IDEAL MODEL FOR PLUGGABLE TYPES

TABLE 1. Correspondence between syntactic and semantic objects

Syntactic object	Semantic object
Term	Value
`plus 3 5`	the number 8
$\lambda x.\ x$	identity function on V
$\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$	the function mapping each $f \in V \to V$ to its least fixed point, and each $v \notin V \to V$ to WRONG
Type expression	Mathematical object, possibly a type
`Nat`	$\mathbb{N}_\perp = \{\perp, 0, 1, 2, \ldots\}$
`Int`	$\mathbb{Z}_\perp = \{\perp, 0, 1, -1, 2, -2, \ldots\}$
`Nat` $\to$ `Int`	$\{f \in (V \to V)_\perp \mid f(\mathbb{N}_\perp) \subseteq \mathbb{Z}_\perp\}$
$\forall\alpha.\ \alpha \to \alpha$	$\bigcap_{T \in \mathbf{Type}} \{f \in (V \to V)_\perp \mid f(T) \subseteq T\}$
$\alpha \to \beta$	A certain mapping from type environments to types
Judgement	Statement
`plus 3 5 : Nat`	$8 \in \mathbb{N}_\perp$
x `: Nat` $\vdash x$ `: Nat`	If $\rho(x) \in \mathbb{N}_\perp$, then $\rho(x) \in \mathbb{N}_\perp$
$\alpha; \beta \in$ `[Bot,` α`]` $\vdash_S \beta \leq \beta \to \alpha$	For all $T_\alpha \in \mathbf{Type}$, for all $T_\beta \in \mathbf{Type}$ with $\{\perp\} \subseteq T_\beta \subseteq T_\alpha$, we have $T_\beta \subseteq \{f \in (V \to V)_\perp \mid f(T_\beta) \subseteq T_\alpha\}$ (It is a false statement.)
Typing rule	Inference rule
$\dfrac{s : \texttt{Nat} \to \texttt{Int} \qquad t : \texttt{Nat}}{s\ t : \texttt{Int}}$	If the term s denotes $f \in \{g \mid g(\mathbb{N}_\perp) \subseteq \mathbb{Z}_\perp\}$ and t denotes $v \in \mathbb{N}_\perp$, then draw the conclusion that $s\ t$ denotes $f(v) \in Z_\perp$.

Contents

1. Pros and cons

Bracha (Pluggable type systems) advocates that the type system of a language should be optional and should not affect the language's runtime behavior. There are several benefits to such an arrangement.

(1) A useful program can be executed even if it has no type.
(2) Type systems can evolve faster than the language itself.
(3) Type inference can be made optional as well, so that the expressiveness of the type system is not bounded by the power of the inference algorithm.

We demonstrate a technique of pluggable types for purely functional languages. It achieves all of the above and more:

(4) It is safe for type systems to work together. The composite of several sound subsystems will continue to reject all programs with runtime type errors.
(5) Types defined in different subsystems can interact with each other. It is possible to call library functions defined in another type system.

The technique is applicable to a wide variety of type systems. It can express subtyping, universal types, union and intersection types, recursive types, dependent types, and higher kinds.

There are some restrictions on what a type can be. For example, if a program f has type T, then T must admit every program that terminates less often than f but behaves otherwise in the same way. Our types cannot separate terminating programs from nonterminating ones.

The technique guarantees nothing about the type checker's performance; it may run forever. Writing a nonterminating type checker is as easy as writing a nonterminating program in Java. It remains to investigate how to compose non-brute-force type checkers in a modular and scalable way.

2. Background

A runtime type error occurs when a value is used in an unintended manner, such as adding an integer and a truth value, dereferencing a non-pointer, and calling a non-function. A type system is *sound* if no well-typed program encounters runtime type errors. Soundness is one of the most important design goals of type systems.

There are several methods to prove a type system sound. We will discuss two: the syntactic approach, and the domain-theoretic approach.

2.1. The syntactic approach. It is the current standard framework for soundness proofs.

(1) Capture the runtime behavior of the language in a small-step semantics such that terms with runtime type errors are *stuck*: Neither are they values, nor can they reduce to other terms.
(2) Demonstrate *progress*: A well-typed term is either a value or reduces to something else.
(3) Demonstrate *preservation*: If a term has type τ then it continues to have type τ after one reduction.

Together, progress and preservation imply that well-typed terms never get stuck, and thus cannot experience runtime type errors.

In most circumstances, we can "append" to a syntactic soundness proof to accommodate new runtime behaviors without modifying existing arguments. But we cannot take two systems proven sound by the syntactic approach, take the union of their typing rules, and expect soundness to hold for the result. The two syntactic soundness proofs gave us progress and preservation for terms typed with rules in one single system; they say nothing about terms typed with a mixture of rules from both systems.

2.2. The domain-theoretic approach.

(1) Capture the runtime behavior of the language in a Scott domain.
(2) Define types as certain sets of values in the semantic domain. Designate a special value WRONG for runtime type errors, and make sure it is not a member of any type.
(3) Construct an interpretation from terms to values in the semantic domain. Show that if a term t has type τ, then t interprets to a member of τ.

Since no type contains WRONG, well-typed programs do not denote WRONG, and their evaluation may not encounter runtime type errors.

A domain-theoretic soundness proof is not extensible with new runtime behaviors. Adding mutation to a purely functional language, for example, requires a completely new domain. The old arguments have to be rewritten, because their foundation—the old domain equation—has become obsolete.

However, if we extend the type system without modifying the runtime behavior, then we can keep interpreting terms into the old semantic domain. If the new typing rules are sound on their own, then they already meet the expectation of the old soundness proof, namely that they assign type τ only to terms interpreting to a member of τ. In this way, the old proof carries over even to terms typed with a mixture of old and new rules, and type soundness continues to hold.

If types are pluggable, then the language's runtime behavior has to stay constant in all possible type systems. Here, a domain-theoretic type soundness proof is more extensible than a syntactic proof.

3. Roadmap

To achieve safely pluggable types, we exploit the extensibility of domain-theoretic type soundness proofs when the runtime system never changes. These are the steps:

(1) Choose a semantic domain. For purely functional languages, a domain for untyped lambda calculus suffices.
(2) Choose a theory of types for the semantic domain. We employ the ideal model by MacQueen, Plotkin and Sethi. It can easily express subtyping, universal types, union and intersection types, recursive types, dependent types, and higher kinds.
(3) Develop type systems such that each typing rule corresponds to a true statement in the theory of types. This property often follows from the domain-theoretic soundness proof and incurs no effort beyond that.

Thus typing rules become lemmas, judgements become statements, and typing derivations become proofs. Typing terms with a mixture of type systems is no more than writing proofs with a larger collection of lemmas.

The next section rephrases the ideal model of types by MacQueen et al. (Ideal model for recursive polymorphic types) in a more modern framework, so as to avoid using 2 different notions of limit in the semantic domain.

The section after that describes how to make type systems pluggable.

The other sections give examples of pluggable type systems. We will look at System F, unboxed impredicative polymorphism, and optional type arguments. It will be interesting to explore subtyping, continuations, mutation, dependent types and higher kinds.

4. Ideal model of types

This section recounts the ideal model of types by MacQueen et al. (Ideal model for recursive polymorphic types) with modern notions. After an informal overview of the most important concepts, we will work toward a proof that recursive types are well-defined. The next section starts on page 12.

4.1. Informal overview. The domain V contains values denoted by terms of untyped lambda calculus. It satisfies the isomorphism

$$V \simeq (V \to V) + B + \{\bot, \text{WRONG}\}, \tag{1}$$

where $+$ is disjoint union, $(V \to V)$ is the set of "continuous" functions from V to V, B is the set of base values (usually numbers, truth values and strings), $\bot$ is the value of nonterminating programs, and WRONG signifies runtime type errors. Let us assume implicit conversion between V and B, $(V \to V)$ etc., so that we may treat base values, functions, $\bot$ and WRONG as if they were actual values in V and save space.

Types are sets of values under certain closure conditions. The operator $\boxrightarrow$ constructs a type from two types.

$$\textbf{Type} = \text{set of subsets of } V \text{ that are types}$$
$$T_1 \boxrightarrow T_2 = \{f \in V \mid f = \bot \quad \text{or} \quad f \in V \to V \text{ and } f(T_1) \subseteq T_2\}$$

Let us state the definition of types for future reference. A subset $T \subseteq V$ is a type if it is a nonempty closed set under the Scott toplogy and WRONG $\notin T$.

4.2. Domain-theoretic background. Here we list facts from domain theory. Some results will depend on the particular construction of V as a *consistently complete algebraic cpo*, but we will not go into details. Chapters 2–4 of Stoltenberg-Hansen et al. (Mathematical Theory of Domains) contain a careful development.

There is a *definedness* partial order $\sqsubseteq$ on V such that

D1. $\bot \sqsubseteq v$ for all $v \in V$,
D2. a base value in B is only comparable with $\bot$ and itself,
D3. if $f, g \in V \to V$, then $f \sqsubseteq g$ if and only if $f(v) \sqsubseteq g(v)$ for all $v \in V$.

A set $A \subseteq S$ is *directed* if A is nonempty and for all $x, y \in A$ there exists $z \in A$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. Intuitively, a directed set contains successive approximations of the same value.

A partially ordered set is a *cpo* (complete partial order) if its every directed subset has a supremum. V is a cpo.

There is a topology on V called the *Scott topology*. A set $S \subseteq V$ is *open* in the Scott topology if

- S is *upward-closed*: $v \in S$ whenever $u \sqsubseteq v$ for some $u \in S$, and
- S is *unreachable by directed limit*: $\sup A \notin S$ for every directed set A disjoint from S.

A set $T \subseteq V$ is *closed* if its complement $V - T$ is open. T is closed if and only if

- T is *downward-closed*: $u \in T$ whenever $u \sqsubseteq v$ for some $v \in T$, and
- T is *closed under directed limit*: $\sup A \in T$ for every directed set $A \subseteq T$.

A subset $T \subseteq V$ is a *type* if T is a nonempty closed set and WRONG $\notin T$. Both downward-closure and closure under directed limit are necessary to establish that each recursive type equation is satisfied by a unique type. For example, one and only one type satisfies the following version of Curry's paradox:

$$T = T \boxrightarrow \{\bot\}.$$

It allows us to type Ω with recursive types (folklore on Haskell mailing list):

$$(\lambda x : \mu\alpha.\alpha \to \texttt{Bot}.\ x\ x)\ (\lambda x : \mu\alpha.\alpha \to \texttt{Bot}.\ x\ x).$$

Closure under directed limit is a natural strengthening of the *admissibility* criterion for contracts in HALO (Haskell to logic through denotational semantics). In fact, all contracts described in HALO can be strengthened to types by imposing downward-closure without disturbing the theories surrounding them.

Scott topology satisfies the 3 axioms of topology:

- $\emptyset$ and V are open,
- the union of an arbitrary family of open sets are open,
- the intersection of finitely many open sets is open.

On the flip side,

- the intersection of an arbitrary family of types is a type,
- the union of finitely many types is a type.

A function f from V to V is *continuous* if for every open set $S \subseteq V$, the preimage $f^{-1}(S)$ is an open set. The function f is continuous if and only if

- f is *monotone*: $f(u) \sqsubseteq f(v)$ whenever $u \sqsubseteq v$, and
- f *preserves directed limits*: $\sup f(A) = f(\sup A)$ for every directed $A \subseteq V$.

Write $V \to V$ for the set of continuous functions from V to V. The domain equation (1) states that a subset of V is isomorphic to the set of continuous functions on V.

A value $v \in V$ is *compact* if for all directed set A with $v \sqsubseteq \sup A$, there exists $a \in A$ such that $v \sqsubseteq a$. A compact element of V is either $\bot$, WRONG, a base value, or a function f such that there exists compact values $u_1, v_1, \ldots, u_n, v_n$ and f is the

least function mapping each u_i to v_i. Informally, all information about a compact function lies within a finite number of argument-result pairs. This characterization of compact elements is a corollary of theorem 3.8 in chapter 3 on page 67 of Stoltenberg-Hansen et al. (Mathematical Theory of Domains).

A cpo is *algebraic* if every element is the supremum of a directed set of compact elements. V is an algebraic cpo. In particular, the behavior of every function in $V \to V$ is determined by its value on compact elements. This agrees with the intuition that a terminating higher-order function f can only call its argument g finitely many times. Even if g is more complex than finite collections of argument-result pairs, f cannot detect that extra complexity and cannot reflect it in $f(g)$.

Since V is algebraic, every closed set is completely determined by its compact elements. In particular, if T_1 and T_2 are distinct types, then there exists a compact element that belongs to one and not the other.

A cpo is *consistently complete* if every set of elements have a supremum whenever they are bounded above. V is consistently complete.

These are all the concepts necessary to comprehend the statement "V is a consistently complete algebraic cpo."

4.3. Rank of compact elements. MacQueen et al. (Ideal model for recursive polymorphic types) proposed the notion of *rank* of compact elements so as to define the distance between types and discuss the convergence of sequences of types. They presented the domain V as some limit of $V_0, V_1, \ldots$, where

$$\begin{aligned} V_0 &= \{\bot\}, \\ V_1 &= (V_0 \to V_0) + B + \{\bot, \textsc{wrong}\}, \\ &\vdots \\ V_n &= (V_{n-1} \to V_{n-1}) + B + \{\bot, \textsc{wrong}\}, \\ &\vdots \end{aligned}$$

and defined the rank of a compact element v as the number i such that v "appears" in V_i for the first time. It is possible to formalize this intuition in terms of commuting embedding-projection pairs between each V_i and V akin to those described in §12.3 on page 319 of Stoltenberg-Hansen et al. (Mathematical theory of domains). We will not discuss the formalities in detail, only list the relevant properties of the rank function.

R1. $\bot$ has rank 0.

R2. Base values in B and WRONG have rank 1.

R3. A function $f \in V \to V$ has rank at most n if and only if there exists a set P of value pairs of rank at most $n-1$ such that f is the least function satisfying $f(x) = y$ for all $(x, y) \in P$.

R4. If $f \in V \to V$ does not have rank n for every $n \in \mathbb{N}$, then f has rank ∞.

R5. Every compact element has finite rank.

Properties R1–4 can be regarded as a practical definition of rank. R5 is a consequence of theorems 6.5 and 6.7 in chapter 4 on page 107 of Stoltenberg-Hansen et al. (Mathematical Theory of Domains).

The next lemma exercises the properties of rank and will be used in §4.13.

4.4. Lemma. *Let $S \subseteq V$ be a set of rank-n elements. If S has an upper bound in V, then* $\sup S$ *exists and has rank at most n.*

Proof. Since S has an upper bound in V, either S is disjoint from $V \to V$ or is a subset of it.

Case $S \cap (V \to V) = \emptyset$. If S has no rank-1 element, then $S \subseteq \{\bot\}$ and $\sup S = \bot$. If S has a rank-1 element v, then v is maximal in V and has to be the upper bound of S. Thus $v = \sup S$.

Case $S \subseteq V \to V$. Let $g \in V$ be an upper bound of S. By R3, for each $v \in S$ there exists a set P_v of argument-result pairs of rank at most $n-1$ such that v is the least function agreeing with P_v. Let

$$P = \bigcup_{v \in S} P_v,$$
$$f(x) = \sup \{y' \mid (x', y') \in P \text{ and } x' \sqsubseteq x\}.$$

Since g is an upper bound of S, each set $\{y' \mid (x', y') \in P \text{ and } x' \sqsubseteq x\}$ is bounded above by $g(x)$ and has a supremum in V by consistent completeness. Thus f is well-defined. By property D3 of the definedness partial order, f is the least upper bound of S. By R4, f has rank $\leq n$. □

4.5. Metric space of types. Recall from §4.2 that types are nonempty closed sets excluding WRONG. Following MacQueen et al. (Ideal model for recursive polymorphic types), we will define a distance function between types so that they form a metric space.

Let T_1, T_2 be types. If $T_1 = T_2$, then their *proximity* is ∞. If $T_1 \neq T_2$, then their proximity is the smallest rank among elements in the symmetric difference

$$T_1 \triangle T_2 = (T_1 - T_2) \cup (T_2 - T_1).$$

The proximity of two types signifies how hard it is to tell them apart. No two types have proximity 0, because $\bot$ is a member of all types. It is easiest to tell types of proximity 1 apart, because they are separated by a visible base value. If a function f separates two types, then we will have a hard time verifying it if we have to call f on many complicated arguments before arriving at some visible evidence.

The distance $d(T_1, T_2)$ between types T_1, T_2 is inverse exponential in the types' proximity.

$$d(T_1, T_2) = \frac{1}{2^{\text{proximity}(T_1, T_2)}} \tag{2}$$

The distance function d satisfies the conditions required of one in a metric space:

M1. $d(T_1, T_2) \geq 0$.

M2. $d(T_1, T_2) = 0$ if and only if $T_1 = T_2$.

M3. $d(T_1, T_2) = d(T_2, T_1)$.

M4. $d(T_1, T_3) \leq d(T_1, T_2) + d(T_2, T_3)$.

M1 is obvious. M2 follows from algebraicity of V, as all pairs of distinct types are separated by a compact element, which has finite rank. M3 follows from symmetry of symmetric difference. To see M4, let v be the least-ranked element separating T_1 and T_3. Assume without loss of generality that $v \in T_3 - T_1$. If $v \in T_2$, then $d(T_1, T_3) \leq d(T_1, T_2)$. If $v \notin T_2$, then $d(T_1, T_3) \leq d(T_2, T_3)$.

4.6. Converging sequences of types. Recall the standard definition of convergence of a sequence of sets:

The *limit superior* of an infinite sequence of sets $S_1, S_2, \ldots$ is

$$\limsup_{n\to\infty} S_n = \bigcap_{n=1}^{\infty} \bigcup_{i=n}^{\infty} S_i.$$

The *limit inferior* is

$$\liminf_{n\to\infty} S_n = \bigcup_{n=1}^{\infty} \bigcap_{i=n}^{\infty} S_i.$$

If the limit superior and limit inferior are equal, then the sequence $S_1, S_2, \ldots$ *converges*, and its *limit* is

$$S = \limsup_{n\to\infty} S_n = \liminf_{n\to\infty} S_n.$$

Let $T_1, T_2, \ldots$ types. They *converge as a sequence of types* if they converge to a limit S as a sequence of sets. The limit of $T_1, T_2, \ldots$ is the closure T of S in the Scott topology:

$$T = \text{closure}(S) = \bigcap \{S' \mid S \subseteq S' \text{ and } S' \text{ is closed}\}.$$

4.7. Remark. The closure operation is necessary in the definition of the limit of a sequence of types. In general, the set limit S of types $T_1, T_2, \ldots$ may not be a type. Let id_i be the rank-i approximation of the identity function and consider

$$T_i = \{v \in V \mid v \sqsubseteq id_i\}.$$

The sequence $T_1, T_2, \ldots$ converges to a set S containing every id_i but not their limit, namely the identity function on V. In other words, S is not closed under directed limit and is not a type.

4.8. Cauchy sequences. An infinite sequence $T_1, T_2, \ldots$ of types is Cauchy if for every $\epsilon > 0$ there exists $n \in \mathbb{N}$ such that for all $i, j \geq n$, the distance between T_i and T_j is smaller than ϵ.

4.9. Theorem. *Every Cauchy sequence of types converges.*

Proof. This corresponds to theorem 1 of MacQueen et al. (Ideal model for recursive polymorphic types). There are two proof goals.

G1. Every Cauchy sequence of types has a set limit.

G2. The closure of the set limit is a type.

G1. Let $T_1, T_2, \ldots$ be a Cauchy sequence of types. Knowing the standard result $\liminf T_n \subseteq \limsup T_n$, we need only show $\limsup T_n \subseteq \liminf T_n$.

Choose an arbitrary value

$$v \in \limsup T_n = \bigcap_{n=1}^{\infty} \bigcup_{i=n}^{\infty} T_i.$$

Let $\epsilon = 2^{-\text{rank}(v)} > 0$. There exists a natural number `cauchy` such that for all $i, j \geq$ `cauchy`, we have $d(T_i, T_j) < \epsilon$. Since v is a member of the limit superior, there exists a natural number `has_`$v \geq$ `cauchy` such that $v \in T_{\texttt{has\_}v}$. If we pick any $i \geq$ `has_`v, then we have $v \in T_i$, because otherwise the contradiction

$$d(T_{\texttt{has\_}v}, T_i) \geq 2^{-\text{rank}(v)} = \epsilon$$

would arise. Thus

$$v \in \bigcap_{i=\texttt{has\_}v}^{\infty} T_i \subseteq \bigcup_{n=1}^{\infty} \bigcap_{i=n}^{\infty} T_i. = \liminf T_n.$$

G2. We will verify that the limit

$$T = \text{closure}(\limsup T_n) = \text{closure}(\liminf T_n)$$

is a nonempty closed subset of V excluding WRONG. There are 3 properties to check.

G2.1. T is a closed set.
G2.2. $T \neq \emptyset$.
G2.3. WRONG $\notin T$.

T is closed by construction. Since each T_n is nonempty and downward closed, $\perp \in T_n$ for all $n \in \mathbb{N}$. Thus $\perp \in T$, and G2.2 follows. For G2.3, note that $V - \{\text{WRONG}\}$ is a closed set containing all T_i, and thus a superset of T. □

4.10. Banach fixed-point theorem. A metric space is *complete* if all Cauchy sequences converge. By theorem 4.9, the metric space of types is complete.

A function F mapping types to types is *nonexpansive* if there exists $0 \leq q \leq 1$ such that for all types T_1, T_2,

$$d(f(T_1), f(T_2)) \leq q\, d(T_1, T_2).$$

If $q < 1$, then F is *contractive*.

The Banach fixed-point theorem states that every contractive function on a nonempty complete metric space has a unique fixed point. Therefore a recursive type equation $T = F(T)$ defines a unique type if F is contractive. In the next few paragraphs, we will show record, variant and function type constructions to be contractive, substantiating the claim that recursive algebraic data types are well-defined.

4.11. KISP: composing binary type operators. We want to show record, variant and function type constructions to be contractive even when nested to arbitrary depth. The KISP framework in Yakushev et al. (Generic programming with fixed points for mutually recursive data types) provides a good structure to the argument. KISP stands for the type-level combinators K, I, sum and product, which are sufficient to express all strictly positive algebraic data types. Instead of sum and product, we will use combinators lifted from $\boxed{\to}$, union and intersection, so as to support recursion in contravariant positions as well.

Suppose we want to establish that a unique type T_* satisfies

$$T_* = (T_* \boxed{\to} \mathbb{Z}_\perp) \boxed{\to} \mathbb{Z}_\perp.$$

For that, we need to demonstrate contraction of the type-level function F mapping each T to $(T \boxed{\to} \mathbb{Z}_\perp) \boxed{\to} \mathbb{Z}_\perp$. Let us express F in point-free style:

$$F = (I :\boxed{\to}: K(\mathbb{Z}_\perp)) :\boxed{\to}: K(\mathbb{Z}_\perp)$$

where

$$I(T) = T$$
$$K(T_1)(T_2) = T_1$$
$$(F_1 :\boxed{\to}: F_2)(T) = F_1(T) \boxed{\to} F_2(T)$$

$$(F_1 :\cup: F_2)(T) = F_1(T) \cup F_2(T)$$
$$(F_1 :\cap: F_2)(T) = F_1(T) \cap F_2(T)$$

The lifted union and intersection will be relevant later for records and variants. We can show contraction of F—or of other type functions built with the five combinators above—with the following lemma.

4.12. Lemma.

L1. *$K(T)$ is contractive for all T.*
L2. *I is nonexpansive.*
L3. *If F and G are nonexpansive, then $F :\boxrightarrow: G$ is contractive.*
L4. *If F and G are nonexpansive, then $F :\cup: G$ and $F :\cap: G$ are nonexpansive. If G and G are contractive, then $F :\cup: G$ and $F :\cap: G$ are contractive.*

$K(T)$ is contractive because it takes all distances to 0. I is nonexpansive because it preserves distance. For L3 and L4, imagine H to be a binary type operator like $\boxrightarrow$, $\cup$ or $\cap$. If H is nonexpansive in both arguments, then the type combinator lifted from H preserves nonexpansion and contraction. Such is the case in L4. If H is contractive in both arguments, then the type combinator lifted from H takes two nonexpansive type functions to a contractive one. Such is the case in L3. We will prove neither the preceding remarks nor the nonexpansion of union and intersection, because I do not think they are hard and do not know how to present them in an interesting way. The next theorem discusses contraction of $\boxrightarrow$.

4.13. Theorem. *The binary type operator $\boxrightarrow$ is contractive in both arguments.*

Proof. We will only prove contraction of $\boxrightarrow$ in its contravariant first argument; the case for the covariant second argument is similar. Let

$$f \in (A \boxrightarrow C) - (B \boxrightarrow C)$$

be the least-ranked element in the symmetric difference between $A \boxrightarrow C$ and $B \boxrightarrow C$. There exists a pair of elements $x, y \in V$ such that $x \in B - A$, $y \notin C$ and $f(x) = y$. Let n be the rank of f. There exists a set P of argument-result pairs of rank at most $n - 1$ such that f is the least function agreeing with P. Let

$$X = \{x' \mid x' \sqsubseteq x \text{ and } (x', y') \in P \text{ for some } y'\},$$
$$Y = \{y' \mid (x', y') \in X \text{ for some } x'\}.$$

X and Y have supremums because x is an upper bound of X and y is an upper bound of Y. Since f is least,

$$y = f(x) = \sup Y = f(\sup X).$$

Since $\sup X \sqsubseteq x$ and B is downward closed, we have $\sup X \in B$. If $\sup X \in A$, then $f(\sup X) = y \notin C$ contradicts the fact that $f \in A \boxrightarrow C$. Therefore $\sup X$ is in the symmetric difference between A and B. By lemma 4.4, the rank of $\sup S$ is at most $n - 1$. Thus

$$d(A \boxrightarrow C, B \boxrightarrow C) = \frac{1}{2^n} = \frac{1}{2} \cdot \frac{1}{2^{n-1}} \leq \frac{1}{2} \cdot \frac{1}{2^{\mathrm{rank}(\sup X)}} \leq \frac{1}{2}\, d(A, B).$$

□

4.14. Contraction of record and variant constructions. Using lemma 4.12 and theorem 4.13, we can encode record and variant constructions into type functions and show them to be contractive.

For each value $v \in V$, write

$$v{\downarrow} = \{u \in V \mid u \sqsubseteq v\}.$$

Clearly $v{\downarrow}$ is a type. The record type $\{\texttt{real} : \mathbb{R}_\perp, \texttt{img} : \mathbb{R}_\perp\}$ is encoded as intersection of function types

$$(\texttt{"real"}{\downarrow} \boxrightarrow \mathbb{R}_\perp) \cap (\texttt{"img"}{\downarrow} \boxrightarrow \mathbb{R}_\perp),$$

where `"real"` and `"img"` are string literals. In general, record types are encoded thus:

$$\{l_i : T_i \ {}^{1 \le i \le n}\} = \bigcap_{i=1}^{n} l_i{\downarrow} \boxrightarrow T_i.$$

If all F_i are nonexpansive, then the function taking T to $\{l_i : F_i(T)\}$ is contractive by L3 and L4 of lemma 4.12.

The variant type $\langle \texttt{rational} : \mathbb{Q}_\perp, \texttt{irrational} : \mathbb{R}_\perp \rangle$ is encoded as the union of intersections of function types

$$\begin{aligned} &\big((\texttt{"tag"}{\downarrow} \boxrightarrow \texttt{"rational"}) \cap (\texttt{"content"}{\downarrow} \boxrightarrow \mathbb{Q}_\perp)\big) \\ \cup\ &\big((\texttt{"tag"}{\downarrow} \boxrightarrow \texttt{"irrational"}) \cap (\texttt{"content"}{\downarrow} \boxrightarrow \mathbb{R}_\perp)\big)\,. \end{aligned}$$

Since the strings `"rational"` and `"irrational"` are maximal in V, no maximal value can be both variants at the same time. In general,

$$\langle l_i : T_i \ {}^{1 \le i \le n} \rangle = \bigcup_{i=1}^{n} (\texttt{"tag"}{\downarrow} \boxrightarrow l_i{\downarrow}) \ \cap \ (\texttt{"content"}{\downarrow} \boxrightarrow T_i).$$

Just like record types, the function taking T to $\langle l_i : F_i(T) \rangle$ is contractive if each F_i is nonexpansive.

This encoding of records and variants has structural subtyping. If we want to make algebraic data types "disjoint" as they are in Haskell, we need only add the name of the data type as a field to every constructor.

4.15. Beyond algebraic data types. We have shown that the recursive type $\mu\alpha.\ \sigma$ is well-defined if σ is built from record, variant and function type constructions. In a language with universal types such as System F, We may want to allow types like

$$\mu\alpha.\ \alpha \to (\forall\beta.\ \beta \to \alpha) \qquad \text{or} \qquad \forall\alpha.\ \alpha \to (\mu\beta.\ \beta \to \alpha).$$

Proving them well-defined involves reasoning about more than one bound variable at a time, which requires a more powerful framework than KISP.

Write

$$T_1 \boxtimes T_2 = \{\texttt{fst} : T_1, \texttt{snd} : T_2\}.$$

To encode dependent vectors of real numbers, we might attempt to take the infinite union

$$(0 \boxtimes \mathit{Unit}) \cup (1 \boxtimes \mathbb{R}_\perp) \cup (2 \boxtimes \mathbb{R}_\perp \boxtimes \mathbb{R}_\perp) \cup (3 \boxtimes \mathbb{R}_\perp \boxtimes \mathbb{R}_\perp \boxtimes \mathbb{R}_\perp) \cup \cdots.$$

Although the arbitrary union of types is not always a type, there exists a smallest type containing the sets $(0 \boxtimes \mathit{Unit})$, $(1 \boxtimes \mathbb{R}_\perp)$ etc. that we can take as the type

of dependent vectors. This is a consequence of example 1.4(ii) in chapter 5 on page 117 of Stoltenberg-Hansen et al. (Mathematical Theory of Domains).

4.16. The importance of being pluggable. We have seen that the ideal model can express function types, records, variants, recursive types and dependent types. With such power, why do we need pluggable types at all? Why not a monolithic type system to mirror the ideal model exactly? MacQueen et al. (Ideal model for recursive polymorphic types) states that it is undecidable to check whether an untyped lambda term denotes a member of any given type. While I lack the expertise to verify, it seems reasonable to assume that the undecidability result would survive the port of the ideal model to consistently complete algebraic cpos. If this is the case, then we may never enjoy a complete type checker for the ideal model, but only use fragments of it appropriate to the situation at hand.

5. How to craft pluggable types

We have learnt the notion of types, the nature of function types, and the construction of recursive types. We will see how to make type systems pluggable.

Our core language is untyped lambda calculus with constants. Its standard denotational semantics serves as the interface to the runtime system modeled by the value domain V. Each lambda term denotes a value in V under a term environment env, which maps variables to values in V.

$$\begin{array}{rcll} t & ::= & & \text{untyped lambda term} \\ & & c & \text{constant} \\ & | & x & \text{variable} \\ & | & \lambda x.\ t & \text{abstraction} \\ & | & t\ t & \text{application} \end{array}$$

$$\begin{aligned} [\![\ c\ ]\!]\ env &= v_c && \text{designated value for constant } c \\ [\![\ x\ ]\!]\ env &= env(x) && \text{variable look-up} \\ [\![\ \lambda x.\ t\ ]\!]\ env &= \text{function mapping } v \in V \text{ to } [\![\ t\ ]\!]\ (env \text{ updated } x \mapsto v) \\ [\![\ t_1\ t_2\ ]\!]\ env &= \begin{cases} \bot & \text{if } [\![\ t_1\ ]\!]\ env = \bot \\ f(v) & \text{if } [\![\ t_1\ ]\!]\ env = f \in V \to V \text{ and } [\![\ t_2\ ]\!]\ env = v \\ \text{WRONG} & \text{if } [\![\ t_1\ ]\!]\ env \notin \{\bot\} \cup (V \to V) \end{cases} \end{aligned}$$

A pluggable type system may rely on syntax extensions. Each extended term must "erase" to an untyped lambda term, so that the runtime knows how to execute it. A syntax extension may be an optional type annotation, such as one on the argument of a lambda abstraction.

$$\begin{array}{rcll} \sigma & ::= & \cdots & \text{type expression} \\ t & += & \lambda x : \sigma.\ t & \text{annotated abstraction} \\ \text{erase}(\lambda x : \sigma.\ t) & = & \lambda x.\ \text{erase}(t) & \text{erasure to core language} \end{array}$$

Type expressions have no core syntax; they are not even required to denote a type. For our purpose of ruling out runtime errors, it suffices that *some* type expressions denote types. A type expressions of System F denotes a type only if it is *closed*, when all its type variables are universally quantified somewhere. Table 1 on page 1 contains more examples of type expressions.

Pluggable type systems produce *Judgements*. Judgements are abbreviations of mathematical statements. To treat statements as concrete data, we can represent them by well-formed formulas of ZFC (Zermelo-Fraenkel set theory with the axiom of choice). Each pluggable type system is free to produce its own flavor of judgements, but all systems should strive to eventually produce T-judgements, or typing judgements about closed terms.

$$J_0 \quad ::= \quad t : \sigma \quad \begin{array}{l}\text{typing judgement about a closed term} \\ \text{where } t \text{ is closed and } \sigma \text{ denotes a type}\end{array}$$

The judgement $t : \sigma$ abbreviates "The value denoted by t under the empty environment is a member of the type denoted by σ." We say a closed term t is *well-typed* if the type system in use can produce some T-judgement $t : \sigma$. Since WRONG is not a member of any type, well-typed expressions do not denote WRONG.

A pluggable type system produces judgements according to a collection of *typing rules*. In fact, we can think of the type system *as* the collection of typing rules. A typing rule is an inference rule in the style of natural deduction. Each rule has zero or more antecedents, zero or more side conditions, and one conclusion. The antecedents are judgements, the side conditions are statements (i. e., well-formed formulas of ZFC), and the conclusion is a judgement. A typing rule corresponds to the statement "If all antecedents and side conditions are true, then the conclusion is true." We call a typing rule *sound* if it corresponds to a provable statement. Sound typing rules are admissible with respect to the axioms and inference rules of ZFC (Does this claim need justification?). The typing rule in the last row of table 1 on page 1 is sound.

Judgements are produced by *derivations*. A derivation is a natural deduction proof: a finite tree built from instances of typing rules, where the antecedents of every rule instance coincide with the conclusions of instances immediately above it, and all side conditions are true. The final conclusion is the product of the derivation. If the typing rules are sound, then the product of every derivation has a proof in ZFC. A pluggable type system is *sound* if all its typing rules are sound. A sound type system only produces true judgements.

We mix several type systems together by taking union of their typing rules. The mixture of sound type systems is clearly sound. If it produces the T-judgement $t : \sigma$, then we are certain that t denotes an element of some type, which cannot be WRONG. In this sense, mixing pluggable type systems preserves type safety.

We know a term t never raises runtime type errors if some typing derivation produces a T-judgement about t. It does not matter how the typing derivation is produced. A practical implementation of a pluggable type system may be incomplete without compromising type safety. The type checker may reject some terms by mistake and loop forever on others, but the terms it does accept are guaranteed to be type safe. Since the type system is extensible, writing code with an incomplete type checker will not create maintenance hell in the future. The programmer can start using a type system without waiting for its type checker to become complete and decidable.

6. Warm-up example: System F

We will phrase System F in the framework outlined in §5 so that it can be used together with other type systems. This particular semantic model of System F was sketched in Girard (System F of variable types: fifteen years later, §3.1).

6.1. Syntax extension. System F requires annotated abstraction, type abstraction and type application. The interpretation of any term with these extensions is identical to the interpretation of its erasure.

$$\begin{array}{rcll} t & += & & \text{syntax extension} \\ & & \lambda x : \sigma.\ t & \text{annotated abstraction} \\ & | & \Lambda\alpha.\ t & \text{type abstraction} \\ & | & t\ [\sigma] & \text{type application} \end{array}$$

$$\begin{aligned} \text{erase}(\lambda x : \sigma.\ t) &= \lambda x.\ \text{erase}(t) \\ \text{erase}(\Lambda\alpha.\ t) &= \text{erase}(t) \\ \text{erase}(t\ [\sigma]) &= \text{erase}(t) \end{aligned}$$

6.2. Type expressions.

$$\begin{array}{rcll} \sigma & ::= & & \text{open type expression} \\ & & \iota & \text{base type} \\ & | & \alpha & \text{type variable} \\ & | & \sigma \to \sigma & \text{function type} \\ & | & \forall\alpha.\ \sigma & \text{universal type} \end{array}$$

6.3. Interpretation of open types. A type environment *Env* is a *total*[1] function mapping type variables to types. An open type denotes a function mapping type environments to types.

$$\begin{aligned} [\![\ \iota\ ]\!]\ Env &= T_\iota \subseteq B && \text{designated base type} \\ [\![\ \alpha\ ]\!]\ Env &= Env(\alpha) \\ [\![\ \sigma_0 \to \sigma_1\ ]\!]\ Env &= ([\![\ \sigma_0\ ]\!]\ Env) \ \boxrightarrow\ ([\![\ \sigma_1\ ]\!]\ Env) \\ &= \{\bot\} \cup \{f \in V \to V \mid f([\![\ \sigma_0\ ]\!]\ Env) \subseteq [\![\ \sigma_1\ ]\!]\ Env\} \\ [\![\ \forall\alpha.\ \sigma\ ]\!]\ Env &= \bigcap_{T \in \mathbf{Type}} [\![\ \sigma\ ]\!]\ (Env \text{ updated } \alpha \mapsto T) \end{aligned}$$

If an open type expression σ has no free type variable, then we say σ is *closed* and denotes the type $[\![\ \sigma\ ]\!]\ Env_\bot$, where $Env_\bot$ is the type environment mapping every type variable to $\{\bot\}$.

6.4. Judgements.

$$\begin{array}{rcll} \Gamma & ::= & & \text{typing context} \\ & & \emptyset & \text{empty context} \\ & | & \Gamma, x : \sigma & \text{term variable binding} \end{array}$$

$$\begin{array}{rcll} J_{\mathrm{F}} & ::= & \Gamma \vdash t : \sigma & \text{F-judgement} \end{array}$$

[1]If we allowed type environments to be partial functions, then many typing judgements would have to begin with "for every type environment defined on all relevant type variables."

6.5. Interpretion of F-judgements. Write

$$[\![\Gamma]\!]\ Env = \{env \mid env(x) \in [\![\sigma]\!]\ Env \text{ for all } x : \sigma \in \Gamma\}.$$

If $env \in [\![\Gamma]\!]\ Env$, then we say that the term environment env is *compatible* with the typing context Γ under the type environment Env.

The F-judgement $\Gamma \vdash t : \sigma$ abbreviates the following statement.

For every type environment Env and every $env \in [\![\Gamma]\!]\ Env$,

$$[\![t]\!]\ env\ \in\ [\![\sigma]\!]\ Env.$$

6.6. Typing rules of Pluggable F.

$$\frac{\emptyset \vdash t : \sigma \qquad t, \sigma \text{ closed}}{t : \sigma} \tag{T-F}$$

$$\frac{v_c \in [\![\sigma]\!]\ Env \qquad \sigma \text{ closed}}{\Gamma \vdash c : \sigma} \tag{F-Con}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \tag{F-Var}$$

$$\frac{\Gamma, x : \sigma_1 \vdash t : \sigma_2 \qquad x \notin \Gamma}{\Gamma \vdash \lambda x : \sigma.\ t : \sigma_1 \to \sigma_2} \tag{F-Abs}$$

$$\frac{\Gamma \vdash t_1 : \sigma_2 \to \sigma_3 \qquad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash t_1\ t_2 : \sigma_3} \tag{F-App}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash \Lambda\alpha.\ t : \forall\alpha.\ \sigma} \tag{F-Tabs}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\ \sigma_0}{\Gamma \vdash t\ [\sigma_1] : \sigma_0[\alpha \mapsto \sigma_1]} \tag{F-Tapp}$$

6.7. Soundness of Pluggable F. We outline the proof of each statement corresponding to a typing rule of Pluggable F.

T-F: Since t and σ are closed, they denote a value v_t and a type T_σ under whatever environments. The antecedent says $v_t \in T_\sigma$; the conclusion says the same thing.

F-Con: Since c and σ are both closed, their denotations do not depend on type or term environments. The conclusion is a consequence of the first side condition.

F-Var: The side condition implies that no matter which term environment env compatible with Γ is chosen, the value $env(x)$ must be a member of the type $[\![\sigma]\!]\ Env$. The conclusion follows.

F-Abs: Write $T_1 = [\![\sigma_1]\!]\ Env$ and $T_2 = [\![\sigma_2]\!]\ Env$. By the antecedent, t denotes a value in T_2 under every environment env that is compatible with Γ and maps x to a member of T_1. It follows that the denotation of $\lambda x.\ t$ under every environment compatible with Γ is a function whose image of T_1 is a subset of T_2.

F-App: The antecedents say that under every compatible environment, t_1 denotes a function f whose image of $[\![\sigma_2]\!]\ Env$ is a subset of $[\![\sigma_3]\!]\ Env$, and t_2 denotes a value $v \in [\![\sigma_2]\!]\ Env$. As desired, $f(v) \in [\![\sigma_3]\!]\ Env$.

F-Tabs: Let *env* be an arbitrary term environment compatible with Γ under *Env*. Let $v = [\![t]\!]\ env$. For every type $T \in \mathbf{Type}$, the antecedent guarantees that

$$v \in [\![\sigma]\!]\ (Env \text{ updated } (\alpha \mapsto T)),$$

which implies

$$v \in \bigcap_{T \in \mathbf{Type}} [\![\sigma]\!]\ (Env \text{ updated } (\alpha \mapsto T)) = [\![\forall\alpha.\ \sigma]\!]\ Env.$$

F-Tapp: Let *env* be a term environment compatible with Γ under *Env*. Write

$$T_1 = [\![\sigma_1]\!]\ Env,$$
$$v = [\![t]\!]\ env.$$

By the antecedent,

$$v \in \bigcap_{T \in \mathbf{Type}} [\![\sigma_0]\!]\ (Env \text{ updated } (\alpha \mapsto T)) \ \subseteq\ [\![\sigma_0]\!]\ (Env \text{ updated } \alpha \mapsto T_1).$$

By the correctness of capture-avoiding substitution,

$$[\![\sigma_0]\!]\ (Env \text{ updated } \alpha \mapsto T_1) = [\![\sigma_0[\alpha \mapsto \sigma_1]]\!]\ Env,$$

which gives us $v \in [\![\sigma_0[\alpha \mapsto \sigma_1]]\!]\ Env$ as desired.

7. Constrained type system version 1

To demonstrate what pluggable types are possible, we present a novel type system CT1 (Constrained Type System Version 1). It supports the impredicative universal types of System F, but does not demand type arguments from the user. The metatheory of CT1 is far from mature. We do not know whether it can type all terms that are typeable in System F with extra type arguments, or whether it admits a terminating type checking algorithm. Since CT1 is pluggable, however, we can start reaping its the benefit without waiting for metatheoretic development. If it so happened that CT1 could not handle certain useful programs, then we need only import a new type system to deal with those, confident that any legacy code typed under CT1 will continue to enjoy strong type soundness.

7.1. Syntax extension.

$$\begin{array}{rcll} t & += & \lambda x : \sigma.\ t & \text{annotated abstraction} \\ & | & \Lambda\alpha.\ t & \text{type abstraction} \end{array}$$

$$\text{erase}(\lambda x : \sigma.\ t) = \lambda x.\ \text{erase}(t)$$
$$\text{erase}(\Lambda\alpha.\ t) = \text{erase}(t)$$

7.2. Type expressions. CT1 divides type expressions into two camps. If I am a value, left types stand for what I am, right types stand for what I aspire to be. Open type expressions (§6.2) are both left and right. We divide the left from the right to confine union types to the left hand side of any subtype constraint, and intersection types to the right hand side. This way, every subtype constraint on compound types can be expressed as the conjunction of constraints on simpler types.

ρ	::=		left type expression
		ι	base type
	|	α	type variable
	|	$\tau \to \rho$	left function type
	|	$\forall\alpha.\ \rho$	left universal type
	|	`Bot`	smallest type
	|	$\rho \cup \rho$	union of left types
	|	$\forall\alpha \in I.\ \rho$	interval-bounded universal type

τ	::=		right type expression
		ι	base type
	|	α	type variable
	|	$\rho \to \tau$	right function type
	|	$\forall\alpha.\ \tau$	right universal type
	|	`Top`	largest type
	|	$\tau \cap \tau$	intersection of right types

I	::=	$[\rho, \tau]$	type interval

7.3. Interpretation of type expressions. Left and right type expressions have the same interpretation as open types; they are mappings from type environments to types. Recall that the union of two types is a type.

$$\begin{aligned}
[\![\ \texttt{Bot}\ ]\!]\ Env &= \{\bot\} \\
[\![\ \rho_1 \cup \rho_2\ ]\!]\ Env &= ([\![\ \rho_1\ ]\!]\ Env) \cup ([\![\ \rho_2\ ]\!]\ Env) \\
[\![\ \texttt{Top}\ ]\!]\ Env &= V - \{\text{WRONG}\} \\
[\![\ \tau_1 \cap \tau_2\ ]\!]\ Env &= ([\![\ \tau_1\ ]\!]\ Env) \cap ([\![\ \tau_2\ ]\!]\ Env)
\end{aligned}$$

The interpretation of base types, left/right function types and left/right universal types are identical to the interpretation of base types, function types and universal types in open type expressions (§6.3). Left and right type expressions can be regarded as extension-refinements of open type expressions, where the constructors $\to$ and $\forall$ retain their meanings.

A type interval interprets to a set of types sandwiched between its lower and upper bounds. If the lower bound is not a subset of the upper bound, then the type interval interprets to the empty set.

$$[\![\ [\rho, \tau]\ ]\!]\ Env = \{T \in \mathbf{Type} \mid [\![\ \rho\ ]\!]\ Env\ \subseteq\ T\ \subseteq\ [\![\ \tau\ ]\!]\ Env\}$$

An interval-bounded universal type interprets to the intersection of a family of types indexed by members of the interval.

$$[\![\forall\alpha \in I.\ \rho]\!]\ Env = \begin{cases} \bigcap_{T \in ([\![I]\!]\ Env)} [\![\rho]\!]\ (Env \text{ updated } \alpha \mapsto T) & \text{if } [\![I]\!]\ Env \neq \emptyset, \\ V - \{\text{WRONG}\} & \text{if } [\![I]\!]\ Env = \emptyset. \end{cases}$$

7.4. Judgements. CT1 produces two species of judgements. The S-judgements are about relative containment between types, and the CT-judgements are conditional statements about the membership of values in types. Typing contexts are identical to those in System F.

$$\begin{array}{rcll} \Gamma & ::= & \emptyset & \text{typing context} \\ & | & \Gamma, x : \sigma & \end{array}$$

$$\begin{array}{rcll} E & ::= & \emptyset & \text{prefix of interval bounds} \\ & | & E, \alpha \in I & \end{array}$$

$$\begin{array}{rcll} C & ::= & \emptyset & \text{S-constraints} \\ & | & C, \rho \leq \tau & \end{array}$$

$$J_{\text{CT}} ::= \Gamma \vdash t : \rho \text{ given } C \qquad \text{CT-judgement}$$

$$J_{\text{S}} ::= E \vdash C \qquad \text{S-judgement}$$

7.5. Interpretation of CT- and S-judgements. A type environment Env satisfies the list of constraints C if for each $\rho_i \leq \tau_i \in C$,

$$[\![\rho_i]\!]\ Env \ \subseteq\ [\![\sigma_i]\!]\ Env.$$

The CT-judgement $(\Gamma \vdash t : \rho \text{ given } C)$ abbreviates:

For every Env satisfying C and every $env \in [\![\Gamma]\!]\ Env$,

$$[\![t]\!]\ env\ \in\ [\![\rho]\!]\ Env.$$

Let $E = \beta_1 \in I_1, \ldots, \beta_n \in I_n$ be an arbitrary prefix of interval bounds. A type environment Env is *compatible* with E if for all $1 \leq j \leq n$ we have

$$Env(\beta_j)\ \in\ [\![I_j]\!]\ Env.$$

(See §7.3 for the meaning $[\![I_j]\!]\ Env$. Note that Env appears on both sides of the membership relation.)

Define *well-formed prefixes* recursively thus:

- $\emptyset$ is well-formed.
- If E is well-formed, α is a fresh type variable, and $[\![I]\!]\ Env \neq \emptyset$ for every Env compatible with E, then $(E, \alpha \in I)$ is well-formed.

The S-judgement $E \vdash C$ abbreviates:

E is well-formed. Every Env compatible with E satisfies C.

7.6. CT-rules of CT1.

$$\frac{\emptyset \vdash t : \rho \text{ given } \emptyset \qquad t, \rho \text{ closed}}{t : \rho} \tag{T-CT}$$

$$\frac{v_c \in [\![\ \rho\ ]\!]\ Env \qquad \rho \text{ closed}}{\Gamma \vdash c : \rho \text{ given } \emptyset} \tag{CT-Con}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \text{ given } \emptyset} \tag{CT-Var}$$

$$\frac{\Gamma, x : \sigma \vdash t : \rho \text{ given } C \qquad x \notin \Gamma}{\Gamma \vdash \lambda x : \sigma.\ t : \sigma \to \rho \text{ given } C} \tag{CT-Abs}$$

$$\frac{\Gamma \vdash t_1 : \rho_1 \text{ given } C_1 \qquad \Gamma \vdash t_2 : \rho_2 \text{ given } C_2 \qquad \alpha, \beta \text{ fresh}}{\Gamma \vdash t_1\ t_2 : \beta \text{ given } C_1 \cup C_2, \rho_1 \le \alpha \to \beta, \rho_2 \le \alpha} \tag{CT-App}$$

$$\frac{\Gamma \vdash t : \rho \text{ given } C \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(C)}{\Gamma \vdash \Lambda\alpha.\ t : \forall\alpha.\ \rho \text{ given } C} \tag{CT-Tabs}$$

$$\frac{\Gamma \vdash t : \rho \text{ given } C \qquad \beta_1 \in I_1, \ldots, \beta_n \in I_n \vdash C \qquad \beta_1, \ldots, \beta_n \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash t : \forall\beta_1 \in I_1.\ \cdots \forall\beta_n \in I_n.\ \rho \text{ given } \emptyset} \tag{CT-S}$$

7.7. Soundness of CT-rules. We outline the proof of each statement corresponding to a constrained typing rule of CT1.

T-CT: Since t and σ are closed, they denote a value v and a type T under whatever environments. The antecedent states that $v \in T$ under a vacuously true condition, so we may conclude $v \in T$.

CT-Con and CT-Var: Analogous to F-Con and F-Var discussed in §6.7.

CT-Abs: Let Env be a type environment satisfying C. If Env does not exist, then the conclusion is vacuously true. If Env exists, then let env be compatible with Γ under Env. Write

$$\begin{aligned} T_\sigma &= [\![\ \sigma\ ]\!]\ Env, \\ T_\rho &= [\![\ \rho\ ]\!]\ Env, \\ f &= [\![\ \lambda x.\ t\ ]\!]\ env. \end{aligned}$$

Choose arbitrary $v \in T_\sigma$. Since

$$(env \text{ updated } x \mapsto v) \ \in\ [\![\ \Gamma, x : \sigma\ ]\!]\ Env,$$

the antecedent implies $f(v) \in T_\rho$, and we may conclude $f \in T_\sigma \boxrightarrow T_\rho$.

CT-App: Let Env be a type environment satisfying $C_1 \cup C_2, \rho_1 \le \alpha \to \beta, \rho_2 \le \alpha$. Choose arbitrary $env \in [\![\ \Gamma\ ]\!]\ Env$. Write

$$\begin{aligned} T_1 &= [\![\ \rho_1\ ]\!]\ Env, & v_1 &= [\![\ t_1\ ]\!]\ env, \\ T_2 &= [\![\ \rho_2\ ]\!]\ Env, & v_2 &= [\![\ t_2\ ]\!]\ env. \end{aligned}$$

Since Env satisfies $\rho_1 \le \alpha \to \beta$ and $\rho_2 \le \alpha$,

$$T_1 \subseteq Env(\alpha) \boxrightarrow Env(\beta), \qquad T_2 \subseteq Env(\alpha).$$

Since *Env* satisfies both C_1 and C_2, the antecedents give us

$$v_1 \in T_1 \subseteq Env(\alpha) \boxrightarrow Env(\beta), \qquad v_2 \in T_2 \subseteq Env(\alpha),$$

which yield $v_1(v_2) \in Env(\beta)$ as desired.

CT-Tabs: Similar to T-Tabs discussed in §6.7.

CT-S: The assumed S-judgement $\beta_1 \in I_1, \ldots, \beta_n \in I_n \vdash C$ states that the prefix of interval bounds

$$E = \beta_1 \in I_1, \ldots, \beta_n \in I_n$$

is well-formed, and that every type environment compatible with E satisfies C. Consider an arbitrary type environment *Env*. Let

$$\begin{aligned} T &= [\![\ \forall\beta_1 \in I_1.\ \cdots \forall\beta_n \in I_n.\ \rho\ ]\!]\ Env \\ &= \bigcap_{T_1 \in [\![I_1]\!]\ Env} \quad \bigcap_{T_2 \in [\![I_2]\!]\ (Env \text{ updated } \beta_1 \mapsto T_1)} \cdots \\ &\quad \cdots \bigcap_{T_n \in [\![I_n]\!]\ (Env \text{ updated } \beta_1 \mapsto T_1, \ldots, \beta_{n-1} \mapsto T_{n-1})} \\ &\qquad\qquad [\![\ \rho\ ]\!]\ (Env \text{ updated } \beta_1 \mapsto T_1, \ldots, \beta_n \mapsto T_n). \end{aligned} \tag{3}$$

Note that in (3), all index sets of the form

$$[\![\ I_j\ ]\!]\ (Env \text{ updated } \beta_1 \mapsto T_1, \ldots, \beta_{j-1} \mapsto T_{j-1})$$

are nonempty due to well-formedness of E. Choose arbitrary $env \in [\![\ \Gamma\ ]\!]\ Env$. We want to prove that $[\![\ t\ ]\!]\ env \in T$. Pick one particular instantiation of $T_1, \ldots, T_n$ in equation (3). Let

$$Env' = (Env \text{ updated } \beta_1 \mapsto T_1, \ldots, \beta_n \mapsto T_n).$$

Env' is compatible with E and satisfies C. Since $\beta_1, \ldots, \beta_n \notin \mathrm{FTV}(\Gamma)$,

$$[\![\ \Gamma\ ]\!]\ Env' = [\![\ \Gamma\ ]\!]\ Env \ni env.$$

By the antecedent ($\Gamma \vdash t : \rho$ given C),

$$[\![\ t\ ]\!]\ env \in [\![\ \rho\ ]\!]\ Env'.$$

Since $T_\alpha, T_1, \ldots, T_n$ are arbitrary, $[\![\ t\ ]\!]\ env \in T$.

7.8. S-rules of CT1.

$$\frac{}{\emptyset \vdash \emptyset} \tag{S-Vacuous}$$

$$\frac{E \vdash C}{E \vdash C, \sigma \le \sigma} \tag{S-Refl}$$

$$\frac{E \vdash C}{E \vdash C, \mathtt{Bot} \le \tau} \tag{S-Bot}$$

$$\frac{E \vdash C}{E \vdash C, \rho \le \mathtt{Top}} \tag{S-Top}$$

$$\frac{E \vdash C, \rho_1 \le \tau_1, \rho_2 \le \tau_2}{E \vdash C, \tau_1 \to \rho_2 \le \rho_1 \to \tau_2} \tag{S-Arrow}$$

$$\frac{E \vdash C, \rho_1 \le \tau, \rho_2 \le \tau}{E \vdash C, \rho_1 \cup \rho_2 \le \tau} \quad \text{(S-Union)}$$

$$\frac{E \vdash C, \rho \le \tau_1, \rho \le \tau_2}{E \vdash C, \rho \le \tau_1 \cap \tau_2} \quad \text{(S-Intersect)}$$

$$\frac{E, \alpha \in I \vdash C, \rho \le \tau \qquad \alpha \text{ occurs only in } \rho}{E \vdash C, (\forall \alpha.\ \rho) \le \tau} \quad \text{(S-L0)}$$

$$\frac{E, \alpha \in [\rho_1 \cup \rho_2, \tau_1 \cap \tau_2] \vdash C, \rho \le \tau \qquad \alpha \text{ occurs only in } \rho}{E \vdash C, (\forall \alpha \in [\rho_1, \tau_1].\ \rho) \le \tau} \quad \text{(S-L1)}$$

$$\frac{E \vdash C, \rho \le \tau \qquad \alpha \text{ occurs only in } \tau}{E \vdash C, \rho \le (\forall \alpha.\ \tau)} \quad \text{(S-R)}$$

$$\frac{E \vdash C, \rho \le \tau \qquad \alpha \text{ fresh}}{E, \alpha \in [\rho, \tau] \vdash C, \rho \le \alpha, \alpha \le \tau} \quad \text{(S-Loner)}$$

7.9. Soundness of S-rules.

S-Vacuous: The empty prefix of interval bounds is well-formed. Every statement in an empty collection is true.

S-Refl: The subset relation $\subseteq$ is reflexive.

S-Bot: $\{\bot\}$ is a subset of every type.

S-Top: Every type is a subset of $V - \{\text{WRONG}\}$.

S-Arrow: E is well-formed. Let *Env* be compatible with E. Let

$$f \in [\![\ \tau_1\ ]\!]\ Env \ \rightrightarrows\ [\![\ \rho_2\ ]\!]\ Env.$$

For every

$$v \in [\![\ \rho_1\ ]\!]\ Env \subseteq [\![\ \tau_1\ ]\!]\ Env,$$

we know

$$f(v) \in [\![\ \rho_2\ ]\!]\ Env \subseteq [\![\ \tau_2\ ]\!]\ Env.$$

The desired result follows:

$$[\![\ \tau_1\ ]\!]\ Env \ \rightrightarrows\ [\![\ \rho_2\ ]\!]\ Env \qquad \subseteq \qquad [\![\ \rho_1\ ]\!]\ Env \ \rightrightarrows\ [\![\ \tau_2\ ]\!]\ Env$$

S-Union, S-Intersect: Obvious, by definitions of set union and intersection.

S-L0: This is a special case of S-L1 with $\rho_1 = \texttt{Bot}$ and $\tau_1 = \texttt{Top}$.

S-L1: Let *Env* be a type environment compatible with E. Let

$$I_1 = [\![\ [\rho_1, \tau_1]\ ]\!]\ Env,$$
$$I_2 = [\![\ [\rho_1 \cup \rho_2, \tau_1 \cap \tau_2]\ ]\!]\ Env.$$

Since $(E, \alpha \in [\rho_1 \cup \rho_2, \tau_1 \cap \tau_2])$ is well-formed, E is well-formed and $I_2 \subseteq I_1$ is nonempty. With any $T_\alpha \in I_2$,

$$\begin{aligned}
[\![\forall\alpha \in [\rho_1, \tau_1].\ \rho]\!]\ Env &= \bigcap_{T \in I_1} [\![\rho]\!]\ (Env \text{ updated } \alpha \mapsto T) \\
&\subseteq [\![\rho]\!]\ (Env \text{ updated } \alpha \mapsto T_\alpha) \\
&\subseteq [\![\tau]\!]\ (Env \text{ updated } \alpha \mapsto T_\alpha) \\
&= [\![\tau]\!]\ Env,
\end{aligned}$$

where the last equality is due to α not being free in τ. So Env satisfies $(\forall\alpha.\ \rho) \leq \tau$. Since α is not free in C and $(Env \text{ updated } \alpha \mapsto T_\alpha)$ satisfies C, Env satisfies C as well.

S-R: Let Env be compatible with E. Then Env satisfies C. Choose arbitrary $T_\alpha \in \mathbf{Type}$. Since α does not occur in E, the type environment

$$Env' = (Env \text{ updated } \alpha \mapsto T_\alpha)$$

is compatible with E. By the antecedent,

$$[\![\rho]\!]\ Env = [\![\rho]\!]\ Env' \subseteq [\![\tau]\!]\ Env'.$$

Since T_α is arbitrary, we may conclude

$$[\![\rho]\!]\ Env \subseteq \bigcap_{T \in \mathbf{Type}} [\![\tau]\!]\ (Env \text{ updated } \alpha \mapsto T) = [\![\forall\alpha.\ \tau]\!]\ Env.$$

S-Loner: Let Env be compatible with E. By the antecedent,

$$[\![\rho]\!]\ Env \subseteq [\![\tau]\!]\ Env.$$

Therefore the interval $[\![[\rho, \tau]]\!]\ Env$ is nonempty, and the prefix $(E, \alpha \in [\rho, \tau])$ is well-formed. Moreover,

$$[\![\rho]\!]\ Env \subseteq T_\alpha \qquad \text{and} \qquad T_\alpha \subseteq [\![\tau]\!]\ Env$$

for every $T_\alpha \in [\![[\rho, \tau]]\!]\ Env$ by definition. Hence the conclusion.

8. Mixing F and CT1 for optional type arguments

CT1 is friendlier than F for programming with impredicative polymorphism, because the user does not have to provide type arguments. However, we do not know whether CT1 subsumes F, whether all terms well-typed in F remain well-typed in CT1 if we leave out type applications. This lack of knowledge discourages a commitment to CT1, but it does not prevent us from enjoying its benefit. By mixing F and CT1 together, user program fragments will not be problematic if they can be typed in F and not in CT1.

F consumes and produces F-judgements, while CT1 consumes and produces S- and CT-judgements. To have rules from both systems in the typing derivation of a single term, we need to make F- and CT-judgements talk to each other by establishing some entailment relations.

Going from F-judgements to CT-judgements is easy, because the former are stronger than the latter.

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \sigma \text{ given } \emptyset} \qquad \text{(CT-F)}$$

It is trickier to go from CT-judgements to F-judgements. Not only do we have to resolve accumulated constraints (via CT-S for example), we must also eliminate union, intersection and interval-bounded universal type expressions from the type of a term, so that F-rules can deal with it comfortably. Ascription suffices.

$$t \quad += \quad t : \sigma \quad \text{ascription}$$

$$\text{erase}(t : \sigma) = t$$

$$\frac{\Gamma \vdash t : \rho \text{ given } \emptyset \qquad \emptyset \vdash \rho \leq \sigma}{\Gamma \vdash (t : \sigma) : \sigma} \qquad \text{(F-Ascr)}$$

The soundness of these rules is obvious after §6.7, 7.7 and 7.9.

9. Subtyping

Records, variants, subtyping. Subsumption and supertype-bounded quantification in F and CT1.

10. Parametric types

It may be possible to refine "equal up to termination" relation in "fast and loose reasoning" to the non-transitive consistency relation. Free theorems should hold. Parametric types are not universal types.

11. Category of types

Category theory has been applied to programming languages and is useful to reason about data type generic programming (bananas et co.). Types form a category.

- Objects are types.
- Morphisms are triples (f, T_1, T_2) where $f \in V \to V$ and $f(T_1) \subseteq T_2$.
- Composition of morphisms is function composition on the first component: $(f, T_2, T_3) \circ (g, T_1, T_2) = (f \circ g, T_1, T_3)$.

Each function in $V \to V$ gives rise to multiple morphisms. The composition of morphisms is clearly associative and always produces a morphism. The identity morphism on the object T is (id_V, T, T), where id_V is the identity function on V.

An endofunctor F is a function mapping types to types and morphisms to morphisms such that identity morphisms and composition are preserved. If F is contractive, then it has a unique fixed point T. If we write id_T for the identity morphism (id_V, T, T), then (T, id_T) is both the initial algebra and the final coalgebra. Lists *are* streams.