4th Year Project

Oxford Computer Science

CoPolymorphic Types

By Eason Kamander

With many thanks to Bartek Klin and Reuben Hillyard
for a great deal of help, direction, and support

Oct 2024 — May 2025
Word Count: 7938

Contents

1 Introduction

This project is a continuation of previous work, investigating cosimple types with guarded recursion. Similar to Nakano's Later Modality [3], cosimple types utilize a modal operator to guarantee that each term gradually makes progress under lazy evaluation, even if it never reaches a normal form. The idea at the center of this research is a new, local perspective on equi-recursive types, which we describe in much more detail. We also expand our focus, incorporating polymorphic type inference into this type system.

Polymorphism is usually formalized in the lambda calculus through universally quantified types, as in System F. However, there are a variety of issues with System F which hinder this strategy. First off, type checking and inference in System F are both undecidable. Even more significantly, System F lacks principal types, without which type checking and inference become a matter of guesswork.

For example [5], the self-application function $\omega \equiv \lambda x.xx$ is typeable as both $(\forall a.\ a) \to (\forall a.\ a)$ and $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$, representing void and unit respectively. Unfortuantly, there is no more general unifier for these two type assignments.

Intersection types [7] take a much more direct approach, recording a collection of distinct types for each argument to use. Thus, we have $\omega : \{a \to b,\ a\} \to b$, whose input is specified by the formal intersection of $a \to b$ and a; that is to say, something which is both a function and an argument for that function. These types much more closely stick to the premise of polymorphism, since after all, polymorphism literally means many forms.

Intersection types resemble multi-functions, where each domain is specified by a set while each codomain is only a singleton. This assymetry between inputs and outputs reflects a basic assymetry of the lambda calculus itself: a function may use its variables many times, but only returns a single expression as output. In this paper, we integrate the power of intersection types with the philosophy of the cosimple types.

First, following in the footsteaps of the cosimple types, we introduce type environments, using them to parametrize the recursively defined intersection types we have at our disposal. Then, we discuss principal type inference, developing a straightforward algorithm and formalizing its correctness. Along the way, we uncover a categorical perspective on type environments, and reproduce the undecidability result of ordinary intersection type. Finally, we use this new framework to propose a new, hopefully decidable, approximation system.

2 Environments

In the simply typed lambda calculus, we begin by choosing a fixed set of primitive type constants, and then inductively construct our universe of function types starting from this set. Some authors include primitives that represent specific objects of study, such as a built-in type of natural numbers or booleans, usually accompanied by term constants to populate these primitives with their intended inhabitants [2] [6]. Other portrayals focus exclusively on pure lambda terms, and treat the primitive types as meaningless, interchangeable variables [1] [4]. However, there is still a choice to be made regarding the cardinality of our set of type variables.

Typically, we assume a countably infinite set of type variables, allowing us to summon arbitrarily many fresh variables as we need them. However, any individual type derivation is finite, and thus

only mentions finitely many type variables; we only chose an infinite set because no finite number of variables would be large enough for all possible derivations. But instead of fixing one set of type variables up front, we could choose our own set of type variables for each derivation, allowing us to select exactly as many as we need for the derivation at hand. For example, we can use three type variables $\{A, B, C\}$ for the principal type derivation on the first-of-three projection map:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A,\, y : B,\, z : C \vdash x : A}\,(\text{var})}{x : A,\, y : B \vdash \lambda z.x : C \to A}\,(\text{abs})}{x : A \vdash \lambda yz.x : B \to C \to A}\,(\text{abs})}{\vdash \lambda xyz.x : A \to B \to C \to A}\,(\text{abs})$$

Meanwhile, the principal type derivation for the first-of-four projection map requires an additional variable. Thus, we can construct our universe of types in many different ways, each of which gives rise to its own system of derivations. Taking this idea further, we introduce type environments, an algebraic structure representing a generalized universe of types. Just as a field provides a notion of scalars to be used in the definition of vector spaces, an environment provides a notion of types to be used in the definition of derivations.

Definition 2.1. A type environment consists of:

1. A set $\mathcal{T}$ of types
2. A function $\mathsf{Dom} : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ assigning each type its domain
3. A function $\mathsf{cod} : \mathcal{T} \to \mathcal{T}$ assigning each type its codomain

Whereas the ordinary inductive definition of the types relies on variables as a starting point, type environments have no need for variables. Instead, every type is interpreted as a multi-function in terms of the other types provided in that environment. Thus, type environments can be viewed as the coalgebras for the functor $X \mapsto \mathcal{P}(X) \times X$.

Alternatively, the domain function can be equivalently characterized by its uncurrying, as the binary relation $(x, y) \mapsto x \in \mathsf{Dom}(y)$. From this perspective, a type environment is a setting where we can formulate the question "does this type belong to the domain of that type?" as one of our primitive judgements. Thus, type environments can also be viewed as the models of the first-order signature consisting of one binary relation and one unary function.

We write $A \to b$ to denote any type whose domain is the set A and whose codomain is the type b, generally using uppercase for sets of types and lowercase for individual types. We may also omit brackets, abbreviating the singleton domain $\{a\} \to b$ as $a \to b$.

Example 2.1.1. The trivial type environment contains a single type, which is both an element of its own domain and equal to its own codomain. In terms of components, the trivial type environment is defined by:

1. The singleton set of types, $\mathcal{T} \equiv \{*\}$
2. The total domain relation, $* \in \mathsf{Dom}(*)$
3. The unique codomain function, $\mathsf{cod}(*) \equiv *$

For simplicity, we prefer to define each type alongside its domain and codomain. We may notate a type environment with a set of definitions of the form $f \equiv A \to b$, introducing the type f while simultaneously assigning its interpretation as a multi-function with domain A and codomain b. Thus, the trivial type environment is given by the single recursive type definition:

$$* \equiv \{*\} \to *$$

Example 2.1.2. The single-variabled type environment is defined by:

$$x_n \equiv \varnothing \to x_{n+1} \quad \text{for } n \in \mathbb{N}$$

Intuitively, this environment is constructed by starting from a single type x_0 and repeatedly adding in one additional element, to ensure that each type has a codomain. Thus, the single-variabled type environment is a term model, equivalently defined as the Herbrand structure for the first-order signature of a type environment, extended with a single constant symbol x_0.

Now, consider a fixed type environment $\mathcal{T}$.

Definition 2.2. A sequent internal to $\mathcal{T}$ consists of:

1. A lambda term t with free variables in $\mathcal{V}$
2. A consequence type $c \in \mathcal{T}$
3. A context $\Gamma : \mathcal{V} \to \mathcal{P}(\mathcal{T})$

We write $\Gamma \vdash t : c$ to denote the sequent with context Γ, term t, and consequence c. As usual, this expression is intended to represent the claim "the assumptions in Γ imply that t has type c."

Just as we sometimes regard the domain function as a binary relation, our context Γ defines a relation between free variables and types. The elements of this relation are the assumptions present in Γ, that each free variable can be used with each of its associated types.

As is customary, we typically write our context Γ as a list of components, reflecting the total order in which the free term variables in $\mathcal{V}$ were introduced.

Example 2.2.1. The sequent "$x : \{a, b\}$, $y : \{c\} \vdash x(\lambda z.z) : d$" can be read as:

Whenever
- x can be used with type a
- x can be used with type b
- y can be used with type c

Then
- $x(\lambda z.z)$ is a term of type d

Definition 2.3. A derivation in $\mathcal{T}$ is a proof tree according to the inference rules:

$$\frac{c \in \Gamma(x)}{\Gamma \vdash x : c}\ (\text{var})$$

$$\frac{\Gamma,\, x : A \vdash s : b}{\Gamma \vdash \lambda x.s : A \to b}\ (\text{abs})$$

$$\frac{\Gamma \vdash p : A \to b \qquad \forall a \in A.\, \Gamma \vdash q : a}{\Gamma \vdash pq : b}\ (\text{app})$$

Note: The premise on the right side of the application rule is not a sequent. Instead, the universal quantifier represents the conjunction of many sequents. Thus, in some environments, proof trees may be infinitely wide.

Nonetheless, derivations are still essentially trees in the sense that they are built by induction out of some assortment of sub-derivations. Just as in the simply typed lambda calculus, our three inference rules correspond exactly to the three constructors of lambda terms, so that the required assortment of sub-derivations for a sequent is determined by inspecting its term.

This property, known as Subject Construction, allows us to view each inference rule as an explicit description of the data contained within any derivation for the corresponding class of terms. Thus, the required assortment of sub-derivations for a given sequent $\Gamma \vdash t : c$ can be equivalently seen as a witness for the proposition $[\![\Gamma \vdash t : c]\!]$, defined by pattern matching on t:

$$[\![\Gamma \vdash x : a]\!] \equiv a \in \Gamma(x) \qquad (\text{var})$$

$$[\![\Gamma \vdash \lambda x.s : f]\!] \equiv [\![\Gamma,\, x : \mathsf{Dom}(f) \vdash s : \mathsf{cod}(f)]\!] \qquad (\text{abs})$$

$$[\![\Gamma \vdash pq : b]\!] \equiv \exists f.\, \mathsf{cod}(f) = b \wedge [\![\Gamma \vdash p : f]\!] \qquad (\text{app})$$
$$\wedge\, \forall a \in \mathsf{Dom}(f).\, [\![\Gamma \vdash q : a]\!]$$

By recursively unraveling these definitions, we can associate each closed lambda term t with its characteristic sentence $\exists c.\, [\![\vdash t : c]\!]$, representing the entire structure of a derivation on t. Thus, the derivations on t in any type environment $\mathcal{T}$ are exactly the witnesses for this characteristic sentence in our chosen model $\mathcal{T}$.

Example 2.3.1. If our type environment is simple, in the sense that the domain of each type is always a singleton set, then the structure of our derivations mirror those of the simply typed lambda calculus. The universal quantifier in the application rule can be ignored, since it will only ever operate on singleton sets. Similarly, our abstraction rule will only introduce singleton components into our contexts. Thus, the elementhood constraints in the variable rule will always amount to equality checking.

However, we can still form non-standard derivations using recursively defined types, as shown in the next example. Instead, if our type environment is also inductive, in the sense that it is possible to place an ordering on the set of types such that each element $A \to b$ occurs strictly after all of its dependencies $a \in A$ and b, then our derivations follow the familiar behavior of the simple types. In practice, all we have done is pre-define a collection of simple types we are able to use.

In particular, we can formulate a type for the church encoded natural numbers in the following simple, inductive environment. Here, a_0 resembles a type variable, with $c \equiv (a_0 \to a_0) \to (a_0 \to a_0)$:

$$\begin{aligned} a_n &\equiv \{a_{2n+1}\} \to a_{2n+2} \quad \text{for } n \in \mathbb{N} \\ b &\equiv \{a_0\} \to a_0 \\ c &\equiv \{b\} \to b \end{aligned}$$

Now, we have enough structure to recreate the principal type derivation of any church encoded natural number. Here, we encode the principal type derivation for the number $2 \equiv \lambda fx.f(fx)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{b \in \{b\}}{f : \{b\},\, x : \{a_0\} \vdash f : b}\,(\text{var}) \quad \dfrac{\dfrac{b \in \{b\}}{f : \{b\},\, x : \{a_0\} \vdash f : b}\,(\text{var}) \quad \dfrac{a_0 \in \{a_0\}}{f : \{b\},\, x : \{a_0\} \vdash x : a_0}\,(\text{var})}{f : \{b\},\, x : \{a_0\} \vdash fx : a_0}\,(\text{app})}{f : \{b\},\, x : \{a_0\} \vdash f(fx) : a_0}\,(\text{app})}{f : \{b\} \vdash \lambda x.f(fx) : b}\,(\text{abs})}{\vdash \lambda fx.f(fx) : c}\,(\text{abs})$$

However, we can only guarantee derivations for these normal forms. Meanwhile, an unevaluated, simply-typeable term would likely utilize additional function types along the intermediate steps of its derivation, which would need to be included in our environment.

Example 2.3.2. The trivial type environment is simple but not inductive. Thus, the structure of its derivations is similar to the simply typed lambda calculus, but the recursive type $* \equiv * \to *$ introduces unusual behavior. In fact, this type environment provides a unique type derivation for each closed term.

Proof.

For any set of term variables $\mathcal{V}$, let $\top$ denote the total context $\mathsf{const}\ \{*\} : \mathcal{V} \to \mathcal{P}(\{*\})$, assigning each component $v \in \mathcal{V}$ the full set of types $v : \{*\}$.

Now, we argue by structural induction that for any term t with free variables in $\mathcal{V}$, the sequent $\top \vdash t : *$ has a unique derivation:

- For a variable x, we claim $\top \vdash x : *$ has a unique derivation:
  By definition $\top(x) \equiv \{*\}$, so the condition $* \in \top(x)$ must hold.
  So, by the variable rule, we immediately have a derivation for $\top \vdash t : *$.
  And, by Subject Construction, this derivation is necessarily unique.
- For an abstraction $\lambda x.s$, we claim $\top \vdash \lambda x.s : *$ has a unique derivation:
  By the inductive hypothesis, we have a unique derivation for $\top, x : \{*\} \vdash s : *$.
  Then, by the abstraction rule, we can produce a derivation for $\top \vdash \lambda x.s : \{*\} \to *$.
  But since $*$ interprets as $\{*\} \to *$, we can use this derivation with the consequence type $*$.
  And, by Subject Construction, this is the unique way such a derivation can be formed.
- For an application pq, we claim $\top \vdash pq : *$ has a unique derivation:
  By the inductive hypothesis, we have unique derivations for $\top \vdash p : *$ and $\top \vdash q : *$.
  But since $*$ interprets as $\{*\} \to *$, we can use our derivation for p with type $\{*\} \to *$.

Then, our derivation for q is exactly the singleton conjunction $\forall c \in \{*\}.\ \top \vdash q : c$.
So, by the application rule, we can produce a derivation for $\top \vdash pq : *$.

By Subject Construction, any derivation must assign a type to the subterm p.
But since $*$ is the unique type, we could only choose $\top \vdash p : *$, and then $\top \vdash q : *$.
But since these sub-derivations are unique, our derivation is also unique.

In particular, for any closed term t, we have a unique derivation for $\top \vdash t : *$.
But any context over an empty set of variables is empty, so this choice of context is unique.
And $*$ is the unique type, so this entire sequent is unique for the closed term t.
Therefore, there is a unique derivation for every closed term. □

Example 2.3.3. While we can simply assign a recursive type to a badly-behaved term, our type system also allows us some flexibility to defer this choice. In a non-simple (but potentially inductive) environment containing some type of the form $\{a \to b,\ a\} \to b$, we have the following derivation for the self-application function $\omega \equiv \lambda x.xx$:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \to b \in \{a \to b,\ a\}}{x : \{a \to b,\ a\} \vdash x : a \to b}\ (\text{var}) \quad \dfrac{a \in \{a \to b,\ a\}}{x : \{a \to b,\ a\} \vdash x : a}\ (\text{var})}{x : \{a \to b,\ a\} \vdash xx : b}\ (\text{app})}{\vdash \lambda x.xx : \{a \to b,\ a\} \to b}\ (\text{abs})$$

Theorem 2.4. Subject Reduction

At any beta reduction step $(\lambda x.s)q \to_\beta s[q/x]$,
Every derivation for the redex $\Gamma \vdash (\lambda x.s)q : c$
Induces a derivation for the reduct $\Gamma \vdash s[q/x] : c$.

Proof. By Subject Construction, any derivation for a redex $(\lambda x.s)q$ has the form:

$$\dfrac{\dfrac{\dfrac{\phi}{\Gamma,\ x : A \vdash s : b}}{\Gamma \vdash \lambda x.s : A \to b}\ (\text{abs}) \quad \dfrac{\psi_a \text{ for } a \in A}{\forall a \in A.\ \Gamma \vdash q : a}}{\Gamma \vdash (\lambda x.s)q : b}\ (\text{app})$$

Now, we construct a derivation for the reduct $s[q/x]$ by starting from the sub-derivation ϕ for the subterm s and replacing each occurrence of the variable x with the corresponding sub-derivation ψ for the subterm q. We argue by structural induction on s that $\Gamma \vdash s[q/x] : b$.

- For the substituted variable x, we claim $\Gamma \vdash q : b$.
  By Subject Construction, ϕ must have used the variable rule.
  Therefore, $b \in A$ so we have a sub-derivation ψ_b proving $\Gamma \vdash q : b$.
- For a distinct variable y, we claim $\Gamma \vdash y : b$.
  By Subject Construction, ϕ must use the variable rule.
  By definition, $(\Gamma, x : A)(y) = \Gamma(y)$ since x is distinct from y.
  Therefore, $b \in \Gamma(y)$ so by the variable rule we have $\Gamma \vdash y : b$.

- For an abstraction $\lambda y.t$, we claim $\Gamma \vdash (\lambda y.t)[q/x] : b$.
  By Subject Construction, ϕ must use the abstraction rule.
  Therefore, we have a derivation for $\Gamma, x : A, y : \mathsf{Dom}(b) \vdash t : \mathsf{cod}(b)$.
  By the inductive hypothesis, we have a derivation for $\Gamma, y : \mathsf{Dom}(b) \vdash t[q/x] : \mathsf{cod}(b)$.
  Then by the abstraction rule, we can produce a derivation for $\Gamma \vdash \lambda y.t[q/x] : b$.
- For an application uv, we claim $\Gamma \vdash (uv)[q/x] : b$.
  By Subject Construction, ϕ must use the application rule.
  Therefore, we have some type $f \equiv C \to b$ with a derivation for $\Gamma, x : A \vdash u : f$.
  Additionally, for each element $c \in C$, we have a derivation for $\Gamma, x : A \vdash v : c$.
  By the inductive hypothesis, we have a derivation for $\Gamma \vdash u[q/x] : f$.
  Additionally, for each element $c \in C$, we have a derivation for $\Gamma \vdash v[q/x] : c$.
  Then by the application rule, we have a derivation for $\Gamma \vdash (u[q/x])(v[q/x]) : b$

□

Theorem 2.5. Subject Expansion

At any beta reduction step $(\lambda x.s)q \to_\beta s[q/x]$,
Every derivation for the reduct $\Gamma \vdash s[q/x] : c$
Assigns a set of types D to the substitution occurrences of q.
If our type environment has an element with interpretation $D \to c$,
then there is a derivation for the redex $\Gamma \vdash (\lambda x.s)q : c$.

Proof. Say that a factorization for our reduct $s[q/x]$ consists of:

1. A set of types $D \subseteq \mathcal{T}$
2. A family of derivations $\Gamma \vdash q : d$ for $d \in D$
3. An inner derivation for $\Gamma, x : D \vdash s : c$

We argue by induction that any derivation for our reduct $\Gamma \vdash s[q/x] : c$ has a factorization:

- For the substituted variable x, we have a derivation for $\Gamma \vdash x[q/x] : c$.
  We select the singleton set $D \equiv \{c\}$, with our given derivation already having term q.
  Then, by the variable rule, we obtain an inner derivation $\Gamma, x : \{c\} \vdash x : c$.
- For a distinct variable y, we have a derivation for $\Gamma \vdash y[q/x] : c$.
  We select the empty set $D \equiv \varnothing$, requiring no derivations for q.
  By Subject Construction, the constraint $c \in \Gamma(y)$ must be satisfied.
  Then, by the variable rule, we have an inner derivation for $\Gamma, x : \varnothing \vdash y : c$.
- For an abstraction $\lambda y.t$, we have a derivation for $\Gamma \vdash (\lambda y.t)[q/x] : A \to b$.
  By Subject Construction, we have a subderivation for $\Gamma, y : A \vdash t[q/x] : b$.
  Then, by the inductive hypothesis, we have a factorization for this derivation.
  Applying the abstraction rule to our inner derivation, we obtain $\Gamma, x : D \vdash \lambda y.t : b$.
  And since y does not occur in q, it can be removed from our family of derivations.
- For an application uv, we have a derivation for $\Gamma \vdash uv[q/x] : c$.
  By Subject Construction, we have some $f = A \to c$ with $\Gamma \vdash u[q/x] : f$.
  Additionally, for each element $a \in A$, we have a derivation for $\Gamma \vdash v[q/x] : a$.

By the inductive hypothesis, we have factorizations for each of these derivations.
Applying the application rule to our inner derivations, we obtain $\Gamma, x : D \vdash uv : c$.
And, by taking the union of all our families, we obtain a new family of derivations for q.

Once we have a factorization for our reduct, we can attempt to produce a complete a derivation for our redex. If we can find a type $f \equiv D \to c$, we can apply the abstraction rule to our inner derivation, yielding a derivation for $\Gamma \vdash \lambda x.s : f$. Then, we can use the application rule with our family of derivations to produce our final derivation for $\Gamma \vdash (\lambda x.s)q : c$. □

3 Inference

In this section, we explore principal type inference from a computational perspective. It's not too difficult to construct principal type derivations algorithmically, following the derivation rules while making as few assumptions as possible. In the next section, we will more rigorously justify these derivations status as principal.

Definition 3.1. We construct principal type derivations according to the following algorithm:

The principal typing algorithm takes in a lambda term t and produces a type environment $\mathcal{T}$ along with a derivation for the given term t in the chosen environment $\mathcal{T}$. Throughout the algorithm, we maintain an incomplete derivation tree with a single unproven sequent, gradually filling in new branches of our tree, and simultaneously adding more structure to our environment. Initially, we start with only the root of our tree: an unproven sequent $\vdash t : b$ in the type environment generated by a single type b, as in example 2.1.2. Then, we proceed by repeatedly pattern matching on the unique unproven sequent:

- For an unproven application $\Gamma \vdash pq : b$,

  By Subject Construction, we must select some type of the form $A \to b$, whose codomain is our current consequence type b. In order to satisfy this equational constraint while making as few assumptions as possible, we ought to introduce a new type f to our environment, setting $\mathsf{cod}(f) \equiv b$. However, we still must provide an interpretation for the domain $\mathsf{Dom}(f) \subseteq \mathcal{T}$.

  The more types there are in $\mathsf{Dom}(f)$, the more subderivations we will need to provide for our argument q. However, if we eventually use f as the type of an abstraction, our new variable will only be useable with the types contained in $\mathsf{Dom}(f)$.

  Without trying to predict the future, there is only one sensible choice we can make: we initialize f with an empty domain $\mathsf{Dom}(f) \equiv \varnothing$, but reserve the right to insert more elements as they are needed. Thus, we extend our tree with the rule:

  $$\frac{\Gamma \vdash p : f \qquad \overline{\forall a \in \varnothing.\, \Gamma \vdash q : a}}{\Gamma \vdash pq : b}\ (\text{app})$$

  At the moment, the argument q is unused, vacuously satisfying our empty set of premises. Luckily, this leaves us with exactly one unproven sequent for our function p.

- For an unproven abstraction $\Gamma \vdash \lambda x.s : f$,

  By Subject Construction, we must use the abstraction rule, where all of our choices are fully determined. Thus, we continue onwards into the body of the abstraction s:

$$\frac{\Gamma,\, x : \mathsf{Dom}(f) \vdash s : \mathsf{cod}(f)}{\Gamma \vdash \lambda x.s : f}\ (\text{abs})$$

- For an unproven variable $\Gamma \vdash x : b$,

  By Subject Construction, we must ensure that $b \in \Gamma(x)$.

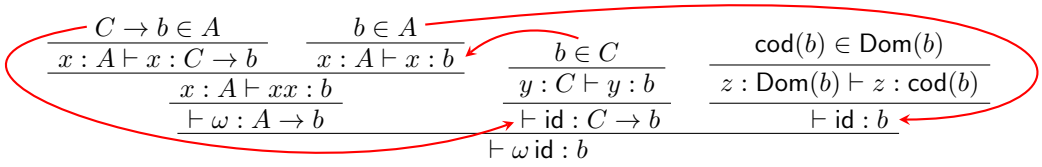
  Retracing our steps, there must have been some abstraction rule where x was added to our context. But in the abstraction rule, we must use the domain of our consequence type as the new component for our context. Thus, the component $\Gamma(x)$ was originally defined as $\mathsf{Dom}(c)$ for some consequence type c.

  Therefore, in order to justify our sequent, we assert $b \in \mathsf{Dom}(c)$ as a new constraint on our type environment. If c was introduced during an application rule, then adding an element to our domain will require us to prove an additional premise in the argument of that application. This becomes our next unproven sequent.

  Otherwise, c must have been one of the types we started out with in our initial environment, as either the root consequence type or one of its codomains. In this case, every sequent has already been proven, and our algorithm can successfully terminate.

This algorithm defines an iterative process, which does not always terminate. However, it does have well-defined limiting behaviour, incrementally building up a more completed proof. Thus, we can regard its output as a potentially infinite derivation.

Example 3.1.1. The principal type derivation for $\omega\,\mathsf{id} \equiv (\lambda x.xx)(\lambda y.y)$ is constructed as follows:



First, we follow the leftmost path through our term, leading to the variable constraint $C \to b \in A$. But since we declared that each element $a \in A$ must be a valid type for id, we now must initiate a new subderivation for $\mathsf{id} : C \to b$, in the middle column. This leads us to conclude that $b \in C$, but when we introduced C during our first pass through, we declared that each of its elements can be used for the variable x in the context where $x : A$. Therefore, another short subderivation tells us that $b \in A$. This leads to our final derivation in the rightmost column, where we show $\mathsf{id} : b$, culminating in the variable constraint $\mathsf{cod}(b) \in \mathsf{Dom}(b)$. But since we did not introduce $\mathsf{Dom}(b)$ ourselves, this does not lead to any more callbacks.

Throughout the runtime of our algorithm, we introduce new types and elementhood constraints to our environment. Looking back at our derivation, we can recover a record of these changes. Thus, the principal type environment for $\omega\,\mathsf{id}$ can be described in our original notation as:

$$\begin{aligned} a &\equiv A \to b \quad \text{where } A \equiv \{c, b\} \\ c &\equiv C \to b \quad \text{where } C \equiv \{b\} \\ b &\equiv D \to e \quad \text{where } D \equiv \{e\} \\ e &\equiv \text{free} \end{aligned}$$

This algorithm is reminiscent of a lazy evalution strategy. Rather than producing a full derivation tree up front, the subderivations for an application argument are deferred until the last possible moment, when the occurence of some matching variable invokes them. Taking this idea further, we introduce an alternative system of inference rules, where this order of operations is hardcoded in.

In order to retrace our steps from a variable occurence back to its matching application argument, we annotate each sequent with our accumulated list of application arguments. We write $(A \cap t)$ for the set of types A, with the additional condition that any element $a \in A$ will have to be a valid type for the term t. For clarity, we distingush an unannotated set with a checkmark $(A \cap \checkmark)$.

Definition 3.2. A trace internal to $\mathcal{T}$ is a potentially infinite proof according to the rules:

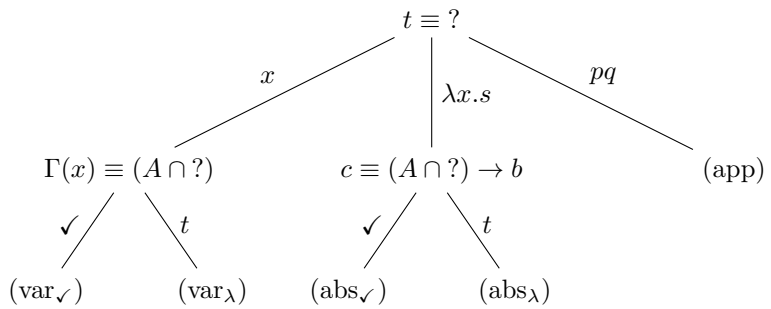
$$\frac{\Gamma(x) \equiv A \cap \checkmark \qquad a \in A}{\Gamma \vdash x : a}\,(\mathrm{var}_\checkmark) \qquad \frac{\Gamma(x) \equiv A \cap t \qquad a \in A \qquad \Gamma \vdash t : a}{\Gamma \vdash x : a}\,(\mathrm{var}_\lambda)$$

$$\frac{\Gamma, x : (A \cap \checkmark) \vdash s : b}{\Gamma \vdash \lambda x.s : (A \cap \checkmark) \to b}\,(\mathrm{abs}_\checkmark) \qquad \frac{\Gamma, x : (A \cap t) \vdash s : b}{\Gamma \vdash \lambda x.s : (A \cap t) \to b}\,(\mathrm{abs}_\lambda)$$

$$\frac{\Gamma \vdash p : (A \cap q) \to b}{\Gamma \vdash pq : b}\,(\mathrm{app})$$

It can be helpful to think of a trace as a sequence of steps during the execution of a computer program. Disregarding the side conditions in the variable rule, each of these rules has at most one premise. Thus, we can think of these inference rules as dictating the next state of our program.

Although we now have five inference rules instead of three, there is still a clean seperation between variables, abstractions, and applications. Thus, we obtain a new variant of Subject Construction, where the appropriate rule for a given annotated sequent $\Gamma \vdash t : c$ is determined by inspecting its term and annotations as follows:



Example 3.2.1. The principal derivation for $\omega\,\mathsf{id}$ can be translated into a trace:

$$
\dfrac{\dfrac{\dfrac{C \to b \in A \quad \dfrac{\dfrac{b \in C \quad \dfrac{b \in A \quad \dfrac{\dfrac{\mathsf{cod}(b) \in \mathsf{Dom}(b)}{x : (A \cap \mathsf{id}),\, y : (C \cap x),\, z : (\mathsf{Dom}(b) \cap \checkmark) \vdash z : \mathsf{cod}(b)}\,(\text{var}_\checkmark)}{x : (A \cap \mathsf{id}),\, y : (C \cap x) \vdash \mathsf{id} : b}\,(\text{abs}_\checkmark)}{x : (A \cap \mathsf{id}),\, y : (C \cap x) \vdash x : b}\,(\text{var}_\lambda)}{x : (A \cap \mathsf{id}),\, y : (C \cap x) \vdash y : b}\,(\text{var}_\lambda)}{x : (A \cap \mathsf{id}) \vdash \mathsf{id} : (C \cap x) \to b}\,(\text{abs}_\lambda)}{x : (A \cap \mathsf{id}) \vdash x : (C \cap x) \to b}\,(\text{var}_\lambda)}{x : (A \cap \mathsf{id}) \vdash xx : b}\,(\text{app})}{\dfrac{\vdash \omega : (A \cap \mathsf{id}) \to b}{\vdash \omega\,\mathsf{id} : b}\,(\text{app})}\,(\text{abs}_\lambda)
$$

Derivations and traces behave very similarly, especially in a principal type environment as above. However, in a non-principal type environment, traces tend to be more permissive than derivations. For example, consider the lambda term $\mathsf{id}\,\mathsf{id}$. In any environment containing a type of the form $\{a \to a,\, b \to b\} \to a \to a$, we have a non-standard derivation with an extra, unused subderivation:

$$
\dfrac{\dfrac{\dfrac{a \to a \in \{a \to a,\, b \to b\}}{x : \{a \to a,\, b \to b\} \vdash x : a \to a}\,(\text{var})}{\vdash \lambda x.x : \{a \to a,\, b \to b\} \to a \to a}\,(\text{abs}) \quad \dfrac{\dfrac{a \in \{a\}}{y : \{a\} \vdash y : a}\,(\text{var})}{\vdash \lambda y.y : a \to a}\,(\text{abs}) \quad \dfrac{\dfrac{b \in \{b\}}{y : \{b\} \vdash y : b}\,(\text{var})}{\vdash \lambda y.y : b \to b}\,(\text{abs})}{\vdash (\lambda x.x)(\lambda y.y) : a \to a}\,(\text{app})
$$

Meanwhile for traces, this unused subderivation would never be reached, since there is no annotation callback which mandates its existence. Since we never rely on the fact that $b \to b$ is a valid type for id, we have a similar non-standard trace in the broader class of environments which contain a type of the form $\{a \to a,\, c\} \to a \to a$.

Lemma 3.3. Componentwise Weakening

If $\Gamma(x) \subseteq \Delta(x)$ for all components x,
Then every trace for a sequent $\Gamma \vdash t : c$
Induces a trace for the sequent $\Delta \vdash t : c$

Proof. We proceed by structural co-induction on our sequent $\Gamma \vdash t : c$.

- For an application pq, we assume $\Gamma \vdash pq : b$.
  By Subject Construction, we must have $\Gamma \vdash p : (A \cap q) \to b$.
  Then, by the inductive hypothesis, $\Delta \vdash p : (A \cap q) \to b$.
  Therefore, by the application rule, $\Delta \vdash pq : b$.
- For an abstraction $\lambda x.s$, we assume $\Gamma \vdash \lambda x.s : (A \cap m) \to b$.
  By Subject Construction, we must have $\Gamma, x : (A \cap m) \vdash s : b$.
  Then, by the inductive hypothesis, $\Delta, x : (C \cap m) \vdash s : b$ for any superset $C \supseteq A$.
  In particular, $A \supseteq A$ so we have $\Delta, x : (A \cap m) \vdash s : b$.
  Therefore, by the appropriate abstraction rule, $\Delta \vdash \lambda x.s : (A \cap m) \to b$.

- For an annotated variable x, we assume $\Gamma \vdash x : a$ with $\Gamma(x) \equiv A \cap t$.
  By Subject Construction, we must have $a \in A$ and $\Gamma \vdash t : a$.
  Then, by the inductive hypothesis, $\Delta \vdash t : a$.
  And since $\Gamma \subseteq \Delta$, we know that $a \in \Gamma(x) \subseteq \Delta(x)$.
  Therefore, by the annotated abstraction rule, $\Delta \vdash x : a$.
- For an unannotated variable x, we assume $\Gamma \vdash x : a$ with $\Gamma(x) \equiv A \cap \checkmark$.
  By Subject Construction, the constraint $a \in A$ must hold.
  Then since $\Gamma \subseteq \Delta$, we know that $a \in \Gamma(x) \subseteq \Delta(x)$.
  Therefore, by the annotated abstraction rule, $\Delta \vdash x : a$.

□

Theorem 3.4. Derivation $\implies$ Trace

Every derivation contains a series of sequents which assemble into a trace.

Proof. We decompose our derivation into a set of subderivations, corresponding to the annotations on our sequent. This way, we can maintain the validity of our annotations as an invariant.

Say that a "state" for an annotated sequent $\Gamma \vdash t : c$ consists of:

- A main derivation for the sequent $\Gamma \vdash t : c$ with its annotations removed.
- For any annotation $(A \cap t)$ in Γ or c, and any $a \in A$, a derivation for $\Gamma \vdash t : a$.

Now, given any derivation ϕ for a closed sequent $\vdash t : c$, we can form a state for the sequent $\vdash t : c$ using ϕ as our main derivation, and with no auxillary derivations since we do not yet have any annotations. Then, we argue by induction that states can be evaluated in accordance with the inference rules of traces, such that a state for any conclusion sequent can be transformed into a state for the associated premise.

- For an application $\Gamma \vdash pq : b$,
  We expand our main derivation into its immediate subderivations.
  Thus, we obtain $\Gamma \vdash p : A \to b$ and $\Gamma \vdash q : a$ for each $a \in A$.
  The former becomes the main derivation for our next sequent $\Gamma \vdash p : (A \cap q) \to b$.
  And the latter become auxillary derivations for our new annotation $(A \cap q)$.
- For either abstraction $\Gamma \vdash \lambda x.s : (A \cap m) \to b$,
  We replace our main derivation with its immediate subderivation.
  Thus, we obtain our next main derivation $\Gamma, x : A \vdash s : b$.
  Next, we apply weakening to all of our auxillary derivations.
  Now, we have a state for our next sequent $\Gamma, x : (A \cap m) \vdash s : b$.
- For an annotated variable $\Gamma \vdash x : b$ with $\Gamma(x) \equiv A \cap t$,
  By Subject Construction, our main derivation implies that $b \in A$.
  Therefore, we have an auxillary derivation for $\Gamma \vdash t : b$.
  We make this our new main derivation, while retaining our auxillary derivations.
  Now, we have a state for our next sequent $\Gamma \vdash t : b$.

- For an unannotated variable $\Gamma \vdash x : b$ with $\Gamma(x) \equiv A \cap \checkmark$,
  By Subject Construction, our main derivation implies that $b \in A$.
  Therefore, we can complete our trace with the unannotated variable rule.

By iteratively applying this procedure, we can extract an arbitrarily long prefix of a trace. Thus, in the limit, every derivation can be transformed into a trace. □

4 Principality

In the simply typed lambda calculus, the principal type derivations are defined using most general unifiers, motivated from the perspective of type substitutions. The idea is that a type derivation is principal when every other derivation looks like it, with some substitution applied to the type variables. To formulate an analogous notion of type substitution for our derivations, we consider the homomorphisms between type environments. However, our two equivalent conceptions of type environments induce distinct notions of homomorphism:

Definition 4.1. A coalgebra homomorphism is a set function $H : \mathcal{T} \to \mathcal{U}$ such that

$$\mathsf{cod}(H(f)) = H(\mathsf{cod}(f))$$
$$\mathsf{Dom}(H(f)) = H[\mathsf{Dom}(f)]$$

A coalgebra homomorphism preserves the domain function $\mathsf{Dom} : \mathcal{T} \to \mathcal{P}(\mathcal{T})$.
Each input type gets sent to an output with an exactly matching interpretation.

Definition 4.2. A graph homomorphism is a set function $H : \mathcal{T} \to \mathcal{U}$ such that

$$\mathsf{cod}(H(f)) = H(\mathsf{cod}(f))$$
$$a \in \mathsf{Dom}(f) \Rightarrow H(a) \in \mathsf{Dom}(H(f))$$

A graph homomorphism preserves the domain relation $(x, y) \mapsto x \in \mathsf{Dom}(y)$.
Related elements stay related, but new elements can be added to a domain set.

Notably, every coalgebra homomorphism is a graph homomorphism. Intuitively, coalgebra maps behave like embeddings, while graph homomorphisms form a much more familiar kind of morphism. Thus, the category of type environments and graph homomorphisms has a very rich structure, where many of our constructions can be formalized.

Example 4.2.1. In the category of type environments and graph homomorphisms, the trivial type environment $* \equiv \{*\} \to *$ is the terminal object.

Example 4.2.2. A graph homomorphism from the single-variabled type environment to any other type environment $\mathcal{T}$ is exactly a way of picking out an element of $\mathcal{T}$. Thus, the single-variabled type environment is free on one generator.

Theorem 4.3. Graph Homomorphisms Preserve Traces

For any graph homomorphism $H : \mathcal{T} \to \mathcal{U}$,
Each trace in $\mathcal{T}$ on an arbitrary sequent $\Gamma \vdash t : c$
Induces a trace in $\mathcal{U}$ for the corresponding sequent $H[\Gamma] \vdash t : H(c)$.

Proof. Graph homomorphisms only preserve the relatedness of elements.
Thus, for $f \equiv A \to b \in \mathcal{T}$, we do not always have $H(f) \equiv H[A] \to H(b)$.
Instead, we only know that whenever $a \in \mathsf{Dom}(f)$, we have $H(a) \in \mathsf{Dom}(H(f))$.
Therefore, $H(f) \equiv C \to H(b)$ for some superset $C \supseteq H[A]$.
For convenience, we write C for this superset of $H(A)$ throughout.

We proceed by induction on our sequent:

- In the application case, we assume $\Gamma \vdash pq : b$
  By Subject Construction, our immediate subtrace is for $\Gamma \vdash p : (A \cap q) \to b$.
  By the inductive hypothesis, we obtain a trace for $H[\Gamma] \vdash p : (C \cap q) \to H(b)$.
  Therefore, by the application rule, we have $H[\Gamma] \vdash pq : H(b)$.

- In either abstraction case, we assume $\Gamma \vdash \lambda x.s : (A \cap m) \to b$.
  By Subject Construction, our immediate subtrace is for $\Gamma, x : (A \cap m) \vdash s : b$.
  By the inductive hypothesis, we obtain a trace for $H[\Gamma], x : (H[A] \cap m) \vdash s : H(b)$.
  Then, by the weakening lemma, we can obtain $H[\Gamma], x : (C \cap m) \vdash s : H(b)$.
  Therefore, by the appropriate abstraction rule, $H[\Gamma] \vdash \lambda x.s : (C \cap m) \to H(b)$.

- For an annotated variable, we assume $\Gamma \vdash x : b$ with $\Gamma(x) \equiv A \cap t$.
  By Subject Construction, we have $b \in A$ and a subtrace for $\Gamma \vdash t : b$.
  Then, by the inductive hypothesis, we obtain a trace for $H[\Gamma] \vdash t : H(b)$.
  And, by definition of a graph homomorphism, we have $H(b) \in H[A]$.
  Therefore, by the annotated variable rule, we have $H[\Gamma] \vdash x : H(b)$.

- For an unannoted abstraction, we assume $\Gamma \vdash x : b$ with $\Gamma(x) \equiv A \cap \checkmark$.
  By Subject Construction, we know that $b \in A$ holds.
  Then, by definition of a graph homomorphism, we have $H(b) \in H[A]$.
  Therefore, by the unannotated variable rule, we have $H[\Gamma] \vdash x : b$.

□

Theorem 4.4. Coalgebra Homomorphisms Preserve Derivations

For any coalgebra homomorphism $H : \mathcal{T} \to \mathcal{U}$,
Each derivation in $\mathcal{T}$ on an arbitrary sequent $\Gamma \vdash t : c$
Induces a derivation in $\mathcal{U}$ for the corresponding sequent $H[\Gamma] \vdash t : H(c)$.

Proof. We show that the characteristic proposition $[\![\Gamma \vdash t : c]\!]$ is preserved.

First, we argue that formula in regular logic (consisting of conjunction and existential quantification) are preserved by either kind of homomorphism. Formally, we show that if $\mathcal{T} \vDash \phi(c_1, \ldots, c_n)$ then $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n))$.

- For atomic formula,
  Even for a graph homomorphism, if $x \in \mathsf{Dom}(y)$ then $H(x) \in \mathsf{Dom}(H(y))$.
  Similarly for equalities, if $x = \mathsf{cod}(y)$ then $H(x) = \mathsf{cod}(H(y))$.

- For conjunction,
  Assume $\mathcal{T} \vDash \phi(c_1, \ldots, c_n) \wedge \psi(c_1, \ldots, c_n)$.
  By definition, $\mathcal{T} \vDash \phi(c_1, \ldots, c_n)$ and $\mathcal{T} \vDash \psi(c_1, \ldots, c_n)$.

By the inductive hypothesis, $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n))$ and $\mathcal{U} \vDash \psi(H(c_1), \ldots, H(c_n))$.
Therefore, $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n)) \wedge \psi(H(c_1), \ldots, H(c_n))$.

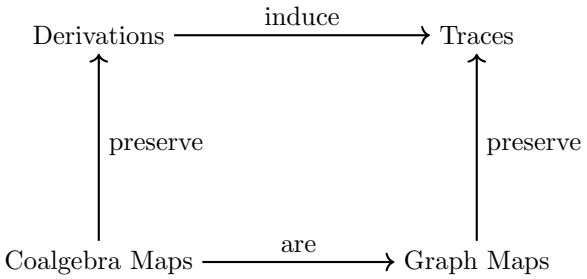
- For existential quantifiers,
  Assume $\mathcal{T} \vDash \exists x.\, \phi(c_1, \ldots, c_n, x)$.
  By definition, there is some $c_{n+1} \in \mathcal{T}$ such that $\mathcal{T} \vDash \phi(c_1, \ldots, c_n, c_{n+1})$.
  By the inductive hypothesis, $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n), H(c_{n+1}))$.
  Therefore, $\mathcal{U} \vDash \exists x.\, \phi(H[\vec{c}], x)$ with $H(c_{n+1})$ as the witness for x.

Next, we use the specific properties of coalgebra homomorphisms to prove the case for a "bounded" universal quantifier, as occurs in the argument premise of the application rule:

Assume $\mathcal{T} \vDash \forall x \in \mathsf{Dom}(f).\, \phi(c_1, \ldots, c_n, x)$.
By definition, $\mathcal{T} \vDash \phi(c_1, \ldots, c_n, a)$ for all $a \in \mathsf{Dom}(f)$.
By the inductive hypothesis, $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n), H(a))$ for all $a \in \mathsf{Dom}(f)$.
But in a coalgebra homomorphism, $H[\mathsf{Dom}(f)] = \mathsf{Dom}(H(f))$.
So, we equivalently have $\mathcal{U} \vDash \phi(H(c_1), \ldots, H(c_n), a)$ for all $a \in \mathsf{Dom}(H(f))$.
Therefore, $\mathcal{U} \vDash \forall x \in \mathsf{Dom}(H(f)).\, \phi(H(c_1), \ldots, H(c_n), x)$.

Since any characteristic proposition $[\![\Gamma \vdash t : c]\!]$ has this form, we know that derivations transport along coalgebra homomorphisms. □

Thus, our two notions of homomorphism correspond to our two type assignment systems. Coalgebra homomorphisms are a special case of graph homomorphisms, where each domain set is perfectly preserved. Similarly, derivations resemble traces with an additional exactness condition, where every elementhood domain is used by some subderivation. Informally, the picture we have so far is:



Theorem 4.5. Principal Means Initial

For any environment $\mathcal{T}$ along with any trace ϕ for a lambda term t,
There is a unique graph homomorphism from the principal environment for t into $\mathcal{T}$
Such that the principal derivation is mapped onto ϕ.

Proof. Since the principal type environment is defined by an iterative process (as a directed colimit), we define an increasing sequence of partial functions for each step of the iteration.

First, we ensure the root sequent $\vdash t : c$ of the principal derivation is sent to the root sequent of ϕ, by sending the consequence type c to the consequence type at the root of ϕ. But then, since every graph homomorphism must preserve codomains, we have uniquely determined our map for the first

iteration step, since we started with the type environment generated by a single variable. Now, we proceed by induction, maintaining a preserved prefix of our trace as an invariant:

- For an application pq,
  We have the principal sequent $\Gamma \vdash pq : b$ sent to the ϕ-sequent $H[\Gamma] \vdash pq : H(b)$.
  By Subject Construction, ϕ uses the application rule next with $H[\Gamma] \vdash p : C \to H(b)$.
  In the next iteration step, we must send the freshly introduced type $H(A \to b) \equiv C \to H(b)$.
  Since A is introduced empty and $\mathsf{cod}(H(A \to b)) \equiv H(b)$, H is still a graph homomorphism.

- For either abstraction $\lambda x.s$,
  Our principal sequent $\Gamma \vdash \lambda x.s : (A \cap m) \to b$ is sent to $H[\Gamma] \vdash \lambda x.s : (H[A] \cap m) \to H(b)$.
  By Subject Construction, ϕ uses an abstraction rule next with $H[\Gamma], x : H[A] \cap m \vdash s : H(b)$.
  But the graph homomorphism H already preserves this sequent.
  Since nothing new is introduced, we can proceed.

- For an unannotated variable x,
  Our principal sequent $\Gamma \vdash x : b$ is sent to the ϕ-sequent $H[\Gamma] \vdash x : H(b)$.
  By Subject Construction, ϕ uses the unannotated variable rule, with $H[\Gamma](x) \equiv H[A] \cap \checkmark$.
  Since $H(b) \in H[A]$, we can safely add the constraint $b \in A$ to our environment.

- For an annotated variable x,
  Our principal sequent $\Gamma \vdash x : b$ is sent to the ϕ-sequent $H[\Gamma] \vdash x : H(b)$.
  Similar as to unannotated variables, we can safely introduce our new constraint.
  By Subject Construction, ϕ uses the annotated variable rule next, with $H[\Gamma] \vdash t : H(b)$.
  But since we already map b to $H(b)$, we continue with our trace for t.

□

Another way to interpret this result, is as an adjunction between the category of type environments and the category of presheaves on the discrete set of lambda terms.

$$\mathsf{TypeEnv} \underset{\mathsf{Traces}}{\overset{\mathsf{Free}}{\perp}} [\Lambda, \mathsf{Set}]$$

As shown in Theorem 4.3, traces are preserved by graph homomorphisms, so by currying, we obtain a functor $\mathsf{Traces} : \mathsf{TypeEnv} \to [\Lambda, \mathsf{Set}]$. Then, by the universal property of the co-completion, we have a unique colimit preserving functor $\mathsf{Free} : [\Lambda, \mathsf{Set}] \to \mathsf{TypeEnv}$ sending the representable objects (the terms) to their principal type environments.

$$\mathsf{TypeEnv}(\mathsf{Free}(t),\, \mathcal{T}) \cong \mathsf{Traces}(\mathcal{T})(t) \cong [\Lambda, \mathsf{Set}](t,\, \mathsf{Traces}(\mathcal{T}))$$

Finally, our initiality condition establishes an isomorphism between morphisms out of the free type environment and elements of the set of traces, by associating each trace with the unique morphism that maps the principal type derivation onto it.

5 Normalization

Throughout the evaluation of a trace, we unpack the structure of our term into different parts of our sequent, seperating application arguments off into type annotations and moving bound variables into our context. Unlike most type systems where derivation trees can branch off in many different directions, all the information traces can retain about the original term must be passed along into the next sequent. Putting these pieces back together, we can reconstruct a closed lambda term which the sequent is operating on, although this is not always exactly the same as the closed term we started with.

Definition 5.1. The amalgamated term of an annotated sequent $\Gamma \vdash t : c$ is given as $\Gamma \blacktriangleright (t \blacktriangleleft c)$, as can be computed using the following two inductive definitions:

First, for $t \blacktriangleleft c$, we shuffle the annotations in c back onto t.
$s \blacktriangleleft (A \cap t) \to b \equiv (st) \blacktriangleleft b$
$s \blacktriangleleft (A \cap \checkmark) \to b \equiv s$

Then, for $\Gamma \blacktriangleright u$, we provide a semantics for each variable in Γ.
$\Gamma, x : (A \cap t) \blacktriangleright u \equiv \Gamma \blacktriangleright u[t/x]$
$\Gamma, x : (A \cap \checkmark) \blacktriangleright u \equiv \Gamma \blacktriangleright \lambda x.u$
$\blacktriangleright u \equiv u$

Example 5.1.1. At the beginning of any trace, our context is empty and our type is unannotated, so our amalgamated term is exactly just our term.

Example 5.1.2. In a trace for $\omega\,\mathsf{id}$, we eventually reach some sequent of the form:

$$x : (C \cap \mathsf{id}), y : (D \cap x) \vdash \mathsf{id} : (A \cap \checkmark) \to b$$

Here, our type $(A \cap \checkmark) \to b$ is unannotated, so $\mathsf{id} \blacktriangleleft (A \cap \checkmark) \to b$ is just id.
Then, each component of our context is annotated, so we substitute $[x/y]$ and then $[\mathsf{id}/x]$.
However, id is already closed, so our amalgamated term is just id.

Lemma 5.2. Head Preservation of Amalgamation

For any context Γ and annotated type c, the operation $C[X] \equiv \Gamma \blacktriangleright (X \blacktriangleleft c)$ places a substitution instance of X at the head, with the substitution acting on exactly the annotated variables in Γ.

Proof. First, $X \blacktriangleleft c \equiv Xt_1 \dots t_n$ for some $t_1, \dots, t_n$.
Here, X occurs at the head with no substitution applied.

Then, $\Gamma \blacktriangleright (Xt_1 \dots t_n)$ performs a sequence of operations.
Each unannotated variable results in an abstraction, keeping X at the head.
Each annotated variable results in a substitution, which distributes down to X. □

Lemma 5.3. Substitution Invariance of Amalgamation

If $\Gamma(x) \equiv A \cap t$, then $\Gamma \blacktriangleright s \equiv \Gamma \blacktriangleright s[t/x]$.

Proof. We proceed by induction on the components of Γ.

- In the case for $\Gamma, x : (A \cap t)$ where the context ends with an entry for x:

$$\begin{array}{c} \Gamma, x : (A \cap t) \blacktriangleright s \equiv \Gamma \blacktriangleright s[t/x] \\ \Vert\vert \\ \Gamma, x : (A \cap t) \blacktriangleright s[t/x] \equiv \Gamma \blacktriangleright s[t/x][t/x] \end{array}$$

  Here, the vertical equality holds because in a well-formed context, x does not occur in t.

- In the case for $\Gamma, y : (C \cap \checkmark)$ where we already have $\Gamma(x) \equiv A \cap t$:

$$\begin{array}{c} \Gamma, y : (C \cap \checkmark) \blacktriangleright s \equiv \Gamma \blacktriangleright \lambda y.s \overset{\text{IH}}{\equiv} \Gamma \blacktriangleright (\lambda y.s)[t/x] \\ \Vert\vert \\ \Gamma, y : (C \cap \checkmark) \blacktriangleright s[t/x] \equiv \Gamma \blacktriangleright \lambda y.(s[t/x]) \end{array}$$

  Here, the vertical equality holds by the definition of substitution for abstractions.

- In the case for $\Gamma, y : (C \cap u)$ where we already have $\Gamma(x) \equiv A \cap t$:

$$\begin{array}{c} \Gamma, y : (C \cap u) \blacktriangleright s \equiv \Gamma \blacktriangleright s[u/y] \overset{\text{IH}}{\equiv} \Gamma \blacktriangleright s[u/y][t/x] \\ \Vert\vert \\ \Gamma, y : (C \cap u) \blacktriangleright s[t/x] \equiv \Gamma \blacktriangleright s[t/x][u/y] \overset{\text{IH}}{\equiv} \Gamma \blacktriangleright s[t/x][u/y][t/x] \end{array}$$

  Here, the vertical equality holds because in a well-formed context, y does not occur in t.

□

Theorem 5.4. Finite Trace $\iff$ Solvable

For any closed term t, traces of t are finite if and only if t is solvable.

Proof. We formalize the exact correspondence between trace evaluation and head reduction.

- In the unannotated variable rule:

$$\frac{\Gamma(x) \equiv A \cap \checkmark \qquad a \in A}{\Gamma \vdash x : a} \ (\text{var}_\checkmark)$$

  Let t denote the amalgamation of the conclusion $\Gamma \blacktriangleright (x \blacktriangleleft a)$.
  By lemma 5.2, t contains a substitution instance of x at the head.
  But this substitution only affects variables which appear annotated in Γ.
  Since x is unannotated in Γ, the head of t is exactly the variable x.
  Therefore, the amalgamated term t is in head normal form.

- In the annotated abstraction rule:

$$\frac{\Gamma, x : (A \cap t) \vdash s : b}{\Gamma \vdash \lambda x.s : (A \cap t) \to b} \ (\text{abs}_\lambda)$$

  Our premise has amalgamated term $(\Gamma, x : (A \cap t)) \blacktriangleright (s \blacktriangleleft b)$.
  By definition of $(\blacktriangleright)$, this simplifies to $\Gamma \blacktriangleright ((s \blacktriangleleft b)[t/x])$.
  But b is well-formed terms in Γ, so x does not occur in b.
  Therefore, we can further simplify to $\Gamma \blacktriangleright (s[t/x] \blacktriangleleft b)$.

Our conclusion has amalgamated term $\Gamma \blacktriangleright (\lambda x.s \blacktriangleleft (A \cap t) \to b)$.
By definition of $(\blacktriangleleft)$, this simplifies to $\Gamma \blacktriangleright ((\lambda x.s)t \blacktriangleleft b)$.

By lemma 5.2, the head subterm for our conclusion is the redex $(\lambda x.s)t$.
Therefore, the amalgamation of our conclusion is not in head normal form.
Further, its single-step head reduct is the amalgamation of our premise.

So far, we have shown that these two rules help us deconstruct the (possibly infinite) number of head reduction steps needed until we reach normal form. The unannotated variable rule tells us this number is zero, by demonstrating that we are already in head normal form. The annotated abstraction rule tells us that this number is the successor of the number for our immediate subtrace, since we perform exactly one step of head reduction to get there.

Now, we will show that our other three rules are irrelevant to this deconstruction process. First, we argue that each of these three rules exactly preserve the amalgamated term.

- In the annotated variable rule:

$$\frac{\Gamma(x) \equiv A \cap t \qquad a \in A \qquad \Gamma \vdash t : a}{\Gamma \vdash x : a} \;(\text{var}_\lambda)$$

  Whenever we use this rule, we know that $\Gamma(x) \equiv A \cap t$.
  Lemma 5.3 tells us that $\Gamma \blacktriangleright s \equiv \Gamma \blacktriangleright s[t/x]$ for any term s.
  Therefore, we can form the chain of equalities:

$$\begin{array}{c} \Gamma \blacktriangleright (x \blacktriangleleft a) \overset{5.3}{\equiv} \Gamma \blacktriangleright ((x \blacktriangleleft a)[t/x]) \equiv \Gamma \blacktriangleright (x[t/x] \blacktriangleleft a[t/x]) \\ \hfill \parallel\!| \\ \Gamma \blacktriangleright (t \blacktriangleleft a) \overset{5.3}{\equiv} \Gamma \blacktriangleright ((t \blacktriangleleft a)[t/x]) \equiv \Gamma \blacktriangleright (t[t/x] \blacktriangleleft a[t/x]) \end{array}$$

  This vertical equality holds because in a well-formed context, x does not occur in t.

- In the unannotated abstraction rule:

$$\frac{\Gamma, x : (A \cap \checkmark) \vdash s : b}{\Gamma \vdash \lambda x.s : (A \cap \checkmark) \to b} \;(\text{abs}_\checkmark)$$

  We claim that $(\Gamma, x : (A \cap \checkmark)) \blacktriangleright (s \blacktriangleleft b) \equiv \Gamma \blacktriangleright (\lambda x.s \blacktriangleleft (A \cap \checkmark) \to b)$.
  Observe that $(A \cap \checkmark) \to b$, and therefore also b, are already out of annotations.
  So, by definition of $(\blacktriangleleft)$, we can simplify $\lambda x.s \blacktriangleleft (A \cap \checkmark) \to b \equiv \lambda x.s$ and $s \blacktriangleleft b \equiv s$.
  But then, by definition of $(\blacktriangleright)$, we have $(\Gamma, x : (A \cap \checkmark)) \blacktriangleright s \equiv \Gamma \blacktriangleright \lambda x.s$.

- In the application rule:

$$\frac{\Gamma \vdash p : (A \cap q) \to b}{\Gamma \vdash pq : b} \;(\text{app})$$

  We claim that $\Gamma \blacktriangleright (p \blacktriangleleft (A \cap q) \to b) \equiv \Gamma \blacktriangleright (pq \blacktriangleleft b)$.
  But by definition of $(\blacktriangleleft)$, $p \blacktriangleleft (A \cap q) \to b \equiv pq \blacktriangleleft b$.

Therefore, at any point in our trace, our amalgamated term is exactly the n-th head reduct of our original term, where n is the number of annotated abstraction steps (abs_λ) we have performed.

Additionally, if we reach the end of our trace with the unannotated variable rule ($\mathrm{var}_{\checkmark}$), we know that our amalgamated term has now reached head normal form.

Finally, we argue that every trace makes progress towards normalization, in the sense that our three irrelevant rules will only ever occur finitely many times in a row.

In any trace, the number of consecutive (abs) or (app) steps is always bounded by the nesting depth of the term, since both of these rules select a new subterm one step down the syntax tree. Thus, it suffices to bound the number of (var_λ) steps which can occur without making progress, that is to say, without any intermediate (abs_λ) steps.

In a well-formed context, every (var_λ) step calls back to an annotation which was defined before the current variable entered the scope. So without an intermediate (abs_λ) step to introduce a new annotated variable, the next variable we reach must appear earlier in our context. Therefore, the length of the context bounds the number of (var_λ) steps we can take before progress is made.

In conclusion, if our trace is finite, we can observe its amalgamated term reach head normal form in a finite number of (abs_λ) steps. Meanwhile, if our trace is infinite, this progress bound shows that we will necessarily perform an infinite number of head reduction steps. $\square$

6 Decidability

Since principal type inference is undecidable, we consider decidable approximations, analogous to the Hindley-Milner approximation of System F. Motivated by the simply typed lambda calculus, as well as first-hand experience computing head normal forms, we seek to identify the occurences of self-application, in order to prevent any excessively recursive reduction sequences. To see how this manifests within a trace, we observe the simplest unsolvable term $\Omega \equiv \omega\omega$:

$$
\begin{array}{rrl}
 & \ddots \qquad\qquad\qquad\qquad \vdots & \\
\hline
E \to b \in A & x : (A \cap \omega),\ y : (C \cap x),\ z : (D \cap y) \vdash \omega : (E \cap z) \to b & (\mathrm{var}_\lambda) \\
\hline
E \to b \in C & x : (A \cap \omega),\ y : (C \cap x),\ z : (D \cap y) \vdash x : (E \cap z) \to b & (\mathrm{var}_\lambda) \\
\hline
E \to b \in D & x : (A \cap \omega),\ y : (C \cap x),\ z : (D \cap y) \vdash y : (E \cap z) \to b & (\mathrm{var}_\lambda) \\
\hline
 & x : (A \cap \omega),\ y : (C \cap x),\ z : (D \cap y) \vdash z : (E \cap z) \to b & (\mathrm{app}) \\
\hline
 & x : (A \cap \omega),\ y : (C \cap x),\ z : (D \cap y) \vdash zz : b & (\mathrm{abs}_\lambda) \\
\hline
D \to b \in A & x : (A \cap \omega),\ y : (C \cap x) \vdash \omega : (D \cap y) \to b & (\mathrm{var}_\lambda) \\
\hline
D \to b \in C & x : (A \cap \omega),\ y : (C \cap x) \vdash x : (D \cap y) \to b & (\mathrm{var}_\lambda) \\
\hline
 & x : (A \cap \omega),\ y : (C \cap x) \vdash y : (D \cap y) \to b & (\mathrm{app}) \\
\hline
 & x : (A \cap \omega),\ y : (C \cap x) \vdash yy : b & (\mathrm{abs}_\lambda) \\
\hline
C \to b \in A & x : (A \cap \omega) \vdash \omega : (C \cap x) \to b & (\mathrm{var}_\lambda) \\
\hline
 & x : (A \cap \omega) \vdash x : (C \cap x) \to b & (\mathrm{app}) \\
\hline
 & x : (A \cap \omega) \vdash xx : b & (\mathrm{abs}_\lambda) \\
\hline
 & \vdash \omega : (A \cap \omega) \to b & (\mathrm{app}) \\
\hline
 & \vdash \omega\omega : b &
\end{array}
$$

While head reduction for Ω follows a fixed loop, repeating the same head reduction step in an

endless cycle, the repititions within this trace grow over time. We continue to place new variables in the context, and thus must follow an ever growing chain of variables back to the beginning, before we can start a new phase with another occurence of ω. Along each step in this chain, we insert our latest type into all of the previous annotated domain sets we have introduced, forming a triangle of constraints, which happens to roughly mirror the shape of our trace:

$$\begin{array}{lll} \ddots & & \vdots \\ E \to b \in D & E \to b \in C & E \to b \in A \\ & D \to b \in C & D \to b \in A \\ & & C \to b \in A \end{array}$$

The trace for $\omega\,\mathsf{id}$ begins similarly with the first three constraints assembling into a triangle, but then diverges once id stops introducing new self-applications, terminating soon after. Thus, we can think of these triangles as representing a chain of nested self-applications.

In general, self-application might not always happen in the exact same way as Ω. So, we consider all of the ways that a type can be contained within another type:

Definition 6.1. A containment relation is a binary relation on $\mathcal{T}$ given by the rules:

- The domain relation $(x, y) \mapsto x \in \mathsf{Dom}(y)$ is a containment relation.
- The codomain relation $(x, y) \mapsto x = \mathsf{cod}(y)$ is a containment relation.
- The containment relations are closed under binary composition.

Example 6.1.1. The composition $\mathsf{cod} \circ \mathsf{Dom}$ is characterized as follows:

$$\begin{aligned} (\mathsf{cod} \circ \mathsf{Dom})(x, z) &\equiv \exists y.\, x \in \mathsf{Dom}(y) \wedge y = \mathsf{cod}(z) \\ &\equiv x \in \mathsf{Dom}(\mathsf{cod}(z)) \end{aligned}$$

We might describe this situation as "x occurs in the second argument to z" since expanding our interpretation $z \equiv A \to B \to c$ reveals the constraint $x \in B$.

Earlier, we said that a type environment is inductive if the set of types can be ordered such that each element $A \to b$ occurs strictly after all of its dependencies $a \in A$ and b. Collectively, the containment relations capture the transitive closure of these dependencies. Thus, we may equivalently say that a type environment is inductive when all its containment relations are irreflexive. Informally, this means no type is contained within itself.

Definition 6.2. A descending chain in $\mathcal{T}$ consists of:

- An ordinal α.
- A containment relation R.
- A homomorphism of binary relations $f : (\alpha, \ni) \to (\mathcal{T}, R)$.

Example 6.2.1. In the principal type environment for Ω, our triangle of constraints represents a descending chain with ordinal ω for the relation Dom.

Example 6.2.2. In the trivial type environment, the unique element $*$ is related to itself by every containment relation. Thus, for any ordinal α, the unique map $\mathsf{const}\ * : \alpha \to \{*\}$ always forms a descending chain.

Definition 6.3. The rank of a type environment $\mathcal{T}$ is the supremum over the class of ordinals that have a descending chain in $\mathcal{T}$.

Example 6.3.1. The rank of the trivial type environment is the class ordinal Ord of all the small ordinals, since every small ordinal has a descending chain in the trivial environment.

Theorem 6.4. Small Rank $\iff$ Inductive

A type environment is inductive if and only if it has small rank.

Proof. If a type environment is inductive, then every containment relation is irreflexive. But then, since no type is related to itself, every descending chain is injective. Therefore, by Hartogs's theorem, there is an upper bound on the length of the descending chains. In particular, the rank of an inductive type environment is bounded by the Hartogs number of its underlying set.

Conversely, if a type environment is not inductive, there must be some type f which is related to itself by the containment relation R. But then, the function $\mathsf{const}\ f : (\alpha, \ni) \to (\mathcal{T}, R)$ is always a descending chain, for any ordinal α. Therefore, the rank of $\mathcal{T}$ is the large ordinal Ord. □

Theorem 6.5. The rank of a principal type environment is at least 2.

Proof. As argued in Theorem 5.4, every trace eventually uses the variable rule, introducing some constraint $a \in \mathsf{Dom}(f)$. But then, sending $\{0 \mapsto f,\ 1 \mapsto a\}$ is a descending chain of length 2 for the containment relation Dom. □

Example 6.5.1. The principal type environment for id has rank 2, since this single usage of the variable rule is essentially the only containment relation imposed by the principal trace.

Additionally, there are terms of every finite rank. Consider the recursive function definition:

```
slow :: Nat -> Nat
slow n = match (pred n) {
    Nothing => Zero,
    Just m => Succ (slow m),
}
```

Rather than formulating this definition in terms of induction on the natural numbers, or even just using the id combinator, we can also choose to encode this function as a fixed point, directly representing the self-reference in the definition of `slow`. Now, every time we recurse, expanding our fixed point combinator $\mathsf{fix}\ f \to_\beta f\ (\mathsf{fix}\ f)$ increases the length of our descending chain. In a sense, this allows us to measure how much recursion our term uses.

Conjecture 6.6. Finite Rank $\iff$ Solvable

If a term is solvable, then its principal type environment is finite, and therefore has finite rank.

Conversely, if a term is unsolvable, then its principal type environment must be infinite. As shown in Theorem 5.4, it uses the annotated abstraction rule (abs_λ) infinitely often, so we also know that its context grows without bound.

We claim without proof that it eventually starts producing a descending chain over some containment relation. Intuitively, each trace must follow some infinite path through a finite set of subterms. In the eventuality of an infinite trace, we need only consider subterms which occur infinitely often. But since these terms appear again and again, they must eventually lead to some callback, and therefore must be solvable on their own. Thus, the situation always follows the same general pattern as Ω, where we repeatedly type check the solvable subterm ω, leading to further callbacks in an endless loop. However, we have so far been unable to prove this result.

If we assume this conjecture holds, we obtain a decidable approximation of traces with rank at most n for any natural number n. Rank 2 traces somewhat correspond to the simply typed lambda calculus, where no apparent self-applications are allowed. But upgrading to include rank 3 traces permits many terms which nominally contain some self-application, but do not truly rely on a recursive definition, such as $\omega\,\mathsf{id}$.

If we do detect a finite chain above our chosen threshold, rather than rejecting the term outright, we can simply weaken our principality requirement. For example, once we have formed a descending chain of a certain size in the trace for Ω, we can get ahead of the infinite descending chain issue by unifying all of the elements in our finite descending chain. Thus, we obtain the finite derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \equiv a \to b \in \{a\}}{x : \{a\} \vdash x : a \to b}\,(\text{var}) \quad \dfrac{a \in \{a\}}{x : \{a\} \vdash x : a}\,(\text{var})}{x : \{a\} \vdash xx : b}\,(\text{app})}{\vdash \omega : a \to b}\,(\text{abs}) \quad \dfrac{\dfrac{\dfrac{\dfrac{a \equiv a \to b \in \{a\}}{y : \{a\} \vdash y : a \to b}\,(\text{var}) \quad \dfrac{a \in \{a\}}{y : \{a\} \vdash y : a}\,(\text{var})}{y : \{a\} \vdash yy : b}\,(\text{app})}{\vdash \omega : a \to b}\,(\text{abs})}{\vdash \omega : a}\,(a \equiv a \to b)}{\vdash \Omega : b}\,(\text{app})$$

7 Conclusion

Overall, this project embodies a handful of core ideas:

1. While it's very common to give categorical semantics for a type theory, here we have stayed closer to the implementation level, essentially giving categorical semantics for the algorithmic process of type inference.
2. One of the central motifs of the lambda calculus is the implementation of recursion in terms of iteration. Primarily, the iterative process of beta reduction is sufficient to capture the breadth of computational possibility. Similarly, traces capture derivations iteratively.
3. Ultimately, the practical application of this type system hinges on the correctness of the infinite descending chain conjecture. Thus, we can view this project as an attempt to develop a logical framework which can reason about self-application, not just as a syntactic property, but in terms of its fundamental, underlying behavior.

The main result we still hope to acheive is a proof of the infinite descending chain conjecture. This work is crucial in order to guarantee the termination of the principal type approximation algorithm. Additionally, while our current results focus purely on type inference, we would also like to consider type checking. Checking our terms against explictly provided types is a crucial aspect of real world programming languages. In particular, we'd like to consider types containing a combination of multi-functions and quantifiers. While type checking with quantifiers is undecidable in System F, there are already a variety of issues in System F, such as the absence of principal type derivations. It's plausible that incorporating quantifiers into our type system could have more success. Here, we consider a variety of use cases for this potential system of type checking with quantifiers:

1. Efficiency:

   Unless a term crosses the descending chain threshold, principal type inference essentially performs the entire computation of the term. This can be very advantageous for providing extremly detailed types, but it can also be very costly. For example, the principal trace for the church encoded exponential 3^{3^3} requires trillions of steps. In these cases, we'd almost certainly prefer to use a less precise type, most likely expecting to type these arithmetic expressions as simply a natural number.

2. Encapsulation:

   In fact, our type system already allows us to formulate a broad spectrum of choices between the most computationally intensive principal trace and the weakest simple approximation. In our example, we might want to validate our exponential expression as a non-zero natural number $\{a \to a,\, a \to b\} \to a \to b$, or as a multiple of three $\{a \to b,\, b \to c,\, c \to a\} \to a \to a$. Depending on the circumstances, certain properties might be more important than others. Thus, we might assign several different types to the same term, where the specificity of our type determines how much of our implementation internals we expose.

   In particular, we could assign a more precise type to the state monad, as shown below. Since our operations on integers are not well-defined for the newly introduced type X, the only way to return an output state is to use the exact same input state we were given. Thus, we have a type for state computations which are statically verified as read-only, without any need for conversion between encodings.

$$\begin{aligned}
\mathsf{State}(A) &\equiv \mathsf{Int} \to A \times \mathsf{Int} \\
&\equiv \mathsf{Int} \to \forall C.\, (A \to \mathsf{Int} \to C) \to C \\
\mathsf{State}'(A) &\equiv \forall X.\, \{\mathsf{Int},\, X\} \to A \times \{\mathsf{Int},\, X\} \\
&\equiv \forall X.\, \{\mathsf{Int},\, X\} \to \forall C.\, (A \to \{\mathsf{Int},\, X\} \to C) \to C
\end{aligned}$$

3. Random Access Arrays:

   In functional programming languages such as Haskell, data structures tend to have tree-like implementations. This makes it difficult to utilize imperative arrays with random access by arbitrary indices. In Scala, arrays are treated as functions out of (some of) the natural numbers. Similarly, we could encode arrays as an existentially quantified type of indices. This basic building block would facilitate the creation of many imperative data structures.

4. Automatic Primitives:

   Many frequently used data types such as lists are built recursively out of sums and products. A good compiler should be able to automatically detect these kinds of primitives, and use a more efficient memory representation, hopefully optimizing batched operations. Ideally, these batching representations would interface smoothly with the previous encoding of arrays.

5. Dynamic Reflection:

   We might also want to use existentially quantified types to represent a lack of information, as would be needed for the purposes of reflection. For example, we might want to define an evaluation map which takes a tree-like encoding of a lambda term and returns the actual program that term represents. In general, the output of this type would be a mystery ($\exists a.\, a$) but at each invocation we could use more specific knowledge about the type of our input to produce a better, safer type for our output.

6. Arithmetic Strength:

   Finally, we are curious about the arithmetic capabilities of these decidable approximations. In the same way that System F corresponds to the universal fragment of second order logic, it would be interesting to know how much of this logic is captured by the rank 3 descending chain. Similarly, each descending chain cut-off defines a certain ordinal strength, collectively representing an increasing sequence of ordinals.

References

[1] Henk Barendregt. "Lambda Calculi with Types". In: *Handbook of Logic in Computer Science*. Oxford University Press, Dec. 1992. ISBN: 9780198537618. DOI: 10.1093/oso/9780198537618.003.0002. eprint: https://academic.oup.com/book/0/chapter/421961562/chapter-pdf/52352320/isbn-9780198537618-book-part-2.pdf. URL: https://doi.org/10.1093/oso/9780198537618.003.0002.

[2] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[3] Adrien Guatto. "A Generalized Modality for Recursion". In: *CoRR* abs/1805.11021 (2018). arXiv: 1805.11021. URL: http://arxiv.org/abs/1805.11021.

[4] Andrew D. Ker. "Lambda Calculus and Types". Course Lecture Notes. URL: https://www.cs.ox.ac.uk/teaching/materials21-22/lambda/notes.pdf.

[5] Didier Le Botlan and Didier Rémy. "MLF: raising ML to the power of system F". In: *SIGPLAN Not.* 38.9 (Aug. 2003), pp. 27–38. ISSN: 0362-1340. DOI: 10.1145/944746.944709. URL: https://doi.org/10.1145/944746.944709.

[6] Benjamin C. Pierce. "Programming Language Foundations". In: vol. 2. Software Foundations. University of Pennsylvania, 2017. Chap. The Simply Typed Lambda-Calculus. URL: https://softwarefoundations.cis.upenn.edu/plf-current/Stlc.html.

[7] Steffen van Bakel. "Intersection type assignment systems". In: *Theoretical Computer Science* 151.2 (1995). 13th Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 385–435. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(95)00073-6. URL: https://www.sciencedirect.com/science/article/pii/0304397595000736.