Specifically, Felleisen's interpretation requires a term of the form

$$M = \mathfrak{C}(\lambda k^{A\to B}.N) : A$$

to be evaluated as follows. To evaluate M, first evaluate N. Note that both M and N have type A. If N returns a result, then this immediately becomes the result of M as well. On the other hand, if during the evaluation of N, the function k is ever called with some argument $x : A$, then the further evaluation of N is aborted, and x immediately becomes the result of M.

In other words, the final result of M can be calculated anywhere inside N, no matter how deeply nested, by passing it to k as an argument. The function k is known as a *continuation*.

There is a lot more to programming with continuations than can be explained in these lecture notes. For an interesting application of continuations to compiling, see e.g. [9] from the bibliography (Section 1.8). The above explanation of what it means to "evaluate" the term M glosses over several details. In particular, we have not given a reduction rule for $\mathfrak{C}$ in the style of β-reduction. To do so is rather complicated and is beyond the scope of these notes.

7 Polymorphism

The polymorphic lambda calculus, also known as "System F", is obtained extending the Curry-Howard isomorphism to the quantifier $\forall$. For example, consider the identity function $\lambda x^A.x$. This function has type $A \to A$. Another identity function is $\lambda x^B.x$ of type $B \to B$, and so forth for every type. We can thus think of the identity function as a family of functions, one for each type. In the polymorphic lambda calculus, there is a dedicated syntax for such families, and we write $\Lambda\alpha.\lambda x^\alpha.x$ of type $\forall\alpha.\alpha \to \alpha$. Please read Chapter 11 of "Proofs and Types" by Girard, Lafont, and Taylor [2].

8 Weak and strong normalization

8.1 Definitions

As we have seen, computing with lambda terms means reducing lambda terms to normal form. By the Church-Rosser theorem, such a normal form is guaranteed

to be unique if it exists. But so far, we have paid little attention to the question whether normal forms exist for a given term, and if so, how we need to reduce the term to find a normal form.

Definition. Given a notion of term and a reduction relation, we say that a term M is *weakly normalizing* if there exists a finite sequence of reductions $M \to M_1 \to \ldots \to M_n$ such that M_n is a normal form. We say that M is *strongly normalizing* if there does not exist an infinite sequence of reductions starting from M, or in other words, if *every* sequence of reductions starting from M is finite.

Recall the following consequence of the Church-Rosser theorem, which we stated as Corollary 4.2: If M has a normal form N, then $M \twoheadrightarrow N$. It follows that a term M is weakly normalizing if and only if it has a normal form. This does not imply that every possible way of reducing M leads to a normal form. A term is strongly normalizing if and only if every way of reducing it leads to a normal form in finitely many steps.

Consider for example the following terms in the untyped lambda calculus:

1. The term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is neither weakly nor strongly normalizing. It does not have a normal form.

2. The term $(\lambda x.y)\Omega$ is weakly normalizing, but not strongly normalizing. It reduces to the normal form y, but it also has an infinite reduction sequence.

3. The term $(\lambda x.y)((\lambda x.x)(\lambda x.x))$ is strongly normalizing. While there are several different ways to reduce this term, they all lead to a normal form in finitely many steps.

4. The term $\lambda x.x$ is strongly normalizing, since it has no reductions, much less an infinite reduction sequence. More generally, every normal form is strongly normalizing.

We see immediately that strongly normalizing implies weakly normalizing. However, as the above examples show, the converse is not true.

8.2 Weak and strong normalization in typed lambda calculus

We found that the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not weakly or strongly normalizing. On the other hand, we also know that this term is not typeable in the simply-typed lambda calculus. This is not a coincidence, as the following theorem shows.

Theorem 8.1 (Weak normalization theorem). *In the simply-typed lambda calculus, all terms are weakly normalizing.*

Theorem 8.2 (Strong normalization theorem). *In the simply-typed lambda calculus, all terms are strongly normalizing.*

Clearly, the strong normalization theorem implies the weak normalization theorem. However, the weak normalization theorem is much easier to prove, which is the reason we proved both these theorems in class. In particular, the proof of the weak normalization theorem gives an explicit measure of the complexity of a term, in terms of the number of redexes of a certain degree in the term. There is no corresponding complexity measure in the proof of the strong normalization theorem.

Theorem 8.3 (Strong normalization theorem for System F). *In the polymorphic lambda calculus (System F), all terms are strongly normalizing.*

Please refer to Chapters 4, 6, and 14 of "Proofs and Types" by Girard, Lafont, and Taylor [2] for the proofs of Theorems 8.1, 8.2, and 8.3, respectively.

9 Denotational semantics

We introduced the lambda calculus as the "theory of functions". But so far, we have only spoken of functions in abstract terms. Do lambda terms correspond to any *actual* functions, such as, functions in set theory? And what about the notions of β- and η-equivalence? We intuitively accepted these concepts as expressing truths about the equality of functions. But do these properties really hold of real functions? Are there other properties that functions have that that are not captured by $\beta\eta$-equivalence?

The word "semantics" comes from the Greek word for "meaning". *Denotational semantics* means to give meaning to a language by interpreting its terms as mathematical objects. This is done by describing a function that maps syntactic objects (e.g., types, terms) to semantic objects (e.g., sets, elements). This function is called an *interpretation* or *meaning function*, and we usually denote it by $[\![-]\!]$. Thus, if M is a term, we will usually write $[\![M]\!]$ for the meaning of M under a given interpretation.

Any good denotational semantics should be *compositional*, which means, the interpretation of a term should be given in terms of the interpretations of its subterms. Thus, for example, $[\![MN]\!]$ should be a function of $[\![M]\!]$ and $[\![N]\!]$.

Suppose that we have an axiomatic notion of equality $\simeq$ on terms (for instance, $\beta\eta$-equivalence in the case of the lambda calculus). With respect to a particular class of interpretations, *soundness* is the property

$$M \simeq N \qquad \Rightarrow \qquad [\![M]\!] = [\![N]\!] \text{ for all interpretations in the class.}$$

Completeness is the property

$$[\![M]\!] = [\![N]\!] \text{ for all interpretations in the class} \qquad \Rightarrow \qquad M \simeq N.$$

Depending on our viewpoint, we will either say the axioms are sound (with respect to a given interpretation), or the interpretation is sound (with respect to a given set of axioms). Similarly for completeness. Soundness expresses the fact that our axioms (e.g., β or η) are true with respect to the given interpretation. Completeness expresses the fact that our axioms are sufficient.

9.1 Set-theoretic interpretation

The simply-typed lambda calculus can be given a straightforward set-theoretic interpretation as follows. We map types to sets and typing judgments to functions. For each basic type ι, assume that we have chosen a non-empty set S_ι. We can then associate a set $[\![A]\!]$ to each type A recursively:

$$\begin{array}{lcl} [\![\iota]\!] & = & S_\iota \\ [\![A \to B]\!] & = & [\![B]\!]^{[\![A]\!]} \\ [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\ [\![1]\!] & = & \{*\} \end{array}$$

Here, for two sets X, Y, we write Y^X for the set of all functions from X to Y, i.e., $Y^X = \{f \mid f : X \to Y\}$. Of course, $X \times Y$ denotes the usual cartesian product of sets, and $\{*\}$ is some singleton set.

We can now interpret lambda terms, or more precisely, typing judgments, as certain functions. Intuitively, we already know which function a typing judgment corresponds to. For instance, the typing judgment $x{:}A, f{:}A \to B \vdash fx : B$ corresponds to the function that takes an element $x \in [\![A]\!]$ and an element $f \in [\![B]\!]^{[\![A]\!]}$, and that returns $f(x) \in [\![B]\!]$. In general, the interpretation of a typing judgment

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : B$$

will be a function

$$[\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![B]\!].$$

Which particular function it is depends of course on the term M. For convenience, if $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ is a context, let us write $[\![\Gamma]\!] = [\![A_1]\!] \times \ldots \times [\![A_n]\!]$. We now define $[\![\Gamma \vdash M : B]\!]$ by recursion on M.

- If M is a variable, we define

$$[\![x_1{:}A_1, \ldots, x_n{:}A_n \vdash x_i : A_i]\!] = \pi_i : [\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![A_i]\!],$$

  where $\pi_i(a_1, \ldots, a_n) = a_i$.

- If $M = NP$ is an application, we recursively calculate

$$\begin{array}{rcl} f & = & [\![\Gamma \vdash N : A \to B]\!] : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]}, \\ g & = & [\![\Gamma \vdash P : A]\!] : [\![\Gamma]\!] \to [\![A]\!]. \end{array}$$

  We then define
$$[\![\Gamma \vdash NP : B]\!] = h : [\![\Gamma]\!] \to [\![B]\!]$$
  by $h(\bar{a}) = f(\bar{a})(g(\bar{a}))$, for all $\bar{a} \in [\![\Gamma]\!]$.

- If $M = \lambda x^A.N$ is an abstraction, we recursively calculate

$$f \quad = \quad [\![\Gamma, x{:}A \vdash N : B]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!].$$

  We then define

$$[\![\Gamma \vdash \lambda x^A.N : A \to B]\!] = h : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]}$$

  by $h(\bar{a})(a) = f(\bar{a}, a)$, for all $\bar{a} \in [\![\Gamma]\!]$ and $a \in [\![A]\!]$.

- If $M = \langle N, P\rangle$ is an pair, we recursively calculate

$$\begin{array}{rcl} f & = & [\![\Gamma \vdash N : A]\!] : [\![\Gamma]\!] \to [\![A]\!], \\ g & = & [\![\Gamma \vdash P : B]\!] : [\![\Gamma]\!] \to [\![B]\!]. \end{array}$$

  We then define

$$[\![\Gamma \vdash \langle N, P\rangle : A \times B]\!] = h : [\![\Gamma]\!] \to [\![A]\!] \times [\![B]\!]$$

  by $h(\bar{a}) = (f(\bar{a}), g(\bar{a}))$, for all $\bar{a} \in [\![\Gamma]\!]$.

- If $M = \pi_i N$ is a projection (for $i = 1, 2$), we recursively calculate

$$f \quad = \quad [\![\Gamma \vdash N : B_1 \times B_2]\!] : [\![\Gamma]\!] \to [\![B_1]\!] \times [\![B_2]\!].$$

  We then define
$$[\![\Gamma \vdash \pi_i : B_i]\!] = h : [\![\Gamma]\!] \to [\![B_i]\!]$$
  by $h(\bar{a}) = \pi_i(f(\bar{a}))$, for all $\bar{a} \in [\![\Gamma]\!]$. Here π_i in the meta-language denotes the set-theoretic function $\pi_i : [\![B_1]\!] \times [\![B_2]\!] \to [\![B_i]\!]$ given by $\pi_i(b_1, b_2) = b_i$.

- If $M = *$, we define

$$[\![\Gamma \vdash * : 1]\!] = h : [\![\Gamma]\!] \to \{*\}$$

  by $h(\bar{a}) = *$, for all $\bar{a} \in [\![\Gamma]\!]$.

To minimize notational inconvenience, we will occasionally abuse the notation and write $[\![M]\!]$ instead of $[\![\Gamma \vdash M : B]\!]$, thus pretending that terms are typing judgments. However, this is only an abbreviation, and it will be understood that the interpretation really depends on the typing judgment, and not just the term, even if we use the abbreviated notation.

9.2 Soundness

Lemma 9.1 (Context change). *The interpretation behaves as expected under reordering of contexts and under the addition of dummy variables to contexts. More precisely, if $\sigma : \{1, \ldots, n\} \to \{1, \ldots, m\}$ is an injective map, and if the free variables of M are among $x_{\sigma 1}, \ldots, x_{\sigma} n$, then the interpretations of the two typing judgments,*

$$\begin{array}{l} f = [\![x_1{:}A_1, \ldots, x_m{:}A_m \vdash M : B]\!] : [\![A_1]\!] \times \ldots \times [\![A_m]\!] \to [\![B]\!], \\ g = [\![x_{\sigma 1}{:}A_{\sigma 1}, \ldots, x_{\sigma n}{:}A_{\sigma n} \vdash M : B]\!] : [\![A_{\sigma 1}]\!] \times \ldots \times [\![A_{\sigma n}]\!] \to [\![B]\!] \end{array}$$

are related as follows:

$$f(a_1, \ldots, a_m) = g(a_{\sigma 1}, \ldots, a_{\sigma n}),$$

for all $a_1 \in [\![A_1]\!], \ldots, a_m \in [\![A_m]\!]$.

Proof. Easy, but tedious, induction on M. □

The significance of this lemma is that, to a certain extent, the context does not matter. Thus, if the free variables of M and N are contained in Γ as well as Γ', then we have

$$[\![\Gamma \vdash M : B]\!] = [\![\Gamma \vdash N : B]\!] \qquad \text{iff} \qquad [\![\Gamma' \vdash M : B]\!] = [\![\Gamma' \vdash N : B]\!].$$

Thus, whether M and N have equal denotations only depends on M and N, and not on Γ.

Lemma 9.2 (Substitution Lemma). *If*

$$\begin{array}{ll} [\![\Gamma, x{:}A \vdash M : B]\!] = f : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!] & \textit{and} \\ [\![\Gamma \vdash N : A]\!] = g : [\![\Gamma]\!] \to [\![A]\!], & \end{array}$$

then

$$[\![\Gamma \vdash M[N/x] : B]\!] = h : [\![\Gamma]\!] \to [\![B]\!],$$

where $h(\bar{a}) = f(\bar{a}, g(\bar{a}))$, for all $\bar{a} \in [\![\Gamma]\!]$.

Proof. Very easy, but very tedious, induction on M. □

Proposition 9.3 (Soundness). *The set-theoretic interpretation is sound for $\beta\eta$-reasoning. In other words,*

$$M =_{\beta\eta} N \qquad \Rightarrow \qquad [\![\Gamma \vdash M : B]\!] = [\![\Gamma \vdash N : B]\!].$$

Proof. Let us write $M \sim N$ if $[\![\Gamma \vdash M : B]\!] = [\![Gamma \vdash N : B]\!]$. By the remark after Lemma 9.1, this notion is independent of Γ, and thus a well-defined relation on terms (as opposed to typing judgments). To prove soundness, we must show that $M =_{\beta\eta} N$ implies $M \sim N$, for all M and N. It suffices to show that $\sim$ satisfies all the axioms of $\beta\eta$-equivalence.

The axioms (*refl*), (*symm*), and (*trans*) hold trivially. Similarly, all the (*cong*) and (ξ) rules hold, due to the fact that the meaning of composite terms was defined solely in terms of the meaning of their subterms. It remains to prove that each of the various (β) and (η) laws is satisfied (see page 57). We prove the rule ($\beta_{\to}$) as an example; the remaining rules are left as an exercise.

Assume Γ is a context such that $\Gamma, x{:}A \vdash M : B$ and $\Gamma \vdash N : A$. Let

$$\begin{array}{l} f = [\![\Gamma, x{:}A \vdash M : B]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!], \\ g = [\![\Gamma \vdash N : A]\!] : [\![\Gamma]\!] \to [\![A]\!], \\ h = [\![\Gamma \vdash (\lambda x^A.M) : A \to B]\!] : [\![\Gamma]\!] \to [\![B]\!]^{[\![A]\!]}, \\ k = [\![\Gamma \vdash (\lambda x^A.M)N : B]\!] : [\![\Gamma]\!] \to [\![B]\!], \\ l = [\![\Gamma \vdash M[N/x] : B]\!] : [\![\Gamma]\!] \to [\![B]\!]. \end{array}$$

We must show $k = h$. By definition, we have $k(\bar{a}) = h(\bar{a})(g(\bar{a})) = f(\bar{a}, g(\bar{a}))$. On the other hand, $l(\bar{a}) = f(\bar{a}, g(\bar{a}))$ by the substitution lemma. □

Note that the proof of soundness amounts to a simple calculation; while there are many details to attend to, no particularly interesting new idea is required. This is typical of soundness proofs in general. Completeness, on the other hand, is usually much more difficult to prove and often requires clever ideas.

9.3 Completeness

We cite two completeness theorems for the set-theoretic interpretation. The first one is for the class of all models with finite base type. The second one is for the single model with one countably infinite base type.

Theorem 9.4 (Completeness, Plotkin, 1973). *The class of set-theoretic models with finite base types is complete for the lambda-$\beta\eta$ calculus.*

Recall that completeness for a class of models means that if $[\![M]\!] = [\![N]\!]$ holds in *all* models of the given class, then $M =_{\beta\eta} N$. This is not the same as completeness for each individual model in the class.

Note that, for each *fixed* choice of finite sets as the interpretations of the base types, there are some lambda terms such that $[\![M]\!] = [\![N]\!]$ but $M \neq_{\beta\eta} N$. For instance, consider terms of type $(\iota \to \iota) \to \iota \to \iota$. There are infinitely many $\beta\eta$-distinct terms of this type, namely, the Church numerals. On the other hand, if S_ι is a finite set, then $[\![(\iota \to \iota) \to \iota \to \iota]\!]$ is also a finite set. Since a finite set cannot have infinitely many distinct elements, there must necessarily be two distinct Church numerals M, N such that $[\![M]\!] = [\![N]\!]$.

Plotkin's completeness theorem, on the other hand, shows that whenever M and N are distinct lambda terms, then there exist *some* set-theoretic model with finite base types in which M and N are different.

The second completeness theorem is for a *single* model, namely the one where S_ι is a countably infinite set.

Theorem 9.5 (Completeness, Friedman, 1975). *The set-theoretic model with base type equal to $\mathbb{N}$, the set of natural numbers, is complete for the lambda-$\beta\eta$ calculus.*

We omit the proofs.

10 The language PCF

PCF stands for "programming with computable functions". The language PCF is an extension of the simply-typed lambda calculus with booleans, natural numbers, and recursion. It was first introduced by Dana Scott as a simple programming language on which to try out techniques for reasoning about programs. Although PCF is not intended as a "real world" programming language, many real programming languages can be regarded as (syntactic variants of) extensions of PCF, and many of the reasoning techniques developed for PCF also apply to more complicated languages.

PCF is a "programming language", not just a "calculus". By this we mean, PCF is equipped with a specific evaluation order, or rules that determine precisely how terms are to be evaluated. We follow the slogan:

Programming language = syntax + evaluation rules.

After introducting the syntax of PCF, we will look at three different equivalence relations on terms.

- *Axiomatic equivalence* $=_{\mathrm{ax}}$ will be given by axioms in the spirit of $\beta\eta$-equivalence.
- *Operational equivalence* $=_{\mathrm{op}}$ will be defined in terms of the operational behavior of terms. Two terms are operationally equivalent if one can be substituted for the other in any context without changing the behavior of a program.
- *Denotational equivalence* $=_{\mathrm{den}}$ is defined via a denotational semantics.

We will develop methods for reasoning about these equivalences, and thus for reasoning about programs. We will also investigate how the three equivalences are related to each other.

10.1 Syntax and typing rules

PCF types are simple types over two base types **bool** and **nat**.

$$A, B ::= \mathbf{bool} \mid \mathbf{nat} \mid A \to B \mid A \times B \mid 1$$

$$(\mathit{true})\ \frac{}{\Gamma \vdash \mathbf{T} : \mathbf{bool}} \qquad (\mathit{pred})\ \frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{pred}\,(M) : \mathbf{nat}}$$

$$(\mathit{false})\ \frac{}{\Gamma \vdash \mathbf{F} : \mathbf{bool}} \qquad (\mathit{iszero})\ \frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{iszero}\,(M) : \mathbf{bool}}$$

$$(\mathit{zero})\ \frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{nat}} \qquad (\mathit{fix})\ \frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathbf{Y}(M) : A}$$

$$(\mathit{succ})\ \frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{succ}\,(M) : \mathbf{nat}}$$

$$(\mathit{if})\ \frac{\Gamma \vdash M : \mathbf{bool} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : A}{\Gamma \vdash \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P : A}$$

Table 6: Typing rules for PCF

The raw terms of PCF are those of the simply-typed lambda calculus, together with some additional constructs that deal with booleans, natural numbers, and recursion.

$$\begin{array}{rcl} M, N, P & ::= & x \mid MN \mid \lambda x^A.M \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M \mid * \\ & & \mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{zero} \mid \mathbf{succ}\,(M) \mid \mathbf{pred}\,(M) \\ & & \mid \mathbf{iszero}\,(M) \mid \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \mid \mathbf{Y}(M) \end{array}$$

The intended meaning of these terms is the same as that of the corresponding terms we used to program in the untyped lambda calculus: **T** and **F** are the boolean constants, **zero** is the constant zero, **succ** and **pred** are the successor and predecessor functions, **iszero** tests whether a given number is equal to zero, **if** M **then** N **else** P is a conditional, and $\mathbf{Y}(M)$ is a fixpoint of M.

The typing rules for PCF are the same as the typing rules for the simply-typed lambda calculus, shown in Table 4, plus the additional typing rules shown in Table 6.

10.2 Axiomatic equivalence

The axiomatic equivalence of PCF is based on the $\beta\eta$-equivalence of the simply-typed lambda calculus. The relation $=_{\mathrm{ax}}$ is the least relation given by the following:

$$\begin{array}{rcl}
\textbf{pred}\,(\textbf{zero}\,) & = & \textbf{zero} \\
\textbf{pred}\,(\textbf{succ}\,(\underline{n})) & = & \underline{n} \\
\textbf{iszero}\,(\textbf{zero}\,) & = & \mathbf{T} \\
\textbf{iszero}\,(\textbf{succ}\,(\underline{n})) & = & \mathbf{F} \\
\textbf{if }\mathbf{T}\textbf{ then }N\textbf{ else }P & = & N \\
\textbf{if }\mathbf{F}\textbf{ then }N\textbf{ else }P & = & P \\
\mathbf{Y}(M) & = & M(\mathbf{Y}(M))
\end{array}$$

Table 7: Axiomatic equivalence for PCF

- All the β- and η-axioms of the simply-typed lambda calculus, as shown on page 57.
- One congruence or ξ-rule for each term constructor. This means, for instance
$$\frac{M =_{\rm ax} M' \qquad N =_{\rm ax} N' \qquad P =_{\rm ax} P'}{\textbf{if } M \textbf{ then } N \textbf{ else } P =_{\rm ax} \textbf{if } M' \textbf{ then } N' \textbf{ else } P'},$$
and similar for all the other term constructors.
- The additional axioms shown in Table 7. Here, $\underline{n}$ stands for a *numeral*, i.e., a term of the form **succ** $(\ldots(\textbf{succ}\,(\textbf{zero}\,))\ldots)$.

10.3 Operational semantics

The operational semantics of PCF is commonly given in two different styles: the *small-step* or *shallow* style, and the *big-step* or *deep* style. We give the small-step semantics first, because it is closer to the notion of β-reduction that we considered for the simply-typed lambda calculus.

There are some important differences between an operational semantics, as we are going to give it here, and the notion of β-reduction in the simply-typed lambda calculus. Most importantly, the operational semantics is going to be *deterministic*, which means, each term can be reduced in at most one way. Thus, there will never be a choice between more than one redex. Or in other words, it will always be uniquely specified which redex to reduce next.

As a consequence of the previous paragraph, we will abandon many of the congruence rules, as well as the (ξ)-rule. We adopt the following informal conventions:

- never reduce the body of a lambda abstraction,
- never reduce the argument of a function (except for primitive functions such as **succ** and **pred**),
- never reduce the "then" or "else" part of an if-then-else statement,
- never reduce a term inside a pair.

Of course, the terms that these rules prevent from being reduced can nevertheless become subject to reduction later: the body of a lambda abstraction and the argument of a function can be reduced after a β-reduction causes the λ to disappear and the argument to be substituted in the body. The "then" or "else" parts of an if-then-else term can be reduced after the "if" part evaluates to true or false. And the terms inside a pair can be reduced after the pair has been broken up by a projection.

An important technical notion is that of a *value*, which is a term that represents the result of a computation and cannot be reduced further. Values are given as follows:

$$\text{Values:}\quad V, W ::= \mathbf{T} \mid \mathbf{F} \mid \textbf{zero} \mid \textbf{succ}\,(V) \mid * \mid \langle M, N\rangle \mid \lambda x^A.M$$

The transition rules for the small-step operational semantics of PCF are shown in Table 8.

We write $M \to N$ if M reduces to N by these rules. We write $M \not\to$ if there does not exist N such that $M \to N$. The first two important technical properties of small-step reduction are summarized in the following lemma.

Lemma 10.1. *1.* Values are normal forms. *If V is a value, then $V \not\to$.*

2. Evalution is deterministic. *If $M \to N$ and $M \to N'$, then $N \equiv N'$.*

Another important property is subject reduction: a well-typed term reduces only to another well-typed term of the same type.

Lemma 10.2 (Subject Reduction). *If $\Gamma \vdash M : A$ and $M \to N$, then $\Gamma \vdash N : A$.*

Next, we want to prove that the evaluation of a well-typed term does not get "stuck". If M is some term such that $M \not\to$, but M is not a value, then we regard this as an error, and we also write $M \to$ **error** . Examples of such terms are $\pi_1(\lambda x.M)$ and $\langle M, N\rangle P$. The following lemma shows that well-typed closed terms cannot lead to such errors.

$$\frac{M \to N}{\textbf{pred}\,(M) \to \textbf{pred}\,(N)}$$

$$\frac{}{\textbf{pred}\,(\textbf{zero}\,) \to \textbf{zero}}$$

$$\frac{}{\textbf{pred}\,(\textbf{succ}\,(V)) \to V}$$

$$\frac{M \to N}{\textbf{iszero}\,(M) \to \textbf{iszero}\,(N)}$$

$$\frac{}{\textbf{iszero}\,(\textbf{zero}\,) \to \textbf{T}}$$

$$\frac{}{\textbf{iszero}\,(\textbf{succ}\,(V)) \to \textbf{F}}$$

$$\frac{M \to N}{\textbf{succ}\,(M) \to \textbf{succ}\,(N)}$$

$$\frac{M \to N}{MP \to NP}$$

$$\frac{}{(\lambda x^A.M)N \to M[N/x]}$$

$$\frac{M \to M'}{\pi_i M \to \pi_i M'}$$

$$\frac{}{\pi_1\langle M,N\rangle \to M}$$

$$\frac{}{\pi_2\langle M,N\rangle \to N}$$

$$\frac{M : 1, \quad M \neq *}{M \to *}$$

$$\frac{M \to M'}{\textbf{if } M \textbf{ then } N \textbf{ else } P \to \textbf{if } M' \textbf{ then } N \textbf{ else } P}$$

$$\frac{}{\textbf{if T then } N \textbf{ else } P \to N}$$

$$\frac{}{\textbf{if F then } N \textbf{ else } P \to P}$$

$$\frac{}{\mathbf{Y}(M) \to M(\mathbf{Y}(M))}$$

Table 8: Small-step operational semantics of PCF

Lemma 10.3 (Progress). *If M is a closed, well-typed term, then either M is a value, or else there exists N such that $M \to N$.*

The Progress Lemma is very important, because it implies that a well-typed term cannot “go wrong”. It guarantees that a well-typed term will either evaluate to a value in finitely many steps, or else it will reduce infinitely and thus not terminate. But a well-typed term can never generate an error. In programming language terms, a term that type-checks at *compile-time* cannot generate an error at *run-time*.

To express this idea formally, let us write $M \to^* N$ in the usual way if M reduces to N in zero or more steps, and let us write $M \to^* \textbf{error}$ if M reduces in zero or more steps to an error.

Proposition 10.4 (Safety). *If M is a closed, well-typed term, then $M \not\to^* \textbf{error}$.*

Exercise 32. Prove Lemmas 10.1–10.3 and Proposition 10.4.

$$\frac{}{\textbf{T} \Downarrow \textbf{T}}$$

$$\frac{}{\textbf{F} \Downarrow \textbf{F}}$$

$$\frac{}{\textbf{zero} \Downarrow \textbf{zero}}$$

$$\frac{}{\langle M,N\rangle \Downarrow \langle M,N\rangle}$$

$$\frac{}{\lambda x^A.M \Downarrow \lambda x^A.M}$$

$$\frac{M \Downarrow \textbf{zero}}{\textbf{pred}\,(M) \Downarrow \textbf{zero}}$$

$$\frac{M \Downarrow \textbf{succ}\,(V)}{\textbf{pred}\,(M) \Downarrow V}$$

$$\frac{M \Downarrow \textbf{zero}}{\textbf{iszero}\,(M) \Downarrow \textbf{T}}$$

$$\frac{M \Downarrow \textbf{succ}\,(V)}{\textbf{iszero}\,(M) \Downarrow \textbf{F}}$$

$$\frac{M \Downarrow V}{\textbf{succ}\,(M) \Downarrow \textbf{succ}\,(V)}$$

$$\frac{M \Downarrow \lambda x^A.M' \qquad M'[N/x] \Downarrow V}{MN \Downarrow V}$$

$$\frac{M \Downarrow \langle M_1, M_2\rangle \qquad M_1 \Downarrow V}{\pi_1 M \Downarrow V}$$

$$\frac{M \Downarrow \langle M_1, M_2\rangle \qquad M_2 \Downarrow V}{\pi_2 M \Downarrow V}$$

$$\frac{M : 1}{M \Downarrow *}$$

$$\frac{M \Downarrow \textbf{T} \qquad N \Downarrow V}{\textbf{if } M \textbf{ then } N \textbf{ else } P \Downarrow V}$$

$$\frac{M \Downarrow \textbf{F} \qquad P \Downarrow V}{\textbf{if } M \textbf{ then } N \textbf{ else } P \Downarrow V}$$

$$\frac{M(\mathbf{Y}(M)) \Downarrow V}{\mathbf{Y}(M) \Downarrow V}$$

Table 9: Big-step operational semantics of PCF

10.4 Big-step semantics

In the small-step semantics, if $M \to^* V$, we say that M *evaluates to* V. Note that by determinacy, for every M, there exists at most one V such that $M \to^* V$.

It is also possible to axiomatize the relation “M evaluates to V” directly. This is known as the big-step semantics. Here, we write $M \Downarrow V$ if M evaluates to V. The axioms for the big-step semantics are shown in Table 9.

The big-step semantics satisfies properties similar to those of the small-step semantics.

Lemma 10.5. 1. Values. *For all values V, we have $V \Downarrow V$.*

2. Determinacy. *If $M \Downarrow V$ and $M \Downarrow V'$, then $V \equiv V'$.*

3. Subject Reduction. *If $\Gamma \vdash M : A$ and $M \Downarrow V$, then $\Gamma \vdash V : A$.*

The analogues of the Progress and Safety properties cannot be as easily stated for big-step reduction, because we cannot easily talk about a single reduction step or about infinite reduction sequences. However, some comfort can be taken in the fact that the big-step semantics and small-step semantics coincide:

Proposition 10.6. *$M \to^* V$ iff $M \Downarrow V$.*

10.5 Operational equivalence

Informally, two terms M and N will be called operationally equivalent if M and N are interchangeable as part of any larger program, without changing the observable behavior of the program. This notion of equivalence is also often called observational equivalence, to emphasize the fact that it concentrates on observable properties of terms.

What is an observable behavior of a program? Normally, what we observe about a program is its output, such as the characters it prints to a terminal. Since any such characters can be converted in principle to natural numbers, we take the point of view that the observable behavior of a program is a natural number that it evaluates to. Similarly, if a program computes a boolean, we regard the boolean value as observable. However, we do not regard abstract values, such as functions, as being directly observable, on the grounds that a function cannot be observed until we supply it some arguments and observe the result.

Definition. An *observable type* is either **bool** or **nat**. A *result* is a closed value of observable type. Thus, a result is either $\mathbf{T}$, $\mathbf{F}$, or $\underline{n}$. A *program* is a closed term of observable type

A *context* is a term with a hole, written $C[-]$. Formally, the class of contexts is defined by a BNF:

$$C[-] \quad ::= \quad [-] \mid x \mid C[-]N \mid MC[-] \mid \lambda x^A.C[-] \mid \ldots$$

and so on, extending through all the cases in the definition of a PCF term.

Well-typed contexts are defined in the same way as well-typed terms, where it is understood that the hole also has a type. The free variables of a context are defined in the same way as for terms. Moreover, we define the *captured variables* of a context to be those bound variables whose scope includes the hole. So for instance, in the context $(\lambda x.[-])(\lambda y.z)$, the variable x is captured, the variable z is free, and y is neither free nor captured.

If $C[-]$ is a context and M is a term of the appropriate type, we write $C[M]$ for the result of replacing the hole in the context $C[-]$ by M. Here, we do not α-rename any bound variables, so that we allow free variables of M to be captured by $C[-]$.

We are now ready to state the definition of operational equivalence.

Definition. Two terms M, N are *operationally equivalent*, in symbols $M =_{\mathrm{op}} N$, if for all closed and closing context $C[-]$ of observable type and all values V,

$$C[M] \Downarrow V \iff C[N] \Downarrow V.$$

Here, by a *closing* context we mean that $C[-]$ should capture all the free variables of M and N. This is equivalent to requiring that $C[M]$ and $C[N]$ are closed terms of observable types, i.e., programs. Thus, two terms are equivalent if they can be used interchangeably in any program.

10.6 Operational approximation

As a refinement of operational equivalence, we can also define a notion of operational approximation: We say that M *operationally approximates* N, in symbols $M \sqsubseteq_{\mathrm{op}} N$, if for all closed and closing contexts $C[-]$ of observable type and all values V,

$$C[M] \Downarrow V \Rightarrow C[N] \Downarrow V.$$

Note that this definition includes the case where $C[M]$ diverges, but $C[N]$ converges, for some N. This formalizes the notion that N is "more defined" than M. Clearly, we have $M =_{\mathrm{op}} N$ iff $M \sqsubseteq_{\mathrm{op}} N$ and $N \sqsubseteq_{\mathrm{op}} M$. Thus, we get a partial order $\sqsubseteq_{\mathrm{op}}$ on the set of all terms of a given type, modulo operational equivalence. Also, this partial order has a least element, namely if we let $\Omega = \mathbf{Y}(\lambda x.x)$, then $\Omega \sqsubseteq_{\mathrm{op}} N$ for any term N of the appropriate type.

Note that, in general, $\sqsubseteq_{\mathrm{op}}$ is not a complete partial order, due to missing limits of ω-chains.

10.7 Discussion of operational equivalence

Operational equivalence is a very useful concept for reasoning about programs, and particularly for reasoning about program fragments. If M and N are operationally equivalent, then we know that we can replace M by N in any program without affecting its behavior. For example, M could be a slow, but simple subroutine for sorting a list. The term N could be a replacement that runs much faster. If we can prove M and N to be operationally equivalent, then this means we can safely use the faster routine instead of the slower one.

Another example are compiler optimizations. Many compilers will try to optimize the code that they produce, to eliminate useless instructions, to avoid duplicate calculations, etc. Such an optimization often means replacing a piece of code M by another piece of code N, without necessarily knowing much about the context in which M is used. Such a replacement is safe if M and N are operationally equivalent.

On the other hand, operational equivalence is a somewhat problematic notion. The problem is that the concept is not stable under adding new language features. It can happen that two terms, M and N, are operationally equivalent, but when a new feature is added to the language, they become unequivalent, *even if M and N do not use the new feature*. The reason is the operational equivalence is defined in terms of contexts. Adding new features to a language also means that there will be new contexts, and these new contexts might be able to distinguish M and N.

This can be a problem in practice. Certain compiler optimizations might be sound for a sequential language, but might become unsound if new language features are added. Code that used to be correct might suddenly become incorrect if used in a richer environment. For example, many programs and library functions in C assume that they are executed in a single-threaded environment. If this code is ported to a multi-threaded environment, it often turns out to be no longer correct, and in many cases it must be re-written from scratch.

10.8 Operational equivalence and parallel or

Let us now look at a concrete example in PCF. We say that a term **POR** implements the *parallel or* function if it has the following behavior:

$$\begin{array}{lcll} \textbf{POR}\ \mathbf{T}P & \to & \mathbf{T}, & \text{for all } P \\ \textbf{POR}\ N\mathbf{T} & \to & \mathbf{T}, & \text{for all } N \\ \textbf{POR}\ \mathbf{FF} & \to & \mathbf{F}. & \end{array}$$

Note that this in particular implies $\textbf{POR}\ \mathbf{T}\Omega = \mathbf{T}$ and $\textbf{POR}\ \Omega\mathbf{T} = \mathbf{T}$, where Ω is some divergent term. It should be clear why **POR** is called the "parallel" or: the only way to achieve such behavior is to evaluate both its arguments in parallel, and to stop as soon as one argument evaluates to $\mathbf{T}$ or both evaluate to $\mathbf{F}$.

Proposition 10.7. **POR** *is not definable in PCF.*

We do not give the proof of this fact, but the idea is relatively simple: one proves by induction that every PCF context $C[-,-]$ with two holes has the following property: either, there exists a term N such that $C[M,M'] = N$ for all M,M' (i.e., the context does not look at M,M' at all), or else, either $C[\Omega,M]$ diverges for all M, or $C[M,\Omega]$ diverges for all M. Here, again, Ω is some divergent term such as $\mathbf{Y}(\lambda x.x)$.

Although **POR** is not definable in PCF, we can define the following term, called the *POR-tester*:

```
POR-test = λx.if xTΩ then
                   if xΩT then
                       if xFF then Ω
                       else T
                   else Ω
               else Ω
```

The POR-tester has the property that **POR-test** $M = \mathbf{T}$ if M implements the parallel or function, and in all other cases **POR-test** M diverges. In particular, since parallel or is not definable in PCF, we have that **POR-test** M diverges, for all PCF terms M. Thus, when applied to any PCF term, **POR-test** behaves precisely as the function $\lambda x.\Omega$ does. One can make this into a rigorous argument that shows that **POR-test** and $\lambda x.\Omega$ are operationally equivalent:

$$\textbf{POR-test} =_{\text{op}} \lambda x.\Omega \qquad \text{(in PCF)}.$$

Now, suppose we want to define an extension of PCF called *parallel PCF*. It is defined in exactly the same way as PCF, except that we add a new primitive function **POR**, and small-step reduction rules

$$\frac{M \to M' \qquad N \to N'}{\textbf{POR}\ MN \to \textbf{POR}\ M'N'}$$

$$\frac{}{\textbf{POR}\ \mathbf{T}N \to \mathbf{T}}$$

$$\frac{}{\textbf{POR}\ M\mathbf{T} \to \mathbf{T}}$$

$$\frac{}{\textbf{POR}\ \mathbf{FF} \to \mathbf{F}}$$

Parallel PCF enjoys many of the same properties as PCF, for instance, Lemmas 10.1–10.3 and Proposition 10.4 continue to hold for it.

But notice that

$$\textbf{POR-test} \neq_{\text{op}} \lambda x.\Omega \qquad \text{(in parallel PCF).}$$

This is because the context $C[-] = [-]\,\textbf{POR}$ distinguishes the two terms: clearly, $C[\textbf{POR-test}\,] \Downarrow \textbf{T}$, whereas $C[\lambda x.\Omega]$ diverges.

11 Complete partial orders

11.1 Why are sets not enough, in general?

As we have seen in Section 9, the interpretation of types as plain sets is quite sufficient for the simply-typed lambda calculus. However, it is insufficient for a language such as PCF. Specifically, the problem is the fixpoint operator $\mathbf{Y} : (A \to A) \to A$. It is clear that there are many functions $f : A \to A$ from a set A to itself that do not have a fixpoint; thus, there is no chance we are going to find an interpretation for a fixpoint operator in the simple set-theoretic model.

On the other hand, if A and B are types, there are generally many functions $f : [\![A]\!] \to [\![B]\!]$ in the set-theoretic model that are not definable by lambda terms. For instance, if $[\![A]\!]$ and $[\![B]\!]$ are infinite sets, then there are uncountably many functions $f : [\![A]\!] \to [\![B]\!]$; however, there are only countably many lambda terms, and thus there are necessarily going to be functions that are not the denotation of any lambda term.

The idea is to put additional structure on the sets that interpret types, and to require functions to preserve that structure. This is going to cut down the size of the function spaces, decreasing the “slack” between the functions definable in the lambda calculus and the functions that exist in the model, and simultaneously increasing the chances that additional structure, such as fixpoint operators, might exist in the model.

Complete partial orders are one such structure that is commonly used for this purpose. The method is originally due to Dana Scott.

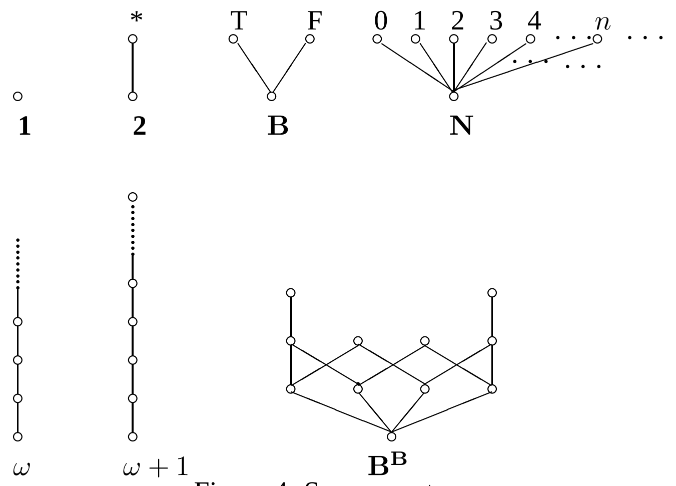

Figure 4: Some posets

11.2 Complete partial orders

Definition. A *partially ordered set* or *poset* is a set X together with a binary relation $\sqsubseteq$ satisfying

- *reflexivity:* for all $x \in X$, $x \sqsubseteq x$,
- *antisymmetry:* for all $x, y \in X$, $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$,
- *transitivity:* for all $x, y, z \in X$, $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.

The concept of a partial order differs from a total order in that we do not require that for any x and y, either $x \sqsubseteq y$ or $y \sqsubseteq x$. Thus, in a partially ordered set it is permissible to have incomparable elements.

We can often visualize posets, particularly finite ones, by drawing their line diagrams as in Figure 4. In these diagrams, we put one circle for each element of X, and we draw an edge from x upward to y if $x \sqsubseteq y$ and there is no z with $x \sqsubseteq z \sqsubseteq y$. Such line diagrams are also known as *Hasse diagrams*.

The idea behind using a partial order to denote computational values is that $x \sqsubseteq y$ means that x is *less defined than* y. For instance, if a certain term diverges, then its denotation will be less defined than, or below that of a term that has a definite value. Similarly, a function is more defined than another if it converges on more inputs.

Another important idea in using posets for modeling computational value is that of *approximation*. We can think of some infinite computational object (such as, an infinite stream), to be a limit of successive finite approximations (such as, longer and longer finite streams). Thus we also read $x \sqsubseteq y$ as x *approximates* y. A complete partial order is a poset in which every countable chain of increasing elements approximates something.

Definition. Let X be a poset and let $A \subseteq X$ be a subset. We say that $x \in X$ is an *upper bound* for A if $x \sqsubseteq a$ for all $a \in A$. We say that x is a *least upper bound* for A if x is an upper bound, and whenever y is also an upper bound, then $x \sqsubseteq y$.

Definition. An ω*-chain* in a poset X is a sequence of elements $x_0, x_1, x_2, \ldots$ such that

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$$

Definition. A *complete partial order (cpo)* is a poset such that every ω-chain of elements has a least upper bound.

If $x_0, x_1, x_2, \ldots$ is an ω-chain of elements in a cpo, we write $\bigvee^{\uparrow}_{i\in\mathbb{N}} x_i$ for the least upper bound. We also call the least upper bound the *limit* of the ω-chain.

Not every poset is a cpo. In Figure 4, the poset labeled ω is not a cpo, because the evident ω-chain does not have a least upper bound (in fact, it has no upper bound at all). The other posets shown in Figure 4 are cpos.

11.3 Properties of limits

Proposition 11.1. *1.* Monotonicity. *Suppose $\{x_i\}_i$ and $\{y_i\}_i$ are ω-chains in a cpo C, such that $x_i \sqsubseteq y_i$ for all i. Then*

$$\bigvee^{\uparrow}_{i} x_i \sqsubseteq \bigvee^{\uparrow}_{i} y_i.$$

2. Exchange. *Suppose $\{x_{ij}\}_{i,j\in\mathbb{N}}$ is a doubly monotone double sequence of elements of a cpo C, i.e., whenever $i \leqslant i'$ and $j \leqslant j'$, then $x_{ij} \sqsubseteq x_{i'j'}$. Then*

$$\bigvee^{\uparrow}_{i\in\mathbb{N}} \bigvee^{\uparrow}_{j\in\mathbb{N}} x_{ij} = \bigvee^{\uparrow}_{j\in\mathbb{N}} \bigvee^{\uparrow}_{i\in\mathbb{N}} x_{ij} = \bigvee^{\uparrow}_{k\in\mathbb{N}} x_{kk}.$$

In particular, all limits shown are well-defined.

Exercise 33. Prove Proposition 11.1.

11.4 Continuous functions

If we model data types as cpo's, it is natural to model algorithms as functions from cpo's to cpo's. These functions are subject to two constraints: they have to be monotone and continuous.

Definition. A function $f : C \to D$ between posets C and D is said to be *monotone* if for all $x, y \in C$,

$$x \sqsubseteq y \qquad \Rightarrow \qquad f(x) \sqsubseteq f(y).$$

A function $f : C \to D$ between cpo's C and D is said to be *continuous* if it is monotone and it preserves least upper bounds of ω-chains, i.e., for all ω-chains $\{x_i\}_{i\in\mathbb{N}}$ in C,

$$f(\bigvee^{\uparrow}_{i\in\mathbb{N}} x_i) = \bigvee^{\uparrow}_{i\in\mathbb{N}} f(x_i).$$

The intuitive explanation for the monotonicity requirement is that information is "positive": more information in the input cannot lead to less information in the output of an algorithm. The intuitive explanation for the continuity requirement is that any particular output of an algorithm can only depend on a finite amount of input.

11.5 Pointed cpo's and strict functions

Definition. A cpo is said to be *pointed* if it has a least element. The least element is usually denoted $\bot$ and pronounced "bottom". All cpo's shown in Figure 4 are pointed.

A coninuous function between pointed cpo's is said to be *strict* if it preserves the bottom element.

11.6 Products and function spaces

If C and D are cpo's, then their *cartesian product* $C \times D$ is also a cpo, with the pointwise order given by $(x, y) \sqsubseteq (x', y')$ iff $x \sqsubseteq x'$ and $y \sqsubseteq y'$. Least upper bounds are also given pointwise, thus

$$\bigvee^{\uparrow}_{i}(x_i, y_i) = (\bigvee^{\uparrow}_{i} x_i, \bigvee^{\uparrow}_{i} y_i).$$

Proposition 11.2. *The first and second projections, $\pi_1 : C \times D \to C$ and $\pi_2 : C \times D \to D$, are continuous functions. Moreover, if $f : E \to C$ and $g : E \to D$ are continuous functions, then so is the function $h : E \to C \times D$ given by $h(z) = (f(z), g(z))$.*

If C and D are cpo's, then the set of continuous functions $f : C \to D$ forms a cpo, denoted D^C. The order is given pointwise: given two functions $f, g : C \to D$, we say that

$$f \sqsubseteq g \qquad \text{iff} \qquad \text{for all } x \in C,\ f(x) \sqsubseteq g(x).$$

Proposition 11.3. *The set D^C of continuous functions from C to D, together with the order just defined, is a complete partial order.*

Proof. Clearly the set D^C is partially ordered. What we must show is that least upper bounds of ω-chains exist. Given an ω-chain $f_0, f_1, \ldots$ in D^C, we define $g \in D^C$ to be the pointwise limit, i.e.,

$$g(x) = \bigvee_{i\in\mathbb{N}}^{\uparrow} f_i(x),$$

for all $x \in C$. Note that $\{f_i(x)\}_i$ does indeed form an ω-chain in C, so that g is a well-defined function. We claim that g is the least upper bound of $\{f_i\}_i$. First we need to show that g is indeed an element of D^C. To see that g is monotone, we use Proposition 11.1(1) and calculate, for any $x \sqsubseteq y \in C$,

$$g(x) = \bigvee_{i\in\mathbb{N}}^{\uparrow} f_i(x) \sqsubseteq \bigvee_{i\in\mathbb{N}}^{\uparrow} f_i(y) = g(y).$$

To see that g is continuous, we use Proposition 11.1(2) and calculate, for any ω-chain $x_0, x_1, \ldots$ in C,

$$g(\bigvee_j^{\uparrow} x_j) = \bigvee_i^{\uparrow}\bigvee_j^{\uparrow} f_i(x_j) = \bigvee_j^{\uparrow}\bigvee_i^{\uparrow} f_i(x_j) = \bigvee_j^{\uparrow} g(x_j).$$

Finally, we must show that g is the least upper bound of the $\{f_i\}_i$. Clearly, $f_i \sqsubseteq g$ for all i, so that g is an upper bound. Now suppose $h \in D^C$ is any other upper bound of $\{f_i\}$. Then for all x, $f_i(x) \sqsubseteq h(x)$. Since $g(x)$ was defined to be the least upper bound of $\{f_i(x)\}_i$, we then have $g(x) \sqsubseteq h(x)$. Since this holds for all x, we have $g \sqsubseteq h$. Thus g is indeed the least upper bound.

Exercise 34. Recall the cpo $\mathbf{B}$ from Figure 4. The cpo $\mathbf{B}^{\mathbf{B}}$ is also shown in Figure 4. Its 11 elements correspond to the 11 continuous functions from $\mathbf{B}$ to $\mathbf{B}$. Label the elements of $\mathbf{B}^{\mathbf{B}}$ with the functions they correspond to.

Proposition 11.4. *The application function $D^C \times C \to D$, which maps (f, x) to $f(x)$, is continuous.*

Proposition 11.5. *Continuous functions can be continuously curried and uncurried. In other words, if $f : C \times D \to E$ is a continuous function, then $f^* : C \to E^D$, defined by $f^*(x)(y) = f(x, y)$, is well-defined and continuous. Conversely, if $g : C \to E^D$ is a continuous function, then $g_* : C \times D \to E$, defined by $g_*(x, y) = g(x)(y)$, is well-defined and continuous. Moreover, $(f^*)_* = f$ and $(g_*)^* = g$.*

11.7 The interpretation of the simply-typed lambda calculus in complete partial orders

The interpretation of the simply-typed lambda calculus in cpo's resembles the set-theoretic interpretation, except that types are interpreted by cpo's instead of sets, and typing judgments are interpreted as continuous functions.

For each basic type ι, assume that we have chosen a pointed cpo S_ι. We can then associate a pointed cpo $[\![A]\!]$ to each type A recursively:

$$\begin{array}{lcl}
[\![\iota]\!] & = & S_\iota \\
[\![A \to B]\!] & = & [\![B]\!]^{[\![A]\!]} \\
[\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![1]\!] & = & \mathbf{1}
\end{array}$$

Typing judgments are now interpreted as continuous functions

$$[\![A_1]\!] \times \ldots \times [\![A_n]\!] \to [\![B]\!]$$

in precisely the same way as they were defined for the set-theoretic interpretation. The only thing we need to check, at every step, is that the function defined is indeed continuous. For variables, this follows from the fact that projections of cartesian products are continuous (Proposition 11.2). For applications, we use the fact that the application function of cpo's is continuous (Proposition 11.4), and for lambda-abstractions, we use the fact that currying is a well-defined, continuous operation (Proposition 11.5). Finally, the continuity of the maps associated with products and projections follows from Proposition 11.2.

Proposition 11.6 (Soundness and Completeness). *The interpretation of the simply-typed lambda calculus in pointed cpo's is sound and complete with respect to the lambda-$\beta\eta$ calculus.*

11.8 Cpo's and fixpoints

One of the reasons, mentioned in the introduction to this section, for using cpo's instead of sets for the interpretation of the simply-typed lambda calculus is that cpo's admit fixpoint, and thus they can be used to interpret an extension of the lambda calculus with a fixpoint operator.

Proposition 11.7. *Let C be a pointed cpo and let $f : C \to C$ be a continuous function. Then f has a least fixpoint.*

Proof. Define $x_0 = \bot$ and $x_{i+1} = f(x_i)$, for all $i \in \mathbb{N}$. The resulting sequence $\{x_i\}_i$ is an ω-chain, because clearly $x_0 \sqsubseteq x_1$ (since x_0 is the least element), and if $x_i \sqsubseteq x_{i+1}$, then $f(x_i) \sqsubseteq f(x_{i+1})$ by monotonicity, hence $x_{i+1} \sqsubseteq x_{i+2}$. It follows by induction that $x_i \sqsubseteq x_{i+1}$. Let $x = \bigvee^{\uparrow}_i x_i$ be the limit of this ω-chain. Then using continuity of f, we have

$$f(x) = f(\bigvee^{\uparrow}_i x_i) = \bigvee^{\uparrow}_i f(x_i) = \bigvee^{\uparrow}_i x_{i+1} = x.$$

To prove that it is the least fixpoint, let y be any other fixpoint, i.e., let $f(y) = y$. We prove by induction that for all i, $x_i \sqsubseteq y$. For $i = 0$ this is trivial because $x_0 = \bot$. Assume $x_i \sqsubseteq y$, then $x_{i+1} = f(x_i) \sqsubseteq f(y) = y$. It follows that y is an upper bound for $\{x_i\}_i$. Since x is, by definition, the least upper bound, we have $x \sqsubseteq y$. Since y was arbitrary, x is below any fixpoint, hence x is the least fixpoint of f. □

If $f : C \to C$ is any continuous function, let us write $f^\dagger$ for its least fixpoint. We claim that $f^\dagger$ depends continuously on f, i.e., that $\dagger : C^C \to C$ defines a continuous function.

Proposition 11.8. *The function $\dagger : C^C \to C$, which assigns to each continuous function $f \in C^C$ its least fixpoint $f^\dagger \in C$, is continuous.*

Exercise 35. Prove Proposition 11.8.

Thus, if we add to the simply-typed lambda calculus a family of fixpoint operators $Y_A : (A \to A) \to A$, the resulting extended lambda calculus can then be interpreted in cpo's by letting

$$[\![Y_A]\!] = \dagger : [\![A]\!]^{[\![A]\!]} \to [\![A]\!].$$

11.9 Example: Streams

Consider streams of characters from some alphabet A. Let $A^{\leqslant\omega}$ be the set of finite or infinite sequences of characters. We order A by the *prefix ordering*: if s and t are (finite or infinite) sequences, we say $s \sqsubseteq t$ if s is a prefix of t, i.e., if there exists a sequence s' such that $t = ss'$. Note that if $s \sqsubseteq t$ and s is an infinite sequence, then necessarily $s = t$, i.e., the infinite sequences are the maximal elements with respect to this order.

Exercise 36. Prove that the set $A^{\leqslant\omega}$ forms a cpo under the prefix ordering.

Exercise 37. Consider an automaton that reads characters from an input stream and writes characters to an output stream. For each input character read, it can write zero, one, or more output characters. Discuss how such an automaton gives rise to a continuous function from $A^{\leqslant\omega} \to A^{\leqslant\omega}$. In particular, explain the meaning of monotonicity and continuity in this context. Give some examples.

12 Denotational semantics of PCF

The denotational semantics of PCF is defined in terms of cpo's. It extends the cpo semantics of the simply-typed lambda calculus. Again, we assign a cpo $[\![A]\!]$ to each PCF type A, and a continuous function

$$[\![\Gamma \vdash M : B]\!] : [\![\Gamma]\!] \to [\![B]\!]$$

to every PCF typing judgment. The interpretation is defined in precisely the same way as for the simply-typed lambda calculus. The interpretation for the PCF-specific terms is shown in Table 10. Recall that $\mathbf{B}$ and $\mathbf{N}$ are the cpos of lifted booleans and lifted natural numbers, respectively, as shown in Figure 4.

Definition. Two PCF terms M and N of equal types are denotationally equivalent, in symbols $M =_{\text{den}} N$, if $[\![M]\!] = [\![N]\!]$. We also write $M \sqsubseteq_{\text{den}} N$ if $[\![M]\!] \sqsubseteq [\![N]\!]$.

12.1 Soundness and adequacy

We have now defined the three notions of equivalence on terms: $=_{\text{ax}}$, $=_{\text{op}}$, and $=_{\text{den}}$. In general, one does not expect the three equivalences to coincide. For example, any two divergent terms are operationally equivalent, but there is no

Types:	$[\![\textbf{bool}]\!]$	=	$\mathbf{B}$
	$[\![\textbf{nat}]\!]$	=	$\mathbf{N}$
Terms:	$[\![\mathbf{T}]\!]$	=	$T \in \mathbf{B}$
	$[\![\mathbf{F}]\!]$	=	$F \in \mathbf{B}$
	$[\![\textbf{zero}]\!]$	=	$0 \in \mathbf{N}$
	$[\![\textbf{succ}\,(M)]\!]$	=	$\begin{cases} \bot & \text{if } [\![M]\!] = \bot, \\ n+1 & \text{if } [\![M]\!] = n \end{cases}$
	$[\![\textbf{pred}\,(M)]\!]$	=	$\begin{cases} \bot & \text{if } [\![M]\!] = \bot, \\ 0 & \text{if } [\![M]\!] = 0, \\ n & \text{if } [\![M]\!] = n+1 \end{cases}$
	$[\![\textbf{iszero}\,(M)]\!]$	=	$\begin{cases} \bot & \text{if } [\![M]\!] = \bot, \\ \mathbf{T} & \text{if } [\![M]\!] = 0, \\ \mathbf{F} & \text{if } [\![M]\!] = n+1 \end{cases}$
	$[\![\textbf{if } M \textbf{ then } N \textbf{ else } P]\!]$	=	$\begin{cases} \bot & \text{if } [\![M]\!] = \bot, \\ [\![N]\!] & \text{if } [\![M]\!] = \mathbf{F}, \\ [\![P]\!] & \text{if } [\![M]\!] = \mathbf{T}, \end{cases}$
	$[\![\mathbf{Y}(M)]\!]$	=	$[\![M]\!]^{\dagger}$

Table 10: Cpo semantics of PCF

reason why they should be axiomatically equivalent. Also, the POR-tester and the term $\lambda x.\Omega$ are operationally equivalent in PCF, but they are not denotationally equivalent (since a function representing POR clearly exists in the cpo semantics). For general terms M and N, one has the following property:

Theorem 12.1 (Soundness). *For PCF terms M and N, the following implications hold:*

$$M =_{\text{ax}} N \qquad \Rightarrow \qquad M =_{\text{den}} N \qquad \Rightarrow \qquad M =_{\text{op}} N.$$

Soundness is a very useful property, because $M =_{\text{ax}} N$ is in general easier to prove than $M =_{\text{den}} N$, and $M =_{\text{den}} N$ is in turns easier to prove than $M =_{\text{op}} N$. Thus, soundness gives us a powerful proof method: to prove that two terms are operationally equivalent, it suffices to show that they are equivalent in the cpo semantics (if they are), or even that they are axiomatically equivalent.

As the above examples show, the converse implications are not in general true. However, the converse implications hold if the terms M and N are closed and of observable type, and if N is a value. This property is called computational adequacy. Recall that a program is a closed term of observable type, and a result is a closed value of observable type.

Theorem 12.2 (Computational Adequacy). *If M is a program and V is a result, then*

$$M =_{\text{ax}} V \qquad \Longleftrightarrow \qquad M =_{\text{den}} V \qquad \Longleftrightarrow \qquad M =_{\text{op}} V.$$

Proof. First note that the small-step semantics is contained in the axiomatic semantics, i.e., if $M \to N$, then $M =_{\text{ax}} N$. This is easily shown by induction on derivations of $M \to N$.

To prove the theorem, by soundness, it suffices to show that $M =_{\text{op}} V$ implies $M =_{\text{ax}} V$. So assume $M =_{\text{op}} V$. Since $V \Downarrow V$ and V is of observable type, it follows that $M \Downarrow V$. Therefore $M \to^* V$ by Proposition 10.6. But this already implies $M =_{\text{ax}} V$, and we are done. □

12.2 Full abstraction

We have already seen that the operational and denotational semantics do not coincide for PCF, i.e., there are some terms such that $M =_{\text{op}} N$ but $M \neq_{\text{den}} N$. Examples of such terms are **POR-test** and $\lambda x.\Omega$.

But of course, the particular denotational semantics that we gave to PCF is not the only possible denotational semantics. One can ask whether there is a better one. For instance, instead of cpo's, we could have used some other kind of mathematical space, such as a cpo with additional structure or properties, or some other kind of object altogether. The search for good denotational semantics is a subject of much research. The following terminology helps in defining precisely what is a "good" denotational semantics.

Definition. A denotational semantics is called *fully abstract* if for all terms M and N,

$$M =_{\mathrm{den}} N \qquad \Longleftrightarrow \qquad M =_{\mathrm{op}} N.$$

If the denotational semantics involves a partial order (such as a cpo semantics), it is also called *order fully abstract* if

$$M \sqsubseteq_{\mathrm{den}} N \qquad \Longleftrightarrow \qquad M \sqsubseteq_{\mathrm{op}} N.$$

The search for a fully abstract denotational semantics for PCF was an open problem for a very long time. Milner proved that there could be at most one such fully abstract model in a certain sense. This model has a syntactic description (essentially the elements of the model are PCF terms), but for a long time, no satisfactory semantic description was known. The problem has to do with sequentiality: a fully abstract model for PCF must be able to account for the fact that certain parallel constructs, such as parallel or, are not definable in PCF. Thus, the model should consist only of "sequential" functions. Berry and others developed a theory of "stable domain theory", which is based on cpo's with a additional properties intended to capture sequentiality. This research led to many interesting results, but the model still failed to be fully abstract.

Finally, in 1992, two competing teams of researchers, Abramsky, Jagadeesan and Malacaria, and Hyland and Ong, succeeded in giving a fully abstract semantics for PCF in terms of games and strategies. Games capture the interaction between a player and an opponent, or between a program and its environment. By considering certain kinds of "history-free" strategies, it is possible to capture the notion of sequentiality in just the right way to match PCF. In the last decade, game semantics has been extended to give fully abstract semantics to a variety of other programming languages, including, for instance, Algol-like languages.

Finally, it is interesting to note that the problem with "parallel or" is essentially the *only* obstacle to full abstraction for the cpo semantics. As soon as one adds "parallel or" to the language, the semantics becomes fully abstract.

Theorem 12.3. *The cpo semantics is fully abstract for parallel PCF.*