Bachelor Project in Computer Science

Danny Nygård Hansen

22nd September 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 $A \subseteq_\kappa X$. We do not distinguish between the ordinal ω and the cardinal $\aleph_0$, so $\mathcal{P}_\omega(X)$ 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 $X \rightharpoonup Y$. For $f \in X \rightharpoonup Y$ we let $\operatorname{dom} f$ and $\operatorname{ran} f$ denote the domain and range of f, respectively. We also write $X \rightharpoonup_\kappa Y$ for the partial functions $f \in X \rightharpoonup Y$ 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 $f \in X \rightharpoonup_\omega Y$, then also $f[x_0 \mapsto y_0] \in X \rightharpoonup_\omega Y$.

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

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.

For readers not interested in the technical details, we note that the main results that will be used in the sequel are Theorem 2.5 and Corollary 2.9.

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 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 set under a monotone map is also directed. 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.

Let $F \colon P \to P$ be a monotone[3] 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.

[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 directness, σ-algebras would have this property as well, while we for topologies would need 'arbitrary' subsets.

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

(b) We begin by studying dcpo'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.

Now 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[5]

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

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

2.1 • 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. 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. ∎

(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 2.1 does not apply. However, F still has fixpoints, as the following theorem shows:

2.2 • 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.

[4] Also called ***Scott-continuous*** after Dana Scott.
[5] A ***chain*** is a totally ordered set.

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.■

2.3 • 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

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

where $x_1, \ldots, x_k$ and y are elements of X, and $k \in \mathbb{N}$.[6] 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***. Given a (possibly infinite) collection 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 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.

2.4 • 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 $x_1, \ldots, x_k \in A$ 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.

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

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 $x_1, \ldots, x_k \in A$. But since A is $\mathcal{R}$-closed, this implies that $y \in A$, and so $F(A) \subseteq A$. ∎

2.5 • 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 2.4. But then Corollary 2.3 implies that $\mu F \subseteq A$. ∎

(b) 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 property is captured by F in the following way:

2.6 • 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 in $\mathcal{R}$ with antecedents $x_1, \ldots, x_k \in \bigsqcup \mathcal{D}$ and consequent y. Since the (directed) join in $\mathcal{P}(X)$ is just union, there are sets $A_1, \ldots, A_k \in \mathcal{D}$ 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 2.1 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 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 $\operatorname{con} D$, and the set of assumptions of D is denoted $\operatorname{asm} D$. If $\operatorname{asm} D = \emptyset$, then we say that D is ***closed***.

2.7 • 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 $i \in \{1,\dots,n\}$, $D' = (R_1,\dots,R_i)$ 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 $R_1,\dots,R_{i-1}$, 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

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

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 $D = (R_1,\dots,R_n)$ 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 $l(D \circ E) = l(D) + l(E)$. ⌟

2.8 • 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 2.7(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 $D = (R_1,\dots,R_{n+1})$. If x is an antecedent of R_{n+1}, then by Remark 2.7(c) D must have a proper subderivation D' whose final conclusion is x. By Remark 2.7(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. ■

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.[7] The main consequence we shall use is the following:

2.9 • 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 2.8 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 2.7(b). Another application of Proposition 2.8 then implies that $x \in \mu F$. ■

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 *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 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***[8], which is an element of $Loc \rightharpoonup_\omega Exp$. We simply write *Sto* for this set of partial 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***. 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

[7] Compare 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).

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

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. 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 $\mathit{Var} \rightharpoonup_\omega \mathit{Type}$. 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 $Loc \rightharpoonup_\omega Type$ 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

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

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:

$$
\begin{array}{ll}
 & x \in \mathit{Var} \\
 & l \in \mathit{Loc} \\
 & X \in \mathit{TVar} \\
\mathit{Exp} & e ::= () \mid x \mid l \mid (e,e) \mid \cdots \\
\mathit{Val} & v ::= () \mid l \mid (v,v) \mid \mathsf{inj}_1\, v \mid \cdots \\
\mathit{Type} & \tau ::= \mathsf{Unit} \mid X \mid \mathsf{ref}(\tau) \mid \tau \times \tau \mid \cdots \\
\mathit{ECtx} & K ::= - \mid (K,e) \mid (v,K) \mid \cdots
\end{array}
$$

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[9] on $V \cup \Sigma$, and $S \in V$ is the start symbol. 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[10] as written has infinitely many productions: For instance, there is a production (e, x) for each of the infinitely many variables $x \in \mathit{Var}$. 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

$$x ::= \mathsf{x} \mid x\text{'},$$

yielding the countably many variables x, x', x'', and so on. Including these productions instead of the postulate '$x \in \mathit{Var}$' 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.

[9] 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 $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$. This defines another binary relation $\Rightarrow$ on A^*, the reflexive and transitive closure of which is denoted $\Rightarrow^*$.

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

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*

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 2.5. ■

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

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

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* $e = \mathsf{fst}\, e'$, *then* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau \times \tau_2$.

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

(f) *If* $e = \mathsf{inj}_1\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_1$.

(g) *If* $e = \mathsf{inj}_2\, e'$, *then* $\tau = \tau_1 + \tau_2$ *and* $\Xi \mid \Gamma \mid \Sigma \vdash e' : \tau_2$.

(h) *If* $e = \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, x \Rightarrow e_3\ \mathsf{end}$, *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* $v = \mathsf{inj}_1\, v'$ *or* $v = \mathsf{inj}_2\, v'$.

(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 $\mathsf{inj}_1\, v'$ 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

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 $\Xi, X \mid \Gamma \mid \Sigma \vdash \mathsf{rec}\, f\,(x) := e : \tau_1 \to \tau_2$. 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'.[11] 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.[12]

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 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 $\mathsf{fst}\, e = \mathsf{fst}\,(v_1, v_2)$, and this reduces via a head-step to v_1. Choosing the evaluation context $K = -$, HEAD-STEP-STEP implies that $\mathsf{fst}\,(v_1, v_2) \to v_1$. 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

[11] 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.

[12] 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.

that $e = K[d]$, $e' = K[d']$ and $d \to_h d'$. Letting $K' = \mathsf{fst}\, K$ we have

$$\mathsf{fst}\, e = K'[d] \to K'[d'] = \mathsf{fst}\, e',$$

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

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 4.4(c) it must be on the form $\mathsf{inj}_1\, v$ or $\mathsf{inj}_2\, v$ for a value v. Hence the expression $\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, x \Rightarrow e_3\ \mathsf{end}$ 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 $K' = \mathsf{match}\ K\ \mathsf{with}\ \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, x \Rightarrow e_3\ \mathsf{end}$, it follows that $\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, x \Rightarrow e_3\ \mathsf{end}$ 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 $\Sigma \vdash e_1 : \mathrm{ref}(\tau)$ 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 $\Sigma \vdash e : \mathrm{ref}(\tau)$, 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.

$K = \mathsf{fst}\, K'$: Then $K[e] = \mathsf{fst}\, K'[e]$, 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.

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

$K = \mathsf{match}\, K'\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\, \mathsf{end}$: 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.[13] [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 $e = \mathsf{fst}\,(v_1, v_2)$ 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 $e = \mathsf{ref}\, v$, $e' = l$, $\sigma' = \sigma[l \mapsto v]$, and $l \notin \mathrm{dom}\,\sigma$. By inversion we have $\Xi \mid \Gamma \mid \Sigma \vdash \mathsf{ref}\, v : \mathsf{ref}(\tau')$ 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 $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathsf{ref}(\Sigma'(l))$, so we have both $\Xi \mid \Gamma \mid \Sigma' \vdash \sigma'$ and $\Xi \mid \Gamma \mid \Sigma' \vdash l : \mathsf{ref}(\tau')$.

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 $\Xi \mid \Gamma \mid \Sigma \vdash () : \mathsf{Unit}$ as required.

[13] 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.

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 ■

5 ◇ Misc

5.1 • Lambda calculus

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

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

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 $\rightarrow$ 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

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

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

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

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