outermost) expressions that appear in the left-hand side of the rules are well-typed (e.g., we assume that FE in the application rule is well-typed, hence we also implicitly assume that E and F are typeable to some types $t \rightarrow s$ and t, respectively):

$$\mathrm{pred}\ 0 \rightarrow 0$$

$$\mathrm{pred}\ \mathrm{succ}\ V \rightarrow V \qquad \text{, if } V \text{ is a value}$$

$$\mathrm{zero?}\ 0 \rightarrow \mathrm{true}$$

$$\mathrm{zero?}\ \mathrm{succ}\ V \rightarrow \mathrm{false} \qquad \text{, if } V \text{ is a value}$$

$$\mathrm{cond}_t\ \mathrm{true}\ E\ E' \rightarrow E$$

$$\mathrm{cond}_t\ \mathrm{false}\ E\ E' \rightarrow E'$$

$$\frac{E \to E'}{\sigma E \to \sigma E'} \quad , \text{ if } \sigma \in \{\mathrm{succ}\,, \mathrm{pred}\,, \mathrm{zero?}\,\}$$

$$\frac{C \to C'}{\mathrm{cond}_t\ C E E' \to \mathrm{cond}_t\ C' E E'}$$

$$\frac{F \to F'}{F E \to F' E}$$

$$(\lambda\, x : t.E)E' \to E[x \leftarrow E']$$

$$(\mu x : t.E) \to E[x \leftarrow (\mu x : t.E)]$$

Note that, unlike in the case of general term rewriting, the above rules cannot be applied inside *any* context (subterm), but actually apply “on top”. Some incursions into the to-be-rewritten term context are possible, but only those obeying some restrictions:

- never evaluate the passed argument (call-by-name),
- never evaluate the second and third arguments of a condition,
- never apply β-reduction or μ-unwinding in the body of another λ or μ-abstraction, etc.

And all the above “negative” facts are just implicit in the rules, which do not allow arbitrary congruence.

A property that one wants to be satisfied by the one-step-transition relation is that the latter is *sound*, in the sense that it rewrites a program into an equivalent program.

Proposition 1 *(Soundness of the one-step transition relation) Suppose* $X \triangleright E\!:\!t$. *If* $E \to E'$, *then* $\vdash (\forall X)E =_t E'$.

(Proof hint: Define the relation R on terms as $E \simeq E'$ iff there exists a type assignment X such that $\vdash (\forall X)E =_t E'$, and show that $\simeq$ is closed under all the transition rules.)

An important consequence is the fact that types are preserved by transitions, which allows one to type-check the program only once, at the begining, and then not worry about types during the execution:

Corollary 1 *If* $X \triangleright E\!:\!t$ *and* $E \to E'$, *then* $X \triangleright E' : t$.

Another fact to notice is that we were able to define our values (i.e. normal forms) *a priori*, and then guide our notion of computation towards them. And indeed, what we initially called “values” are precisely those closed terms that cannot be further reduced:

Proposition 2 *The following are equivalent for any* closed *(i.e., without free variables) expression E:*

- *E is a value;*
- *there is no E' such that $E \to E'$.*

(Proof hint: Show that, on the one hand, a value cannot be rewritten and, on the other hand, that any closed non-value can be rewritten.)

Notice that there are non-closed expressions, like the variables or applications between two variables, that are neither values nor can transit in one step into anything. Moreover, there are closed expressions that are neither values nor can transit in one step into any expression *other than themselves*, like, for instance, $\mu x\!:\!t.x$.

Exercise 9 *Characterize the non-closed expressions satisfying the former property. Is there any other expression, not α-equivalent to $\mu x\!:\!t.x$, satisfying the latter property?*

Define $\rightarrow^*$, the *transition relation*, to be the reflexive and transitive closure of $\rightarrow$. By the soundness of the one-step-transition relation, one can immediately infer

Theorem(*Soundness of the transition relation*) Suppose $X \triangleright E\!:\!t$. If $E \rightarrow^* E'$, then $\vdash (\forall X)\ E =_t E'$.

Corollary 2 *(Subject reduction) If $X \triangleright E\!:\!t$ and $E \rightarrow^* E'$, then $X \triangleright E' : t$.*

Since our language is *deterministic*, any expression can evaluate to *at most* one value:

Proposition 3 *If E is an expression and V, V' are values such that $E \rightarrow^* V$ and $E \rightarrow^* V'$, then $V = V'$.*

(Proof hint: First show that, for each expression E, there exists *at most one* expression E' such that $E \to E'$.)

If $E \to^* V$, then we say that *E evaluates to V*. There are expressions (like $\mu x : t.x$) that do not evaluate to any value; this is natural, since not all programs terminate (on all inputs). But if an expression E evaluates to some value V, then this unique value is considered to be the result of the computation. Thus we have a *partial function* mapping expressions to values, which could be regarded as the final product of the transitional semantics.

What about completeness?

We saw that the transition relation is sound w.r.t. equivalence of programs. But how complete is it? That is, what can we infer about equivalence of two programs by evaluating them transitionally? Well, according to this restricted and pragmatic form of evaluation, not much. And here are two reasons for that (can you find more reasons?):

- η-equivalence is not in the scope of our transitional semantics; for instance, $\lambda\, f\!:\!\mathrm{nat} \to \mathrm{nat}.\lambda\, y\!:\!\mathrm{nat}.f y$ and $\lambda\, f\!:\!\mathrm{nat} \to \mathrm{nat}.f$ are different values;
- even without the η equation, two different values can be proved equivalent, if they are λ-abstractions; for instance, $\lambda\, x\!:\!\mathrm{nat}.(\lambda\, y\!:\!\mathrm{nat}.y)x$ and $\lambda\, x\!:\!\mathrm{nat}.x$; this is actually a *semantical decision*, since we do not want to look into the body

of functions, unless we have to apply them.

However, one can prove a restricted form of completeness:

> ***Theorem***(*Basic-type-completeness of the transition)* If $\vdash (\forall X)\ E =_t E'$ and $t \in \{\mathrm{nat}, \mathrm{bool}\}$, then, for each value V, E evaluates to V if and only if E' evaluates to V. (Proof hint: First show that, if V is a value of type $t \in \{\mathrm{nat}, \mathrm{bool}\}$ and $\vdash (\forall X)\ E =_t V$, then $E \to^* V$.)

Hence, if we restrict our class to programs of basic types only, two programs that are provably equivalent either both do not terminate, or they both terminate and return the same value. Note that the restriction of returning basic typed values does not inhibit higher-order programing: diverse functions can be passed as arguments and returned as values in the process of calculating an integer value! Note that the restriction to basic types is crucial.

Natural Semantics

The two-step process of first defining a transition on expressions, and then extracting from it a partial map from expressions to values, can be performed in only one step. The *natural semantics* captures directly the idea of evaluation $E \rightarrow^* V$, by giving rules in terms of pairs (Expression,Value). The binary relation $\Downarrow$ between expressions and values is defined as the smallest closed under the following rules (below, V denotes a value, and E, E_1, E_2 expressions; obviously, succ V is then also a value; we assume that all expressions below are well-typed:

$$V \Downarrow V \qquad \frac{E \Downarrow V}{\text{succ } E \Downarrow \text{succ } V}$$

$$\frac{E \Downarrow 0}{\text{pred } E \Downarrow 0} \qquad \frac{E \Downarrow \text{succ } V}{\text{pred } E \Downarrow V}$$

$$\frac{E \Downarrow 0}{\text{zero? } E \Downarrow \text{true}} \qquad \frac{E \Downarrow \text{succ } V}{\text{zero? } E \Downarrow \text{false}}$$

$$\frac{C \Downarrow \text{true} \ , \ E_1 \Downarrow V}{\text{cond}_t \ C \ E_1 \ E_2 \Downarrow V} \qquad \frac{C \Downarrow \text{false} \ \ , \ E_2 \Downarrow V}{\text{cond}_t \ C \ E_1 \ E_2 \Downarrow V}$$

$$\frac{E_1 \Downarrow \lambda\, x\!:\!s.E \ , \ E[x \leftarrow E_2] \Downarrow V}{E_1 E_2 \Downarrow V} \qquad \frac{E[x \leftarrow (\mu x\!:\!t.E)] \Downarrow V}{\mu x\!:\!t.E \Downarrow V}$$

The above rules describe the evaluation process in a bottom-up fashion, starting from the values. One can show that transitional and natural semantics are equivalent:

Proposition 4 *For each expression E and value V, $E \Downarrow V$ iff $E \rightarrow^* V$.*

The next corollary, expressing the determism of evaluation, might be instructive to be proved directly about natural semantics:

Corollary 3 *If E is an expression and V, V' are values such that $E \Downarrow V$ and $E \Downarrow V'$, then $V = V'$.*

Complete Partial Orders

In what follows we recall some mathematical concepts related to complete partial orders and the fixed-point theorem, which will be useful to give a denotational semantics to PCF. For more details on how these concepts relate to recursive/iterative behavior of programs, you can also check the corresponding lecture notes in CS422 (lectures 21 and 22).

Let $(D, \sqsubseteq)$ be a partial order, that is, a set D together with a binary relation $\sqsubseteq$ on it which is reflexive, transitive and anti-symmetric. Partial orders are also called *posets*. Given a set of elements $X \subseteq D$, an element $p \in D$ is called an *upper bound (ub)* of X if and only if $x \sqsubseteq p$ for any $x \in X$. Furthermore, $p \in D$ is called a *least upper bound (lub)* of X if and only if p is an upper bound and for any other upper bound q of X it is the case that $p \sqsubseteq q$.

Note that upper bounds and least upper bounds may not always exist. For example, if $D = X = \{x, y\}$ and $\sqsubseteq$ is the identity relation, then X has no upper bounds. Least upper bounds may not exist even though upper bounds exist. For example, if $D = \{a, b, c, d, e\}$ and $\sqsubseteq$ is defined by $a \sqsubseteq c$, $a \sqsubseteq d$, $b \sqsubseteq c$, $b \sqsubseteq d$, $c \sqsubseteq e$, $d \sqsubseteq e$, then any subset X of D admits upper bounds, but the set $X = \{a, b\}$ does not have a least upper bound.

Due to the anti-symmetry property, least upper bounds are unique when they exist. For that reason, we let $\sqcup X$ denote the lub of X.

Given a poset $(D, \sqsubseteq)$, a *chain* in D is an infinite sequence $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$ of elements in D, also written using set notation as $\{d_n \mid n \in \mathbb{N}\}$. Such a chain is called *stationary* when there is some $n \in \mathbb{N}$ such that $d_m = d_{m+1}$ for all $m \geq n$.

A poset $(D, \sqsubseteq)$ is called a *complete partial order (cpo)* if and only if any of its chains has a lub. $(D, \sqsubseteq)$ is said to *have bottom* if and only if it has a minimal element. Such element is typically denoted by $\bot$, and the poset with bottom $\bot$ is written $(D, \sqsubseteq, \bot)$. If $\{d_n \mid n \in \mathbb{N}\}$ is a chain in $(D, \sqsubseteq)$, then we also let $\displaystyle\bigsqcup_{n\in\mathbb{N}} d_n$ or even $\sqcup d_n$ denote its lub $\sqcup\{d_n \mid n \in \mathbb{N}\}$.

Examples.

$(\mathcal{P}(S), \subseteq, \emptyset)$ is a cpo with bottom, where $\mathcal{P}(S)$ is the set of subsets of a set S and $\emptyset$ is the empty set.

$(\mathbb{N}, \leq)$, the set of natural numbers ordered by "less than or equal to", has bottom 0 but is not complete: the sequence $0 \leq 1 \leq 2 \leq \cdots \leq n \leq \cdots$ has no upper bound in $\mathbb{N}$.

$(\mathbb{N} \cup \{\infty\}, \leq, 0)$, the set of natural numbers plus infinity, where infinity is larger than any number, is a cpo with bottom 0. It is a cpo because any chain is either stationary, in which case its lub is obvious, or is unbounded by any natural number, in which case ∞ is its lub.

$(\mathbb{N}, \geq)$ is a cpo but has no bottom.

$(\mathbb{Z}, \leq)$ is not a cpo and has no bottom.

$(S, =)$, a flat set S where the only partial ordering is the identity, is a cpo. It has bottom if and only if S has only one element.

Most importantly, $(A \rightharpoondown B, \preceq, \bot)$, the set of partial functions $A \rightharpoondown B$ ordered by the informativeness relation "$f \preceq g$ iff $f(a)$ defined implies $g(a)$ defined and $g(a) = f(a)$" is a cpo with bottom $\bot : \; A \rightharpoondown B$, the function which is undefined in each state.

If $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are two posets and $f : \; D \to D'$ is a function, then f is called *monotone* if and only if $f(x) \sqsubseteq' f(y)$ for any $x, y \in D$ with $x \sqsubseteq y$. If f is monotone, then we simply write $f : \; (D, \sqsubseteq) \to (D', \sqsubseteq')$.

Monotone functions *preserve chains*, that is, $\{f(d_n) \mid n \in \mathbb{N}\}$ is a chain in $(D', \sqsubseteq')$ whenever $\{d_n \mid n \in \mathbb{N}\}$ is a chain in $(D, \sqsubseteq)$. Moreover, if $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ are cpos then for any chain $\{d_n \mid n \in \mathbb{N}\}$ in $(D, \sqsubseteq)$, we have

$$\bigsqcup_{n\in\mathbb{N}} f(d_n) \sqsubseteq' f(\bigsqcup_{n\in\mathbb{N}} d_n)$$

Indeed, since f is monotone and since $d_n \sqsubseteq \sqcup d_n$ for each $n \in \mathbb{N}$, it follows that $f(d_n) \sqsubseteq' f(\sqcup d_n)$ for each $n \in \mathbb{N}$. Therefore, $f(\sqcup d_n)$ is an upper bound for the chain $\{f(d_n) \mid n \in \mathbb{N}\}$. The rest follows because $\sqcup f(d_n)$ is the lub of $\{f(d_n) \mid n \in \mathbb{N}\}$.

Note that $\sqcup f(d_n) \sqsupseteq' f(\sqcup d_n)$ does not hold in general. Let, for example, $(D, \sqsubseteq)$ be $(\mathbb{N} \cup \{\infty\}, \leq)$, $(D', \sqsubseteq')$ be $(\{0, \infty\}, 0 \leq \infty)$, and f be the monotone function taking any natural number to 0 and ∞ to ∞. For the chain $\{n \mid n \in \mathbb{N}\}$, note that $\sqcup n = \infty$, so $f(\sqcup n) = \infty$. On the other hand, the chain $\{f(n) \mid n \in \mathbb{N}\}$ is stationary in 0, so $\sqcup f(n) = 0$.

One can think of a lub of a chain as a “limit” of that chain. Inspired by the analogous notion of continuous function in mathematical analysis, which is characterized by the property of preserving limits, we say that a monotone function $f : (D, \sqsubseteq) \rightarrow (D', \sqsubseteq')$ is *continuous* if and only if $\sqcup f(d_n) \sqsupseteq' f(\sqcup d_n)$, which is equivalent to $\sqcup f(d_n) = f(\sqcup d_n)$, for any chain $\{d_n \mid n \in \mathbb{N}\}$ in $(D, \sqsubseteq)$.

The Fixed-Point Theorem

Any monotone function $f : (D, \sqsubseteq, \bot) \to (D, \sqsubseteq, \bot)$ defined on a cpo with bottom to itself admits an implicit and important chain, namely $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdot \sqsubseteq f^n(\bot) \sqsubseteq \cdots$, where f^n denotes n compositions of f with itself. The next is a key result in denotational semantics.

Theorem. *Let $(D, \sqsubseteq, \bot)$ be a cpo with bottom, let $f : (D, \sqsubseteq, \bot) \to (D, \sqsubseteq, \bot)$ be a continuous function, and let fix(f) be the lub of the chain $\{f^n(\bot) \mid n \in \mathbb{N}\}$. Then fix$(f)$ is the least fix-point of f.*

Proof. We first show that $\mathit{fix}(f)$ is a fix-point of f:

$$\begin{array}{rcl} f(\mathit{fix}(f)) & = & f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) \\ & = & \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) \\ & = & \bigsqcup_{n\in\mathbb{N}} f^n(\bot) \\ & = & \mathit{fix}(f). \end{array}$$

Next we show that $\mathit{fix}(f)$ is the least fix-point of f. Let d be another fix-point of f, that is, $f(d) = d$. We can show by induction that $f^n(\bot) \sqsubseteq d$ for any $n \in \mathbb{N}$: first note that $f^0(\bot) = \bot\bot \sqsubseteq d$; assume $f^n(\bot) \sqsubseteq d$ for some $n \in \mathbb{N}$; since f is monotone, it follows that $f(f^n(\bot)) \sqsubseteq f(d) = d$, that is, $f^{n+1}(\bot) \sqsubseteq d$. Thus d is an upper bound of the chain $\{f^n(\bot) \mid n \in \mathbb{N}\}$, so $\mathit{fix}(f) \sqsubseteq d$.

In CS422 we have discussed how the fixed-point theorem above plays a crucial role in defining the denotational semantics of `while`. In the sequel we will see that the same powerful theorem allows us to give denotational semantics to the more general form of recursion that we have in PCF. Before we do that, let us first see two other interesting applications of the fixed-point theorem, mentioning that it actually has many applications in both computer science and mathematics (there is even a conference, called FIX, dedicated to the use of fixed-points in computer science!).

Examples. Consider the following common definition of the factorial:

$$f(n) = \begin{cases} 1 & , \text{ if } n = 0 \\ n * f(n-1) & , \text{ if } n > 0 \end{cases}$$

How does one know that such a mathematical object, i.e., a function satisfying the above property, actually exists ? According to the fixed-point theorem, since the operator $\mathcal{F}$

defined on the set of partial functions between $I\!N$ and $I\!N$ as

$$\mathcal{F}(g)(n) = \begin{cases} 1 & \text{, if } n = 0 \\ n * g(n-1) & \text{, if } n > 0 \text{ and } g \text{ defined} \\ \text{undefined} & \text{, if } n > 0 \text{ and } g \text{ undefined} \end{cases}$$

is continuous, hence it has a least fixed point. We thus can take $f = fix(\mathcal{F})$, and get

$$f(n) = \mathcal{F}(f)(n) = \begin{cases} 1 & \text{, if } n = 0 \\ n * f(n-1) & \text{, if } n > 0 \text{ and } f(n) \text{ defined} \end{cases}$$

Here it happens that f is total, thus it is the *unique* fixed point of $\mathcal{F}$.

Any context-free language over an (possibly infinite) alphabet can be defined as the least fixed point of some continuous operator on the power set of the set of words over the given alphabet. Let for instance the alphabet T be $Var \cup Z \cup \{+, -, *\}$, where Z is the set of integers and Var is a set of variables. Consider the following BNF syntax for arithmetic expressions:

- $Exp ::= Z \mid Var \mid Exp + Exp \mid -Exp \mid Exp * Exp$

Then the language coinsisting of all arithmetic expressions can of course be defined, as usually, using the notion of derivation. But it is also the least fixed point of the continuous operator $\mathcal{F} : (\mathcal{P}(T^*), \subseteq, \emptyset) \to (\mathcal{P}(T^*), \subseteq, \emptyset)$, defined by $\mathcal{F}(L) = Z \cup Var \cup L\{+\}L \cup \{-\}L \cup L\{*\}L$. Notice that the iterations $\mathcal{F}(\emptyset)$, $\mathcal{F}^2(\emptyset)$, ... correspond to the one-step, two-steps, ... derivations applying the grammar's productions.

Fixed-Point Semantics of PCF

We are now ready to define a canonical model of PCF. Unlike the canonical functional models of simply-typed λ-calculus, our model should be able to capture non-termination of programs; non-termination will be handled by the interpretation of the recursion operator μ. In our semantical framework,

- basic types will be interpreted as certain CPOs with bottom element $\bot$, where $\bot$ will stand for "undefined";
- types will be interpreted as continuous functions between the interpretations of their component types;
- environments will be, as usual, (*Type*-indexed) mappings from type assignments into the model;
- well-typed expressions will be interpreted as mappings from environments into the interpretation of their types.

In what follows, we consider *bottomed CPOs* (BCPOs for short), i.e., structures of the form $(P, \sqsubseteq, \bot)$, where $(P, \leq)$ is a CPO and $\bot$ is its bottom element. Note that $\mathit{Cont}((P, \leq, \bot_P), (Q, \leq, \bot_Q))$, the set of continuous functions from P to Q, can be naturally endowed with a BCPO structure, if we let $f \leq g$ iff $f(p) \leq g(p)$ for all $p \in P$; the bottom element will be the function $\bot$ defined by $\bot(p) = \bot_Q$ for all $p \in P$. In what follows, this will be the BCPO structure that we will implicitly assume on $\mathit{Cont}((P, \leq, \bot_P), (Q, \leq, \bot_Q))$.

Exercise 10 *Prove that $\mathit{Cont}((P, \leq, \bot_P), (Q, \leq, \bot_Q))$, with the indicated structure, is indeed a BCPO.*

Exercise 11 *Prove that BCPOs, together with continuous functions between them, form a category.*

Recall that $I\!\!N$ and $I\!\!B$ denote the sets of natural numbers and booleans respectively. Let $\perp \notin I\!\!N \cup I\!\!B$, and let $I\!\!N_\perp$ and $I\!\!B_\perp$ be the sets $I\!\!N \cup \{\perp\}$ and $I\!\!B \cup \{\perp\}$ respectively, endowed with the partial orders $\{(\perp, i) \mid i \in I\!\!N\} \cup \{(i, i) \mid i \in I\!\!N \cup \{\perp\}\}$ and $\{(\perp, \text{false }), (\perp, \text{true}), (\perp, \perp), (\text{true}, \text{true}), (\text{false }, \text{false })\}$ respectively.

Exercise 12 *Find a natural correspondence between the set of partial functions from X to Y and the set of continuous functions from $X_\perp$ to $Y_\perp$, where $X, Y \in \{I\!\!N, I\!\!B\}$. Is this correspondence bijective?*

Interpreting types

We are now ready to define $[\![\_]\!] : \mathit{Type} \to$ [The class of BCPOs]:

- $[\![\mathrm{nat}]\!] = \mathbb{N}_\perp$, $[\![\mathrm{bool}]\!] = \mathbb{B}_\perp$;
- $[\![s \to t]\!] = \mathit{Cont}([\![s]\!], [\![t]\!])$.

We let $\mathcal{HO}_\perp$ denote the *Type*-indexed set $\{[\![s]\!]\}_{s \in \mathit{Type}}$.

Interpreting constants

- $[\![\mathrm{succ}]\!], [\![\mathrm{pred}]\!] \in [\![\mathrm{nat} \to \mathrm{nat}]\!] = \mathit{Cont}(I\!N_\perp, I\!N_\perp)$,

$$[\![\mathrm{succ}]\!](v) = \begin{cases} v+1 & \text{, if } n \in I\!N \\ \perp & \text{, if } v = \perp \end{cases}$$

$$[\![\mathrm{pred}]\!](v) = \begin{cases} v-1 & \text{, if } n \in I\!N - \{0\} \\ 0 & \text{, if } n = 0 \\ \perp & \text{, if } v = \perp \end{cases}$$

- $[\![\mathrm{true}]\!], [\![\mathrm{false}\ ]\!] \in [\![\mathrm{bool}]\!]$, $[\![\mathrm{true}]\!] = \mathrm{true}$, $[\![\mathrm{false}\ ]\!] = \mathrm{false}$

- $[\![\mathrm{cond}_t]\!] \in [\![\mathrm{bool} \to t \to t \to t]\!] =$ $\mathit{Cont}(I\!B_\perp, \mathit{Cont}([\![t]\!], \mathit{Cont}([\![t]\!], [\![t]\!])))$,

$$[\![\mathrm{cond}_t]\!](b)(v_1)(v_2) = \begin{cases} v_1 & \text{, if } b = \text{true} \\ v_2 & \text{, if } b = \text{false} \\ \bot & \text{, if } b = \bot \end{cases}$$

Exercise 13 *Show that all the above are correct interpretations, in the sense that* $[\![\mathrm{succ}]\!]$, $[\![\mathrm{pred}]\!]$, $[\![\mathrm{cond}_t]\!]$, $[\![\mathrm{cond}_t]\!](b)$, $[\![\mathrm{cond}_t]\!](b)(v)$ *are indeed continuous functions.*

Interpreting well-typed terms

We define $[\![X \triangleright E : t]\!] : [X \to \mathcal{HO}_\perp] \to [\![t]\!]$ recursively on the structure of $X \triangleright E : t$.

- $[\![X \triangleright x : t]\!](\rho) = \rho(x)$ if x is a variable;
- $[\![X \triangleright \sigma : t]\!](\rho) = [\![\sigma]\!] \in [\![t]\!]$ if σ is a constant;
- $[\![X \triangleright E_1 E_2 : t]\!](\rho) = ([\![X \triangleright E_1 : s \to t]\!](\rho))([\![X \triangleright E_2 : s]\!](\rho))$, if $X \triangleright E_1 : s \to t$ and $X \triangleright E_2 : s$;
- $[\![X \triangleright \lambda\, x : s.E : s \to t]\!](\rho)(v) = [\![X, x : s \triangleright E : t]\!](\rho[x \leftarrow v])$ for each $v \in [\![s]\!]$;
- $[\![X \triangleright \mu x : t.E : t]\!](\rho) = fix([\![\lambda\, x : t.E : t \to t]\!](\rho))$.

Exercise 14 *Show that the above mapping is correctly defined, in the sense that the returned values indeed belong to the specified codomains. (Hint: The proof will proceed, as usually, by induction* *on E; note that there is nothing to prove for the case of a μ-operator on top.)*

Let us try to explain the intuition behind the definition of $[\![X \rhd \mu x\!:\!t.E\!:\!t]\!](\rho)$. To parallel the "syntactic" intuition of μ (given formally in the equational proof system which acts like a guide for all our semantic frameworks), the desired denotation would be a fixed point of the function whose "law" is expressed by E, that is, a fixed point of $[\![\lambda\, x\!:\!t.E\!:\!t \to t]\!](\rho)$ (since we want to be able to unwind $\mu x\!:\!t.E$ into $E[x \leftarrow \mu x\!:\!t.E]$, i.e., into $(\lambda\, x\!:\!t.E)\mu x\!:\!t.E$). But why *the least fixed point*? Intuitively, we do not want $[\![X \rhd \mu x\!:\!t.E\!:\!t]\!](\rho)$ to possess more information than the one provided by iterated unwindings.

Exercise 15 *The denotation of $X \rhd \mu x\!:\!t.E\!:\!t$ could be equivalently expressed as $fix(g)$, where $g : [\![t]\!] \to [\![t]\!]$ is defined by $g(v) = [\![X, x\!:\!t \rhd E\!:\!t]\!][\rho(x \leftarrow v)]$ for all $v \in [\![t]\!]$.*

As usually, we say that $\mathcal{HO}_\perp$ satisfies an equation $(\forall X)E =_t E'$, denoted $\mathcal{HO}_\perp \models (\forall X)E =_t E'$, iff $[\![X \rhd E\!:\!t]\!] = [\![X \rhd E'\!:\!t]\!]$.

Theorem $\mathcal{HO}_\perp$ is a model of PCF, in the sense that it satisfies all the PCF rules.

Corollary 4 *If PCF $\vdash (\forall X)E =_t E'$, then $[\![X \rhd E\!:\!t]\!] = [\![X \rhd E'\!:\!t]\!]$.*

Corollary 5 *If $E \to E'$, $X \rhd E\!:\!t$ and $X \rhd E'\!:\!t$, then $[\![X \rhd E\!:\!t]\!] = [\![X \rhd E'\!:\!t]\!]$.*

Corollary 6 *If $E \Downarrow V$, $X \rhd E\!:\!t$ and $X \rhd V\!:\!t$, then $[\![X \rhd E\!:\!t]\!] = [\![X \rhd V\!:\!t]\!]$.*

Proposition 5 *If V is a value and $X \rhd V\!:\!t$, where*

$t \in \{\text{nat}, \text{bool}\}$, *then* $[\![X \ \triangleright \ V\!:\!t]\!] \neq \bot$.

Theorem If $X \ \triangleright \ E\!:\!t$, $X \ \triangleright \ V\!:\!t$ and $[\![X \ \triangleright \ E\!:\!t]\!] = [\![X \ \triangleright \ V\!:\!t]\!]$, then $E \Downarrow V$.