

Bachelor Project in Computer Science

Danny Nygård Hansen

21st September 2023

1 ◇ PRELIMINARIES

If X and A are sets, then we write $A \subseteq_\kappa X$ if A is a subset of X with cardinality strictly less than κ . We denote the power set of X by $\mathcal{P}(X)$, and we furthermore write $\mathcal{P}_\kappa(X)$ for the collection of sets A with $A \subseteq_\kappa X$. We do not distinguish between the ordinal ω and the cardinal \aleph_0 , so we write $A \subseteq_\omega X$ if A is a finite subset of X , and $\mathcal{P}_\omega(X)$ similarly denotes the set of finite subsets of X . If Y is another set, then we denote the set of partial maps from X to Y by $X \rightarrow Y$. For $f \in X \rightarrow Y$ we let $\text{dom } f$ and $\text{ran } f$ denote the domain and range of f , respectively. We also write $X \rightarrow_\kappa Y$ for the subset of $X \rightarrow Y$ of partial maps whose domain has cardinality strictly less than κ , and elements in $X \rightarrow_\omega Y$ will be called **finite partial maps**. For $x_0 \in X$ and $y_0 \in Y$ we denote by $f[x_0 \mapsto y_0]$ the partial map given by

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

Notice in particular that $\text{dom } f[x_0 \mapsto y_0] = \text{dom } f \cup \{x_0\}$, so that if $f \in X \rightarrow_\omega Y$, then also $f[x_0 \mapsto y_0] \in X \rightarrow_\omega Y$.

If $\Xi \subseteq_\omega TVar$, 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 $\text{dom } \sigma = \text{dom } \Sigma$ and $\Xi \mid \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)$ for all $l \in \text{dom } \sigma$. In this case we write $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$, and if Ξ and Γ are both empty we simply write $\Sigma \vdash \sigma$.

Let X be a set, and let P be a predicate on X (i.e., a unary relation on – or simply a subset of – X). For $x \in X$ we write $P(x)$ to mean $x \in P$. If $f : X^k \rightarrow X$ is a partial function for some $k \in \mathbb{N}$, then we say that P **satisfies** the inference rule R_f if $P(x_1) \wedge \dots \wedge P(x_k)$ implies $P(f(x_1, \dots, x_k))$ for all $(x_1, \dots, x_k) \in \text{dom } f$. We write the inference rule on the form

$$\frac{P(x_1) \quad \dots \quad P(x_k)}{P(f(x_1, \dots, x_k))},$$

where it is implicit that (x_1, \dots, x_k) lies in $\text{dom } f$.

As an example, in the simply typed lambda calculus X is the set of triples (Γ, e, τ) , where Γ is a type context, e an expression and τ a type. The function f might be unary and have as domain the set of triples such that τ is a product type, say $\tau_1 \times \tau_2$, and it might then map $(\Gamma, e, \tau_1 \times \tau_2)$ to $(\Gamma, \text{snd } e, \tau_2)$. Of course, using the conventional notation we would write e.g. the statement $P(\Gamma, e, \tau_1 \times \tau_2)$ as $\Gamma \mid \Sigma \vdash e : \tau_1 \times \tau_2$, and $P(\Gamma, \text{snd } e, \tau_2)$ would be written $\Gamma \mid \Sigma \vdash \text{snd } e : \tau_2$, so the rule R_f would then just be the rule T-SND.

We also wish to require a given predicate to satisfy a collection of axioms. As motivation, notice that the antecedent of the rule T-VAR is not on the same form as the consequent. That is, it is not an inductive rule, and it can thus be stated as an axiom. To wit, if $(e : \tau) \in \Gamma$ then $P(\Gamma, e, \tau)$. Another way to describe this is as the set $\{(\Gamma, e, \tau) \mid (e : \tau) \in \Gamma\}$. Hence we may simply use subsets of the universal set X as axioms. Of course, if \mathcal{A} is a collection of axioms, then $\bigcup \mathcal{A}$ is also an axiom. Hence if given any number [TODO set many?] of axioms, we can always take their union and assume that there is only one axiom.

Given an axiom [TODO or set of axioms?] A and a collection \mathcal{R} of inference rules, we first define a function $F : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ by

$$F(B) = \bigcup_{R \in \mathcal{R}} f_R(B^{\rho(R)}),$$

where f_R is some partial function inducing R , and $\rho(R)$ is the arity of f_R . For $A \subseteq X$ we further let $F_A(B) = F(B) \cup A$ [TODO alternatively consider sublattice of sets $\supseteq A$]. It is clear that a set P is a fixed-point of F if and only if P is closed under application of the inference rules in \mathcal{R} . It is furthermore easy to show that P is a fixed-point of F_A if and only if P is a fixed-point of F which contains A [TODO need that $B \subseteq F(B)$, maybe assume that \mathcal{R} contains the identity/trivial rule?]. Now notice that $\mathcal{P}(X)$ is a complete lattice (with respect to set inclusion), and that F_A is order-preserving. The Knaster–Tarski fixed-point theorem [TODO ref] then implies that F_A has a least fixed-point¹. We say that this fixed-point is the predicate **induced** by \mathcal{R} and A .

We would also like a way to generate the predicate P induced by \mathcal{R} and A . Let $A_0 = A$, and for $n \in \mathbb{N}$ define

$$A_{n+1} = F_{A_n}(A_n) = A_n \cup \bigcup_{R \in \mathcal{R}} f_R(A_n^{\rho(R)}).$$

The sequence $(A_n)_{n \in \mathbb{N}}$ is thus increasing. Letting $P = \bigcup_{n \in \mathbb{N}} A_n$, we then have the following result:

1.1 • PROPOSITION. *P is the smallest predicate on X which is true on all elements in A and which satisfies all inference rules in \mathcal{R} .*

¹ It also has a greatest fixed-point, but this is trivially $\mathcal{P}(X)$ itself and is of no interest to us.

Proof. Clearly P is true on A , so consider an inference rule $R \in \mathcal{R}$ and let $(x_1, \dots, x_k) \in \text{dom } f_R$ be such that $P(x_1), \dots, P(x_k)$ hold, i.e. such that $x_1, \dots, x_k \in P$. We must then show that also $f_R(x_1, \dots, x_k) \in P$. Then there must be indices n_1, \dots, n_k such that $x_i \in A_{n_i}$. Letting $n = \max\{n_1, \dots, n_k\}$ we thus have² $x_1, \dots, x_k \in A_n$. Hence $f_R(x_1, \dots, x_k) \in f_R(A_n^k)$, so by definition of A_{n+1} we have $f_R(x_1, \dots, x_k) \in A_{n+1} \subseteq P$ as desired.

Conversely we show that P is the smallest such predicate. But this is clear since it must be contained in every predicate containing A and satisfying the inference rules. ■

1.2 • COROLLARY. *For every $x \in P$ either $x \in A$ or there is a smallest $n \in \mathbb{N}$ such that $x \in f_R(A_n^{\rho(R)})$ for some $R \in \mathcal{R}$.* ■

Note that while n is uniquely determined, R may not be. We say that an inference rule R satisfying the corollary is the rule ***applied last*** in the derivation of x . The intuition is that we construct a derivation tree for x , and R is the rule at the root of the tree.

We are of course used to these kinds of results in computer science, and we rarely stop to spell out the details. Since P is the smallest predicate on X with the above properties, we obtain a notion of structural induction or rule induction, and this is often sufficient to prove properties of P . But sometimes it is at least much more convenient to appeal to the ‘last inference rule applied’, as we shall see.³

2 ◇ GENERAL STUFF ABOUT LANGUAGES

²This is where the proof fails if the arity of f_R is infinite. For instance, the proof does not go through for σ -algebras – where f_R might map a sequence $(B_n)_{n \in \mathbb{N}}$ of sets to the union $\bigcup_{n \in \mathbb{N}} B_n$ – nor for topologies.

³As mentioned, the proof of TODO does not go through for σ -algebras. The best we can do is apparently the following: If $\mathcal{D} = \mathcal{D}_0$ is a collection of subsets of X , let $\sigma(\mathcal{D})$ denote the smallest σ -algebra on X containing \mathcal{D} . If α is a countable ordinal with an immediate predecessor β , then we define \mathcal{D}_α by transfinite induction to be the set of countable unions of elements of \mathcal{D}_β and complements of such sets, and otherwise we let $\mathcal{D}_\alpha = \bigcup_{\beta < \alpha} \mathcal{D}_\beta$. Then one can show that $\sigma(\mathcal{D}) = \bigcup_{\alpha \in \Omega} \mathcal{D}_\alpha$, where Ω is the set of countable ordinals. Cf. Folland TODO. Note however that we by minimality of $\sigma(\mathcal{D})$ do get a notion of induction, just as for predicates defined by inference rules.

For topologies we do have a simple characterisation of the topology \mathcal{T} generated by a collection of sets \mathcal{S} . In this case \mathcal{S} is a subbasis for \mathcal{T} , i.e., every set in \mathcal{T} is a (generally infinite) union of finite intersections of elements in \mathcal{S} . But notice that infinitely many set operations (finite intersections or arbitrary unions) are still required to obtain a general open set, since we take a finite intersection infinitely many times.

On the other hand, results such as TODO ref are of course standard fare in algebra, where we obtain a similar description of the subalgebra generated by a subset of some algebra. See e.g. Bergman TODO

2.1 • Syntax

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.

We first fix countable sets Var of variables and Loc of locations. The *expressions* of the language will be a set Exp containing both Var and Loc , and we designate some of these expressions to be *values*, collected in a set Val . We think of an expression as specifying (part of) the state of the 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*⁴, which is an element of $Loc \rightarrow_{\omega} Exp$. We simply write Sto for this set of maps.

For $e \in Exp$ we define the set $FV(e) \subseteq Var$ 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*.

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 of the language might be $e_1 e_2 e_3$, but this might have two different derivations corresponding to $(e_1 e_2) e_3$ and $e_1 (e_2 e_3)$, and to disambiguate the expression $e_1 e_2 e_3$ we must TODO [TODO Mogensen].

2.2 • Type system

We next 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. It also includes reference types, which are the types of locations. For $\tau \in Type$ we define the set of *free type variables* $FTV(\tau) \subseteq TVar$. As with free variables, these can be bound in various ways. 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.

⁴Sometimes called a *heap*, but this has nothing to do with the heap *data structure*.

If an expression e has free variables, then to specify the type of e it is necessary to first specify the types of the free variables in e . This is done using a **type context**, which is an element of $Var \rightarrow_{\omega} Type$. If e is supposed to be well-typed in a type context Γ , then we of course require that $FV(e) \subseteq \text{dom } \Gamma$. That is, Γ must in fact specify the types of the variables that occur free in e .

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 will also depend on the store in question. However, it is not in general possible to deduce the type of e in this case: If σ is a store, $l_1, l_2 \in \text{dom } \sigma$, 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 the type of e will instead depend on a given **store typing**, which is an element of $Loc \rightarrow_{\omega} Type$. 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.

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

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

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

The semantics of the type system is captured by a five-place relation on the set

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

where an element $(\Xi, \Gamma, \Sigma, e, \tau)$ of this relation is written $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and is called a **type derivation**. This relation is usually defined recursively, by specifying a series of axioms and inference rules⁵. If $\Xi = \emptyset$, then we write $\Gamma \mid \Sigma \vdash e : \tau$ [TODO do we need this?], and we furthermore write $\Sigma \vdash e : \tau$ if also $\Gamma = \emptyset$.

⁵Or more usually axiom and rule *schemas*. TODO Pierce

2.3 • 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 \rightarrow_p on Exp , such that $e \rightarrow_p e'$ is supposed to mean that e evaluates to or reduces to e' .

Going one level up we define the relation \rightarrow_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 $e \rightarrow_p e'$, then we have $(\sigma, e) \rightarrow_h (\sigma, e')$, 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 $Exp \rightarrow Exp$. 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 \rightarrow on $Sto \times Exp$ by letting $(\sigma, K[e]) \rightarrow (\sigma', K[e'])$ if $(\sigma, e) \rightarrow_h (\sigma', e')$.

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

2.4 • Lambda calculus

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

$$\begin{aligned} x &\in Var \\ Exp \quad e &::= x \mid \lambda x. e \mid e e \\ Val \quad v &::= \lambda x. e \end{aligned}$$

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 \rightarrow on expressions formalises how to reduce expressions. For instance, the rule

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

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

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \text{ E-APP1} \quad \text{and} \quad \frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2} \text{ E-APP2}.$$

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

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

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 \rightarrow_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

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

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 \rightarrow_h .

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

2.5 • Recursion

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

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

On the other hand we also have

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

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

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

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

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

3 ◇ SYSTEM F

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

$$\begin{aligned}
& x \in \text{Var} \\
& l \in \text{Loc} \\
& X \in \text{TVar} \\
\text{Exp} \quad e &::= () \mid x \mid l \mid (e, e) \mid \dots \\
\text{Val} \quad v &::= () \mid l \mid (v, v) \mid \text{inj}_1 v \mid \dots \\
\text{Type} \quad \tau &::= \text{Unit} \mid X \mid \text{ref}(\tau) \mid \tau \times \tau \mid \dots \\
\text{ECtx} \quad K &::= - \mid (K, e) \mid (v, K) \mid \dots
\end{aligned}$$

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

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

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 $\text{Exp} \rightarrow \text{Exp}$ 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:

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

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

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

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

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

satisfying the following inference rules:

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

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

Lemma: If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$, then $FTV(\Gamma) \subseteq \Xi$ and $FV(e) \subseteq \text{dom } \Gamma$. 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?

3.1 • Lemmas

3.1 • 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 $\tau = \text{Unit}$.
- (c) If $e = (e_1, e_2)$, then $\tau = \tau_1 \times \tau_2$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$.
- (d) If $e = \text{fst } e'$, then $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau \times \tau_2$.
- (e) If $e = \text{snd } e'$, then $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1 \times \tau$.
- (f) If $e = \text{inj}_1 e'$, then $\tau = \tau_1 + \tau_2$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1$.
- (g) If $e = \text{inj}_2 e'$, then $\tau = \tau_1 + \tau_2$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_2$.
- (h) If $e = \text{match } e_1 \text{ with } \text{inj}_1 x \Rightarrow e_2 \mid \text{inj}_2 x \Rightarrow e_3 \text{ end}$, 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 $e = \text{rec } f(x) := e'$, then $\tau = \tau_1 \rightarrow \tau_2$ and $\Xi \mid \Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \mid \Sigma \vdash e' : \tau_2$.
- (j) If $e = e_1 e_2$, then $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \rightarrow \tau$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau_1$.
- (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 $e = \text{pack } e'$, then $\tau = \exists X. \tau'$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'[\tau''/X]$.

- (n) If $e = \text{unpack } e_1 \text{ as } x \text{ in } e_2$, 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 $l \in \text{dom } \Sigma$ and $\tau = \text{ref}(\Sigma(l))$.
- (r) If $e = \text{ref } e'$, then $\tau = \text{ref}(\tau')$ and $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau'$.
- (s) If $e = e_1 := e_2$, then $\tau = \text{Unit}$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_1 : \text{ref}(\tau')$ and $\Xi \mid \Gamma \mid \Sigma \vdash e_2 : \tau'$.
- (t) If $e = !e'$, then $\Xi \mid \Gamma \mid \Sigma \vdash e' : \text{ref}(\tau)$.

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.

3.2 • LEMMA: Canonical forms.

Assume that $\Xi \mid \Gamma \mid \Sigma \vdash v : \tau$ where v is a value.

- (a) If $\tau = \text{Unit}$, 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 $v = \text{inj}_1 v'$ or $v = \text{inj}_2 v'$.
- (d) If $\tau = \tau_1 \rightarrow \tau_2$, then $v = \text{rec } f(x) := e$.
- (e) If $\tau = \forall X. \tau'$, then $v = \Lambda e$.
- (f) If $\tau = \exists X. \tau'$, then $v = \text{pack } e$.
- (g) *TODO fold*
- (h) If $\tau = \text{ref}(\tau')$, 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 3.1(f) implies that a value $\text{inj}_1 v'$ 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

3.3 • 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 3.1(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 $\Xi, X \mid \Gamma \mid \Sigma \vdash \text{rec } f(x) := e : \tau_1 \rightarrow \tau_2$. By the inversion lemma [TODO ref – also can't we just do induction??] we have $\Xi, X \mid \Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \mid \Sigma \vdash e : \tau_2$, so by induction it follows that $\Xi \mid \Gamma[\tau'/X], f : \tau_1[\tau'/X] \rightarrow \tau_2[\tau'/X], x : \tau_1[\tau'/X] \mid \Sigma \vdash e : \tau_2[\tau'/X]$ [TODO by substitution on envs, function types etc.]. Applying T-REC we obtain the desired claim. ■

TODO rest

3.2 • Progress

3.4 • 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 $(\sigma, e) \rightarrow (\sigma', e')$.

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'.⁶ 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.

⁶This 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.

T-VAR: Since we may assume that Γ is empty, this implication is vacuously true.⁷

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 $e_2 \rightarrow e'_2$. Since the only rule that generates instances of the one-step relation \rightarrow 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 $d_2 \rightarrow_h d'_2$. 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

$$(v_1, e_2) = K'[d_2] \rightarrow K'[d'_2] = (v_1, e'_2)$$

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 3.2(b), where v_1 and v_2 are values. Hence $\text{fst } e = \text{fst}(v_1, v_2)$, and this reduces via a head-step to v_1 . Choosing the evaluation context $K = -$, HEAD-STEP-STEP implies that $\text{fst}(v_1, v_2) \rightarrow v_1$. If instead e is not a value, then by induction there is some e' such that $e \rightarrow e'$. Hence there are subexpressions d and d' and an evaluation context K such that $e = K[d]$, $e' = K[d']$ and $d \rightarrow_h d'$. Letting $K' = \text{fst } K$ we have

$$\text{fst } e = K'[d] \rightarrow K'[d'] = \text{fst } e',$$

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 $\text{inj}_1 v$. If instead $e \rightarrow e'$, then as before $e = K[d]$, $e' = K[d']$ and $d \rightarrow_h d'$. Letting $K' = \text{inj}_1 K$ we get $\text{inj}_1 e = K'[d]$ and $\text{inj}_1 e' = K'[d']$, so $\text{inj}_1 e \rightarrow \text{inj}_1 e'$.

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 3.2(c) it must be on the form $\text{inj}_1 v$ or $\text{inj}_2 v$ for a value v . Hence the expression $\text{match } e_1 \text{ with } \text{inj}_1 x \Rightarrow e_2 \mid \text{inj}_2 x \Rightarrow e_3 \text{ end}$ 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 $e_1 \rightarrow e'_1$, then by the same argument as in previous cases with $K' = \text{match } K \text{ with } \text{inj}_1 x \Rightarrow e_2 \mid \text{inj}_2 x \Rightarrow e_3 \text{ end}$, it follows that $\text{match } e_1 \text{ with } \text{inj}_1 x \Rightarrow e_2 \mid \text{inj}_2 x \Rightarrow e_3 \text{ end}$ reduces.

⁷In 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.

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

T-APP: Assume that the claim holds for $\Sigma \vdash e_1 : \tau_1 \rightarrow \tau_2$ and $\Sigma \vdash e_2 : \tau_1$. If e_1 is a value, then by Lemma 3.2(d) it must be on the form $\text{rec } f(x) := e$. 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 $e_2 \rightarrow e'_2$. 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 $e_1 \rightarrow e'_1$, and choosing $K' = K e_2$ proves the claim.

T-TLAM: This is obvious since Δe 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 3.2(e) it must be on the form $\Delta e'$, so the claim follows from E-TAPP-TLAM (via HEAD-STEP-STEP using the evaluation context $K = -$). If e is not a value, then $e \rightarrow e'$ for some expression e' by induction. These expressions then have subexpressions d and d' respectively such that $d \rightarrow_h d'$, 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 $\text{pack } e$. Otherwise $e \rightarrow e'$ for some expression e' . The same argument as before using the evaluation context $\text{pack } K$ 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 $\text{pack } v$ by Lemma 3.2(f), so an application of E-UNPACK-PACK yields the claim. Otherwise $e \rightarrow e'$ for some e' , and we use the evaluation context $\text{unpack } K \text{ as } x \text{ in } e_2$ 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 $l \notin \text{dom } \sigma$. Otherwise there is an expression e' and a store σ' such that $(\sigma, e) \rightarrow (\sigma', e')$. 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 $(\sigma, d) \rightarrow_h (\sigma', d')$. Letting $K' = \text{ref } K$ we have $\text{ref } e = K'[d]$ and $\text{ref } e' = K'[d']$, so HEAD-STEP-STEP implies that $(\sigma, \text{ref } e) \rightarrow (\sigma', \text{ref } e')$.

T-STORE: Assume that the claim holds for $\Sigma \vdash e_1 : \text{ref}(\tau)$ 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 3.2(h) it is a location l , and by Lemma 3.1(q) we have $l \in \text{dom } \Sigma = \text{dom } \sigma$. 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 $(\sigma, e_2) \rightarrow (\sigma', e'_2)$. Again writing $e_2 = K[d_2]$ and

$e'_2 = K[d'_2]$ with $(\sigma, d) \rightarrow_h (\sigma', d')$, 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 $\Sigma \vdash e : \text{ref}(\tau)$, and let σ be a store with $\Sigma \vdash \sigma$. If e is a value, then as before it is a location l , and $l \in \text{dom } \sigma$. It then follows from E-LOAD that $(\sigma, !l) \rightarrow_h (\sigma, v)$, where $v = \sigma(l)$. If instead $(\sigma, e) \rightarrow (\sigma', e')$ for an expression e' and a store σ' , then we simply use the evaluation context $!K$ for an appropriate K . ■

3.3 • Preservation

3.5 • THEOREM: *Preservation*.

If

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

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

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

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 $(\sigma, d) \rightarrow_h (\sigma', d')$. By Lemma 3.6 there is some type ρ such that $\Xi \mid \Gamma \mid \Sigma \vdash d : \rho$. Next it follows from Lemma 3.9 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 3.7 we also have $\Xi \mid \Gamma \mid \Sigma' \vdash d : \rho$, so it follows from Lemma 3.8 that $\Xi \mid \Gamma \mid \Sigma' \vdash K[d'] : \tau$ as desired. ■

3.6 • 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 3.1(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.

$K = \text{fst } K'$: Then $K[e] = \text{fst } K'[e]$, so Lemma 3.1(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 .

$K \in \{\text{snd } K', \text{inj}_1 K', \text{inj}_2 K'\}$: Similar to the above.

$K = \text{match } K' \text{ with } \text{inj}_1 x \Rightarrow e_1 \mid \text{inj}_2 x \Rightarrow e_2 \text{ end}$: Here Lemma 3.1(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 3.1(j) implies that $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau_1 \rightarrow \tau$ for some type τ_1 . The claim follows by induction as before.

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

$K = K' _$: Lemma 3.1(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

$K = \text{ref } K'$: Then $K[e] = \text{ref } K'[e]$, 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 $\Xi \mid \Gamma \mid \Sigma \vdash K'[e] : \tau' \dots$ TODO

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

3.7 • 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' . ■

3.8 • 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.⁸ [TODO need lemma saying that $\tau[\tau'/X]$ is a type!]

TODO rest ■

3.9 • LEMMA: *Preservation for head-steps*.

If $\Xi \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\Xi \mid \Gamma \mid \Sigma \vdash \sigma$ and $(\sigma, e) \rightarrow_h (\sigma', e')$, then there exists a store typing Σ' such that $\Sigma \subseteq \Sigma'$, $\Xi \mid \Gamma \mid \Sigma' \vdash e' : \tau$, and $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$.

⁸TODO 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.

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

E-FST: In this case $e = \text{fst}(v_1, v_2)$ 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 $\Xi \mid \Gamma \mid \Sigma \vdash v_1 : \tau$.

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 $e = \text{ref } v$, $e' = l$, $\sigma' = \sigma[l \mapsto v]$, and $l \notin \text{dom } \sigma$. By inversion we have $\Xi \mid \Gamma \mid \Sigma \vdash \text{ref } v : \text{ref}(\tau')$ 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 $\Xi \mid \Gamma \mid \Sigma' \vdash l : \text{ref}(\Sigma'(l))$, so we have both $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash l : \text{ref}(\tau')$.

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

By inversion we also have $\Xi \mid \Gamma \mid \Sigma \vdash l : \text{ref}(\tau')$ 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 $\Xi \mid \Gamma \mid \Sigma \vdash l : \text{ref}(\tau)$, 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

■

4 ◇ FIXED-POINTS AND INDUCTION

Overview: generating functions, in terms of inference rules

complete lattice -> Knaster-Tarski -> induction -> rule induction

dcp(p)o -> Kleene and continuity -> finite derivations -> inversion

Note for readers not interested in details.

4.1 • Abstract theory

(a) Let (P, \leq) be a partially ordered set⁹. 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

⁹That is, \leq is a reflexive, transitive and anti-symmetric binary relation on P .

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 is clearly unique if it exists (by anti-symmetry), and we write $\bigvee A$ for the join of 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 subset $D \subseteq P$ is said to be **directed** if, for every pair of elements $x, y \in D$, there exists a $z \in D$ with $x \leq z$ and $y \leq z$. That is, D is directed if every two-element subset of D has an upper bound in D (but not necessarily a *least* upper bound, i.e. a join). By induction any *finite* subset of D also has an upper bound in D .¹⁰ We further note that the image of a directed set under a monotone map is also directed. If D is a directed set for which the join $\bigvee D$ exists, we often write $\bigsqcup D$ 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 \perp , then P is called a **dcppo** (the extra ‘p’ is for ‘pointed’). Notice that complete lattices are dcpo’s.

Let $F: P \rightarrow P$ be a monotone¹¹ 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) We begin by studying dcpo’s. If P and Q are dcpo’s, then a map $f: P \rightarrow Q$ is **continuous**¹² if, for every directed $D \subseteq P$, the image $f[D]$ is directed and

$$f\left(\bigsqcup D\right) = \bigsqcup f[D].$$

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.

Now let P be a dcpo and $F: P \rightarrow P$ a monotone function. We clearly have $\perp \leq F(\perp)$, and since F is monotone we get the chain¹³

$$\perp \leq F(\perp) \leq \dots \leq F^n(\perp) \leq F^{n+1}(\perp) \leq \dots$$

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

¹⁰As an example of why directedness is 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.

¹¹A map $f: P \rightarrow Q$ between posets is **monotone** if $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in P$

¹²Also called **Scott-continuous** after Dana Scott.

¹³A **chain** is a totally ordered set.

4.1 • THEOREM: Kleene's fixed-point theorem.

If P is a dcppo and $F: P \rightarrow P$ is continuous, then F has a least fixed-point μF and

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

Proof. First notice that, since F is continuous,

$$F\left(\bigsqcup_{n \in \mathbb{N}} F^n(\perp)\right) = \bigsqcup_{n \in \mathbb{N}} F^{n+1}(\perp) = \bigsqcup_{n \in \mathbb{N}^+} F^n(\perp) = \bigsqcup_{n \in \mathbb{N}} F^n(\perp),$$

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

(c) Next, let L be a complete lattice, and let $F: L \rightarrow L$ be a monotone map. Even though L is also a dcppo, if F is not continuous then Theorem 4.1 does not apply. However, F still has fixpoints, as the following theorem shows:

4.2 • THEOREM: Knaster–Tarski's fixed-point theorem.

If L is a complete lattice and $F: L \rightarrow L$ is monotone, then F has a least and a greatest fixed-point, and these are given by

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

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

Proof. 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. ■

4.3 • COROLLARY: Principle of induction.

If $y \in L$ is F -closed, then $\mu F \leq y$.¹⁴ ■

4.2 • 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

$$\text{RULE} \frac{x_1 \quad x_2 \quad \cdots \quad x_k}{y}$$

¹⁴We dually have a **principle of coinduction**: If $y \in L$ is F -consistent, then $y \leq \nu F$.

where x_1, \dots, x_k and y are elements of X , and $k \in \mathbb{N}$.¹⁵ 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, \dots, x_k the **antecedents** of the rule and y the **consequent**. Given a (possibly infinite) collection of inference rules, we construct a generating function $F: \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ 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 an inference rule whose consequent is y and whose antecedents lie in A . We say that F is **represented by** this collection of inference rules.

Clearly F is monotone if it is represented by a collection of inference rules, so it has a least fixed-point μF and we get a principle of induction. However, 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 $R \in \mathcal{R}$ with antecedents lying in A , the consequent of R also lies in A .

4.4 • 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 from \mathcal{R} with antecedents $x_1, \dots, x_k \in A$ and consequent y . 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 in \mathcal{R} with consequent y and antecedents $x_1, \dots, x_k \in A$. But since A is \mathcal{R} -closed, this implies that $y \in A$, and so $F(A) \subseteq A$. ■

4.5 • THEOREM: Principle of rule induction.

If F is represented by \mathcal{R} and $A \subseteq X$, then to show that $\mu F \subseteq A$, it suffices to show the following condition: For every inference rule $R \in \mathcal{R}$ with antecedents lying in A , the consequent of R also lies in A .

Proof. The condition says precisely that A is \mathcal{R} -closed, which implies that A is F -closed by Lemma 4.4. But then Corollary 4.3 implies that $\mu F \subseteq A$. ■

(b) [TODO rewrite] But sometimes it is necessary (or at least useful) to have an alternative characterisation of μF , one that makes use of the fact that the number of antecedents of an inference rule is finite. This has the following consequence:

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

¹⁵We could equivalently view an inference rule as an element of the product $X^k \times X$.

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 in \mathcal{R} with antecedents $x_1, \dots, x_k \in \bigsqcup \mathcal{D}$ and consequent y . Since the (directed) join in $\mathcal{P}(X)$ is just union, there are sets $A_1, \dots, A_k \in \mathcal{D}$ 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 4.1 implies that $\mu F = \bigsqcup_{n \in \mathbb{N}} F^n(\emptyset)$.

We now make this more concrete. If \mathcal{R} is a collection of inference rules, then a **derivation** from \mathcal{R} is a finite sequence $D = (R_1, \dots, R_n)$ 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, \dots, R_{i-1} has x as its antecedent. 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 $\text{con } D$, and the set of assumptions of D is denoted $\text{asm } D$. If $\text{asm } D = \emptyset$, then we say that D is **closed**.

4.7 • REMARK. Let $D = (R_1, \dots, R_n)$ be a derivation.

- (a) If D is closed, then R_1 must be an axiom.
- (b) For any $i \in \{1, \dots, n\}$, $D' = (R_1, \dots, R_i)$ is called a **subderivation** of D . If $i < n$, then D' is called a **strict subderivation** of D (so D' is a strict subderivation when $D' \neq D$). It is clear that $\text{asm } D' \subseteq \text{asm } D$ and $\text{con } D' \subseteq \text{asm } D$, 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 strict subderivation D' whose final consequence is x : For x must be the consequent of some R_1, \dots, R_{i-1} , say R_j , and (R_1, \dots, R_j) is closed subderivation of D by (b).
- (d) If $E = (S_1, \dots, S_m)$ is another derivation, then

$$D \circ E := (R_1, \dots, R_n, S_1, \dots, S_m)$$

is another derivation. It is clear that

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

that

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

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

- (e) The **length** of a derivation $D = (R_1, \dots, R_n)$ 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 $l(D \circ E) = l(D) + l(E)$. \lrcorner

4.8 • PROPOSITION. *If F is represented by \mathcal{R} and $y \in X$, then $y \in \mu F$ if and only if y is the final conclusion of some closed derivation from \mathcal{R} .*

Proof. 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 with consequent y and antecedents x_1, \dots, x_k that lie in $F^n(\emptyset)$. By induction each x_i is the final conclusion of some derivation D_i , and by Remark 4.7(d) the derivation $D_1 \circ \dots \circ D_k \circ (R)$ 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 $D = (R_1, \dots, R_{n+1})$. If x is an antecedent of R_{n+1} , then by Remark 4.7(c) D must have a strict subderivation D' whose final conclusion is x . By Remark 4.7(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. \blacksquare

It is of course standard to use derivation *trees*, but all that matters is that there is a finite way to obtain elements in μF . 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.¹⁶ The main consequence we shall use is the following:

4.9 • COROLLARY: Inversion.

If F is represented by \mathcal{R} and $y \in \mu F$, then there is a rule R in \mathcal{R} with y as consequent whose antecedents also lie in μF .

Proof. By Proposition 4.8 there is a closed derivation D of y from \mathcal{R} , and we denote its last rule by R . If x is an antecedent of R , then some subderivation of D is a derivation of x by Remark 4.7(b). Another application of Proposition 4.8 then implies that $x \in \mu F$. \blacksquare

¹⁶Compare the role of deductive calculi in first-order logic, where e.g. the compactness theorem can be obtained from the *existence* of a deductive calculus (with finite derivations).