Bachelor Project in Computer Science

Danny Nygård Hansen

5th October 2023

1 ◇ Preliminaries

If X is a set, then we denote by $\mathcal{P}(X)$ the power set of X, and for a cardinal κ we furthermore write $\mathcal{P}_\kappa(X)$ for the collection of subsets of X with cardinality strictly less than κ. If $A \in \mathcal{P}_\kappa(X)$, then we also write $A \subseteq_\kappa X$. For ordinals α we do not distinguish between the ordinal ω_α and the cardinal $\aleph_\alpha$, and we write $\omega \coloneqq \omega_0$. Hence $\mathcal{P}_\omega(X)$ denotes the set of finite subsets of X, and $\mathcal{P}_{\omega_1}(X)$ the set of countable subsets. If Y is another set, then we denote the set of partial maps from X to Y by $P(X, Y)$. For $f \in P(X, Y)$ we let $\operatorname{dom} f$ and $\operatorname{ran} f$ denote the domain and range of f, respectively. We also write $P_\kappa(X, Y)$ for the partial functions $f \in P(X, Y)$ with $\operatorname{dom} f \subseteq_\kappa X$. For $x_0 \in X$ and $y_0 \in Y$ we denote by $f[x_0 \mapsto y_0]$ the partial map given by

$$f[x_0 \mapsto y_0](x) = \begin{cases} y_0, & x = x_0, \\ f(x), & x \in \operatorname{dom} f \setminus \{x_0\}. \end{cases}$$

Hence $\operatorname{dom} f[x_0 \mapsto y_0] = \operatorname{dom} f \cup \{x_0\}$, so that if $f \in P_\omega(X, Y)$, then also $f[x_0 \mapsto y_0] \in P_\omega(X, Y)$.

We let $\mathbb{N}$ denote the set of natural numbers, including 0. The image of a set $A \subseteq X$ under a function $f \colon X \to Y$ is denoted $f[A]$.

If X is a set, then we denote by X^* the set of finite sequences of elements in X, including the empty sequence. We often denote such a sequence by a boldface symbol e.g. $\boldsymbol{x}$. If $\boldsymbol{x} \in X^*$ and $x \in X$, then we write $x \in \boldsymbol{x}$ if x is among the entries in $\boldsymbol{x}$, and $x \notin \boldsymbol{x}$ otherwise. We use the convention that the entries in $\boldsymbol{x}$ are written $x_1, \ldots, x_k$.

If A and B are disjoint sets, then we write $A \# B$.

2 ◇ Induction and Inversion

In this section we study inference rules and the structures that they give rise to, and we prove various theorems for such structures that will be important in the proof of type safety in TODO ref.

We first give an overview of the abstract theory, studying generating functions in complete lattices and in dcpo's. In a complete lattice, Knaster–Tarski's fixed-point theorem yields a principle of induction, which when applied to inference rules gives a notion of rule induction. In dcpo's we study continuous functions in the context of Kleene's fixed-point theorem, which gives a finite description of elements in a structure defined by inference rules. We then study derivations of such elements and prove an inversion theorem. We finally use Kleene's fixed-point theorem to prove that we can define functions recursively over inference rules.

For readers not interested in the technical details, we note that the main results that will be used in the sequel are Theorem 2.7, Corollary 2.11 and Theorem 2.12.

2.1 • Abstract theory

(a) Let $(P, \leq)$ be a partially ordered set[1]. If $A \subseteq P$, then an element $x \in P$ is called an ***upper bound*** of A if $a \leq x$ for all $a \in A$. If there is a least upper bound x (i.e., such that if y is any other upper bound, then $x \leq y$), then x is called the ***join*** of A. The join of A is clearly unique if it exists (by anti-symmetry), and we denote it by $\bigvee A$. We similarly define the ***meet*** of A, denoted $\bigwedge A$, to be the greatest lower bound of A, if it exists. If every two-element subset of P has a join and a meet, then P is called a ***lattice***, and we write $x \vee y = \bigvee\{x, y\}$ and $x \wedge y = \bigwedge\{x, y\}$. If every subset of P has a join and a meet, then P is a ***complete lattice***.

If P is a partially ordered set, then a nonempty [TODO Vickers 7.2.1] subset $D \subseteq P$ is said to be ***directed*** if every finite subset of D has an upper bound (but not necessarily a *least* upper bound, i.e. a join). By induction this is equivalent to the property that, for every *pair* of elements $x, y \in D$, there exists a $z \in D$ with $x \leq z$ and $y \leq z$.[2] We further note that the image of a directed

[1] That is, $\leq$ is a reflexive, transitive and anti-symmetric binary relation on P. We also call P a ***poset***.

[2] This is the usual definition of directedness. As an example of why directedness in interesting, recall that a union of a collection of subspaces of a vector space is not usually a subspace itself, but it is if the collection is directed (with respect to inclusion). Similarly for subgroups and other algebraic structures, but note that the same does *not* hold for e.g. topologies or σ-algebras. If we substituted 'countable' for 'finite' in the definition of directedness, σ-algebras would have this property as well, while we for topologies would need 'arbitrary' subsets.

set under a monotone map[3] is also directed [TODO embedding]. If D is a directed set whose join exists, we often write $\bigsqcup D \coloneqq \bigvee D$ instead. If $\bigsqcup D$ exists for every directed subset D of P, then P is called a ***directly complete partial order***, or ***dcpo*** for short. If P also has a least element, usually written $\bot$, then P is called a ***dcppo*** (the extra 'p' is for 'pointed'). Notice that complete lattices are dcppo's.

If P and Q are dcpo's, then a map $f\colon P \to Q$ is ***continuous***[4] if, for every directed $D \subseteq P$, the image $f[D]$ is directed and

$$f\Big(\bigsqcup D\Big) = \bigsqcup f[D].$$

It is easy to show that continuous maps are monotone (notice that if $x \leq y$, then the set $\{x, y\}$ is directed). If conversely f is monotone, then $f[D]$ is as mentioned also directed, and the inequality '$\geq$' always holds.

2.1 • Example: ***Products of dcpo's.***
Let $(P_i)_{i\in I}$ be a collection dcpo's, and equip the product $P \coloneqq \prod_{i\in I} P_i$ with the product order[5]. We claim that P is also a dcpo. If $D \subseteq P$ is directed, then each $\pi_i[D]$ is also directed and thus has a join x_i. Letting $x = (x_i)_{i\in I}$ it is easy to see that $x = \bigsqcup D$ in P: It is clear that x is an upper bound of D, and furthermore $x \leq y$ if and only if $x_i \leq \pi_i(y)$ for all $y \in P$ and $i \in I$.

In particular,

$$\pi_i\Big(\bigsqcup D\Big) = \pi_i(x) = x_i = \bigsqcup \pi_i[D],$$

so π_i is continuous.[6] ⌟

2.2 • Example: ***Partial functions.***
If X and Y are sets, then we order the set $P(X, Y)$ of partial functions as follows: For $f, g\colon X \rightharpoonup Y$ we let $f \leq g$ if $\operatorname{dom} f \subseteq \operatorname{dom} g$ and $f(x) = g(x)$ for all $x \in \operatorname{dom} f$. The ***graph*** of a partial function f is the set

$$\mathcal{G}(f) = \{(x, y) \in X \times Y \mid x \in \operatorname{dom} f \text{ and } f(x) = y\}.$$

This induces a map $\mathcal{G}\colon P(X, Y) \to \mathcal{P}(X \times Y)$ given by $f \mapsto \mathcal{G}(f)$, which is clearly an order-embedding. If $D \subseteq P(X, Y)$ is a directed set of partial functions, then it is clear that the set $\bigsqcup \mathcal{G}[D] = \bigcup_{d\in D} \mathcal{G}(d)$ is the graph of a partial function. This function is then the join of D in $P(X, Y)$, which is therefore a dcpo, and

[3] A function $f\colon P \to Q$ between posets is ***monotone*** if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in P$.

[4] Also called ***Scott-continuous*** after Dana Scott.

[5] The ***product order*** on $P = \prod_{i\in I} P_i$ is defined as follows: For $(x_i)_{i\in I}, (y_i)_{i\in I} \in P$ we say that $(x_i)_{i\in I} \leq (y_i)_{i\in I}$ if $x_i \leq y_i$ for all $i \in I$. The projections $\pi_i\colon P \to P_i$ are thus monotone, and, as an aside, we mention that P is a categorical product in the category of posets and monotone maps.

[6] It follows that P is a product in the category of dcpo's and continuous maps.

the map $\mathcal{G}$ is thus continuous. In fact, $P(X, Y)$ is a dcppo since the empty map is the least element.

Denoting the projection maps by $\pi_X \colon X \times Y \to X$ and $\pi_Y \colon X \times Y \to Y$, notice that $\operatorname{dom} f = \pi_X[\mathcal{G}(f)]$ and $\operatorname{ran} f = \pi_Y[\mathcal{G}(f)]$. Since images preserve unions, and joins in power sets are just unions, if $D \subseteq P(X, Y)$ is directed then

$$\operatorname{dom} \bigsqcup D = \pi_X\Big[\mathcal{G}\Big(\bigsqcup D\Big)\Big] = \pi_X\Big[\bigsqcup \mathcal{G}[D]\Big] = \bigsqcup \pi_X\Big[\mathcal{G}[D]\Big] = \bigsqcup \operatorname{dom}[D],$$

and we similarly have $\operatorname{ran} \bigsqcup D = \bigsqcup \operatorname{ran}[D]$. ⌟

Let $F \colon P \to P$ be a monotone map. We think of F as a ***generating function***. An element $x \in P$ is said to be F-***closed*** if $F(x) \leq x$, F-***consistent*** if $x \leq F(x)$, and a ***fixed-point*** of F if $F(x) = x$. If F has a least fixed-point, then this is usually denoted μF. Similarly, the greatest fixed-point, if it exists, is denoted νF.

(b) Let P be a dcppo and $F \colon P \to P$ a monotone function. We clearly have $\bot \leq F(\bot)$, and since F is monotone we get the chain[7]

$$\bot \leq F(\bot) \leq \cdots \leq F^n(\bot) \leq F^{n+1}(\bot) \leq \cdots$$

called the ***ascending Kleene chain***. Since it is a chain it is also directed. The main theorem is the following:

2.3 • THEOREM: *Kleene's fixed-point theorem.*
If P is a dcppo and $F \colon P \to P$ is continuous, then F has a least fixed-point μF and

$$\mu F = \bigsqcup_{n \in \mathbb{N}} F^n(\bot).$$

Proof[8]**.** First notice that, since F is continuous,

$$F\Big(\bigsqcup_{n \in \mathbb{N}} F^n(\bot)\Big) = \bigsqcup_{n \in \mathbb{N}} F^{n+1}(\bot) = \bigsqcup_{n \in \mathbb{N}^+} F^n(\bot) = \bigsqcup_{n \in \mathbb{N}} F^n(\bot),$$

where we use that $F^0(\bot) = \bot$. Hence $\bigsqcup_{n \in \mathbb{N}} F^n(\bot)$ is indeed a fixed-point of F. If β is any fixed-point of F, then $\bot \leq \beta$, and hence $F^n(\bot) \leq F^n(\beta) = \beta$ since F is monotone. Taking the join on the left-hand side yields $\bigsqcup_{n \in \mathbb{N}} F^n(\bot) \leq \beta$ as desired. ■

The proof of Theorem 2.3 betrays that it is not necessary to work with directed sets: Indeed, the proof only uses the fact that an increasing sequence – also called an ω-***chain*** – in P has a join, and that F preserves such joins. The theory

[7] A ***chain*** is a totally ordered set.

[8] This proof is based on Davey and Priestley (2002, Theorem 8.15).

would work out in the same way if we instead worked in partial orders in which every ω-chain has a join, called ***ω-complete partial orders*** – for short ***ω-cpo's***, or ***ω-cppo's*** if they have a least element. Our reasons for nonetheless working in dcpo's are (at least) twofold: Firstly, replacing directed sets with ω-chains does not simplify the arguments in any way. Secondly, since we intend to apply this theory to power sets anyway, there is no reason to make the weaker assumption that P is an ω-cppo. The present choice yields the stronger conclusion that the map F in Lemma 2.8 is continuous and hence preserves all *directed* joins and not just joins of ω-chains. [TODO also useful in domain theory maybe??]

(c) Next, let L be a complete lattice, and let $F\colon L \to L$ be a monotone map. Even though L is also a dcppo, if F is not continuous then Theorem 2.3 does not apply. However, F still has fixpoints, as the following theorem shows:

2.4 • THEOREM: *Knaster–Tarski's fixed-point theorem.*
If L is a complete lattice and $F\colon L \to L$ is monotone, then F has a least and a greatest fixed-point, and these are given by

$$\mu F = \bigwedge\{x \in L \mid F(x) \leq x\} \quad \textit{and} \quad \nu F = \bigvee\{x \in L \mid x \leq F(x)\}.$$

In particular, μF is the smallest F-closed element and νF is the greatest F-consistent element in L.

Proof [9]**.** Denote the meet above by α. If x is F-closed, then $\alpha \leq x$, so $F(\alpha) \leq F(x) \leq x$. Taking the meet of x we get $F(\alpha) \leq \alpha$, so α is closed. It follows that $F(F(\alpha)) \leq F(\alpha)$, so $F(\alpha)$ is also closed, and so $\alpha \leq F(\alpha)$. Hence α is a fixed-point. Since every other fixed-point is in particular closed, α is the least fixed-point.■

2.5 • COROLLARY: *Principle of induction.*
If $y \in L$ is F-closed, then $\mu F \leq y$. ■

We dually have a ***principle of coinduction*** – if $y \in L$ is F-consistent, then $y \leq \nu F$ – but we shall not need this in the sequel.

2.2 • In power sets

(a) Specialising to the case where L is a power set $\mathcal{P}(X)$, one way to define a generating function is using inference rules. An ***inference rule*** on X is an expression on the form

$$\textsc{Rule}\ \frac{x_1 \qquad x_2 \qquad \cdots \qquad x_k}{y}$$

[9] This proof is based on Davey and Priestley (2002, Theorem 2.35).

where $x_1, \ldots, x_k$ and y are elements of X, and $k \in \mathbb{N}$.[10] We allow k to be zero, in which case we call the rule an ***axiom***. We have decorated the expression with the label 'RULE' so that we may refer to it later. Let us call $x_1, \ldots, x_k$ the ***antecedents*** of the rule and y the ***consequent***. If R is a rule with antecedents $x_1, \ldots, x_k$ and consequent y, we also write $R\colon x_1, \ldots, x_k \vdash y$, or $R\colon \boldsymbol{x} \vdash y$ for short. We often suppress R from this notation and simply write $\boldsymbol{x} \vdash y$. The number of antecedents of a rule R is called its ***arity*** and is denoted $\rho(R)$.

Given a (possibly infinite) collection $\mathcal{R}$ of inference rules, we construct a generating function $F\colon \mathcal{P}(X) \to \mathcal{P}(X)$ by defining $F(A)$ for a subset $A \subseteq X$ as follows: For $y \in X$ we let $y \in F(A)$ if and only if there is a rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$ with $\boldsymbol{x} \subseteq A$. We say that F is ***represented by*** $\mathcal{R}$. In this case F is clearly monotone, so it has a least and a greatest fixed-point μF and νF, respectively. Since it is often unnecessary to explicitly talk about F, we also write $\mu\mathcal{R}$ and $\nu\mathcal{R}$ for these fixed-points.

By Corollary 2.5 we also get a principle of induction for F. However, as for fixed-points it is useful to restate induction in terms of the inference rules, since we usually have explicit rules in mind when defining F. If $\mathcal{R}$ is a collection of inference rules on X, then we say that a subset $A \subseteq X$ is ***$\mathcal{R}$-closed*** if, given any rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$, $\boldsymbol{x} \subseteq A$ implies that $y \in A$.

2.6 • LEMMA. *If F is represented by a collection $\mathcal{R}$ of inference rules, then a subset $A \subseteq X$ is F-closed if and only if it is $\mathcal{R}$-closed.*

Proof. First assume that A is F-closed so that $F(A) \subseteq A$, and consider a rule $\boldsymbol{x} \vdash y$ from $\mathcal{R}$ with $\boldsymbol{x} \subseteq A$. Since F is represented by $\mathcal{R}$, this implies that $y \in F(A) \subseteq A$, so A is $\mathcal{R}$-closed.

If A is $\mathcal{R}$-closed, then let $y \in F(A)$. Hence there is a rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$ with $\boldsymbol{x} \subseteq A$. But since A is $\mathcal{R}$-closed, this implies that $y \in A$, and so $F(A) \subseteq A$. ∎

2.7 • THEOREM: *Principle of rule induction.*
If $\mathcal{R}$ is a set of inference rules on X and $A \subseteq X$, then $\mu\mathcal{R} \subseteq A$ if the following condition holds: For every inference rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$, $\boldsymbol{x} \subseteq A$ implies $y \in A$.

Proof. Let F be the function represented by $\mathcal{R}$. The condition says precisely that A is $\mathcal{R}$-closed, which implies that A is F-closed by Lemma 2.6. But then Corollary 2.5 implies that $\mu\mathcal{R} = \mu F \subseteq A$. ∎

(b) Sometimes it is necessary (or at least useful) to have an alternative characterisation of $\mu\mathcal{R}$, one that makes use of the fact that the number of antecedents of an inference rule is finite. This property is reflected by the represented function F in the following way:

[10] We could equivalently view an inference rule as an element of the product $X^k \times X$.

2.8 • Lemma. *If F is represented by a collection of inference rules, then F is continuous.*

Proof. It suffices to show that if $\mathcal{D} \subseteq \mathcal{P}(X)$ is directed, then $F(\bigsqcup \mathcal{D}) \subseteq \bigsqcup F[\mathcal{D}]$, so let $y \in F(\bigsqcup \mathcal{D})$. Say that F is represented by a collection $\mathcal{R}$. Then there is a rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$ with $\boldsymbol{x} \subseteq \bigsqcup \mathcal{D}$. Since the (directed) join in $\mathcal{P}(X)$ is just union, there are sets $A_1, \ldots, A_k \in \mathcal{D}$ such that $x_i \in A_i$. And since $\mathcal{D}$ is directed there is a set $A \in \mathcal{D}$ with $A_i \subseteq A$, so that $x_i \in A$ for all i. But then $y \in F(A) \subseteq \bigsqcup F[\mathcal{D}]$ as desired. ∎

Hence, Theorem 2.3 implies that $\mu F = \bigsqcup_{n \in \mathbb{N}} F^n(\emptyset)$.

To make this more concrete, we study how one can use the inference rules to derive that some element of X lies in μF. If $\mathcal{R}$ is a collection of inference rules, then a ***derivation*** from $\mathcal{R}$ is a finite sequence $D = (R_1, \ldots, R_n)$ of inference rules from $\mathcal{R}$. If y is a consequent of some R_i, then we say that y is a ***conclusion*** of D. The consequent of R_n is called the ***final conclusion*** of D. We say that $x \in X$ is an ***assumption*** of D if x is an antecedent of some R_i, and if none of the rules $R_1, \ldots, R_{i-1}$ has x as its consequent. That is, x is an assumption if it is not implied by any of the previous rules in the derivation. The set of conclusion of D is denoted $\operatorname{con} D$, and the set of assumptions of D is denoted $\operatorname{asm} D$. If $\operatorname{asm} D = \emptyset$, then we say that D is ***closed***.

2.9 • Remark. Let $D = (R_1, \ldots, R_n)$ be a derivation.

(a) If D is closed, then R_1 must be an axiom.

(b) For any $i \in \{1, \ldots, n\}$, $D' = (R_1, \ldots, R_i)$ is called a ***subderivation*** of D. If $i < n$, then D' is called a ***proper subderivation*** of D (so D' is a proper subderivation when $D' \neq D$). It is clear that $\operatorname{asm} D' \subseteq \operatorname{asm} D$ and $\operatorname{con} D' \subseteq \operatorname{asm} D$, so D' is closed if D is closed.

(c) If D is closed and x is an antecedent of some R_i, then D has a proper subderivation D' whose final consequence is x: For x must be the consequent of some $R_1, \ldots, R_{i-1}$, say R_j, and $(R_1, \ldots, R_j)$ is closed subderivation of D by (b).

(d) If $E = (S_1, \ldots, S_m)$ is another derivation, then

$$D \circ E \coloneqq (R_1, \ldots, R_n, S_1, \ldots, S_m)$$

is another derivation. It is clear that

$$\operatorname{asm}(D \circ E) = \operatorname{asm} D \cup (\operatorname{asm} E \setminus \operatorname{con} D),$$

that

$$\operatorname{con}(D \circ E) = \operatorname{con} D \cup \operatorname{con} E,$$

and that the final conclusion of $D \circ E$ is the final conclusion of E. In particular, if D is closed and $\operatorname{asm} E \subseteq \operatorname{con} D$, then $D \circ E$ is also closed.

(e) The ***length*** of a derivation $D = (R_1, \ldots, R_n)$ is n, and this is denoted $l(D)$. If D' is a subderivation of D, then $l(D') \leq l(D)$ with equality if and only if $D' = D$. If E is another derivation, then $l(D \circ E) = l(D) + l(E)$. ⌟

2.10 • PROPOSITION. *If $\mathcal{R}$ is a collection of inference rules and $y \in X$, then $y \in \mu\mathcal{R}$ if and only if y is the final conclusion of some closed derivation from $\mathcal{R}$.*

Proof. Let F be the function represented by $\mathcal{R}$. First assume that $y \in \mu F$, and recall that $\mu F = \bigsqcup_{n\in\mathbb{N}} F^n(\emptyset) = \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. We prove by induction in n that every element in $F^n(\emptyset)$ is the final conclusion of a closed derivation. The base case $y \in F^0(\emptyset) = \emptyset$ is vacuous, so let $n \in \mathbb{N}$ and assume that the claim holds for every element of $F^n(\emptyset)$. If $y \in F^{n+1}(\emptyset) = F(F^n(\emptyset))$, then there is an inference rule $R\colon \boldsymbol{x} \vdash y$ with $\boldsymbol{x} \subseteq F^n(\emptyset)$. By induction each x_i is the final conclusion of some derivation D_i, and by Remark 2.9(d) the derivation $D_1 \circ \cdots \circ D_k \circ (R)$ is a closed derivation, and y is its final conclusion.

We prove the converse by (strong) induction on the length of a derivation. If y is the final conclusion of a closed derivation (R) of length 1, then R must be an axiom. But then $y \in F(\emptyset) \subseteq \mu F$ since F is represented by $\mathcal{R}$. Next, let $n \in \mathbb{N}$ and let y be the final conclusion of a closed derivation $D = (R_1, \ldots, R_{n+1})$. If x is an antecedent of R_{n+1}, then by Remark 2.9(c) D must have a proper subderivation D' whose final conclusion is x. By Remark 2.9(b) D' is also closed, and so $x \in \mu F$ by induction. But then we must have $y \in F(\mu F) = \mu F$ as desired. ■

It is of course standard to use derivation *trees*, but all that matters is that there is a finite way to obtain elements in $\mu\mathcal{R}$. Since it is easier to define and reason about linear derivations – and since we will not have to do any actual derivations – we have taken this approach here.[11] The main consequence we shall use is the following:

2.11 • COROLLARY: *Inversion.*
If $\mathcal{R}$ is a collection of inference rules and $y \in \mu\mathcal{R}$, then there is a rule $\boldsymbol{x} \vdash y$ in $\mathcal{R}$ such that $\boldsymbol{x} \subseteq \mu\mathcal{R}$.

Proof. By Proposition 2.10 there is a closed derivation D of y from $\mathcal{R}$, and let its last rule be $\boldsymbol{x} \vdash y$. For each x_i, some subderivation of D is a derivation of x_i by Remark 2.9(b). Another application of Proposition 2.10 then implies that $x_i \in \mu\mathcal{R}$. ■

[11] Compare the role of deductive calculi in first-order logic, where e.g. the compactness theorem can be obtained from the mere *existence* of a deductive calculus (with finite derivations), cf. Leary and Kristiansen (2015, Theorem 3.3.1).

(c) The final construction we need is definition by recursion.[12]

2.12 • THEOREM: ***Definition by recursion.***
Let $\mathcal{R}$ be a set of inference rules such that each $y \in \mu\mathcal{R}$ is the consequent of a unique rule, and let Z be any set. For each $R \in \mathcal{R}$ let $h_R \colon \mu\mathcal{R} \times Z^{\rho(R)} \to Z$ be a function. Then there is a unique function $f : \mu\mathcal{R} \to Z$ such that if $R \colon \boldsymbol{x} \vdash y$, then

$$f(y) = h_R\Big(y, f(x_1), \ldots, f(x_k)\Big).$$

Proof. Define a function $G \colon P(\mu\mathcal{R}, Z) \to P(\mu\mathcal{R}, Z)$ as follows: For $f : \mu\mathcal{R} \rightharpoonup Z$ let $\operatorname{dom} G(f)$ be the set of elements $y \in \mu\mathcal{R}$ such that if $R \colon \boldsymbol{x} \vdash y$ is the unique rule with consequent y, $\boldsymbol{x} \subseteq \operatorname{dom} f$. For $y \in \operatorname{dom} G(f)$ we then let

$$G(f)(y) = h_R\Big(y, f(x_1), \ldots, f(x_k)\Big).$$

If F is the function represented by $\mathcal{R}$, notice that $\operatorname{dom} G(f) = F(\operatorname{dom} f)$. In particular, $\operatorname{dom} G(\bot) = F(\emptyset)$, so it follows by induction that $\operatorname{dom} G^n(\bot) = F^n(\emptyset)$ for $n \in \mathbb{N}$.

We next show that G is continuous, so let $D \subseteq P(\mu\mathcal{R}, Z)$ be directed. First notice that, since F is continuous by [TODO ref], we get from [TODO ref ex] that

$$\begin{aligned}\operatorname{dom} G\Big(\bigsqcup D\Big) &= F\Big(\operatorname{dom} \bigsqcup D\Big) = F\Big(\bigsqcup \operatorname{dom}[D]\Big) \\ &= \bigsqcup F\Big[\operatorname{dom}[D]\Big] = \bigsqcup \operatorname{dom}\Big[G[D]\Big] \\ &= \operatorname{dom} \bigsqcup G[D].\end{aligned}$$

For y in this common domain there is a $d \in D$ such that $y \in \operatorname{dom} G(d)$. If $R \colon \boldsymbol{x} \vdash y$ then $\boldsymbol{x} \subseteq \operatorname{dom} d$, and so

$$\begin{aligned}G\Big(\bigsqcup D\Big)(y) &= h_R\Big(y, (\bigsqcup D)(x_1), \ldots, (\bigsqcup D)(x_k)\Big) \\ &= h_R\Big(y, d(x_1), \ldots, d(x_k)\Big) \\ &= G(d)(y) \\ &= \Big(\bigsqcup G[D]\Big)(y).\end{aligned}$$

Thus $G(\bigsqcup D) = \bigsqcup G[D]$, so G is continuous.

It now follows from Theorem 2.3 that

$$\operatorname{dom} \mu G = \operatorname{dom} \bigsqcup_{n \in \mathbb{N}} G^n(\bot) = \bigsqcup_{n \in \mathbb{N}} \operatorname{dom} G^n(\bot) = \bigsqcup_{n \in \mathbb{N}} F^n(\emptyset) = \mu F.$$

Hence μG is in fact a total function solving the equation [TODO ref], which proves the theorem. ∎

[12] The statement of this result is based on Goldrei (1996, Theorem 3.2), while the proof is inspired by Davey and Priestley (2002, §8.18).

[TODO something about solving fixed-point equations when recursion doesn't work, Moschovakis 6.31.]

We also sometimes wish to define *partial* functions recursively. This is no issue, for as the following shows, there is a correspondence between partial functions and certain total functions.

2.13 • Example. Consider a partial function $f : X \rightharpoonup Y$, let $*$ be some element not in Y, and let $Y_* = Y \cup \{*\}$. We then define a total function $f_* : X \to Y_*$ by letting $f_*(x) = f(x)$ if $x \in \operatorname{dom} f$, and $f_*(x) = *$ otherwise.

Conversely, given a total function $f_* : X \to Y_*$ we obtain the corresponding partial function $f : X \rightharpoonup Y$ by restricting f_* to the set $\{x \in X \mid f_*(x) \neq *\}$.[13] ⌟

Hence, to define a partial function f recursively, we first extend the indented codomain by some new element such as $*$, and define a total function f_* recursively while letting $f_*(x) = *$ whenever f is supposed to be undefined at x. Finally restrict f_* to obtain f.

[TODO example? + nonexample for sigma-algebras/topologies?]

2.3 • Topology and logic

2.14 • Definition: *Frames.*
A poset P is a *frame* if

(a) every subset of P has a join,

(b) every finite subset of P has a meet, and

(c) P satisfies the *join-infinite distributive law*,

$$x \wedge \bigvee Y = \bigvee \{x \wedge y \mid y \in Y\}$$

for all $x \in P$ and $Y \subseteq P$. ▲

Since all joins exist, so do all meets. In particular, a frame is complete distributive lattice.

2.15 • Definition: *Topological systems.*
A *topological system* is a pair $(X, \mathcal{T})$, where X is a set and $\mathcal{T}$ is a frame, equipped with a relation $\prec \subseteq X \times \mathcal{T}$ with the following properties:

(a) For all $\mathcal{S} \subseteq_\omega \mathcal{T}$ and $x \in X$,

$$x \prec \bigwedge \mathcal{S} \quad \text{if and only if} \quad \forall U \in \mathcal{S} \colon a \prec U.$$

[13] This correspondence between partial and total functions indicates that there is some correspondence between the category **Pfn** of sets and partial functions and the category $\mathbf{Set}_* \cong 1/\mathbf{Set}$ of pointed sets. Indeed, while these are not isomorphic, they are *equivalent* (see e.g. Smith 2023b, Theorems 156 and 158).

(b) For all $\mathcal{S} \subseteq \mathcal{T}$ and $x \in X$,

$$x \prec \bigvee \mathcal{S} \quad \text{if and only if} \quad \exists U \in \mathcal{S} \colon a \prec U. \qquad \blacktriangle$$

The elements in X are called ***points***, and the elements of $\mathcal{T}$ are alled ***opens***. If $x \prec U$, then we say that x is an ***inner point*** of U. The set of opens of which x is an inner point is denoted $\mathcal{U}_x$. For $x, y \in X$ we say that y ***specialises*** x, written $x \sqsubseteq y$, if $\mathcal{U}_x \subseteq \mathcal{U}_y$. We call $\sqsubseteq$ the ***specialisation preorder*** on X, and it is indeed a preorder.

In trying to equip X with a logic, we obviously wish to define disjunctions on X. For $A \subseteq X$, we say that $x \in X$ is a ***disjunction*** of A, written $\bigvee A$, if it satisfies

$$\bigvee A \prec U \quad \text{if and only if} \quad \exists a \in A \colon a \prec U,$$

for all $U \in \mathcal{T}$. We are careful to distinguish this from a join $\bigsqcup A$ of A with respect to $\sqsubseteq$.[14]

2.16 • PROPOSITION. *Any disjunction is a join, but not vice-versa.*

Proof. Let $\bigvee A$ be a disjuction. If $a \in A$ and $a \prec U$, then we also have $\bigvee A \prec U$, so $\bigvee A$ is an upper bound of A. Next let x be any upper bound of A. If $\bigvee A \prec U$, then $a \prec U$ for some $a \in A$, and since $a \sqsubseteq x$ we also have $x \prec U$. Hence $\bigvee A \sqsubseteq x$.■

However, arbitrary subsets of X do not have disjunctions, or at least do not have disjunctions that respect the logic on $\mathcal{T}$. For instance, if $x, y \in X$ are incomparable with respect to the specialisation preorder, then there are opens $U \in \mathcal{U}_x \setminus \mathcal{U}_y$ and $V \in \mathcal{U}_Y \setminus \mathcal{U}_X$. If the disjunction $x \vee y$ existed, then we would have $x \vee y \prec U$ and $x \vee y \prec V$, but not $x \vee y \prec U \wedge V$, in contradiction with the assumption that $(X, \mathcal{T})$ is a topological system.

In order to mitigate this problem, we must find

2.4 • Abstract syntax

(a) Let $\mathcal{S}$ be a set. We think of $\mathcal{S}$ as a set of ***sorts***, for instance expressions or types. A ***valence*** is an expression on the form $s_1, \ldots, s_k.s$, or $\boldsymbol{s}.s$ for short, where $s_1, \ldots, s_k, s$ are sorts for $k \in \mathbb{N}$. [TODO formalise expression] If $k = 0$, then instead of '$.s$' we omit the period and simply write 's'. Each argument of an operator is supposed to have a valence which describes both the sort of that argument as well as the sorts of the variables that are bound in that argument (if any). An ***arity*** is then an expression on the form $(v_1, \ldots, v_n)s$, where $v_1, \ldots, v_n$ are valences and s is a sort, for $n \in \mathbb{N}$. [TODO formalise expression] Operators

[14] Note that since $\sqsubseteq$ is not (generally) antisymmetric, the join of A is not necessarily unique if it exists, but it is easy to show that if x and y are both joins of A, then $x \sqsubseteq y$ and $y \sqsubseteq x$.

will be assigned arities, and v_i is then the valence of the ith argument to an operator with the above arity, and s is the sort of the return value of the operator. For each arity α we also fix a set $\mathcal{O}_\alpha$ which we think of as a set of ***operators***. Finally we fix for each sort $s \in \mathcal{S}$ a set $\mathcal{V}_s$ of variables so that we for instance can have both ordinary variables (of expression sort) and type variables (of type sort). Note that we do not [TODO contrary to Harper] require the sets $\mathcal{O}_\alpha$ to be disjoint, so an operator can have multiple arities. [TODO variables of multiple sorts?] We let $\mathcal{O}$ be the set of all operators of any arity, and $\mathcal{X}$ the set of all variables of any sort.

For example, if *Exp* is a sort of expressions and *Type* a sort of types, an operator corresponding to lambda abstraction with type annotation might have arity $(Type, Exp.Exp)Exp$. This indicates that its arguments are a type (the type of the argument) along with an expression in which a variable of expression sort is bound, and the entire abstraction is also of expression sort. If we also allow the abstraction to be given a name (in order to easily allow for recursive functions), then the arity of the operator might instead be $(Type; Exp, Exp.Exp)Exp$. We here separate the valences with a semicolon since the valence of the second argument itself contains a comma.

(b) We first define the set of ***unsorted abstract syntax trees***, or simply ***UASTs***. [TODO universal set] We define this set using inference rules. For each $x \in \mathcal{V}$ we have the axiom

$$\frac{}{x} \tag{1}$$

saying that x itself is always a UAST. For each $\varphi \in \mathcal{O}$ taking n arguments and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathcal{V}^*$ we also have the rule

$$\frac{a_1 \quad \cdots \quad a_n}{\varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)} \tag{2}$$

[TODO formalise expression] We denote the set of UASTs by $\mathcal{A}$. The expression $\varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)$ is supposed to denote the application of the operator φ on the UASTs $a_1, \ldots, a_n$ in which the variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ have been bound.

Using unsorted ASTs we also define ***(well-sorted) abstract syntax trees***, or ***ASTs*** for short, as following. We recursively define a relation on $\mathcal{A} \times \mathcal{S}$, where we write an element (a, s) by $a : s$: For each sort $s \in \mathcal{S}$ and variable $x \in \mathcal{V}_s$ of sort s we have the rule

$$\frac{}{x : s} \tag{3}$$

Next, consider an operator $\varphi \in \mathcal{O}_\alpha$ where $\alpha = (v_1; \ldots; v_n)s$ and $v_i = (s_{i1}, \ldots, s_{ik_i})s_i$. If we apply φ to ASTs of the proper sorts and bind in each argument the correct number of variables of the right sorts, then the resulting UAST should also

be well-sorted with sort s: Hence, if $\boldsymbol{x}_i$ is a sequence of variables $(x_{i1},\dots,x_{ik_i})$ where x_{ij} has sort s_{ij}, then we have the rule

$$\frac{a_1 : s_1 \qquad \cdots \qquad a_n : s_n}{\varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n) : s} \tag{4}$$

If $a : s$, then we say that a is ***well-sorted*** with sort s.

We next define the set of ***free variables***. This will be a function $FV() \colon \mathcal{A} \to \mathcal{P}(\mathcal{V})$ defined recursively using Theorem 2.12. According to this theorem, we must specify for each inference rule R a function $h_R \colon \mathcal{A} \times \mathcal{P}(\mathcal{V})^{\rho(R)} \to \mathcal{P}(\mathcal{V})$. If R is a rule as in (1), then $\rho(R) = 0$ and we let $h_R(a) = \{a\}$. If instead R is as in (2), then we let

$$h_R\Big(\varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n), A_1,\dots,A_n\Big) = \bigcup_{i=1}^{n} (A_i \setminus \boldsymbol{x}_i).$$

(If the first argument to h_R is instead a variable, then we let the value of h_R be an arbitrary element of $\mathcal{P}(\mathcal{V})$.) This implies the existence of a function $FV() \colon \mathcal{A} \to \mathcal{P}(\mathcal{V})$ with the properties

$$FV(x) = \{x\},$$
$$FV\Big(\varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n)\Big) = \bigcup_{i=1}^{n} (FV(a_i) \setminus \boldsymbol{x}_i).$$

The function $FV()$ is usually 'defined' simply by writing down the above two equations, but the precise justification goes through Theorem 2.12 and the functions h_R (or a similar argument). We similarly define the set of ***bound variables*** in a UAST by

$$BV(x) = \emptyset,$$
$$BV\Big(\varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n)\Big) = \bigcup_{i=1}^{n} \Big(\boldsymbol{x}_i \cup BV(a_i)\Big).$$

The set of all variables occurring in a UAST a is then $V(a) \coloneqq FV(a) \cup BV(a)$. If $FV(a) = \emptyset$, then we say that a is ***closed***.

(c) For $y, z \in \mathcal{V}$ we define the ***renaming map*** $a \mapsto a^{y\to z}$ on $\mathcal{A}$ as follows:

$$x^{y\to z} = \begin{cases} z, & x = y, \\ x, & x \neq y, \end{cases}$$
$$\varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n)^{y\to z} = \varphi(\boldsymbol{x}_1.a_1';\dots;\boldsymbol{x}_n.a_n'),$$

where $a_i' = a_i$ if $y \in \boldsymbol{x}$ and $a_i' = a_i^{y\to z}$ otherwise. We think of this map as renaming all the *free* occurrences of y in a UAST. This way of informally

defining the renaming map makes it seem like there is an issue of well-definition: That is, if $a \in \mathcal{A}$ do we indeed have $a^{y\to z} \in \mathcal{A}$? But this 'problem' disappears if we formulate the definition in the precise manner prescribed by Theorem 2.12. For a sequence of variables we also define $(x_1,\dots,x_n)^{y\to z} = (x_1',\dots,x_n')$, where $x_i' = z$ if $x_i = y$ and $x_i' = x_i$ if $x_i \neq y$.

If a is well-sorted and we rename a variable in a to a variable of the same sort, then the result should certainly be well-sorted of the same sort, and this is indeed the case:

2.17 • Lemma. *Let $a : s$, and let y, z be variables of the same sort. Then $a^{y\to z} : s$.*

Proof. Induction on the definition of UASTs. If $a = x$ is a variable such that $x = y$, then x and z have the same sort, so $x^{y\to z} = z$ has the same sort as x. [TODO need that variables have only one sort!] If instead $x \neq y$, then $x^{y\to z} = x$ certainly has the same sort as x.

Assume that $a = \varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n)$ and that the claim holds for $a_1,\dots,a_n$. Since a is assumed well-sorted with some sort s, inversion [TODO ref] on the definition of ASTs implies that $a_i : s_i$ for appropriate sorts s_i. Now consider

$$a^{y\to z} = \varphi(\boldsymbol{x}_1.a_1;\dots;\boldsymbol{x}_n.a_n)^{y\to z} = \varphi(\boldsymbol{x}_1.a_1';\dots;\boldsymbol{x}_n.a_n'),$$

where a_i' are as in the definition of renaming. If $a_i' = a_i$, then there is nothing to prove. If instead $a_i' = a_i^{y\to z}$, then by induction a_i' has the same sort as a_i. But then [TODO rule AST] implies that $a^{y\to z}$ has the same sort as a. ■

While the renaming map allows us to rename *free* occurrences of variables in UASTs, this is not usually something we wish to do, first of all since the ASTs we are ultimately interested in will be closed anyway, and secondly because renaming a free variable makes the UAST behave differently with respect to e.g. substitution. Nonetheless, the renaming map allows to define renaming of *bound* occurrences of variables, and the names of bound variables are of course not supposed to be significant. We usually say that two UASTs (or similar syntactic objects, such as terms in the λ-calculus) are ***α-equivalent*** if one is obtained from the other by renaming of bound variables. We define the α-equivalence relation $=_\alpha$ by the following inference rules:

$$\frac{a_i =_\alpha b}{\varphi(\dots;\boldsymbol{x}_i.a_i;\dots) =_\alpha \varphi(\dots;\boldsymbol{x}_{i-1}.a_{i-1};\boldsymbol{x}_i.b;\boldsymbol{x}_{i+1}.a_{i+1};\dots)}$$

$$\frac{}{\varphi(\dots;\boldsymbol{x}_i.a_i;\dots) =_\alpha \varphi(\dots;\boldsymbol{x}_{i-1}.a_{i-1};\boldsymbol{x}_i^{y\to z}.a_i^{y\to z};\boldsymbol{x}_{i+1}.a_{i+1};\dots)}$$

if either $y = z$ [TODO is this necessary??], or else $y \in \boldsymbol{x}_i$, $z \notin \boldsymbol{x}_i$ and $z \notin V(a_i)$. We also require $=_\alpha$ to be an equivalence relation as described in [TODO ref + write it! use old stuff about fixpoints of joins of functions.]

2.4.1. Substitution

(a) We also want to be able to substitute one UAST b into another UAST a by replacing a (free) variable x in a with b. The resulting UAST is denoted $a[b/x]$, and it is fairly difficult to define precisely, even if the concept is natural. If a is a variable z, then this is easy:

$$z[b/x] = \begin{cases} b, & z = x, \\ z, & z \neq x. \end{cases}$$

However, if a is a more complex UAST, then potential bindings inside a might 'capture' the free variables in b, so we must take care that does not happen. On the other hand, if x is not free in a then there is no issue, since then (as it turns out, cf. Lemma 2.20) b is not substituted into a at all. There are various solutions to this capture problem, and the one we have chosen is to simply not define substitution when it does not make sense. We will see (cf. Lemma 2.18) that even if we cannot substitute b into a, then we can find another UAST a' that is α-equivalent to a into which it is possible to substitute b. It also turns out (cf. Proposition 2.23) that if both $a[b/x]$ and $a'[b/x]$ are defined and $a =_\alpha a'$, then we also have $a[b/x] =_\alpha a'[b/x]$. Hence the *partial* substitution function on UASTs induces a *total* substitution function on α-equivalence classes of UASTs.

Returning to the definition of substitution, we now define $\varphi(\ldots;\boldsymbol{x}_i.a_i;\ldots)[b/x]$ to be either undefined or $\varphi(\ldots;\boldsymbol{x}_i.a_i';\ldots)$, where a_i' is to be defined below. For each $i \in \{1,\ldots,n\}$ we do the following:

(1) If $x \in \boldsymbol{x}_i$, then $a_i' = a_i$.

(2) Otherwise, if $a_i[b/x]$ is undefined, then $\varphi(\ldots;\boldsymbol{x}_i.a_i;\ldots)[b/x]$ is also undefined.

(3) Otherwise, if both $x \in FV(a_i)$ and $\boldsymbol{x}_i \cap FV(b) \neq \emptyset$, then $\varphi(\ldots;\boldsymbol{x}_i.a_i;\ldots)[b/x]$ is undefined.

(4) Otherwise it follows that $a_i[b/x]$ is defined, and that either $x \notin FV(a_i)$ or $\boldsymbol{x}_i \# FV(b)$. In this case we let $a_i' = a_i[b/x]$.

The idea is as follows: If it is impossible to substitute b into any of the arguments to φ, then the entire substitution is undefined. If x is among the variables bound by φ in the ith argument $\boldsymbol{x}_i.a_i$, then there is never any issue, since in this case we do not substitute b into a_i at all. Next, if $x \notin FV(a_i)$ then we can also substitute b into a_i, since no actual substitution will happen: During the recursion we never reach a free occurrence of x. Finally, if $\boldsymbol{x}_i \# FV(b)$, then there is no possibility of capture, and we may substitute b into a_i.

As mentioned, even if substitution is undefined for some UAST, we can always find an α-equivalent UAST into which it is possible to perform substitution.

2.18 • Lemma. *If $a[b/x]$ is undefined, then there is an $a' \in \mathcal{A}$ with $a =_\alpha a'$ such that $a'[b/x]$ is defined. If $a : s$, then we may also choose $a' : s$.*

Proof. We prove this by rule induction on the definition of $\mathcal{A}$. If $a = x$ is a variable, then we may simply choose $a' = x$. Next assume that the claim holds for UASTs $a_1, \dots, a_n$, such that $a_i =_\alpha a_i'$ and $a_i'[b/x]$ is defined. Considering the UAST $\varphi(\boldsymbol{x}_1.a_1; \dots; \boldsymbol{x}_n.a_n)$, successive applications of [TODO rule] (and transitivity of $=_\alpha$) implies that

$$\varphi(\boldsymbol{x}_1.a_1; \dots; \boldsymbol{x}_n.a_n) =_\alpha \varphi(\boldsymbol{x}_1.a_1'; \dots; \boldsymbol{x}_n.a_n').$$

For each $i \in \{1, \dots, n\}$, if $x \in \boldsymbol{x}_i$ then do nothing for that i. Otherwise successively apply [TODO rule] to obtain $\boldsymbol{x}_i'$ and a_i'' such that

$$\varphi(\dots; \boldsymbol{x}_{i-1}.a_{i-1}'; \boldsymbol{x}_i.a_i'; \boldsymbol{x}_{i+1}.a_{i+1}'; \dots) =_\alpha \varphi(\dots; \boldsymbol{x}_{i-1}.a_{i-1}'; \boldsymbol{x}_i'.a_i''; \boldsymbol{x}_{i+1}.a_{i+1}'; \dots),$$

and such that $\boldsymbol{x}_i' \# FV(b)$. Now substitution is possible.

For the final claim we note how to modify the above proof to obtain the stronger conclusion. This is first of all clear for variables. Then choose a_i' to have the same sort as a_i. Finally choose the new variables in $\boldsymbol{x}_i'$ and a_i'' to have the same sorts as the old variables, in which case Lemma 2.17 implies that a_i'' has the same sort as a_i'. Applying [TODO rule] yields the claim. ■

(b) We next prove some lemmas on renaming and substitution. First, renaming free variables in a changes the set $FV(a)$ in a predictable way:

2.19 • Lemma. $FV(a^{y \to z}) \subseteq (FV(a) \setminus \{y\}) \cup \{z\}$. *In particular, if $y \neq z$ then $y \notin FV(a^{y \to z})$.*

Proof. We prove the claim by rule induction in a. If $a = x$ is a variable and $x = y$, then

$$FV(x^{y \to z}) = FV(z) = \{z\} \subseteq (FV(x) \setminus \{y\}) \cup \{z\}.$$

If instead $x \neq y$, then

$$FV(x^{y \to z}) = FV(x) \subseteq (FV(x) \setminus \{y\}) \cup \{z\},$$

since $y \notin \{x\} = FV(x)$. Finally we have

$$FV\Big(\varphi(\boldsymbol{x}_1.a_1; \dots; \boldsymbol{x}_n.a_n)^{y \to z}\Big) = FV\Big(\varphi(\boldsymbol{x}_1.a_1'; \dots; \boldsymbol{x}_n.a_n')\Big) = \bigcup_{i=1}^{n} (FV(a_i') \setminus \boldsymbol{x}_i).$$

For each i we either have $y \in \boldsymbol{x}_i$ and $FV(a_i') = FV(a_i)$, or else $y \notin \boldsymbol{x}_i$ and $a_i' = a_i^{y \to z}$. In the former case $FV(a_i') \setminus \boldsymbol{x}_i \subseteq (FV(a_i') \setminus \boldsymbol{x}_i) \setminus \{y\}$. In the latter case we have $FV(a_i') \subseteq (FV(a_i) \setminus \{y\}) \cup \{z\}$, so that $FV(a_i') \setminus \boldsymbol{x}_i \subseteq ((FV(a_i) \setminus \boldsymbol{x}_i) \setminus \{y\}) \cup \{z\}$. Hence

$$\begin{aligned}\bigcup_{i=1}^{n} (FV(a_i') \setminus \boldsymbol{x}_i) &\subseteq \Big(\bigcup_{i=1}^{n} (FV(a_i) \setminus \boldsymbol{x}_i) \setminus \{y\}\Big) \cup \{z\} \\ &= \Big(FV\big(\varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)\big) \setminus \{y\}\Big) \cup \{z\},\end{aligned}$$

as claimed. ■

2.20 • LEMMA. *If $x \notin FV(a)$, then the substitution $a[b/x]$ is defined and $a[b/x] = a$.*

Proof. The proof is by rule induction on a. If a is a variable z, then we must have $z \neq x$, and so $z[b/x] = z$.

Assume instead that $a = \varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)$ and that the claim holds for $a_1, \ldots, a_n$. For $i \in \{1, \ldots, n\}$, if $x \in \boldsymbol{x}_i$ then we do not substitute b into a_i, so assume that $x \notin \boldsymbol{x}_i$. By the definition of free variables, x cannot be free in a_i since it is not free in a, so by induction $a_i[b/x] = a_i$. The claim follows. ■

2.21 • LEMMA: *The substitution lemma.*
TODO

2.22 • LEMMA. *If $a : s$, $b : t$ and $x \in \mathcal{V}_t$ such that the substitution $a[b/x]$ is defined, then $a[b/x] : s$.*

Proof. Rule induction on the definition of ASTs. If a is a variable z, then either $z = x$ in which case $a[b/x] = b$ indeed has the same sort as a, or else $z \neq x$ in which case $a[b/x] = a$.

Assume that $a = \varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)$ and that the claim holds for $a_1, \ldots, a_n$. In this case

$$a[b/x] = \varphi(\boldsymbol{x}_1.a_1; \ldots; \boldsymbol{x}_n.a_n)[b/x] = \varphi(\boldsymbol{x}_1.a_1'; \ldots; \boldsymbol{x}_n.a_n'),$$

where each a_i' is either a_i or $a_i[b/x]$. By induction these have the same sort as a_i, so the claim follows by an application of [TODO rule]. ■

(c) As promised we now show that substitution respects α-equivalence. This is a fairly technical result, and it is not easy to find detailed proofs of this fact in the literature[15], even for the simpler case of the untyped λ-calculus.

[15] Harper (2016) simply leaves it as an exercise, cf. Exercise 1.3.

The main reference on the λ-calculus, Barendregt (1984), uses the so-called ***variable convention*** (cf. his §2.1), in which all bound variables are chosen to be different from the free variables occurring in a given context. Furthermore, α-equivalent terms are identified. He afterwards goes on to define substitution, but for substitution to be well-defined in this context it must respect α-equivalence. Barendregt relegates this to his Appendix C: Here he defines substitution anew with using the variable convention and describes how substitution respects α-equivalence. However, he simply refers to Curry et al. (1958, §3.E) for a proof of this result, but the proof that the authors give is rather anachronistic, and as is the nature of proofs of these technical results, it depends highly on the precise way in which the relevant concepts are defined. Hence we give a detailed proof of the claim below for UASTs.

2.23 • PROPOSITION. *If $a =_\alpha a'$ and $b =_\alpha b'$, then $a[b/x] =_\alpha a'[b'/x]$ if the substitutions $a[b/x]$ and $a'[b'/x]$, as well as either $a'[b/x]$ or $a[b'/x]$, are all defined.*

Proof. By Lemma 2.24 and Lemma 2.25 below, we get either

$$a[b/x] =_\alpha a'[b/x] =_\alpha a'[b'/x]$$

or

$$a[b/x] =_\alpha a[b'/x] =_\alpha a'[b'/x],$$

depending on which substitutions are defined. ■

2.24 • LEMMA. *If $a =_\alpha a'$, then $a[b/x] =_\alpha a'[b/x]$ if both substitutions are defined.*

Proof. We prove this claim by rule induction on the definition of $=_\alpha$.

Reflexivity: TODO

Transitivity: TODO

Symmetry: TODO

First rule [TODO ref:] Assume that $a_i[b/x] =_\alpha b_i[b/x]$ for some i. Then [TODO rule] implies that

$$\begin{aligned}\varphi(\boldsymbol{x}_1.a_1;\ldots;\boldsymbol{x}_n.a_n)[b/x] &= \varphi(\boldsymbol{x}_1.a_1[b/x];\ldots;\boldsymbol{x}_n.a_n[b/x])\\ &=_\alpha \varphi(\boldsymbol{x}_1.a_1[b/x];\ldots;\boldsymbol{x}_i.b_i[b/x];\ldots;\boldsymbol{x}_n.a_n[b/x])\\ &= \varphi(\boldsymbol{x}_1.a_1;\ldots;\boldsymbol{x}_i.b_i;\ldots;\boldsymbol{x}_n.a_n)[b/x].\end{aligned}$$

Second rule [TODO ref:] Fix $i \in \{1,\ldots,n\}$. We may assume that $y \in \boldsymbol{x}_i$, $z \notin \boldsymbol{x}_i$ and $z \notin V(a_i)$ [TODO from the rule], so that $y \neq z$ in particular. From the definition of substitution we have four cases:

(1) $x \in \boldsymbol{x}_i$ and $x \in \boldsymbol{x}_i^{y\to z}$: In this case no substitution happens, so this is obvious.

(2) $x \in \boldsymbol{x}_i$ and $x \notin \boldsymbol{x}_i^{y\to z}$: By definition of renaming of sequences of variables, this implies that $x = y$. But then x is not free in $a_i^{y\to z}$ by Lemma 2.19, so Lemma 2.20 implies that $a_i^{y\to z}[b/x] = a_i^{y\to z}$. Since $x \in \boldsymbol{x}_i$ we do not substitute b into a_i, so the claim follows from [TODO second rule].

(3) $x \notin \boldsymbol{x}_i$ and $x \in \boldsymbol{x}_i^{y\to z}$: In this case we must have $x = z$, so $x \notin V(a_i)$, and in particular x is not free in a_i. Hence $a_i[b/x] = a_i$ as before, and since $x \in \boldsymbol{x}_i^{y\to z}$ we do not substitute b into $a_i^{y\to z}$. The claim then follows as in the previous case.

(4) $x \notin \boldsymbol{x}_i$ and $x \notin \boldsymbol{x}_i^{y\to z}$: In this case x is neither y nor z, so $x \notin FV(a_i)$ if and only if $x \notin FV(a_i^{y\to z})$ by Lemma 2.19. The claim thus follows from Lemma 2.20 as above.

Hence we may assume that both $\boldsymbol{x}_i \# FV(b)$ and $\boldsymbol{x}_i^{y\to z} \# FV(b)$, and we must show that

$$\varphi(\ldots;\boldsymbol{x}_i.a_i[b/x];\ldots) =_\alpha \varphi(\ldots;\boldsymbol{x}_i^{y\to z}.a_i^{y\to z}[b/x];\ldots).$$

An application of [TODO rule] implies that it suffices to show that $a_i^{y\to z}[b/x] =_\alpha a_i[b/x]^{y\to z}$. Notice that $y \in \boldsymbol{x}_i$ so $z \in \boldsymbol{x}_i^{y\to z}$, and so neither y nor z is free in b. By Lemma 2.21 we thus have $a_i^{y\to z}[b/x] = a_i[b/x]^{y\to z}$ as desired. ■

2.25 • LEMMA. *If $b =_\alpha b'$, then $a[b/x] =_\alpha a[b'/x]$ if both substitutions are defined.*

Proof. TODO ■

(d) It turns out to be useful to substitute an AST of one sort for a variable of another sort. This is permitted since we have defined substitution for general UASTs, but we need this operation to respect sorts in certain circumstances. To describe the next result we first define renaming of sorts in arities: If t, u are sorts, then we define [TODO specify symbols]

$$\begin{aligned}
s^{t\to u} &:= \begin{cases} u, & s = t, \\ s, & s \neq t, \end{cases} \\
(s_1,\ldots,s_k)^{t\to u} &:= (s_1^{t\to u},\ldots,s_k^{t\to u}), \\
(\boldsymbol{s}.s)^{t\to u} &:= \boldsymbol{s}^{t\to u}.s^{t\to u}, \\
\big((v_1;\ldots;v_n)s\big)^{t\to u} &:= (v_1^{t\to u};\ldots,v_n^{t\to u})s^{t\to u}.
\end{aligned}$$

Put simply, substituting one sort t with another u in an arity α just means going through the arity and replacing every occurrence of t with u.

Consider now an AST a of sort t with a single free variable y of sort t, and assume that all operators in a have two arities of a particular type: Each operator can be thought of as taking arguments of sort t and returning sort t, *as well as* taking arguments of sort u and returning sort u. In this case we can substitute y for another variable z of sort u, and the result $a^{y\to z}$ will then have sort u instead. [TODO refer forward to application]

2.26 • Lemma. *Assume that*

(a) $a : t$,

(b) *a has a single free variable y of sort t,*

(c) *z is a variable of sort u, and*

(d) *every operator in $\Phi(a)$ [TODO define this] has an arity α which takes precisely one argument of sort t, it also has arity $\alpha^{t\to u}$, and it does not bind any variables of sort t [TODO make more precise].*

Then $a^{y\to z} : u$.

Proof. Rule induction on the definition of ASTs. If a is a variable z, then the only free variable in a is z, and so $z = y$. But then $a^{y\to z} = z$, and this indeed has sort u.

Assume instead that $a = \varphi(\boldsymbol{x}_1.a_1;\ldots;\boldsymbol{x}_n.a_n)$ and that the claim holds for $a_1,\ldots,a_n$. Furthermore, write

$$a^{y\to z} = \varphi(\boldsymbol{x}_1.a_1;\ldots;\boldsymbol{x}_n.a_n)^{y\to z} = \varphi(\boldsymbol{x}_1.a_1';\ldots;\boldsymbol{x}_n.a_n'),$$

where the a_i' are as in the definition of renaming.

By definition of free variables we must have [TODO define notation for sorts]

$$\bigcup_{i=1}^{n}(FV_t(a_i)\setminus\boldsymbol{x}_i) = FV_t(a) = \{y\},$$

so that $FV_t(a_i)\setminus\boldsymbol{x}_i \subseteq \{y\}$ for each i. If $FV_t(a_i)\setminus\boldsymbol{x}_i = \emptyset$, then either y is not free in a_i, in which case $a_i' = a_i^{y\to z} = a_i$ by Lemma 2.20 [TODO formulate for renaming, not just for substitution], or else $y \in \boldsymbol{x}_i$, in which case also $a_i' = a_i$.

Otherwise we have $FV_t(a_i)\setminus\boldsymbol{x}_i = \{y\}$. Since φ binds no variables of sort t, $\boldsymbol{x}_i$ contains no variables of sort t, and so $FV_t(a_i) = \{y\}$. Hence we may apply the induction hypothesis to a_i, implying that $a_i^{y\to z} : u$. [TODO TODO TODO] ■

> **Convention.** Abstract syntax trees that are α-equivalent are identified. ❄

2.5 • Abstract reduction systems

Given two relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$, we define their composition

$$R \circ S := \{(x,z) \in X \times Z \mid \exists y \in Y \colon (x,y) \in R \wedge (y,z) \in S\},$$

(note the ordering), and we also define the inverse

$$R^{-1} := \{(y,x) \in Y \times X \mid (x,y) \in R\}.$$

If $X = Y$, then we let $R^0 := \{(x,x) \mid x \in X\}$ be the identity relation on X (i.e., equality), and for $n \in \mathbb{N}$ we recursively define $R^{n+1} := R^n \circ R$. We further define the following:

$$\begin{aligned}
R^{=} &:= R \cup R^0 && \text{reflexive closure} \\
R^{+} &:= \bigcup_{n \in \mathbb{N}^+} R^n && \text{transitive closure} \\
R^{*} &:= R^{+} \cup R^0 && \text{reflexive transitive closure}
\end{aligned}$$

If we denote R by an arrow, e.g., $\to$, then we also write $\xrightarrow{=} := (\to)^{=}$, $\xrightarrow{+} := (\to)^{+}$ and $\xrightarrow{*} := (\to)^{*}$, and we further define

$$\begin{aligned}
\leftarrow &:= (\to)^{-1} && \text{inverse} \\
\leftrightarrow &:= \to \cup \leftarrow && \text{symmetric closure} \\
\overset{+}{\leftrightarrow} &:= (\leftrightarrow)^{+} && \text{transitive symmetric closure} \\
\overset{*}{\leftrightarrow} &:= (\leftrightarrow)^{*} && \text{reflexive transitive symmetric closure}
\end{aligned}$$

Clearly $\xrightarrow{*}$ is a preorder on X, while $\overset{*}{\leftrightarrow}$ is an equivalence relation.

An ***abstract reduction system*** is simply a pair $(X, \to)$, where X is a set and $\to$ is a binary relation on X. If X is a set of states of a computer program, then we think of $\to$ as formalising computation, in the sense that if $e \to e'$, then e reduces to e' by performing some sort of computation.

2.27 • Remark.

(a) We say that $x \in X$ is ***reducible*** if there is a $y \in X$ such that $x \to y$, and otherwise x is ***irreducible*** or ***in normal form***. An element $y \in X$ is ***a normal form*** of x if $x \xrightarrow{*} y$ and y is in normal form. In the context of programming languages, normal forms are supposed to model the end result of a successful computation.

(b) If $x, y \in X$, then x and y are ***joinable*** if there is a $z \in X$ with $x \xrightarrow{*} z \xleftarrow{*} y$. In this case we write $x \downarrow y$. Note that this does not mean that z is the join of x and y with respect to the preorder $\xrightarrow{*}$, but simply that z is an upper bound.

We say that $\to$ is ***Church–Rosser*** if $x \overset{*}{\leftrightarrow} y$ implies $x \downarrow y$ for all $x, y \in X$. Seemingly a weaker property, $\to$ is ***confluent*** if instead $x \overset{*}{\leftarrow} z \overset{*}{\to} y$ implies $x \downarrow y$, but one can show that these properties are equivalent, cf. Baader and Nipkow (1998, Theorem 2.1.5).

In a particular state of a computer program we might allow the computation to proceed to multiple different ways. For instance, we might allow the arguments to functions to be evaluated in arbitary order, and if $\to$ is confluent then no matter which computations are performed when, it is always possible to continue the computation in such a way that both branches end up the same place. Of course, confluence does not ensure that this happens, it only ensures that it is possible.

(c) We call $\to$ ***terminating*** if there is no infinite chain $x_1 \to x_2 \to \cdots$. If every such chain terminates, then x_1 must have a normal form. If every $x \in X$ has a normal form, then $\to$ is called ***normalising***, and this is clearly a strictly weaker property than being terminating.[16]

Termination is of course a desirable property, but it is not exhibited by most programming languages. Neither is normalisation. Note the difference between these two properties: Termination says that any computation always terminates, normalisation says that no matter the state of the program, there is a computation which terminates the program, but the program is not required to perform this computation. ⌟

2.28 • EXAMPLE: *λ-calculus.*

In the λ-calculus there are various ways of defining a notion of reduction (for instance corresponding to call-by-value or call-by-name, cf. Pierce 2002, §5.1), but the usual is ***full β-reduction***, which is nondeterministic and allows any subexpression which is not a normal form [TODO redexes?? Bader/Nipkow ch 4 + nederpelt/geuvers] to be reduced. Equipped with this notion of reduction, the untyped λ-calculus and both the first- and second-order simply typed λ-calculi are confluent, and so is the calculus of constructions.

However, while the typed λ-calculi are terminating, the untyped λ-calculus is not. For instance, when the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ is reduced we simply obtain Ω itself again. On the other hand, the termination of e.g. simply typed λ-calculus implies that it is impossible to implement recursive functions using only λ-abstraction and application (since recursive functions may not terminate), and recursive functions must instead be added to the theory 'by hand'. [TODO refer to later, System F]

[16] Termination and normalisation are also called ***strong*** and ***weak normalisation*** respectively.

We refer to Nederpelt and Geuvers (2014) for further discussion and proofs or references to proofs of the above claims. ⌟

3 ◇ General stuff about languages

We describe the general framework in which we may describe various programming languages, introducing the concepts that will later be defined precisely in the concrete setting of System F.

3.1 • Syntax

We first fix countable sets $\mathcal{V}$ of variables and Loc of locations. The ***expressions*** of the language will be a set Exp containing both $\mathcal{V}$ and Loc, and we designate some of these expressions to be ***values***, collected in a set Val. We think of an expression as specifying the state of a program, and values are states in which the computation of the program has finished. All locations will also be values, and these are supposed to model memory addresses. The memory state of the program (i.e. the part of the memory that the program has access to) is modelled by a ***store***[17], which is an element of $P_\omega(Loc, Exp)$. We simply write Sto for this set of partial maps.

For $e \in Exp$ we define the set $FV(e) \subseteq \mathcal{V}$ of ***free variables***. There are various ways of binding variables in expressions, and the notion of free variables is supposed to capture the idea that variables can be bound, e.g. by lambda abstraction, and hence also *not* bound. If $FV(e) = \emptyset$, then we say that e is ***closed***. Complete programs do not have free variables, or if they do we specify their values before running the program, so we may assume that all programs are closed expressions. We will see the importance of this assumption when we prove a progress theorem for System F in [TODO ref].

When defining the set of expressions of a language, we are strictly speaking defining a concrete syntax for the language. However, while we write expressions as a linear sequence of characters, they are thought of as describing an abstract syntax. But since writing abstract syntax trees quickly becomes impractical, we instead express them using a more convenient linear shorthand. If e_1, e_2 and e_3 are expressions, an expression in the concrete syntax of the language could be $e_1 \oplus e_2 \otimes e_3$, but this might have two different derivations from the grammar and hence correspond to two different abstract syntax trees represented by $(e_1 \oplus e_2) \otimes e_3$ and $e_1 \oplus (e_2 \otimes e_3)$. To disambiguate the expression $e_1 \oplus e_2 \otimes e_3$ we must either introduce precedence and associativity rules (external to the grammar itself), or else rewrite the grammar to be unambiguous.

[17] Sometimes called a ***heap***, but this has nothing to do with the heap *data structure*.

Luckily we do not need to deal with these issues since we will not have to parse programs written in our version of System F. If the need arises, we simply use parentheses to make the structure of the abstract syntax tree clear. [TODO Mogensen?]

3.2 • Type system

In order to reason statically about the correctness and safety of a program, we introduce ***types***. Each sufficiently 'well-formed' expression will be assigned a type, and if it is possible to assign a type to an expression, then we say that the expression is ***well-typed***. We specify rules which determine which expressions can be typed based on the types of their subexpressions.

We thus define a set *Type* of types. This includes a countable set *TVar* of type variables, any base types (e.g. unit, integer, and boolean types), as well as more complex types that can be constructed recursively from the base types such as function, product, or sum types. It also includes reference types, which are the types of locations. For $\tau \in \mathit{Type}$ we define the set of ***free type variables*** $FTV(\tau) \subseteq \mathit{TVar}$ in τ. If $FTV(\tau) = \emptyset$, then we also say that τ is ***closed***. A pair (e, τ) of an expression and a type is usually written $e : \tau$. An expression may also have types as subexpressions, for instance if a lambda abstraction has a type annotation on its parameter, though this will not be the case for our version of System F.

If an expression e has free variables, then to assign a type to e it is (usually) necessary to first assign types to the free variables in e. This is done using a ***type context***, which is a partial function $P_\omega(\mathcal{V}, \mathit{Type})$. If e is to be well-typed in a type context Γ, then we (again usually) require that $FV(e) \subseteq \operatorname{dom}\Gamma$. That is, Γ must in fact specify the types of the variables that occur free in e. Notice that it is possible for an expression to be well-typed in one context but not another. If for instance e is the expression $x + 1$, then the typing rules will probably require the free variable x to have some sort of numeric type in the given type context for e to be well-typed.

Furthermore, if e has a location as a subexpression, then we need to be able to look up the type of the expression stored at this location in order to specify the type of e. Hence the type of e can also depend on the store in question. However, it not in general possible to deduce the type of e just by knowing the contents of the store: If σ is a store with $l_1, l_2 \in \operatorname{dom}\sigma$, and if $\sigma(l_1)$ references l_2 and $\sigma(l_2)$ references l_1, then it is impossible to deduce the type of $\sigma(l_1)$. Hence we introduce a ***store typing***, which is an element of $P_\omega(\mathit{Loc}, \mathit{Type})$ that assigns a type to each location. We of course require that the store typing in question actually contains in its domain all locations referenced in e, and we furthermore require that all free variables of e lie in the domain of the current

type context.

Since a store σ specifies the expressions at each location, and a store typing Σ specifies the types of those locations, we of course require $\sigma(l)$ to be of type $\Sigma(l)$. In particular, for σ to be well-typed we must have $\operatorname{dom}\sigma \subseteq \operatorname{dom}\Sigma$. [TODO but why equal? Refer to later proofs where we use this]

It may be that the type of e or of the types of the variables in the type context Γ or of the expressions in the store typing Σ contain type variables. In order to keep track of these we collect these in a (finite) set Ξ and require that the free type variables in Γ and Σ, defined by

$$FTV(\Gamma) = \bigcup_{\tau \in \operatorname{ran}\Gamma} FTV(\tau) \quad \text{and} \quad FTV(\Sigma) = \bigcup_{\tau \in \operatorname{ran}\Sigma} FTV(\tau),$$

are contained in Ξ. A variable x is called ***fresh*** for Γ if $x \notin \operatorname{dom}\Gamma$. The finitude of $\operatorname{dom}\Gamma$ ensures that there always exist fresh variables (recall that there are countably infinitely many variables). If Δ is another type context such that $\operatorname{dom}\Gamma \cap \operatorname{dom}\Delta = \emptyset$, then instead of $\Gamma \cup \Delta$ we simply write Γ, Δ. Furthermore, if $\Delta = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ for distinct x_i, then we omit the braces and write $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$. If Ξ and Φ are finite disjoint subsets of $TVar$, then we similarly write Ξ, Φ for $\Xi \cup \Phi$, and if $\Phi = \{X_1, \ldots, X_n\}$ for distinct X_i, then we also write $\Xi, X_1, \ldots, X_n$.

We are now ready to describe the semantics of the type system precisely. The semantics is captured by a subset of the set

$$\mathcal{P}_\omega(TVar) \times (P_\omega(\mathcal{V}, Type)) \times (P_\omega(Loc, Type)) \times Exp \times Type,$$

where an element $(\Xi, \Gamma, \Sigma, e, \tau)$ of this set is written $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and is called a ***type derivation***. If $\Xi = \emptyset$, then we simply denote the above element by $\Gamma \mid \Sigma \vdash e : \tau$ [TODO do we need this?], and we furthermore write $\Sigma \vdash e : \tau$ if also $\Gamma = \emptyset$.

Say that we have somehow settled on a semantics for the type system, i.e. a subset of the above product. We do not often refer to this set explicitly, but let us temporarily denote it $\mathcal{T}$. Usually $\mathcal{T}$ is defined recursively, by specifying a series of inference rules. Indeed, these inference rules represent a generating function, and $\mathcal{T}$ will be the least fixed-point of this function.

If a type derivation $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ lies in $\mathcal{T}$, then we say that e is ***well-typed*** in Ξ, Γ, Σ with type τ.

3.3 • Dynamics

The operational semantics of the language is specified in a small-step style. We describe this semantics in stages, beginning with the ***pure head reductions***. These are evaluations that can be performed (1) on expressions that have

no subexpressions that can be evaluated, (2) without reading or modifying the store. More precisely, we specify a binary relation $\to_p$ on Exp, such that $e \to_p e'$ is supposed to mean that e evaluates to or reduces to e'.

Going one level up we define the relation $\to_h$ on $Sto \times Exp$ of (not necessarily pure) ***head reductions***. These are reductions that may affect and be affected by the contents of the store. Of course, if σ is a store and $e \to_p e'$, then we have $(\sigma, e) \to_h (\sigma, e')$, but we augment the pure head reductions with reductions that e.g. read from or write to the store.

Finally we need a way to evaluate complex expressions. One way of doing this is to specify the evaluation rules for all expressions immediately instead of going through head reductions (this is the approach taken by Pierce TODO). Another is to introduce ***evaluation contexts***, which are (essentially) maps $Exp \to Exp$. If K is an evaluation context and e is an expression, then we write $K[e]$ for the value of K at e. We then define the final reduction relation $\to$ on $Sto \times Exp$ by letting $(\sigma, K[e]) \to (\sigma', K[e'])$ if $(\sigma, e) \to_h (\sigma', e')$.

One role of evaluation contexts is to specify the evaluation order of complex expressions, e.g. if the evaluation of function applications is call-by-value or call-by-name, or if we evaluate the arguments to functions left-to-right or right-to-left. The possibilities thus depend on the available evaluation contexts.

TODO multiple threads

4 ◇ System F

The following grammar defines the syntax, the sets of values and types, and the evaluation contexts of System F:

$$
\begin{array}{lll}
 & x \in \mathcal{V} & \\
 & l \in Loc & \\
 & X \in TVar & \\
Exp & e ::= () \mid x \mid l \mid (e, e) \mid \cdots & \\
Val & v ::= () \mid l \mid (v, v) \mid \iota_1\, v \mid \cdots & \\
Type & \tau ::= \mathsf{Unit} \mid X \mid \mathsf{ref}(\tau) \mid \tau \times \tau \mid \cdots & \\
ECtx & K ::= - \mid (K, e) \mid (v, K) \mid \cdots &
\end{array}
$$

We first note some difficulties with this and similar definitions. Recall that a ***grammar*** is a tuple $G = (V, \Sigma, P, S)$, where V is a finite set of variables or non-terminal symbols, Σ is a finite set of terminal symbols that is disjoint from V, P is a finite rewriting system[18] on $V \cup \Sigma$, and $S \in V$ is the start symbol.

[18] A ***rewriting system*** P on a set A is a binary relation on A^*, i.e. a subset of $A^* \times A^*$. An element of P is called a ***production***. If $(\alpha, \beta) \in P$ and $\gamma, \delta \in A^*$, then we write $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$.

The language generated by G is the language $L(G) = \{\alpha \in \Sigma^* \mid S \Rightarrow^* \alpha\}$. We say that G is ***context-free*** if every production in P is on the form $A \Rightarrow \alpha$, where $A \in V$.

Notice especially that a grammar must only contain *finitely* many productions, but also that the above 'grammar' defining System F[19] as written has infinitely many productions: For instance, there is a production (e, x) for each of the infinitely many variables $x \in \mathcal{V}$. While this poses no problems for the abstract study of System F that we are undertaking, we note that it is (or at least in practice it will be) possible to rewrite the 'grammar' so that it has only finitely many productions. We may for instance define the countably many variables recursively by the grammar with productions

$$x ::= \texttt{x} \mid x\texttt{'},$$

yielding the countably many variables x, x', x'', and so on. Including these productions instead of the postulate '$x \in \mathcal{V}$' would solve at least this problem. We can do similarly for locations and type variables.

Next we notice that the 'grammar' has no obvious start symbol. Indeed, we may take any of e, v, τ or K to be the start symbol, depending on the type of string we wish to construct.

With these issues dealt with, we now relate grammars and the languages they generate to the results from TODO ref, and in particular to inference rules. In fact, since the 'grammar' defining System F is not a proper grammar, we consider more broadly ***infinite grammars***. An infinite grammar is just a grammar $G = (V, \Sigma, P, S)$, except that we allow V, Σ and P to be infinite. Given G we construct a collection of inference rules as follows: First add the axiom

$$\frac{}{S}$$

saying that we can always derive the start symbol S. Next, for every production $(\alpha, \beta) \in P$ and strings $\gamma, \delta \in (V \cup \Sigma)^*$ we add the rule

$$\frac{\gamma\alpha\delta}{\gamma\beta\delta}$$

Denote the resulting collection of inference rules $\mathcal{R}_G$.

4.1 • Proposition. *$\mu\mathcal{R}_G = \{\alpha \in (V \cup \Sigma)^* \mid S \Rightarrow^* \alpha\}$. In particular, TODO no variables*

This defines another binary relation $\Rightarrow$ on A^*, the reflexive and transitive closure of which is denoted $\Rightarrow^*$.

[19] We put the word 'grammar' in scare quotes since it is not actually a grammar.

Proof. Denote the set $\{\alpha \in (V \cup \Sigma)^* \mid S \Rightarrow^* \alpha\}$ by $\tilde{L}(G)$. We prove the inclusion '$\subseteq$' by rule induction, cf. Theorem 2.7. ■

TODO if all rules have one antecedent, maybe let that define a relation (axioms don't give an element of the relation) + show about transitive closure?

4.2 • Lemma. *$Val \subseteq Exp$.*

Proof.

Notice that $Loc \subseteq Val \subseteq Exp$ as we required in TODO ref. If K is an evaluation context and e is an expression, then we define $K[e]$ recursively by

$$\begin{aligned} -[e] &= e \\ (K, e')[e] &= (K[e], e') \\ (v, K)[e] &= (v, K[e]) \\ &etc. \end{aligned}$$

It is easy to prove (by induction in K) that $K[e]$ is an expression, so every evaluation context K can indeed be thought of as a map $Exp \to Exp$ given by $e \mapsto K[e]$.

If e is an expression, then the set $FV(e)$ of free variables in e is defined recursively as follows:

$$\begin{aligned} FV(()) &= \emptyset \\ FV(x) &= \{x\} \\ FV((e_1, e_2)) &= FV(e_1) \cup FV(e_2) \\ &etc. \end{aligned}$$

Similarly, if τ is a type, then we define the set $FTV(\tau)$ of free type variables in τ as follows:

$$\begin{aligned} FTV(\mathsf{Unit}) &= \emptyset \\ FTV(X) &= \{X\} \\ FTV(\tau_1 \times \tau_2) &= FTV(\tau_1) \cup FTV(\tau_2) \\ &etc. \end{aligned}$$

We are now in a position to define the typing relation. This is the smallest relation on the set

$$\mathcal{P}_\omega(TVar) \times (P_\omega(\mathcal{V}, Type)) \times (P_\omega(Loc, Type)) \times Exp \times Type,$$

satisfying the following inference rules:

$$\frac{FTV(\Gamma) \subseteq \Xi \qquad FTV(\Sigma) \subseteq \Xi \qquad (x : \tau) \in \Gamma}{\Xi \mid \Gamma \mid \Sigma \vdash x : \tau} \text{ T-VAR}$$

$$\frac{FTV(\Gamma) \subseteq \Xi \qquad FTV(\Sigma) \subseteq \Xi \qquad l \in \operatorname{dom} \Sigma}{\Xi \mid \Gamma \mid \Sigma \vdash l : \mathsf{ref}(\Sigma(l))} \text{ T-LOC}$$

Lemma: If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, then $FTV(\Gamma) \subseteq \Xi$ and $FV(e) \subseteq \operatorname{dom} \Gamma$. In particular, if $\Xi = \emptyset$ then τ is closed, and if $\Gamma = \emptyset$ then e is closed. TODO

TODO all type variables in tau also in Xi? All locations in e also in Sigma?

TODO lemma weakening?

4.1 • Lemmas

4.3 • Lemma: *Inversion.*
Assume that $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$.

(a) *If* $e = x$ *is a variable, then* $(x : \tau) \in \Gamma$.

(b) *If* $e = ()$, *then* $\tau = \mathsf{Unit}$.

(c) *If* $e = (e_1, e_2)$, *then* $\tau = \tau_1 \times \tau_1$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$.

(d) *If* $e = \pi_1\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau \times \tau_2$.

(e) *If* $e = \pi_2\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1 \times \tau$.

(f) *If* $e = \iota_1\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1$.

(g) *If* $e = \iota_2\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_2$.

(h) *If* $e = \mathsf{match}\ e_1\ \mathsf{with}\ \iota_1\, x \Rightarrow e_2 \mid \iota_2\, x \Rightarrow e_3\ \mathsf{end}$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e_2 : \tau$ *and* $\Xi \mid \Gamma, x : \tau_2 \mid \Sigma \vdash e_3 : \tau$.

(i) *If* $e = \mathsf{rec}\, f(x) := e'$, *then* $\tau = \tau_1 \to \tau_2$ *and* $\Xi \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \Sigma \vdash e' : \tau_2$.

(j) *If* $e = e_1\, e_2$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \to \tau$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau$.

(k) *If* $e = \Lambda\, e'$, *then* $\tau = \forall X.\tau'$ *and* $\Xi, X \mid \Gamma \mid \Sigma \vdash e' : \tau'$.

(l) *If* $e = e'\ \_$, *then* $\tau = \tau'[\tau''/X]$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \forall X.\tau'$.

(m) *If* $e = \mathsf{pack}\, e'$, *then* $\tau = \exists X.\tau'$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'[\tau''/X]$.

(n) *If* $e = \mathsf{unpack}\, e_1\, \mathsf{as}\, x\, \mathsf{in}\, e_2$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \exists X.\tau'$ *and* $\Xi, X \mid \Gamma, x : \tau' \mid \Sigma \vdash e_2 : \tau$.

(o) *fold TODO*

(p) *unfold TODO*

(q) *If* $e = l$ *is a location, then* $l \in \operatorname{dom} \Sigma$ *and* $\tau = \mathsf{ref}(\Sigma(l))$.

(r) *If* $e = \mathsf{ref}\, e'$, *then* $\tau = \mathsf{ref}(\tau')$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'$.

(s) *If* $e = e_1 := e_2$, *then* $\tau = \mathsf{Unit}$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \mathsf{ref}(\tau')$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau'$.

(t) *If* $e = !e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \mathsf{ref}(\tau)$.

Proof. Notice that since the conclusions of different inference rules are distinct, there is a unique rule that was applied last in the derivation of $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ [TODO ref what that means]. This means that if e has any of the above forms, then it must have the type as assigned by the relevant inference rule, and the assumptions of that rule must also hold.

For instance, if there exist expressions e_1 and e_2 such that $e = (e_1, e_2)$, then the last rule applied must be T-PAIR. But then τ must be on the form $\tau_1 \times \tau_2$ for types τ_1 and τ_2. And furthermore, the assumptions must also hold, implying that $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$. The other cases are proved in the same way. ■

Notice the significance of the inversion lemma: While an expression can generally have many different types (if nothing else then due to substitution into type variables), it seems natural to believe that e.g. pairs cannot be of function type. The inversion lemma says precisely this, that if a pair has any type, then that type must be a product type. Note that the lemma does *not* just say that a well-typed pair is of product type, it rather says that a well-typed pair is *only* of product type.

4.4 • LEMMA: ***Canonical forms.***
Assume that $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau$ *where* v *is a value.*

(a) *If* $\tau = \mathsf{Unit}$, *then* $v = ()$.

(b) *If* $\tau = \tau_1 \times \tau_2$, *then* $v = (v_1, v_2)$.

(c) *If* $\tau = \tau_1 + \tau_2$, *then either* $v = \iota_1\, v'$ *or* $v = \iota_2\, v'$.

(d) *If* $\tau = \tau_1 \to \tau_2$, *then* $v = \mathsf{rec}\, f\, (x) := e$.

(e) *If* $\tau = \forall X.\tau'$, *then* $v = \Lambda\, e$.

(f) *If* $\tau = \exists X.\tau'$, *then* $v = \mathsf{pack}\, e$.

(g) *TODO fold*

(h) *If* $\tau = \mathsf{ref}(\tau')$, *then* v *is a location.*

Proof. We assume that $\tau = \tau_1 \times \tau_2$ for concreteness; the other cases are identical. In this case we simply check for each production of the grammar with non-terminal v whether the relevant value can have type $\tau_1 \times \tau_2$. For instance, Lemma 4.3(f) implies that a value $\iota_1\, v'$ can only be of sum type, and hence v cannot be of this form. The only possibility is that v is in fact a pair. ■

— TODO substitution lemma

4.5 • LEMMA. *If* $\Xi, X \mid \Gamma \mid \Sigma \vdash e : \tau$, *then* $\Xi \mid \Gamma[\tau'/X] \mid \Sigma \vdash e : \tau[\tau'/X]$.

Proof. T-VAR: Assume that $\Xi, X \mid \Gamma \mid \Sigma \vdash x : \tau$. By Lemma 4.3(a) we thus have $(x : \tau) \in \Gamma$, so $(x : \tau[\tau'/X]) \in \Gamma[\tau'/X]$ by definition of substitution. But then T-VAR implies that $\Xi \mid \Gamma[\tau'/X] \mid \Sigma \vdash x : \tau[\tau'/X]$ (where we use that X does not occur in the context or type, so it doesn't need to appear in Ξ).

T-REC: Assume that $\Xi, X \mid \Gamma \mid \Sigma \vdash \mathrm{rec}\, f(x) := e : \tau_1 \to \tau_2$. By the inversion lemma [TODO ref – also can't we just do induction??] we have $\Xi, X \mid \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \Sigma \vdash e : \tau_2$, so by induction it follows that $\Xi \mid \Gamma[\tau'/X], f : \tau_1[\tau'/X] \to \tau_2[\tau'/X], x : \tau_1[\tau'/X] \mid \Sigma \vdash e : \tau_2[\tau'/X]$ [TODO by substitution on envs, function types etc.]. Applying T-REC we obtain the desired claim. ■

TODO rest

4.2 • Progress

4.6 • THEOREM: *Progress.*
If $\Sigma \vdash e : \tau$, *then either* e *is a value or else, for any store* σ *with* $\Sigma \vdash \sigma$, *there exists an expression* e' *and a store* σ' *such that* $(\sigma, e) \to (\sigma', e')$.

Proof. The proof is by induction on the typing relation $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, but the claim to be proved is augmented by 'or either Ξ or Γ is non-empty'.[20] Notice that the induction step for each inference rule is trivial if Ξ or Γ is non-empty, so we need only prove each case when Ξ and Γ are empty. Furthermore, since the store is relevant for only a few reductions, we suppress it from the notation in most of the cases below, simply taking about expressions reducing to other expressions.

T-UNIT: Since $()$ is a value, this follows.

T-VAR: Since we may assume that Γ is empty, this implication is vacuously true.[21]

T-PAIR: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1$ and $\Sigma \vdash e_2 : \tau_2$. If both e_1 and e_2 are values, then (e_1, e_2) is also a value, so assume that only $e_1 = v_1$ is a value and that $e_2 \to e_2'$. Since the only rule that generates instances of

[20] This ensures that we can perform the induction on the entire 4-ary relation, which is important since this relation is the one that is defined by the inference rules, *not* the corresponding binary relation obtained by restricting the ternary relation to the subset where Ξ and Γ are empty.

[21] In the formalism of TODO ref T-VAR would not be an inference rule, it would instead be an axiom requiring the relation to include all quadruples (Ξ, Γ, e, τ) such that $(e : \tau) \in \Gamma$ and all type variables in τ occur in Ξ. This is of course also vacuously true when Γ is empty.

the one-step relation $\to$ is HEAD-STEP-STEP, it follows that e_2 is on the form $K[d_2]$ and e_2' is on the form $K[d_2']$, where K is an evaluation context and d_2 and d_2' are subexpressions of e_2 and e_2' respectively such that $d_2 \to_h d_2'$. Letting $K' = (v_1, K)$ it follows that $(v_1, e_2) = K'[d_2]$ and $(v_1, e_2') = K'[d_2']$, and so

$$(v_1, e_2) = K'[d_2] \to K'[d_2'] = (v_1, e_2')$$

by HEAD-STEP-STEP. If instead e_1 is not a value, then the same argument (using the evaluation context (K, e_2)) yields the same result.

T-FST: Assume that the claim holds for $\Sigma \vdash e : \tau_1 \times \tau_2$. If e is a value, then it is on the form (v_1, v_2) by Lemma 4.4(b), where v_1 and v_2 are values. Hence $\pi_1\, e = \pi_1\, (v_1, v_2)$, and this reduces via a head-step to v_1. Choosing the evaluation context $K = -$, HEAD-STEP-STEP implies that $\pi_1\, (v_1, v_2) \to v_1$. If instead e is not a value, then by induction there is some e' such that $e \to e'$. Hence there are subexpressions d and d' and an evaluation context K such that $e = K[d]$, $e' = K[d']$ and $d \to_h d'$. Letting $K' = \pi_1\, K$ we have

$$\pi_1\, e = K'[d] \to K'[d'] = \pi_1\, e',$$

as desired.

T-SND: Similar to T-FST.

T-INJ1: Assume that the claim holds for $\Sigma \vdash e : \tau_1$. If e is a value v, then so is $\iota_1\, v$. If instead $e \to e'$, then as before $e = K[d]$, $e' = K[d']$ and $d \to_h d'$. Letting $K' = \iota_1\, K$ we get $\iota_1\, e = K'[d]$ and $\iota_1\, e' = K'[d']$, so $\iota_1\, e \to \iota_1\, e'$.

T-INJ2: Similar to T-INJ1.

T-MATCH: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1 + \tau_2$. If e_1 is a value, then by Lemma 4.4(c) it must be on the form $\iota_1\, v$ or $\iota_2\, v$ for a value v. Hence the expression $\mathsf{match}\ e_1\ \mathsf{with}\ \iota_1\, x \Rightarrow e_2 \mid \iota_2\, x \Rightarrow e_3\ \mathsf{end}$ can reduce via a head step by either E-MATCH-INJ1 or E-MATCH-INJ2, so it reduces by HEAD-STEP-STEP (using the evaluation context $K = -$). If instead $e_1 \to e_1'$, then by the same argument as in previous cases with $K' = \mathsf{match}\ K\ \mathsf{with}\ \iota_1\, x \Rightarrow e_2 \mid \iota_2\, x \Rightarrow e_3\ \mathsf{end}$, it follows that $\mathsf{match}\ e_1\ \mathsf{with}\ \iota_1\, x \Rightarrow e_2 \mid \iota_2\, x \Rightarrow e_3\ \mathsf{end}$ reduces.

T-REC: This is obvious since $\mathsf{rec}\, f\, (x) := e$ is a value.

T-APP: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1 \to \tau_2$ and $\Sigma \vdash e_2 : \tau_1$. If e_1 is a value, then by Lemma 4.4(d) it must be on the form $\mathsf{rec}\, f\, (x) := e$. If also e_2 is a value, then the claim follows by E-REC-APP. If $e_1 = v$ is a value but e_2 is not, then $e_2 \to e_2'$. The same argument as in previous cases with $K' = v\, K$ shows that $v\, e_2$ reduces. Finally, if e_1 is not a value, then $e_1 \to e_1'$, and choosing $K' = K\, e_2$ proves the claim.

T-TLAM: This is obvious since $\Lambda\, e$ is a value.

T-TAPP: Assume that the claim holds for $\Sigma \vdash e : \forall X.\tau$. If e is a value, then by Lemma 4.4(e) it must be on the form $\Lambda\, e'$, so the claim follows from E-TAPP-TLAM (via HEAD-STEP-STEP using the evaluation context $K = -$). If e is not a value, then $e \to e'$ for some expression e' by induction. These expressions then have subexpressions d and d' respectively such that $d \to_h d'$, and such that $e = K[d]$ and $e' = K[d']$ for some evaluation context K. Letting $K' = K\,\_$ we thus have $e\,\_ = K'[d]$ and $e'\,\_ = K'[d']$, proving the claim.

T-PACK: Assume that the claim holds for $\Sigma \vdash e : \tau[\tau'/X]$. If e is a value, then so is $\mathsf{pack}\, e$. Otherwise $e \to e'$ for some expression e'. The same argument as before using the evaluation context $\mathsf{pack}\, K$ for an appropriate K yields the claim.

T-UNPACK: Assume that the claim holds for $\Sigma \vdash e_1 : \exists X.\tau$. If e_1 is a value, then it must be on the form $\mathsf{pack}\, v$ by Lemma 4.4(f), so an application of E-UNPACK-PACK yields the claim. Otherwise $e \to e'$ for some e', and we use the evaluation context $\mathsf{unpack}\, K\, \mathsf{as}\, x\, \mathsf{in}\, e_2$ for an appropriate K.

T-FOLD: TODO

T-UNFOLD: TODO

T-LOC: Locations are values, so this is obvious.

T-ALLOC: Assume that the claim holds for $\Sigma \vdash e : \tau$. If e is a value, then the claim follows by applying E-ALLOC, noting that there always exists a location $l \notin \operatorname{dom} \sigma$. Otherwise there is an expression e' and a store σ' such that $(\sigma, e) \to (\sigma', e')$. But then there is some evaluation context K and subexpressions d and d' of e and e' respectively, such that $e = K[d]$, $e' = K[d']$ and $(\sigma, d) \to_h (\sigma', d')$. Letting $K' = \mathsf{ref}\, K$ we have $\mathsf{ref}\, e = K'[d]$ and $\mathsf{ref}\, e' = K'[d']$, so HEAD-STEP-STEP implies that $(\sigma, \mathsf{ref}\, e) \to (\sigma', \mathsf{ref}\, e')$.

T-STORE: Assume that the claim holds for $\Sigma \vdash e_1 : \mathsf{ref}(\tau)$ and $\Sigma \vdash e_2 : \tau$, and let σ be a store with $\Sigma \vdash \sigma$. If e_1 is a value, then by Lemma 4.4(h) it is a location l, and by Lemma 4.3(q) we have $l \in \operatorname{dom} \Sigma = \operatorname{dom} \sigma$. If e_2 is also a value, then the claim follows from E-STORE. If $e_1 = l$ is a value but e_2 is not, then there is some e_2' and σ' such that $(\sigma, e_2) \to (\sigma', e_2')$. Again writing $e_2 = K[d_2]$ and $e_2' = K[d_2']$ with $(\sigma, d) \to_h (\sigma', d')$, we use the evaluation context $l := K$. Finally, if e_1 is not a value, then the same argument using instead $K := e_2$ yields the claim.

T-LOAD: Assume that the claim holds for $\Sigma \vdash e : \mathsf{ref}(\tau)$, and let σ be a store with $\Sigma \vdash \sigma$. If e is a value, then as before it is a location l, and $l \in \operatorname{dom} \sigma$. It then follows from E-LOAD that $(\sigma, !\, l) \to_h (\sigma, v)$, where $v = \sigma(l)$. If instead $(\sigma, e) \to (\sigma', e')$ for an expression e' and a store σ', then we simply use the evaluation context $!K$ for an appropriate K. ∎

4.3 • Preservation

If $\Xi \subseteq_\omega \mathit{TVar}$, Γ is a type context and Σ is a store typing, then we say that a store σ is ***well-typed*** with respect to Ξ, Γ and Σ if $\operatorname{dom}\sigma = \operatorname{dom}\Sigma$ and $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$ for all $l \in \operatorname{dom}\sigma$. In this case we write $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$, and if Ξ and Γ are both empty we simply write $\Sigma \vdash \sigma$.

4.7 • Theorem: *Preservation.*
If

$$\Xi \mid \Gamma \mid \Sigma \vdash e : \tau, \quad \Xi \mid \Gamma \mid \Sigma \vdash \sigma \quad \textit{and} \quad (\sigma, e) \to (\sigma', e'),$$

then there exists some store typing Σ' with $\Sigma \subseteq \Sigma'$ such that

$$\Xi \mid \Gamma \mid \Sigma' \vdash e' : \tau \quad \textit{and} \quad \Xi \mid \Gamma \mid \Sigma' \vdash \sigma'.$$

Proof. By definition of the one-step relation, there exist an evaluation context K and subexpressions d of e and d' of e' such that $e = K[d]$, $e' = K[d']$, and $(\sigma, d) \to_h (\sigma', d')$. By Lemma 4.8 there is some type ρ such that $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$. Next it follows from Lemma 4.11 that $\Xi \mid \Gamma \mid \Sigma' \vdash d' : \rho$ for some store typing Σ' with $\Sigma \subseteq \Sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$. By Lemma 4.9 we also have $\Xi \mid \Gamma \mid \Sigma' \vdash d : \rho$, so it follows from Lemma 4.10 that $\Xi \mid \Gamma \mid \Sigma' \vdash K[d'] : \tau$ as desired. ■

4.8 • Lemma. *If K is an evaluation context, e is an expression and $\Xi \mid \Gamma \mid \Sigma \vdash K[e] : \tau$, then $\Xi \mid \Gamma \mid \Sigma \vdash e : \rho$ for some type ρ.*

Proof. Every evaluation context is obtained from the hole '$-$' by finitely many applications of the productions in the grammar [TODO prove this?]. We prove the claim by induction on the length of such a sequence of productions. If $K = -$, then the claim is obvious, since then $K[e] = e$. Hence we assume that K is obtained from some evaluation context K' by some application of a production, so that the induction hypothesis holds for K'.

$K = (K', e')$: Then $K[e] = (K'[e], e')$, and since this is well-typed with type τ, Lemma 4.3(c) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau_1$ for some type τ_1. By induction applied to K' we have $\Xi \mid \Gamma \mid \Sigma \vdash e : \rho$ for some type ρ.

$K = (v, K')$: Similar to the above.

$K = \pi_1\, K'$: Then $K[e] = \pi_1\, K'[e]$, so Lemma 4.3(d) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau \times \tau_2$ for some τ_2. By induction we have $\Xi \mid \Gamma \mid \Sigma \vdash e : \rho$ for some type ρ.

$K \in \{\pi_2\, K', \iota_1\, K', \iota_2\, K'\}$: Similar to the above.

$K = \mathsf{match}\ K'\ \mathsf{with}\ \iota_1\, x \Rightarrow e_1 \mid \iota_2\, x \Rightarrow e_2\ \mathsf{end}$: Here Lemma 4.3(h) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau_1 + \tau_2$, so the claim follows by induction.

$K = K'\, e'$: Then $K[e] = K'[e]\, e'$, so Lemma 4.3(j) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau_1 \to \tau$ for some type τ_1. The claim follows by induction as before.

$K = v\,K'$: Similar to the above.

$K = K'\,\_$: Lemma 4.3(l) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau_1$ for some type τ_1, so the claim follows by induction.

$K =$: TODO

$K = \mathsf{ref}\,K'$: Then $K[e] = \mathsf{ref}\,K'[e]$, so the inversion lemma implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau'$. The claim follows by induction.

$K = K' := e'$: Then $K[e] = K'[e] := e'$, so the inversion lemma implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau'$ ... TODO

TODO rest – but they are all the same, so maybe just do one? ■

4.9 • LEMMA: *Weakening.*
If Σ and Σ' are store typings with $\Sigma \subseteq \Sigma'$ and $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, then $\Xi \mid \Gamma \mid \Sigma' \vdash e : \tau$.

Proof. This is a straightforward induction on type derivations, in that we notice that in all inference rules, the store typing is the same in the conclusion as it is in the hypotheses. Furthermore, if the axiom T-LOC holds for Σ, then it clearly holds for Σ'. ■

4.10 • LEMMA. *If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau$ for the same type τ, then $\Xi \mid \Gamma \mid \Sigma \vdash K[e] : \rho$ and $\Xi \mid \Gamma \mid \Sigma \vdash K[e'] : \rho$ for the same type ρ.*

Proof. The proof is by induction on K. If $K = -$, then this is obvious.

$K = (K', e'')$: Then $K'[e]$ and $K'[e']$ have the same type, so by T-PAIR, so do $K[e]$ and $K[e']$.

$K = K'\,\_$: Then $K'[e]$ and $K'[e']$ have the same type by induction, and so do $K[e] = K'[e]\,\_$ and $K[e'] = K'[e']\,\_$ by T-TAPP.[22] [TODO need lemma saying that $\tau[\tau'/X]$ is a type!]

TODO rest ■

4.11 • LEMMA: *Preservation for head-steps.*
If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$ and $(\sigma, e) \to_h (\sigma', e')$, then there exists a store typing Σ' such that $\Sigma \subseteq \Sigma'$, $\Xi \mid \Gamma \mid \Sigma' \vdash e' : \tau$, and $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$.

Proof. We simply check all cases. [TODO mention pure cases]

E-FST: In this case $e = \pi_1\,(v_1, v_2)$ and $e' = v_1$ for values v_1, v_2. Then e is a value, so [TODO canonical forms] implies first that $\Xi \mid \Gamma \mid \Sigma \vdash (v_1, v_2) : \tau \times \tau'$ for some type τ', and then that $\Xi \mid \Gamma \mid \Sigma \vdash v_1 : \tau$.

[22] TODO since we just apply it to underscore, it actually has a lot of different types. But we can find one type that works for both.

E-TAPP-TLAM: Write the type of $\Lambda\, e\, \_$ as $\tau[\tau'/X]$. By inversion we have $\Xi \mid \Gamma \mid \Sigma \vdash \Lambda\, e : \forall X.\tau$, and again by inversion this implies that $\Xi, X \mid \Gamma \mid \Sigma \vdash e : \tau$. But then it follows from [TODO lemma 0] that $\Xi \mid \Gamma[\tau'/X] \mid \Sigma \vdash e : \tau[\tau'/X]$, and since Γ does not contain X (since Ξ does not) we have $\Gamma[\tau'/X] = \Gamma$, so the claim follows.

E-ALLOC: In this case $e = \mathrm{ref}\, v$, $e' = l$, $\sigma' = \sigma[l \mapsto v]$, and $l \notin \mathrm{dom}\, \sigma$. By inversion we have $\Xi \mid \Gamma \mid \Sigma \vdash \mathrm{ref}\, v : \mathrm{ref}(\tau')$ for some τ', and we further have $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau'$. Now letting $\Sigma' = \Sigma[l \mapsto \tau']$, it follows from T-LOC that $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathrm{ref}(\Sigma'(l))$, so we have both $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathrm{ref}(\tau')$.

E-STORE: Here $e = l := v$, $e' = ()$ and $\sigma' = \sigma[l \mapsto v]$ with $l \in \mathrm{dom}\, \sigma$. Notice first that $l := v$ and $()$ both have type Unit by inversion, so that $\Xi \mid \Gamma \mid \Sigma \vdash () : \mathrm{Unit}$ as required.

By inversion we also have $\Xi \mid \Gamma \mid \Sigma \vdash l : \mathrm{ref}(\tau')$ and $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau'$ for some type τ', and another application of inversion (TODO via T-LOC, or canonical forms?) implies that $\tau' = \Sigma(l)$. Furthermore, since $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$, we have $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$. Since $v = \sigma'(l)$ it thus follows that $\Xi \mid \Gamma \mid \Sigma \vdash \sigma'(l) : \Sigma(l)$, so $\Xi \mid \Gamma \mid \Sigma \vdash \sigma'$.

E-LOAD: Here $e = !l$, $e' = v$ and $\sigma = \sigma'$ with $\sigma(l) = v$. By inversion we have $\Xi \mid \Gamma \mid \Sigma \vdash l : \mathrm{ref}(\tau)$, so $\tau = \Sigma(l)$ [TODO again]. But since $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$ we have $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$, or in other words, $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau$.

TODO rest ∎

5 ◇ Misc

5.1 • Lambda calculus

The syntax of the untyped lambda calculus consists only of variables, abstractions and applications:

$$\begin{array}{rl} & x \in \mathcal{V} \\ \mathit{Exp} & e ::= x \mid \lambda x.e \mid e\, e \\ \mathit{Val} & v ::= \lambda x.e \end{array}$$

This means that there in particular are expressions on the form $e_1\, e_2\, e_3$, and we use the convention that application is left-associative, i.e. that the above expression is to be read $(e_1\, e_2)\, e_3$. [TODO build into the grammar, Mogensen]

The small-step reduction relation $\to$ on expressions formalises how to reduce expressions. For instance, the rule

$$\frac{}{(\lambda x.e_1)\, e_2 \to e_1[x \mapsto e_2]}\ \text{E-APP-ABS}$$

says that we can apply abstractions to other expressions. Such a rule is called a ***computation rule***, and an expression $(\lambda x.e_1)\,e_2$ is called a ***redex***. Rewriting a redex according to the above rule is called β-***reduction***.

A more complex expression might not itself be a redex but instead have a redex as a subexpression. In this case we need other rules, so-called ***congruence rules***, which tell us how to reduce complex expressions by reducing subexpressions. For instance, in the expression $e_1\,e_2$, do we evaluate e_1 before e_2 or vice versa? That is, does evaluation happen left-to-right or right-to-left, or do we allow this to be determined arbitrarily? This is of course especially important in languages with side-effects. Formally we impose an evaluation order by having either (or both) of the congruence rules

$$\frac{e_1 \to e_1'}{e_1\,e_2 \to e_1'\,e_2}\ \text{E-APP1} \quad \text{and} \quad \frac{e_2 \to e_2'}{e_1\,e_2 \to e_1\,e_2'}\ \text{E-APP2}.$$

If we desire right-to-left evaluation order, then we choose E-APP2, but we also need a restricted form of E-APP1, namely

$$\frac{e \to e'}{e\,v \to e'\,v}\ \text{E-APP1'}.$$

That is, only when the right expression has been reduced to a value v can we evaluate the left expression.

We may also formalise the evaluation order by defining the reduction relation in terms of head reductions $\to_h$, and using evaluation contexts to impose an evaluation order. For instance,

$$K ::= - \mid K\,e \mid v\,K \quad \text{and} \quad K ::= - \mid e\,K \mid K\,v$$

define evaluation contexts for left-to-right and right-to-left evaluation, respectively. The symbol '$-$' is called the ***hole***, and we think of the hole as the place into which we substitute the expression e when writing $K[e]$.

Computation and congruence rules together might also allow for different evaluation strategies, for instance:

(1) Full β-reduction: We may reduce *any* redex contained in an expression.

(2) Normal order reduction: We must reduce the leftmost, outermost redex first.

(3) Call-by-name: The subexpression e_2 of a redex $(\lambda x.e_1)\,e_2$ cannot be reduced. Instead, we must perform β-reduction without reducing e_2.

(4) Call-by-value: Instead, e_2 *must* be reduced to a value before β-reduction can take place.

For instance, in call-by-value we might have a restricted form of the rule E-APP-ABS, namely

$$\frac{}{(\lambda x.e)\, v \to e[x \mapsto v]}\ \text{E-APP-ABS'}.$$

That is, the argument must be a value v for the reduction to take place. If we use evaluation contexts, this computation rule would be a rule concerning head reductions $\to_h$.

Since the untyped lambda calculus does not allow abstractions to be named, it is not obvious how to define recursive functions. TODO

5.2 • Recursion

Call by name: $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$$\begin{aligned} Yg &= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))g \\ &\to (\lambda x.g(xx))(\lambda x.g(xx)) \\ &\to g((\lambda x.g(xx))(\lambda x.g(xx))) \end{aligned}$$

On the other hand we also have

$$\begin{aligned} g(Yg) &= g(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))g) \\ &\to g((\lambda x.g(xx))(\lambda x.g(xx))). \end{aligned}$$

That is, Yg and $g(Yg)$ reduce to the same expression.

Call by value: $Z = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$

$$\begin{aligned} Zg &= \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))g \\ &\to (\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy)) \\ &\to g(\lambda y.(\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy))y) \\ &\to g() \end{aligned}$$

$$\begin{aligned} g(Zg) &= g(\lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))g) \\ &\to g((\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy))) \end{aligned}$$

6 ◇ Logical Relations

Be sure to distinguish between values and irrs! That is the whole point of progress.

But prove: val => irr

[TODO background info on logical predicates/relations]

To each type τ we associate a set $\mathcal{V}[\![\tau]\!]$ of closed values called its ***value interpretation***, which induces a logical predicate $\mathcal{V}[\![-]\!] \colon \mathit{Type} \to \mathcal{P}(\mathit{CVal})$ [TODO define CVal]. We begin by defining the value interpretations of the base types, which for us means just the unit type:

$$\mathcal{V}[\![\mathsf{Unit}]\!] := \{1\}.$$

Given types τ_1 and τ_2, we can also immediately define the value interpretations of the product and sum of τ_1 and τ_2. This is done recursively by letting

$$\begin{aligned}\mathcal{V}[\![\tau_1 \times \tau_2]\!] &:= \{(v_1, v_2) \mid v_1 \in \mathcal{V}[\![\tau_1]\!], v_2 \in \mathcal{V}[\![\tau_1]\!]\},\\ \mathcal{V}[\![\tau_1 + \tau_2]\!] &:= \{\iota_1\, v \mid v \in \mathcal{V}[\![\tau_1]\!]\} \cup \{\iota_2\, v \mid v \in \mathcal{V}[\![\tau_2]\!]\}.\end{aligned}$$

These definitions should seem fairly natural. While we cannot let $\mathcal{V}[\![\tau_1 \times \tau_2]\!]$ be the *Cartesian* product of $\mathcal{V}[\![\tau_1]\!]$ and $\mathcal{V}[\![\tau_2]\!]$ (since 'mathematical' pairs[23] (v_1, v_2) of values are not even expressions in the language, let alone closed values), the obvious solution is to let this set be the pairs in the *object* language. But notice that these are isomorphic as sets: Indeed, $\mathcal{V}[\![\tau_1 \times \tau_2]\!]$ as defined above is a categorical product of $\mathcal{V}[\![\tau_1]\!]$ and $\mathcal{V}[\![\tau_2]\!]$ when equipped with the projections $(v_1, v_2) \mapsto v_i$. On the other hand, disjoint unions are usually only defined up to isomorphism, and we notice that $\mathcal{V}[\![\tau_1 + \tau_2]\!]$ is in fact a disjoint union of the sets $\mathcal{V}[\![\tau_1]\!]$ and $\mathcal{V}[\![\tau_2]\!]$, the operators ι_1 and ι_2 [TODO spacing when no argument] serving as the 'tags' that create isomorphic copies of $\mathcal{V}[\![\tau_1]\!]$ and $\mathcal{V}[\![\tau_2]\!]$. These of course give rise to the injections $v \mapsto \iota_i\, v$.

Before we can define the value interpretation of function types, we need an additional logical predicate: The ***expression interpretation*** of a type τ is the set $\mathcal{E}[\![\tau]\!]$ of expressions e with the property that if e' is an irreducible expression and $e \to^* e'$, then $e' \in \mathcal{V}[\![\tau]\!]$. That is, we get another logical predicate $\mathcal{E}[\![-]\!] \colon \mathit{Type} \to \mathcal{P}(\mathit{Exp})$. We thus define:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] := \{\lambda x.e \mid \forall v \in \mathcal{V}[\![\tau_1]\!] \colon e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}.$$

Notice that we cannot do without the expression interpretation, since the body of a function, even with the argument substituted for the formal parameter, is not generally a value.

Of course, it is not immediate that the sets of expression and value interpretations are well-defined since they are defined mutually recursively. And we cannot simply use the form of mutual recursion described in [TODO], since

[23] Consider using different notation for object pairs to ensure readability in black/white. Also, do I even want punctuation in a different font?

$\mathcal{E}[\![\tau]\!]$ is defined in terms of $\mathcal{V}[\![\tau]\!]$ for the *same* type τ. But notice that the definition of $\mathcal{V}[\![\tau]\!]$ only refers to expression interpretations of types that are *strictly* smaller than τ. This enables us to do the following: Let $\eta : \mathcal{P}(CVal) \to \mathcal{P}(Exp)$ be the map that to a set V of closed values associates the set $\eta(V)$ of expressions e such that if e' is irreducible and $e \to^* e'$, then $e' \in V$. In particular, $\mathcal{E}[\![\tau]\!] = \eta(\mathcal{V}[\![\tau]\!])$.

As in [TODO], we perform the mutual recursion by defining a function $f : Type \times \mathcal{P}(CVal) \times \mathcal{P}(Exp) \to \mathcal{P}(CVal) \times \mathcal{P}(Exp)$. Considering for instance the inference rule $\tau_1, \tau_2 \vdash \tau_1 \to \tau_2$, the corresponding function h we use for recursion [TODO rewrite this] is then

$$h(\tau, (V_1, V_2), (E_1, E_2)) = (V, \eta(V)), \quad \text{where}$$
$$V = \{\lambda x.e \mid \forall v \in V_1 : e[v/x] \in E_2\}.$$

Notice that we do not explicitly use the type τ, nor the sets V_2 and E_1, in this definition. Inference rules for the other types are handled similarly.

[TODO we do recursion for all inference rules for ASTs, not just the ones for types.]

[TODO more types]

[TODO environment int]

6.1 • Lemma. *If $e \in \mathcal{E}[\![\tau]\!]$ is irreducible, then $e \in \mathcal{V}[\![\tau]\!]$.*

Proof. Since $e \to^* e$ and e is irreducible, this follows directly from the definition of $\mathcal{E}[\![\tau]\!]$. ∎

TODO: define gamma(e), envInt, expInt

6.2 • Theorem: *Fundamental property.*
If $\Gamma \vdash e : \tau$, then $\Gamma \vDash e : \tau$.

Proof. TODO ∎

6.3 • Lemma. *If $(e_1, e_2) \to^* e'$, then there are expressions e_1' and e_2' such that $e' = (e_1', e_2')$, and such that $e_i \to^* e_i'$.*

If $\pi_1\, e \to^* e'$, then either $e' = v_1$ is a value and $e \to^* (v_1, v_2)$, or else $e' = \pi_1\, e''$ such that $e \to^* e''$.

If $e_1\, e_2 \to^* e'$, then either $e_1\, e_2 \to^* e''[v/x] \to^* e'$ where $e_1 \to^* \lambda x.e''$ and $e_2 \to^* v$, or else $e' = e_1'\, e_2'$ where $e_i \to^* e_i'$.

If $\iota_1\, e \to^* e'$, then $e' = \iota_1\, e''$ and $e \to^* e''$ for some e''.

If $\mathsf{match}(e; x.e_1; x.e_2) \to^* e'$, then either $\mathsf{match}(e; x.e_1; x.e_2) \to^* e_i[v/x] \to^* e'$ where $e \to^* \pi_i\, v$, or else $\mathsf{match}(e''; x.e_1'; x.e_2')$ where $e \to^* e''$.

Proof. By induction on the length of the reduction $(e_1, e_2) \to^* e'$, it suffices to prove the claim when this is a one-step reduction. There exists an evaluation context K and expressions d and d' such that $(e_1, e_2) = K[d]$ and $e' = K[d']$, and such that $d \to_h d'$. Notice that K must either be on the form (K', e'') or (v, K') for an evaluation context K', an expression e'' and a value v [TODO make connection between K and $K[e]$]. In the former case we have $K[d] = (K'[d], e'')$, so we must have $e'_1 = K'[d]$ and $e'_2 = e''$. We similarly have $e' = (K'[d'], e'')$, and we notice that $K'[d] \to^* K'[d']$ (indeed this happens in one step) and $e'' \to^* e''$ as desired. If instead $K = (v, K')$, then the argument is similar.

The proof is by induction on the length n of the reduction $\pi_1\, e \to^* e'$. For $n = 0$ we have $e' = \pi_1\, e$ and $e \to^* e$, so the claim holds. Assuming that it holds for some n, suppose that $\pi_1\, e \to^n e'_1 \to e'_2$. Then e'_1 cannot be a value since values are irreducible [TODO lemma ref], so $e'_1 = \pi_1\, e''_1$ with $e \to^* e''_1$ by induction. Now, $e'_1 = K[d_1]$ and $e'_2 = K[d_2]$ with $d_1 \to_h d_2$, where K is either the hole or on the form $\pi_1\, K'$. If K is the hole, then $\pi_1\, e''_1 \to_h e'_2$, which is only possible if $e''_1 = (v_1, v_2)$ and $e'_2 = v_1$. In this case we thus indeed have $e \to^* (v_1, v_2)$. If instead $K = \pi_1\, K'$, then $e'_1 = \pi_1\, K'[d_1]$ and $e'_2 = \pi_1\, K'[d_2]$, the first of which implies that $e''_1 = K'[d]$ by the induction hypothesis. But since $d_1 \to_h d_2$ we also have $K'[d_1] \to K'[d_2]$, and so $e \to^* e''_1 = K'[d_1] \to K'[d_2]$ as desired.

By induction on the length n of the reduction $e_1\, e_2 \to^* e'$. If $n = 0$ then the claim is obvious, so assume that is holds for some n and that $e_1\, e_2 \to^n e' \to e'''$. We consider each disjunct in the induction hypothesis: First assume that $e_1\, e_2 \to^* e''[v/x] \to^* e' \to e'''$, where $e_1 \to^* \lambda x.e''$ and $e_2 \to^* v$. Then we have $e''[v/x] \to^* e'''$, proving the claim. Instead assume that $e' = e'_1\, e'_2$ and $e_i \to^* e'_i$. Then $e'_1\, e'_2 \to e'''$, so $e'_1\, e'_2 = K[d]$ and $e''' = K[d']$ with $d \to_h d'$, and K is either the hole or on one of the forms $K'\, e''_2$ and $v_1\, K'$. If K is the hole, then we must have $e'_1 = \lambda x.e''$ and $e'_2 = v_2$, in which case $e''' = e''[v_2/x]$. If instead $K = K'\, e''_2$, then $e'_1\, e'_2 = K'[d]\, e''_2$ and $e''' = K'[d']\, e''_2$, implying that $e'_1 = K'[d] \to K'[d']$ and $e'_2 = e''_2$ as desired. The final case is similar.

Induction on the length of the reduction. If $\iota_1\, e \to e'$, then $\iota_1\, e = K[d]$ and $e' = K[d']$ with $d \to_h d'$. But then we must have $K = \iota_1\, K'$, so $e = K'[d]$ and $e' = \iota_1\, K'[d']$, and hence $e \to K'[d']$.

Induction on the length of the reduction. If $\mathsf{match}(e; x.e_1; x.e_2) \to e'$, then $\mathsf{match}(e; x.e_1; x.e_2) = K[d]$ and $e' = K[d']$ with $d \to_h d'$. Then K is either the hole or on the form $\mathsf{match}(K'; x.e_1; x.e_2)$. In the former case we must have $e = \pi_i\, v$ and $e' = e_i[v/x]$. In the latter case we have $e = K'[d]$, and so $e \to K'[d']$. ∎

6.4 • Lemma: ***Compatibility.***
If $(x : \tau) \in \Gamma$, *then* $\Gamma \vdash x : \tau$.

$\Gamma \vdash 1 : \mathsf{Unit}$

If $\Gamma, x : \tau_1 \vdash e : \tau_2$, *then* $\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2$.

If $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$, *then* $\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$.
If $\Gamma \vdash e : \tau_1 \times \tau_2$, *then* $\Gamma \vdash \pi_1\, e : \tau_1$.
If $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ *and* $\Gamma \vdash e_2 : \tau_1$, *then* $\Gamma \vdash e_1\, e_2 : \tau_2$.
If $\Gamma \vdash e : \tau_1$, *then* $\Gamma \vdash \iota_1\, e : \tau_1 + \tau_2$.
If $\Gamma \vdash e : \tau_1 + \tau_2$, $\Gamma, x : \tau_1 \vdash e_1 : \tau$ *and* $\Gamma, x : \tau_2 \vdash e_2 : \tau$, *then* $\Gamma \vdash \mathsf{match}(e; x.e_1; x.e_2) : \tau$.

Proof. Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Then $x \in \operatorname{dom}\gamma$, so $\gamma(x)$ is a (closed[TODO]) value [TODO by definition of γ], in particular an element in $\mathcal{E}[\![\tau]\!]$.

For every $\gamma \in \mathcal{G}[\![\Gamma]\!]$ we have $\gamma(1) = 1 \in \mathcal{V}[\![\mathsf{Unit}]\!]$.

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and let e' be an irreducible expression such that $\gamma(\lambda x.e) \to^* e'$. Since $\lambda x.\gamma(e)$ is irreducible [TODO proof, values are irr], $e' = \lambda x.\gamma(e) = \lambda x.\gamma(e)$. To show that this lies in $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$, let $v \in \mathcal{V}[\![\tau_1]\!]$ and notice that

$$\gamma(e)[v/x] = \gamma[x \mapsto v](e) \in \mathcal{E}[\![\tau_2]\!]$$

by [TODO subst lemma] and the hypothesis, since $\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma, x : \tau_1]\!]$. Thus $\gamma(\lambda x.e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$ as desired.

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and let e' be an irreducible expression such that $(\gamma(e_1), \gamma(e_2)) = \gamma((e_1, e_2)) \to^* e'$. By [TODO ref lemma] this implies that $e' = (e_1', e_2')$ for appropriate expressions e_1' and e_2', and furthermore that $\gamma(e_1) \to^* e_1'$ and $\gamma(e_2) \to^* e_2'$. The hypothesis then implies that $\gamma(e_i) \in \mathcal{E}[\![\tau_i]\!]$. Notice that e_1' and e_2' are both irreducible since e' is [TODO proof, doesn't hold for all subexps!], so it follows from [TODO ref lemma] that $e_i' \in \mathcal{V}[\![\tau_i]\!]$. But then $(e_1, e_2) \in \mathcal{V}[\![\tau_1 \times \tau_2]\!]$ as desired.

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and let e' be irreducible such that $\pi_1\, \gamma(e) = \gamma(\pi_1\, e) \to^* e'$. We consider each case of [TODO lemma ref]: First assume that $e' = v_1$ is a value and $\gamma(e) \to^* (v_1, v_2)$. Since (v_1, v_2) is a value, and hence irreducible by [TODO lemma], the hypothesis implies that $(v_1, v_2) \in \mathcal{V}[\![\tau_1 \times \tau_2]\!]$. It follows that $v_1 \in \mathcal{V}[\![\tau_1]\!]$ as desired. Next assume that $e' = \pi_1\, e''$ and that $\gamma(e) \to^* e''$. If e'' is reducible then so is $\pi_1\, e''$, so assume that e'' is irreducible. The hypothesis then implies that $e'' \in \mathcal{V}[\![\tau_1 \times \tau_2]\!]$, which means that e'' is on the form (v_1, v_2) with $v_i \in \mathcal{V}[\![\tau_i]\!]$. But then $e' = \pi_1\, (v_1, v_2)$ reduces to v_1, which is a contradiction.[24]

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and let e' be irreducible such that $\gamma(e_1)\, \gamma(e_2) = \gamma(e_1\, e_2) \to^* e'$. We consider each case of [TODO lemma ref]: First assume that $\gamma(e_1)\, \gamma(e_2) \to^* e''[v/x] \to^* e'$ where $\gamma(e_1) \to^* \lambda x.e''$ and $\gamma(e_2) \to^* v$. But each of these reducts [TODO define this] are values and hence irreducible by [TODO ref], so the hypothesis implies that they lie in $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ and $\mathcal{V}[\![\tau_1]\!]$ respectively. But then $e''[v/x] \in \mathcal{E}[\![\tau_2]\!]$, and the hypothesis then implies that $e' \in \mathcal{V}[\![\tau_2]\!]$ as desired. Instead assume that $e' = e_1'\, e_2'$ where $\gamma(e_i) \to^* e_i'$. Since e' is irreducible,

[24] TODO remember that value interpretations are also syntactic values!! they are defined that way.

then so are the e_i' [TODO more thorough], so the hypothesis implies that $e_1' \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$ and $e_2' \in \mathcal{V}[\![\tau_1]\!]$. Hence e_1' is on the form $\lambda x.e_1''$ and e_2' is a value. But then $e_1'\, e_2'$ is reducible, which is a contradiction.

Let $\gamma \in \mathcal{G}[\![\tau_1 + \tau_2]\!]$, and let e' be irreducible such that $\iota_1\, \gamma(e) = \gamma(\iota_1\, e) \to^* e'$. By [TODO lemma] we have $e' = \iota_1\, e''$ with $\gamma(e) \to^* e''$, and since e' is irreducible so is e'' [TODO], and the hypothesis thus implies that $e'' \in \mathcal{V}[\![\tau_1]\!]$. But then $e' \in \mathcal{V}[\![\tau_1 + \tau_2]\!]$ as desired.

Let $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and let e' be irreducible such that $\mathsf{match}(\gamma(e); x.\gamma(e_1); x.\gamma(e_2)) = \gamma(\mathsf{match}(e; x.e_1; x.e_2)) \to^* e'$.[25] We consider each case in [TODO lemma]: Assume first that $\mathsf{match}(\gamma(e); x.\gamma(e_1); x.\gamma(e_2)) \to^* \gamma(e_i)[v/x] \to^* e'$ and $\gamma(e) \to^* \pi_i\, v$ [TODO remember to use the second one below], and notice that $\gamma(e_i)[v/x] = \gamma[x \mapsto v](e_i) \in \mathcal{E}[\![\tau]\!]$. Since this reduces to e' and $\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma, x : \tau_i]\!]$, the hypotheses imply that $e' \in \mathcal{V}[\![\tau]\!]$ as desired. If instead $e' = \mathsf{match}(e''; x.e_1'; x.e_2')$ and $\gamma(e) \to^* e''$, then e'' must also be irreducible since e' is [TODO]. The hypotheses then imply that $e'' \in \mathcal{V}[\![\tau_1 + \tau_2]\!]$, and is thus on the form $\pi_i\, v$. But then e' is reducible, a contradiction. ∎

A ◇ Order Theory

A.1 • Partial orders and lattices

If P is a set, then a ***partial order*** on P is a binary homogeneous relation $\leq$ on P that is reflexive, transitive and anti-symmetric. The pair $(P, \leq)$ is called a ***partially ordered set***, or simply a ***poset***.

If $A \subseteq P$, then an element $x \in P$ is called an ***upper bound*** of A if $a \leq x$ for all $a \in A$. If there is a least upper bound x (i.e., such that if y is any other upper bound, then $x \leq y$), then x is called the ***supremum*** or ***join*** of A. The join of A is clearly unique if it exists (by anti-symmetry), and we denote it by $\bigvee A$. We similarly define the ***infimum*** or ***meet*** of A, denoted $\bigwedge A$, to be the greatest lower bound of A, if it exists. If every two-element subset of P has a join and a meet, then P is called a ***lattice***, and we write $x \vee y := \bigvee\{x, y\}$ and $x \wedge y := \bigwedge\{x, y\}$. If every subset of P has a join and a meet, then P is a ***complete lattice***.

If P is a poset, then a nonempty [TODO Vickers 7.2.1] subset $D \subseteq P$ is said to be ***directed*** if every finite subset of D has an upper bound (but not necessarily a *least* upper bound, i.e. a join). By induction this is equivalent to the property that, for every *pair* of elements $x, y \in D$, there exists a $z \in D$

[25] Note that the domains of γ and Γ are the same, and this is contained in the domain of $\Gamma, x : \tau_i$. We don't need the domains to be the same to write down $\gamma(e_i)$, we just need it when using the semantic typing relation!

with $x \leq z$ and $y \leq z$.[26] We further note that the image of a directed set under a monotone map[27] is also directed [TODO embedding]. If D is a directed set whose join exists, we often write $\bigsqcup D \coloneqq \bigvee D$ instead. If $\bigsqcup D$ exists for every directed subset D of P, then P is called a ***directly complete partial order***, or ***dcpo*** for short. If P also has a least element, usually written $\bot$, then P is called a ***dcppo*** (the extra 'p' is for 'pointed'). Notice that complete lattices are dcppo's.

If P and Q are dcpo's, then a map $f : P \to Q$ is ***continuous***[28] if, for every directed $D \subseteq P$, the image $f[D]$ is directed and

$$f\Big(\bigsqcup D\Big) = \bigsqcup f[D].$$

It is easy to show that continuous maps are monotone (notice that if $x \leq y$, then the set $\{x, y\}$ is directed). If conversely f is monotone, then $f[D]$ is as mentioned also directed, and the inequality '$\geq$' always holds.

A.1 • Example: ***Products of dcpo's.***
Let $(P_i)_{i \in I}$ be a collection dcpo's, and equip the product $P \coloneqq \prod_{i \in I} P_i$ with the product order[29]. We claim that P is also a dcpo. If $D \subseteq P$ is directed, then each $\pi_i[D]$ is also directed and thus has a join x_i. Letting $x = (x_i)_{i \in I}$ it is easy to see that $x = \bigsqcup D$ in P: It is clear that x is an upper bound of D, and furthermore $x \leq y$ if and only if $x_i \leq \pi_i(y)$ for all $y \in P$ and $i \in I$.

In particular,

$$\pi_i\Big(\bigsqcup D\Big) = \pi_i(x) = x_i = \bigsqcup \pi_i[D],$$

so π_i is continuous.[30] ⌟

A.2 • Example: ***Partial functions.***
If X and Y are sets, then we order the set $P(X, Y)$ of partial functions as follows: For $f, g : X \rightharpoonup Y$ we let $f \leq g$ if $\operatorname{dom} f \subseteq \operatorname{dom} g$ and $f(x) = g(x)$ for all $x \in \operatorname{dom} f$. The ***graph*** of a partial function f is the set

$$\mathcal{G}(f) = \{(x, y) \in X \times Y \mid x \in \operatorname{dom} f \text{ and } f(x) = y\}.$$

[26] This is the usual definition of directedness. As an example of why directedness in interesting, recall that a union of a collection of subspaces of a vector space is not usually a subspace itself, but it is if the collection is directed (with respect to inclusion). Similarly for subgroups and other algebraic structures, but note that the same does *not* hold for e.g. topologies or σ-algebras. If we substituted 'countable' for 'finite' in the definition of directedness, σ-algebras would have this property as well, while we for topologies would need 'arbitrary' subsets.

[27] A function $f : P \to Q$ between posets is ***monotone*** if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in P$.

[28] Also called ***Scott-continuous*** after Dana Scott.

[29] The ***product order*** on $P = \prod_{i \in I} P_i$ is defined as follows: For $(x_i)_{i \in I}, (y_i)_{i \in I} \in P$ we say that $(x_i)_{i \in I} \leq (y_i)_{i \in I}$ if $x_i \leq y_i$ for all $i \in I$. The projections $\pi_i : P \to P_i$ are thus monotone, and, as an aside, we mention that P is a categorical product in the category of posets and monotone maps.

[30] It follows that P is a product in the category of dcpo's and continuous maps.

This induces a map $\mathcal{G}\colon P(X,Y) \to \mathcal{P}(X \times Y)$ given by $f \mapsto \mathcal{G}(f)$, which is clearly an order-embedding. If $D \subseteq P(X,Y)$ is a directed set of partial functions, then it is clear that the set $\bigsqcup \mathcal{G}[D] = \bigcup_{d \in D} \mathcal{G}(d)$ is the graph of a partial function. This function is then the join of D in $P(X,Y)$, which is therefore a dcpo, and the map $\mathcal{G}$ is thus continuous. In fact, $P(X,Y)$ is a dcppo since the empty map is the least element.

Denoting the projection maps by $\pi_X\colon X \times Y \to X$ and $\pi_Y\colon X \times Y \to Y$, notice that $\operatorname{dom} f = \pi_X[\mathcal{G}(f)]$ and $\operatorname{ran} f = \pi_Y[\mathcal{G}(f)]$. Since images preserve unions, and joins in power sets are just unions, if $D \subseteq P(X,Y)$ is directed then

$$\operatorname{dom} \bigsqcup D = \pi_X\Big[\mathcal{G}\Big(\bigsqcup D\Big)\Big] = \pi_X\Big[\bigsqcup \mathcal{G}[D]\Big] = \bigsqcup \pi_X\Big[\mathcal{G}[D]\Big] = \bigsqcup \operatorname{dom}[D],$$

and we similarly have $\operatorname{ran} \bigsqcup D = \bigsqcup \operatorname{ran}[D]$. ⌟

Let $F\colon P \to P$ be a monotone map. We think of F as a ***generating function***. An element $x \in P$ is said to be F-***closed*** if $F(x) \leq x$, F-***consistent*** if $x \leq F(x)$, and a ***fixed-point*** of F if $F(x) = x$. If F has a least fixed-point, then this is usually denoted μF. Similarly, the greatest fixed-point, if it exists, is denoted νF.

If P is a poset and $x \in P$, then we write

$$x^{\downarrow} := \{y \in P \mid y \leq x\} \quad \text{and} \quad x^{\uparrow} := \{y \in P \mid x \leq y\}.$$

A.3 • Definition. A poset P is called

(a) *ω-complete*, for short an *ω-cpo*, if every nonempty ω-chain in P has a join,

(b) *chain-complete*, for short a *ccpo*, if every nonempty chain in P has a join, and

(c) *directedly-complete*, for short a *dcpo*, if every ω-chain in P has a join. ▲

A chain-complete poset is also said to be ***inductive*** or ***inductively ordered***. If a poset with one of the above properties also has a least element, then we say that they are ***pointed*** and add an extra 'p' to their abbreviations, so that we get ω-cppo, ccppo and dcppo.

A.2 • Arrows

A.4 • Proposition. *Let P and Q be posets and let $f\colon P \to Q$ be a function. Then the following are equivalent:*

(a) *f is monotone.*

(b) *If $B \subseteq Q$ is upward closed, then $f^{-1}[B]$ is also upward closed.*

(c) *If $B \subseteq Q$ is downward closed, then $f^{-1}[B]$ is also downward closed.*

Proof. First assume that f is monotone, let $B \subseteq Q$ be upward closed, and let $y \in f^{-1}[B]$. If $x \leq y$, then $f(x) \leq f(y)$, and so $f(x) \in B$. But then $x \in f^{-1}[B]$ as required.

Conversely assume that the preimage under f of an upward closed set is upward closed, and let $x, y \in P$ with $x \leq y$. Then $f(y)^{\downarrow}$ is downward closed, and hence so is $f^{-1}[f(y)^{\downarrow}]$. But then $x \in f^{-1}[f(y)^{\downarrow}]$, so $f(x) \in f(y)^{\downarrow}$, implying that $f(x) \leq f(y)$.

Since the complement of an upward closed set is downward closed, and vice-versa, the last two properties are clearly equivalent. ■

A.5 • DEFINITION. Let P be a poset. A subset $U \subseteq P$ is said to be

(a) *inaccessible by joins of ω-chains* if $\bigvee C \in U$ implies $C \cap U \neq \emptyset$ for all ω-chains $C \subseteq P$,

(b) *inaccessible by joins of chains* if $\bigvee C \in U$ implies $C \cap U \neq \emptyset$ for all chains $C \subseteq P$, and

(c) *inaccessible by directed joins* if $\bigsqcup D \in U$ implies $D \cap U \neq \emptyset$ for all directed sets $D \subseteq P$.

[TODO nonempty chains??] ▲

When we write e.g. $\bigvee C$ we implicitly assume that the join in fact exists.

Let $\mathcal{T}$ be the collection of subsets of P that are upward closed and inaccessible by directed joins. Then $\mathcal{T}$ is closed under arbitrary unions, since if $D \subseteq P$ is directed and $\bigsqcup D \in \bigcup_{i \in I} U_i$ with $U_i \in \mathcal{T}$, then $\bigsqcup D \in U_i$ for some i, and hence $\emptyset \neq \bigsqcup D \cap U_i \subseteq \bigsqcup D \cap \bigcup_{i \in I} U_i$. Similarly, if $U, V \in \mathcal{T}$ and $\bigsqcup D \in U \cap V$, then there are elements $x \in D \cap U$ and $y \in D \cap V$. And since D is directed x and y have an upper bound z in D, and since U and V are upward closed we have $z \in U \cap V$. That is, $\mathcal{T}$ is a topology on P.

We similarly find that the collections of subsets of P that are upwards closed and inaccessible by joins of (ω-)chains also constitute topologies on P. At least the latter two topologies are known as the ***Scott topology***, but we use this name for any of the three topologies. To disambiguate, we also call them the ω-, ***chain***, and ***directed Scott topologies*** respectively. If $f : P \to Q$ is a map between posets, then we say that f is ω-, ***chain***, and ***directedly Scott continuous*** if f is continuous when P and Q are equipped with the relevant Scott topology. If f is continuous in any of these ways, then we simply call f ***Scott continuous***.

It is also useful to note that a subset $F \subset P$ is closed in e.g. the directed Scott topology if and only if is downward closed and $\bigsqcup D \in F$ for all directed

subsets $D \subseteq F$. Note that if $x \in P$, then $x^{\downarrow}$ is closed: It is clearly downward closed, and x is an upper bound of every subset $A \subseteq x^{\downarrow}$, so the join of A (if it exists) also lies in $x^{\downarrow}$. This in particular leads to the following:

A.6 • Lemma. *If $f : P \to Q$ is Scott continuous, then f is monotone.*

Proof. Let $x, y \in P$ with $x \leq y$. The set $f(y)^{\downarrow}$ is closed in Q, so its preimage $f^{-1}[f(y)^{\downarrow}]$ is closed in P and is in particular downward closed. Hence it contains x, and so $f(x) \in f(y)^{\downarrow}$, which means that $f(x) \leq f(y)$. ■

In particular, if f is continuous and $B \subseteq Q$ is upward/downward closed, then $f^{-1}[B]$ is also upward/downward closed by [TODO ref].

There is a different characterisation of continuous maps between posets. If $f : P \to Q$ is a map between posets, then we say that f ***preserves existing joins*** if whenever $\bigvee A$ exists in P for a subset $A \subseteq P$, then $\bigvee f[A]$ exists in Q and $f(\bigvee A) = \bigvee f[A]$. We often consider slightly different properties where we require f to preserve certain properties of subsets as well as existing joins of subsets with these properties. For instance, we say that f ***preserves existing directed joins*** if, whenever $D \subseteq P$ is directed and $\bigsqcup D$ exists in P, then $f[D]$ is also directed, $\bigsqcup f[D]$ exists in Q and $f(\bigsqcup D) = \bigsqcup f[D]$. This leads to the following definition:

A.7 • Definition. For posets P and Q, a function $f : P \to Q$ is called

(a) *ω-continuous* if f preserves existing joins of ω-chains.

(b) *chain-continuous* if f preserves existing joins of chains.

(c) *directedly-continuous* if f preserves existing directed joins. ▲

If f has any of the above properties, then we simply call f ***continuous***. Again continuous maps are monotone:

A.8 • Lemma. *If $f : P \to Q$ is continuous, then f is monotone.*

Proof. If $x, y \in P$ with $x \leq y$, then $\{x, y\}$ is a directed set with $\bigsqcup\{x, y\} = y$, and so $f(y) = \bigsqcup\{f(x), f(y)\}$. In particular, $f(x) \leq f(y)$. ■

However, while continuous functions are automatically Scott continuous, the converse does not generally hold. We need to make a further assumption on the codomain of the function in question, namely that is has one of the following properties:

[TODO moved]

A.9 • PROPOSITION. *Let P and Q be posets. If $f : P \to Q$ is ω-, chain- or directedly-continuous, then f is ω-, chain or directedly Scott continuous, respectively.*

If Q is ω-, chain- or directedly-complete, respectively, then the converse also holds.

Proof. We prove the claim in the case where f is directedly-continuous or directedly Scott continuous. The other cases are similar.

First assume that f is directedly-continuous, and let $F \subseteq Q$ be closed in the directed Scott topology. Assume that $D \subseteq f^{-1}[F]$ is directed and that $\bigsqcup D$ exists in P. Then $f[D] \subseteq F$, so

$$f\Big(\bigsqcup D\Big) = \bigsqcup f[D] \subseteq F,$$

which implies that $\bigsqcup D \in f^{-1}[F]$. By [TODO ref], $f^{-1}[F]$ is also closed.

Conversely assume that Q is directedly-complete and that f is directedly Scott continuous. Let $D \subseteq P$ be directed such that $\bigsqcup D$ exists in P. By [TODO ref] f is monotone so $f[D]$ also directed, and hence the join $z \coloneqq \bigsqcup f[D]$ exists in Q. Notice that $D \subseteq f^{-1}[z^{\downarrow}]$, implying that $\bigsqcup D \in f^{-1}[z^{\downarrow}]$ since $f^{-1}[z^{\downarrow}]$ is closed. But then $f(\bigsqcup D) \leq z = \bigsqcup f[D]$, so f is directedly-continuous. ■

A.3 • A survey of fixed-point theorems

A.10 • THEOREM: *Knaster–Tarski's fixed-point theorem.*
If L is a complete lattice and $F : L \to L$ is monotone, then F has a least and a greatest fixed-point, and these are given by

$$\mu F = \bigwedge\{x \in L \mid F(x) \leq x\} \quad \textit{and} \quad \nu F = \bigvee\{x \in L \mid x \leq F(x)\}.$$

In particular, μF is the smallest F-closed element and νF is the greatest F-consistent element in L.

Proof [31]**.** Denote the meet above by α. If x is F-closed, then $\alpha \leq x$, so $F(\alpha) \leq F(x) \leq x$. Taking the meet of x we get $F(\alpha) \leq \alpha$, so α is closed. It follows that $F(F(\alpha)) \leq F(\alpha)$, so $F(\alpha)$ is also closed, and so $\alpha \leq F(\alpha)$. Hence α is a fixed-point. Since every other fixed-point is in particular closed, α is the least fixed-point.■

A.11 • THEOREM: *Kleene's fixed-point theorem I.*
If P is an ω-cppo and $F : P \to P$ is ω-continuous, then F has a least fixed-point given by[32]

$$\mu F = \bigvee_{n \in \mathbb{N}} F^n(\bot).$$

[31] This proof is based on Davey and Priestley (2002, Theorem 2.35).

[32] This theorem is an immediate generalisation of Davey and Priestley (2002, Theorem 8.15) from dcppo's to ω-cppo's.

Proof. First notice that, since F is continuous,

$$F\Big(\bigvee_{n\in\mathbb{N}} F^n(\bot)\Big) = \bigvee_{n\in\mathbb{N}} F^{n+1}(\bot) = \bigvee_{n\in\mathbb{N}^+} F^n(\bot) = \bigvee_{n\in\mathbb{N}} F^n(\bot),$$

where we use that $F^0(\bot) = \bot$. Hence $\bigvee_{n\in\mathbb{N}} F^n(\bot)$ is indeed a fixed-point of F. If β is any fixed-point of F, then $\bot \leq \beta$, and hence $F^n(\bot) \leq F^n(\beta) = \beta$ since F is monotone. Taking the join on the left-hand side yields $\bigvee_{n\in\mathbb{N}} F^n(\bot) \leq \beta$ as desired. ∎

In the case where F is ω-continuous, it is thus easy to show that F has a fixed-point and even give a fairly explicit formula for it. If F is not continuous, we are not so lucky. However, since $\omega = (\mathbb{N}, \leq)$ is an ordinal we may be inspired to extend the recursive definition of $F^n(\bot)$ to ordinal powers $F^\alpha(\bot)$ and attempt to find a fixed-point among these elements. This is indeed possible, but since we cannot be sure that α is countable, we must assume that P is chain-complete. The proof is also significantly more involved, requiring transfinite recursion and induction.

We present a proof based on Davey and Priestley (2002, Exercise 8.19). For a proof of the existence of a least fixed-points using Zermelo [TODO], see Moschovakis (2006, Theorem 7.36)

A.12 • THEOREM: ***Kleene's fixed-point theorem II.***
Let P be a ccppo and let $F\colon P \to P$ be monotone. Let

$$\begin{aligned} F^0(\bot) &= \bot, \\ F^\alpha(\bot) &= F(F^{\alpha-1}(\bot)) && \text{if } \alpha \text{ is a successor,} \\ F^\alpha(\bot) &= \bigvee_{\beta<\alpha} F^\beta(\bot) && \text{if } \alpha \text{ is a limit,} \end{aligned}$$

for all ordinals α. Then $\mu F = F^\alpha(\bot)$ for some ordinal α, and μF is the least F-closed element in P.

Notice that if F is continuous, then [TODO Kleene I] says that $\mu F = F^\omega(\bot)$. Indeed, examining the proof below we see that the definition of $F^\alpha(\bot)$ for *countable* α only requires P to be an ω-cppo.

Proof. We first show that the above definition of $F^\alpha(\bot)$ makes sense. More generally, fix some $x \in P$. We define a binary (definite [TODO define]) operation $\Phi = \Phi_x$ taking as arguments an ordinal α and a function $g\colon \alpha \to P$: We first let $\Phi(g, 0) = x$. If α has an immediate predecessor β, then we let $\Phi(g, \alpha) = F(g(\beta))$. If α is a limit ordinal and g is monotone, then we let $\Phi(g, \alpha) = \bigvee_{\beta<\alpha} g(\beta)$. (Note that we indeed take the join of a chain since α is a chain and g is assumed monotone.) Finally, if α is a limit ordinal but g is not monotone, then we let

$\Phi(g, \alpha)$ be some arbitrary element of P (we will not need to consider such g). By transfinite recursion [TODO ref] there thus exists a unary (definite) operation $\Pi = \Pi_x$ such that

$$\Pi(\alpha) = \Phi(\Pi|_\alpha, \alpha)$$

for all α.

We prove by transfinite induction that $\Pi|_\alpha$ is monotone for all α. For $\alpha = 0$ this is obvious, so assume that $\Pi|_\xi$ is monotone for all $\xi < \alpha$ and let $\beta < \gamma < \alpha$. We consider three cases:

β and γ are successors: Then $\Pi|_\gamma$ is monotone, and since F is also monotone we have

$$\Pi(\beta) = F(\Pi(\beta - 1)) \leq F(\Pi(\gamma - 1)) = \Pi(\gamma).$$

β is a limit, γ is a successor: Since β is the union of all *successor* ordinals smaller than it[33], it suffices to show that $\Pi(\xi) \leq \Pi(\gamma)$ if $\xi < \beta$ is a successor ordinal. But since $\Pi|_\gamma$ is monotone, we similarly to above find that

$$\Pi(\xi) = F(\Pi(\xi - 1)) \leq F(\Pi(\gamma - 1)) = \Pi(\gamma).$$

β is a successor, γ is a limit: Let ξ be a successor ordinal with $\beta \leq \xi < \gamma$. Since $\Pi|_\gamma$ is monotone, we again have

$$\Pi(\beta) = F(\Pi(\beta - 1)) \leq F(\Pi(\xi - 1)) = \Pi(\xi),$$

which implies that

$$\Pi(\beta) \leq \bigvee_{\xi < \gamma} \Pi(\xi) = \Pi(\gamma).$$

Since $\Pi|_\alpha$ is always monotone, we always have $\Pi(\alpha) = \bigvee_{\beta<\alpha} \Pi(\beta)$ when α is a limit ordinal. Writing $F^\alpha(x) \coloneqq \Pi_x(\alpha)$ we thus obtain a map $F^\alpha \colon P \to P$, and we have

$$\begin{aligned}
F^0(x) &= x, \\
F^\alpha(x) &= F(F^{\alpha-1}(x)) \quad \text{if } \alpha \text{ is a successor,} \\
F^\alpha(x) &= \bigvee_{\beta<\alpha} F^\beta(x) \quad \text{if } \alpha \text{ is a limit,}
\end{aligned}$$

for all $x \in P$ and ordinals α.

We next show that $F^\alpha(\bot)$ is a fixed-point of F for some α. By Hartogs' theorem[34] there is some ordinal α such that there is no injection $\alpha \hookrightarrow P$. On the other hand, $\beta \mapsto F^\beta(\bot)$ is a function $\alpha \to P$, so this cannot be injective.

[33] This follows since if $\xi < \beta$, then also $\xi + 1 < \beta$.

[34] TODO

Hence there are distinct ordinals $\beta, \gamma < \alpha$ with $F^\beta(\bot) = F^\gamma(\bot)$. Because α is totally ordered we may assume without loss of generality that $\beta < \gamma$. Since the map $\beta \mapsto F^\beta(\bot)$ is monotone and $\beta + 1 \leq \gamma$, this implies that

$$F^\beta(\bot) \leq F^{\beta+1}(\bot) \leq F^\gamma(\bot),$$

and hence

$$F(F^\beta(\bot)) = F^{\beta+1}(\bot) = F^\beta(\bot).$$

Thus $F^\beta(\bot)$ is a fixed-point of F.

Finally we show that F has a *least* fixed-point. First we show that the map F^α is monotone for all ordinals α, i.e. that if $x, y \in P$ with $x \leq y$, then $F^\alpha(x) \leq F^\alpha(y)$. If $\alpha = 0$, then this is obvious, so assume that it holds for all ordinals $\beta < \alpha$. If α is a successor, then

$$F^\alpha(x) = F(F^{\alpha-1}(x)) \leq F(F^{\alpha-1}(y)) = F^\alpha(y).$$

If instead α is a limit, then

$$F^\beta(x) \leq F^\beta(y) \leq \bigvee_{\beta<\alpha} F^\beta(y) = F^\alpha(y)$$

for all $\beta < \alpha$, and taking the join on the left-hand side we again get $F^\alpha(x) \leq F^\alpha(y)$. Now if $x \in P$ is F-closed, then it is clear by induction that $F^\alpha(x) \leq x$ for any α: If α is a successor, then

$$F^\alpha(x) = F(F^{\alpha-1}(x)) \leq F(x) \leq x,$$

and if α is a limit, then $F^\beta(x) \leq$ for all $\beta < \alpha$, and taking the join on the left-hand side yields $F^\alpha(x) \leq \alpha$. Hence if α is such that $F^\alpha(\bot)$ is a fixed-point, then $\bot \leq x$, and so

$$F^\alpha(\bot) \leq F^\alpha(x) \leq x,$$

so $F^\alpha(\bot)$ is indeed the least fixed-point of F, as well as the least F-closed element in P. ∎

A natural question might be whether we can use this extension of Kleene's fixed-point theorem to obtain inversion-type results for all monotone maps, not just those that are continuous. We might, for instance, allow inference rules to have infinitely many antecedents: If X is the ground[TODO?] set, $S \subseteq X$ and $y \in X$, then we write $R\colon S \vdash y$ to denote that the inference rule R has as antecedents the elements in S, and y as consequent. Given a collection $\mathcal{R}$ of such inference rules, we may again define a function $F\colon \mathcal{P}(X) \to \mathcal{P}(X)$ by letting $F(A)$ consist of those $y \in X$ such that there is a rule $S \vdash y$ in $\mathcal{R}$ with $S \subseteq A$. Clearly F is then monotone, but it is not clear that it is continuous (and indeed it often is not).

We quickly see that there is no general inversion lemma for F: In the continuous case, all elements of ω are successor ordinals, so if $y \in F^n(\emptyset)$ for some $n \in \omega$, then we know that $y \in F(F^{n-1}(\emptyset))$, and hence there is a rule $\boldsymbol{x} \vdash y$ with $\boldsymbol{x} \subseteq F^{n-1}(\emptyset)$. And for successor ordinals in the continuous case we may draw a similar conclusion. But if α is a limit ordinal and $y \in F^\alpha(\emptyset)$, then $y \in \bigcup_{\beta<\alpha} F^\beta(\emptyset)$, which only tells us that $y \in F^\beta(\emptyset)$ for some $\beta < \alpha$. And while we may assume that β is a successor ordinal, this may be the eventual successor of another limit ordinal, and this process may repeat itself infinitely many times. This implies that there is no way to derive y with only *finitely many* applications of the inference rules.

A.13 • Example: *σ-algebras.*
Let X be a set, and let $\mathcal{D}$ be a collection of subsets of X. Consider the inference rules (with potentially infinitely many antecedents)

$$\begin{aligned} &\vdash X, && \\ &\vdash D && \text{for } D \in \mathcal{D}, \\ A &\vdash X \setminus A && \text{for } A \subseteq X, \\ \mathcal{A} &\vdash \bigcup \mathcal{A} && \text{for } \mathcal{A} \subseteq \mathcal{P}_{\omega_1}(X). \end{aligned}$$

The function F generated by these rules then has at its least fixed-point the σ-algebra $\sigma(\mathcal{D})$ generated by $\mathcal{D}$. We claim that $\sigma(\mathcal{D}) = F^{\omega_1}(\emptyset)$. To show the inclusion '$\supseteq$', notice that since ω_1 is the union of all countable ordinals it suffices to show that $F^\alpha(\emptyset) \subseteq \sigma(\mathcal{D})$ for all countable α, which easily follows by transfinite induction. The opposite inclusion follows by rule induction: If $(A_n)_{n\in\mathbb{N}}$ is a sequence in $\sigma(\mathcal{D})$ with $A_n \in F^{\omega_1}(\emptyset)$, then $A_n \in F^{\alpha_n}(\emptyset)$ for some countable α_n. Letting $\alpha = \bigcup_{n\in\mathbb{N}} \alpha_n$, the ordinal α is also countable. Hence $A_n \in F^\alpha(\emptyset)$, and so $\bigcup_{n\in\mathbb{N}} A_n \in F^{\alpha+1}(\emptyset) \subseteq F^{\omega_1}(\emptyset)$.[35]

Of course, this does not show that ω_1 is the smallest ordinal α such that $\sigma(\mathcal{D}) = F^\alpha(\emptyset)$. But notice that if α is countable, then (at least assuming the axiom of countable choice) it follows by induction that $F^\alpha(\emptyset)$ is also countable. And it is easy to show that if a σ-algebra is infinite, then it is uncountable (see e.g. Folland 2007, Exercise 1.3), so it cannot equal $F^\alpha(\emptyset)$. ⌟

A.14 • Example: *Topologies.*
Let X be a set, and let $\mathcal{S}$ be a collection of subsets of X, and consider the

[35] This example is Folland (2007, Proposition 1.23).

inference rules

$$\begin{aligned} &\vdash \emptyset, \\ &\vdash X, \\ &\vdash S && \text{for } S \in \mathcal{S}, \\ U, V &\vdash U \cap V && \text{for } U, V \subseteq X, \\ \mathcal{U} &\vdash \bigcup \mathcal{U} && \text{for } \mathcal{U} \subseteq \mathcal{P}(X). \end{aligned}$$

Let F be the represented generating function, and let $\mathcal{T}$ be its least fixed-point, i.e. the topology generated by $\mathcal{S}$. Then $\mathcal{S}$ is a subbasis for $\mathcal{T}$, so every element $U \in \mathcal{T}$ is a union of finite intersections of elements in $\mathcal{S}$, that is, U is on the form

$$U = \bigcup_{i \in I} (U_{i1} \cap \cdots \cap U_{ij_i})$$

for appropriate $U_{ij} \in \mathcal{S}$. Notice that $\mathcal{S} \subseteq F(\emptyset)$, so intersections of n sets from $\mathcal{S}$ lie in $F^n(\emptyset)$, and hence all finite intersections are contained in $F^\omega(\emptyset)$. Applying F a final time yields the union of these intersections, so $\mathcal{T} = F^{\omega+1}(\emptyset)$. ⌟

A map $f : P \to P$ on a poset P is called ***expansive*** if $x \leq f(x)$ for all $x \in P$.[36]

A.15 • THEOREM: *Zermelo's fixed-point theorem.*
If P is a chain-complete poset and $F : P \to P$ is expansive, then F has a fixed-point.[37]

Proof. Consider instead the lift $P_\perp$, which is chain-complete and also pointed. Extend F ot $P_\perp$ by letting $F(\perp)$ be some element of P (that is, $F(\perp) \neq \perp$). Let P_0 be the smallest F-invariant subset of $P_\perp$ that is also a ccppo.

[36] Some authors also use the adjectives ***extensive***, ***inflationary*** or even ***increasing***.

[37] Davey and Priestley (2002) give this result the name 'CPO Fixpoint Theorem III', and they add the hypothesis that P be a dcppo, but also claim that then F has a *minimal* fixed-point. But this is false: Consider for instance the poset $P = (\omega + 1)^\partial$, i.e., the set $\omega \cup \{\omega\}$ equipped with the ordering $\preceq$ given by

$$\omega \prec \cdots \prec n \prec n-1 \prec \cdots \prec 1 \prec 0.$$

This is clearly a dcppo since every nonempty subset even has a maximum. Define $F : P \to P$ by letting $F(\omega) = 0$ and $F(n) = n$ for $n \in \omega$. Then F is obviously expansive, but every natural number, of which there is no minimal with respect to $\preceq$, is a fixed-point. In the context of the proof below, Davey and Priestley claim that the smallest F-invariant sub-dcppo of P, here $\{\omega, 0\}$, has a greatest element, here 0, and that this is a minimal fixed-point of F. But this is false, since e.g. 1 is also a fixed-point.

Our proof of Zermelo's theorem is, however, based on Davey and Priestley (2002, Exercise 8.20). Note that e.g. Moschovakis (2006, Theorem 7.35) proves our version of the theorem, but using the theory of well-orderings.

The above counterexample is taken from Hansen (2023).

We will call an element $x \in P_0$ a ***roof*** if $y < x$ implies $F(y) \leq x$ for all $y \in P_0$. For a roof x we consider the set

$$Z_x = \{y \in P_0 \mid y \leq x \text{ or } F(x) \leq y\}.$$

We claim that Z_x is an F-invariant ccppo, so let $y \in Z_x$. If $y \leq x$, then either $y = x$ in which case $F(x) \leq F(y)$, or else $y < x$ so that $F(y) \leq x$ since x is a roof. If instead $F(x) \leq y$, then since F is expansive we have $F(x) \leq F(y)$. Hence $F(y) \in Z_x$, so Z_x is F-invariant. Next let $C \subseteq Z_x$ be a chain. If $y \leq x$ for all $y \in x$, then we also have $\bigvee C \subseteq x$. If instead there is some $y \in C$ with $F(x) \leq y$, then we clearly have $F(x) \leq \bigvee C$. Thus Z_x is a ccppo, so by minimality of P_0 we have $P_0 = Z_x$.

Next we claim that every element in P_0 is a roof. Consider the set

$$Z = \{x \in P_0 \mid x \text{ is a roof}\}.$$

We show that Z is an F-invariant ccppo. If x is a roof and $y \in P_0$, then $y \in Z_x$ and so either $y \leq x$ or $F(x) \leq y$. If $y < F(x)$ then we must have $y \leq x \leq F(x)$, so that $F(x)$ is also a roof, and so Z is F-invariant. If $C \subseteq Z$ is a chain and $y < \bigvee C$, then there is some $x \in C$ with $y < x \leq \bigvee C$, so Z is also chain-complete. Again by minimality we have $P_0 = Z$.

Now notice that P_0 is a chain: If $x, y \in P_0$, then x is a roof and so $y \in Z_x$, which implies that either $y \leq x$ or $x \leq F(x) \leq y$. Since $P_\perp$ is chain-complete, P_0 has a greatest element $\top$, which is then also a roof. Since F is expansive and P_0 is F-invariant we have $\top \leq F(\top) \leq \top$, so $\top$ is a fixed-point of F.

Finally notice that since $F(\perp) \neq \perp$, $\perp$ is not a fixed-point of F, and so F has a fixed-point lying in P, proving the original claim. ■

Bibliography

Baader, Franz and Tobias Nipkow (1998). *Term Rewriting and All That*. 1st ed. Cambridge University Press. xii + 301 pp. ISBN: 0-521-45520-0.

Barendregt, H. P. (1984). *The Lambda Calculus. Its Syntax and Semantics*. revised. Elsevier. xv + 621 pp. ISBN: 0-444-86748-1.

Chiswell, Ian (2009). *A Course in Formal Languages, Automata and Groups*. 1st ed. Springer. vii + 157 pp. ISBN: 978-1-84800-939-4. DOI: 10.1007/978-1-84800-940-0.

Curry, Haskell B., Robert Feys and William Craig (1958). *Combinary Logic*. Revised ed. Vol. 1. North-Holland Publishing Company. xvi + 417 pp.

Davey, B. A. and H. A. Priestley (2002). *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press. xii + 298 pp. ISBN: 978-0-521-78451-1.

Folland, Gerald B. (2007). *Real Analysis: Modern Techniques and Their Applications*. 2nd ed. Wiley. xiv + 386 pp. ISBN: 0-471-31716-0.

Goldrei, Derek (1996). *Classic Set Theory*. 1st ed. Chapman & Hall. viii + 502 pp. ISBN: 0-412-60610-0.

Hansen, Danny Nygård (2023). *Minimality in Zermelo's fixed-point theorem*. URL: https://math.stackexchange.com/q/4779743 (visited on 03/10/2023).

Harper, Robert (2016). *Practical Foundations for Programming Languages*. 2nd ed. Cambridge University Press. xviii + 494 pp. ISBN: 978-1-107-15030-0.

Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. 1st ed. Addison–Wesley. x + 418 pp. ISBN: 0-201-02988-X.

Leary, Christopher C. and Lars Kristiansen (2015). *A Friendly Introduction to Mathematical Logic*. 2nd ed. Milne Library. xiv + 365 pp. ISBN: 978-1-942341-07-9.

Mogensen, Torben Ægidius (2017). *Introduction to Compiler Design*. 2nd ed. Springer. xxi + 258 pp. ISBN: 978-3-319-66965-6. DOI: 10.1007/978-3-319-66966-3.

Moschovakis, Yiannis (2006). *Notes on Set Theory*. 2nd ed. Springer. xii + 276 pp. ISBN: 0-387-28722-1.

Nederpelt, Rob and Herman Geuvers (2014). *Type Theory and Formal Proof*. 1st ed. Cambridge University Press. xxv + 436 pp. ISBN: 978-1-107-03650-5.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st ed. The MIT Press. xxi + 623 pp. ISBN: 0-262-16209-1.

Smith, Peter (2023a). *Category Theory I. Notes towards a gentle introduction*. x + 227 pp. URL: https://www.logicmatters.net/categories/ (visited on 26/09/2023).

— (2023b). *Category Theory II. More notes towards a gentle introduction*. vii + 162 pp. URL: https://www.logicmatters.net/categories/ (visited on 26/09/2023).