CoSimple Types With Guarded Recursion

Eason Kamander

Part B Project Report

Computer Science

Trinity 2024

Abstract

The simply typed lambda calculus captures a rich, minimal framework for reasoning about programs as functions. Many alternative systems, both extensions of the lambda calculus and independent strategies, provide the means to represent infinite data structures or ongoing process communication within their computational theory. It's often customary to model these interactions between concurrent processes as communication along named channels, or through other highly synthetic constructions. This paper aims to combine the functional expressivity of the lambda calculus and the recursive capabilities of process based models into an idiomatic, cohesive type system. We propose careful modifications to the rules of the simple types, with a focus on understanding the category theoretic relationships between type systems themselves. We present an elegant formulation of guarded recursion, prove the logical consistency of this new system, and implement a straight-forward algorithm for type inference. Ultimately, this allows us to voice recursive processes natively within the lambda calculus.

Contents

1 Introduction

In many different areas of theoretical computer science, it's valuable to model infinite, recurring behavior. In the Concurrency course, processes use self-referential definitions to represent ongoing communication. In the Computer-Aided Formal Verification course, transition systems use cyclic transitions to represent recurring behavior. Both of these formulations hint at the many potential upsides of combining their first class recursion with other standardized machinery, especially infinite state. For example, the Concurrency course specifies a FIFO queue using an infinite family of processes indexed by the current contents of the queue. However, ultimately neither system is able to concretely integrate into an infinite setting, due to the complexities of recursion.

In this paper, we use function types to represent protocol specifications, and lambda terms to represent their deterministic implementations. In Section 2, we delve into the details of these process-oriented types. Then in Section 3, we focus on the behaviour of terms, proposing a modified system to achieve a trade-off between safety and expressivity. Finally in Section 4, we discuss some of the ways these ideas could be put into practice.

2 CoSimple Types

This section explores the extension of the simply typed lambda calculus with coinductive type formation. This rudimentary construction presents a useful vantage point to survey the landscape of proposition-like type systems, with a focus on recursion.

The simply typed lambda calculus does not admit any fixed-point combinators, but if it did, they would have type $(X \to X) \to X$. At the most intuitive level, this is apparent from the operational interpretation: a fixed-point combinator

takes in an endofunction and returns one of its fixed points. Somewhat more formally, functional programming languages like Haskell validate the legitimacy of this association, seamlessly augmenting many simple types with a variety of additional inhabitants. While some of these terms can be attributed to more advanced language features or axiomatic system libraries, there are also many "pure" functions which find their way into appropriate seeming types despite employing a simply untypeable degree of recursion. Relaxing the requirements on type formation to permit coinduction, the rationale behind these judgements becomes completely evident.

2.1 CoInduction

In general, inductive objects are defined by the steps involved in putting them together and coinductive objects are defined by the steps involved in taking them apart. While induction requires the familiar well-founded structure used in most definitions, coinduction is more permissive, freely assuming the validity of objects which will only be assigned definitions later, potentially in terms of themselves. We will refer to the coinductive variant of the simple types as the cosimple types.

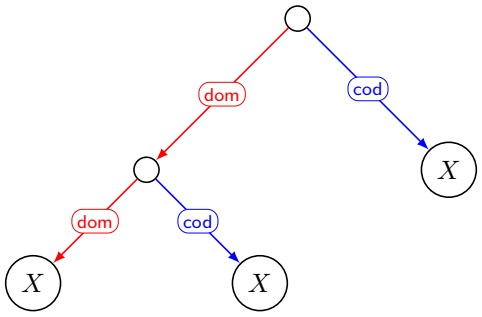


Figure 1: The fixed point type $(X \to X) \to X$ as a tree.

The standard notation used to represent simple types as strings is to directly

encode a binary tree of variables. These underlying mathematical objects can be interpreted as the parse trees which inhabit the grammar of simple types, or as the proof trees which realise simple types under formal inference rules. Switching to a coinductive type formation rule amounts to replacing these trees with graphs, allowing type definitions to loop back on themselves.

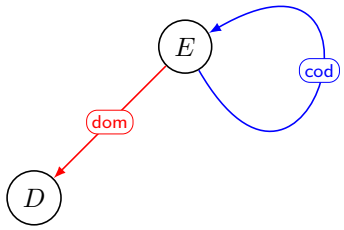


Figure 2: The eater type $E \equiv D \to E$ as a graph.

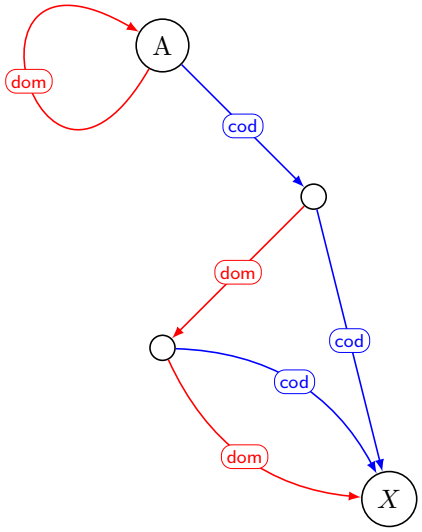
From a set theorist's perspective, this formulation reveals that cosimple types are in fact simpler than simple types: since trees are a special case of graphs, the simple types are exactly the acyclic cosimple types. Similarly, for an imperative programmer, the data type used to encode simple types is almost always better suited for representing cosimple types.

2.2 Syntax

Simple types are naturally serialised using the order in which they were inductively defined, but coinductive definitions are not equipped with this structure. One popular approach is to represent coinductive definitions as inline expressions using an explicit type constructor for recursion. For example, the eater type could be written as $\mu X \,.\, Y \to X$. The μ constructor simulates coinduction by offering up fresh variable bindings for self-referential definitions to latch onto. This way, the recursive expression $Y \to X$ can allude to itself without performing any true self-reference, since the recursion variable X is technically just a variable.

While this can provide a convenient shorthand in some circumstances, it has a few major disadvantages. It complicates a simple idea with variable bindings and scopes. It becomes increasingly cumbersome for mutually recursive definitions. It presents the expansion of recursive types, known as unrolling, as a substantive action rather than an immediate judgemental equality. Most worryingly, it quietly introduces a new type — $\mu X \,.\, X$ — which is neither a variable nor a function, and which unrolls into itself.

Instead, this paper will represent coinduction more directly, permitting cyclic dependencies within arrow type definitions. This way, the serialisation of cosimple types will explicitly characterise their underlying domain and codomain graphs.



(a) Graphical

```
X := new variable();
A := A -> (X -> X) -> X;
```

(b) Concise

```
X := new variable();
F := X -> X;
G := F -> X;
A := A -> G;
```

(c) Verbose

Figure 3: Equivalent notations for a cosimple type.

2.3 Equality

Simple types are usually considered up to extensional equality. Here, the extension of an object refers to its measurable properties, and so this assertion — that function types are identified by their domain and codomain — almost seems to go without saying. This is a natural choice to make for simple types, but it is a choice. Type variables have no extension; their intensional equality is what makes them useful. Similarly, function types could be treated as distinct unless explicitly declared equal. This approach gets the job done for all practical purposes, but a richer understanding requires a more in depth investigation.

When extensional equality is applied to simple types, it propagates in the same direction as induction: equalities on function types are built from smaller equalities between smaller components. But since coinduction is unordered, there is no natural direction for equalities to automatically propagate. However, it is possible to artificially enforce an equality between cosimple types by assuming its existence up front, and then recursively decomposing it into a set of constraints on subexpressions. This process is highly reminiscent of the unification algorithm, motivating an elegant definition of equality in terms of an even more fundamental concept: substitution.

For simple types with extensional equality, substitutions are uniquely determined by their behaviour on variables, extending inductively to act on functions. However, some cosimple types are defined in terms of themselves, so variables alone don't always tell the full story. Instead, we define a substitution as a map on subexpressions which preserves domain and codomain relationships. Since these relationships constitute the edges of the underlying graph, a substitution is exactly a labelled graph homomorphism.

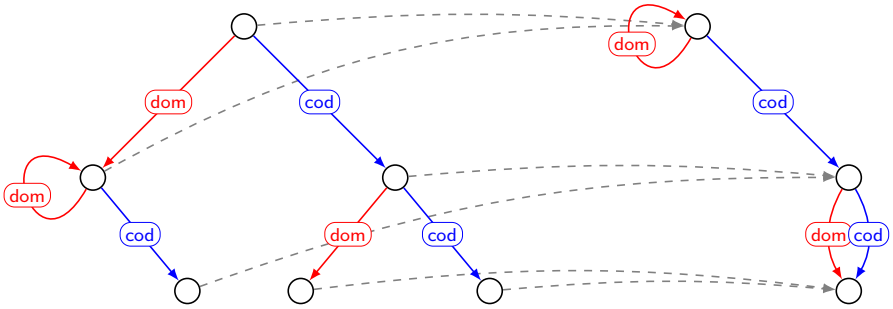


Figure 4: An example substitution.

2.4 Variable Expansion

Consider the substitution sending some type variable V to the function type $F \equiv X \to Y$ with X and Y fresh type variables. This map simply expands V into a function type, which doesn't seem to change its meaning in any significant way. Substitutions acting on F might as well just be acting on V, since the behaviour of X and Y is fully constrained by the requirements of graph homomorphisms. Even though these fresh variables are functionally redundant, their mere existence invalidates the possibility of an inverse substitution from F to V.

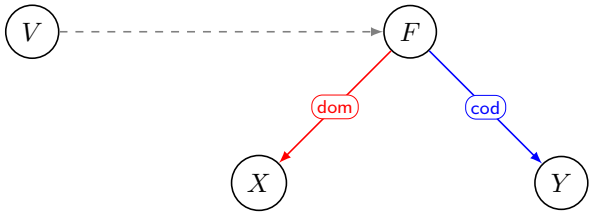


Figure 5: The variable expansion substitution.

In category theory, the substitution from V to F is called an injective epimorphism. A similar example in the category of metric spaces is the standard inclusion from the rationals to the reals, whose image is too small to cover the

entire underlying set, but covers enough of the space to dictate the behaviour of continuous functions. From the perspective of function composition, these maps behave like isomorphisms in the sense that they can safely be cancelled from both sides of an equation. Since these morphisms are so similar to isomorphisms, it's usually preferable to operate in a setting where they really are.

One way to remedy these redundant substitutions is with an explicit quotient, by reasoning about the congruence on graphs generated by the expansion map. This approach does work, but instead we can use this as an opportunity to rephrase our construction in much simpler terms. Informally, by applying the expansion substitution to every graph, imagine nudging all types in the direction of an increasingly expanded form, eventually reaching a system where types consist of functions all the way down.

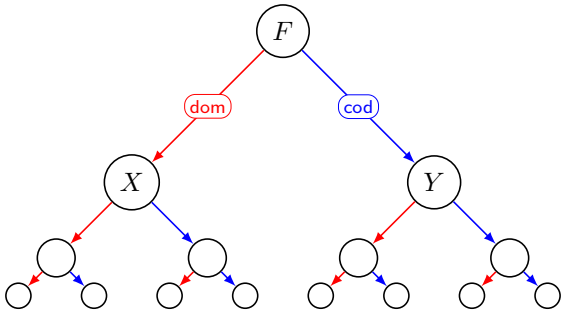


Figure 6: Iterated variable expansion.

2.5 Semantics

A magma is a simple algebraic structure, consisting of a set equipped with a single binary operation. Many common algebraic structures, such as groups and rings, are defined in terms of magmas, since they generalise situations where two objects can be combined into one. Dually, a comagma is an algebraic structure where one object can be separated into two, consisting of a set equipped with a pair of endomorphisms. For example, the reals with sine and exponent form a

comagma.

Every comagma has a natural interpretation as a cosimple type graph, by treating the two endomorphisms as the domain and codomain relationships. In fact, these are exactly the graphs which don't contain type variables. This transformation can be formalised as an embedding from the category of comagmas to the category of type graphs, which gives rise to a free-forgetful adjunction between the two categories. Though the details of adjoint functors are outside the scope of this paper, it suffices to say that the structure of this embedding ensures the existence of a reverse relationship — the free functor — which maps each type graph to the comagma that best approximates it.

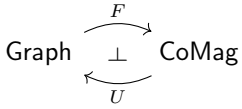


Figure 7: The free-forgetful adjunction.

As it turns out, this free functor is exactly the minimal quotient needed to make variable expansion trivial. As mentioned earlier, the intuition for the free functor is that it expands variables all the way out. Another way to imagine this is that it replaces the native variable representation found in type graphs with meta-theoretic variables, temporary placeholders which could always be expanded further. In this way, type graphs can be seen as a notational shorthand, with the free functor providing a rigorous semantic interpretation.

2.6 Semantics II

Under these definitions, all types have a domain and a codomain, each of which have domains and codomains of their own, and so on, forever. In this way, each type is associated with an infinite binary tree of components. For a type variable, all of these components are distinct: the freest possible configuration. In general,

the structure of a cosimple type is captured by these equality constraints. For example, whenever `id : T`, we must have `dom(T) = cod(T)`. Since `dom` and `cod` must be well-defined functions, this constraint propagates downward, equating each node of the left subtree with the corresponding node of the right subtree. When no additional equalities are imposed, `T` is a principal type for `id`.

These components are naturally indexed by the finite lists of booleans, each representing a path in the infinite binary tree. More abstractly, this is the free monoid with two generators, representing left and right with path concatenation, or equivalently, `dom` and `cod` with function composition. Restricting the structure of a comagma to pick out only the information relevant to an individual node, we can define an isolated cosimple type as an equivalence relation on this monoid of indices, directly representing the unification constraints between components. In order to capture the propagation of equalities, we further require this equivalence relation to satisfy $f \approx g \implies h \cdot f \approx h \cdot g$, defining a congruence under post-composition.

However, since most practical applications operate on multiple interdependent types existing within a shared scope, it's probably more beneficial to continue defining the cosimple types as the elements of some kind of collection. As mentioned earlier, the free monoid with two generators describes all of the functional relationships between types in a comagma. Therefore, in category theory, a comagma can equivalently be defined as a presheaf on this monoid, regarded as a one-object category. This insight leads to many interesting results — most notably for our purposes, the category of cosimple contexts and type substitutions forms an incredibly rich structure known as a topos.

2.7 Derivations

For the simply typed lambda calculus, type inference only fails when the unification of two types would result in a violation of inductive type formation. For the cosimple types, there are no such restrictions, and the most general unifier of any two types can be easily described as a coequalizer, which always exists in a topos. Therefore, every lambda term has a principal cosimple type. This increased ability to carry on performing cyclic unifications is the only difference between simple and cosimple type derivations.

$$\frac{}{\Gamma,\, x : X,\, \Delta \vdash\, x : X}\ (\text{var})$$

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

$$\frac{\Gamma \vdash\, p : A \to B \qquad \Gamma \vdash\, q : A}{\Gamma \vdash\, pq : B}\ (\text{app})$$

Figure 8: Derivation rules for the (co)simple type system.

The quintessential term with no simple type is $\omega \equiv \lambda x.xx$, the self-application function. Since x accepts itself as an argument, the type of x must be equal to its own domain. Regarding this self-referential type $A \equiv A \to B$ as the coinductive analog of a proposition, this derivation has a very natural interpretation in logic. This type is a self-justifying condition, informally corresponding to a statement along the lines of "this sentence implies B." The term ω takes in a proof of this recursive sentence and applies it to itself: since the sentence is true and the sentence implies B, it can extract a proof of B.

$$\dfrac{\dfrac{\dfrac{A \equiv A \to B}{x : A \vdash\, x : A \to B}\ (\text{var}) \qquad \dfrac{}{x : A \vdash\, x : A}\ (\text{var})}{x : A \vdash\, xx : B}\ (\text{app})}{\vdash\, \lambda x.xx : A \to B}\ (\text{abs})$$

Figure 9: Principle type derivation for ω.

However, by proving that one of these sentences would imply B, we've effectively proven the sentence itself. Repeating this process of self-application using ω as a closed proof of A, we produce $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$, the infamous self-replicating term, allowing us to prove B, a type with no constraints. This unfortunate result is known as Curry's paradox. In a classical setting, the coinductive type definition used here could also be written as $A \equiv \neg A \vee B$, combining self-reference and negation to produce an inconsistency.

$$\dfrac{\dfrac{A \equiv A \to B}{\vdash\ \lambda x.xx : A \to B}\,(\omega) \qquad \dfrac{\dfrac{A \equiv A \to B}{\vdash\ \lambda x.xx : A \to B}\,(\omega)}{\vdash\ \lambda x.xx : A}\,(\equiv)}{\vdash\ (\lambda x.xx)(\lambda x.xx) : B}\,(\text{app})$$

Figure 10: Principle type derivation for Ω.

Interestingly, even this inconsistent system still imposes some requirements on the space of possible type judgements. As we've seen, Ω is typeable with any type, but ω requires a function which accepts itself as input. In this way, the cosimple types categorise the ways that terms can be used. As promised, the principal type of the Y combinator reflects its functional signature.

$$\dfrac{\dfrac{\dfrac{}{f : X \to X \vdash\ f : X \to X}\,(\text{var}) \qquad \dfrac{\dfrac{A \equiv A \to X}{x : A \vdash\ x : A \to X}\,(\text{var}) \qquad \dfrac{}{x : A \vdash\ x : A}\,(\text{var})}{x : A \vdash\ xx : X}\,(\text{app})}{f : X \to X,\, x : A \vdash\ f(xx) : X}\,(\text{app})}{f : X \to X \vdash\ \lambda x.f(xx) : A \to X}\,(\text{abs})$$

(Lemma 1)

$$\dfrac{\dfrac{\dfrac{A \equiv A \to X}{f : X \to X \vdash\ \lambda x.f(xx) : A \to X}\,(1) \qquad \dfrac{\dfrac{A \equiv A \to X}{f : X \to X \vdash\ \lambda x.f(xx) : A \to X}\,(1)}{f : X \to X \vdash\ \lambda x.f(xx) : A}\,(\equiv)}{f : X \to X \vdash\ (\lambda x.f(xx))(\lambda x.f(xx)) : X}\,(\text{app})}{\vdash\ \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (X \to X) \to X}\,(\text{abs})$$

Figure 11: Principle type derivation for the Y combinator.

2.8 Subject Construction

For such a foundational property, subject construction is quite difficult to formalise in an elegant way. Instead, definitions for subject construction are often presented as a laundry list of inductive principles. For example, subject construction tells us that the only way to assign a type to an abstraction is using the abstraction rule, resulting in a function type whose codomain is the same as the type of the body of the abstraction when the variable is taken to have the type of the domain.

Colloquially, subject construction tends to refer to the idea that the inductive structure of type derivations and terms are cohesive. This cohesion is what allows us to accurately transport our interpretation of terms as functions into an interpretation of types as function spaces, as in the example above. It is also responsible for the existence of principal types, requiring type derivations over a fixed term to be so similar to each other that almost all of their structure can be factored out, leaving behind only type substitutions.

Perhaps the complexity of these definitions reveals that subject construction is more of a corpus than an individual property. Yet many of these ideas tend to go hand in hand. The cosimple types centralise these principles, disregarding computability concerns in order to isolate the essential nature of arrow types. In some sense, the cosimple type system is the most permissive arrow type system with subject construction. This universal property can be applied in reverse as a kind of definition. If a typing judgement isn't even valid using the cosimple types, then it must be in violation of subject construction. Similarly, the simple types inherit the properties of subject construction as a subsystem of the cosimple types.

3 Guarded Recursion

This section develops a new subsystem of the cosimple types, based on the principles of guarded recursion. This compromise between the simple and cosimple systems allows us to safely handle coinductive reasoning.

Since we've expanded our grammar to include coinductive type formation, it might be natural to ask: why not do the same for terms? Fortunately, there's no need. While the inductive formulation of the lambda calculus does not permit infinite self-referential terms to be defined directly, coinductive constructions are readily mimicked by the semantics of infinitely self-replicating reductions. Under this interpretation, recursive machinery such as the fixed point combinators can be seen as a way of embedding coinductive capabilities into the lambda calculus without compromising its algebraic purity. In a sense, these terms form a Church encoding for recursion, closing the lambda calculus under self-reference.

3.1 Productivity

The acyclicity requirement at the core of the simply typed lambda calculus exists to ensure that type derivations are not only syntactically well-formed, but logically and computationally meaningful. Unfortunately, this approach directly targets the same recursive machinery necessary to encode coinductive definitions.

Rather than eliminating all cyclic behaviour, a more expressive policy could permit recursive definitions which are in some sense productive. For example, the infinite list defined by `x = 1 : x` is productive: its recursive evaluation makes progress, increasing the known prefix of the list one step at a time. As a result, it has a well-defined mathematical interpretation and a realisable computational encoding. On the other hand, the definition `x = head x : x` is not productive:

its recursive evaluation never produces any definitive results.

For many simple data types, productivity checking is fairly straightforward: an expression is productive when all occurrences of self-reference are guarded by a constructor. The untyped lambda calculus, on the other hand, captures a much more expressive style of recursion. Terms grow and shrink in unpredictable ways as they are evaluated, and there is no clear standard for exactly what counts as a constructor.

3.2 Constructors

Abstract data types are usually represented in the lambda calculus by treating their constructors as unspecified inputs. For example, the natural numbers are often represented by the simple type $(X \to X) \to (X \to X)$, whose inhabitants are essentially given the successor and zero constructors as their first two arguments. It makes sense to interpret these types as conditional statements: whenever you find some structure analogous to the zero and successor constructors, the church numeral seven can use this structure to compute the analog for the number seven. This approach has many similarities with continuation passing style, producing abstract terms endlessly waiting to be supplied with the "real version" of the constructors they rely on.

If we choose to consider the successor argument as a productive constructor, a variety of self-referential definitions become permissible, fundamentally altering our semantics. This new type — known as the conatural numbers[2] — is the coinductive dual of the natural numbers, defined as the greatest fixed point of the recursive specification $\mathbb{N} \cong \mathsf{Maybe}\ \mathbb{N}$. Intuitively, it consists of the ordinary natural numbers augmented with a largest element: $\omega \equiv \mathsf{succ}\ \omega$, the fixed point of successor.

However, these new recursive definitions might introduce undesirable behaviour unless we continue to honour our promise of productivity. For example, the isEven function supplies not : Bool → Bool as the value for successor, but applying this substitution to ω would cause recursive evaluation to collapse in place, resulting in an unsolvable term. Even worse, the semantic interpretation of this program leads to immediate inconsistencies, since we've now constructed a boolean which is its own negation. Because of these potential issues, deciding that a constructor is productive involves a trade-off between how terms can be created and how they can be used: the introduction rule for conatural numbers is more permissive, but the elimination rule is more restrictive.

3.3 Modal Logic

In order to dynamically ascertain where productivity promises need to be honoured, we introduce a new modal operator to indicate the presence of self-referential definitions. In this context, we can think of a modal operator as a re-interpretative transformation on types. Our operation ($\bullet$) serves primarily as a placeholder, allowing most types to revert to simple inductive foundations, while recursive definitions are confined inside the modality, exclusively to types of the form $\bullet\, X$. Since the main purpose of this operator is to label and contain potentially unsafe behavior, we refer to this type operation as guarding.

One small detail worth noting is that guarding should not be counted as a fully fledged type constructor, as this would allow the coinductive formation of types such as $X \equiv \bullet\, X$, which have no interpretation as functions, similar to the μ-type syntax described earlier. It's not unreasonable to just explicitly declare these type definitions invalid, but an alternative approach is to treat the guarding operator as more of an annotation. Under this interpretation, types maintain their basic structure as comagmas, but each occurrence of a

type is additionally characterised by some finite number of guards. This can be formalised algebraically by replacing the endomorphisms of the comagma with functions returning both a type and a natural number. In category theory, this corresponds to working with presheaves valued in the Kleisli category for the writer monad over $(\mathbb{N}, +)$.

3.4 Modal Axioms

Modal operators are used to achieve a variety of different logical effects, depending on their algebraic properties. Generally, almost all modalities are assumed to be cohesive with the structure of function application. These additional requirements define an applicative functor, which has many similarities to the modal operators found in Kripke logic[1]. In our case, this added structure permits guards to act as transparent labels, rather than interfering with type derivations.

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash s : \bullet\, A}\ (\iota)$$

$$\frac{\Gamma \vdash t : \bullet(A \to B)}{\Gamma \vdash t : \bullet\, A \to \bullet\, B}\ (\kappa)$$

Figure 12: Derivation rules for guards.

First, the necessitation rule (ι) proclaims that ordinary type derivations remain valid within the world of guards. Corresponding to the unit of the applicative, this rule allows us to impose as many extra guards onto our types as we'd like. Second, the distribution axiom (κ) requires guarded functions to genuinely behave as functions within the guarded world. By distributing guards over an arrow constructor, we can call a guarded function with a guarded argument to obtain a guarded result. This internal model of function application allows guards to naturally propagate along type derivations, ensuring that the byprod-

ucts of guarded subterms remain guarded.

The last remaining piece of the puzzle is to formalise a policy for when guards need to be introduced, since currently their presence is entirely optional. Intuitively, in the coinductive syntax, we'd like to require guards for self-reference, such as the `x` appearing in the definition `x = 1 : x`. This choice encourages the type checker to treat `(1 :)` as removing a guard, which serves as a tentative definition for productivity. Since these self-referential programs can only be encoded using fixed points and other recursive constructions, we can achieve this effect by mirroring the simply typed lambda calculus and forbidding unguarded cyclic type formation.

3.5 Derivations

As expected, recursion guards only become necessary for handling instances of self-application. Crucially, the paradoxical coinductive type $A \equiv \bullet A \to B$ is now guarded, but the type derivation $\omega : A \to B$ is not. In fact, it is no longer possible to type check $\omega : A$ because the argument of A is guarded, but ω cannot remove any guards. Since these types are incompatible, Ω is once again untypeable.

$$\dfrac{\dfrac{\dfrac{A \equiv \bullet A \to B}{x : A \vdash x : \bullet A \to B}\,(\text{var}) \qquad \dfrac{\dfrac{}{x : A \vdash x : A}\,(\text{var})}{x : A \vdash x : \bullet A}\,(\iota)}{x : A \vdash xx : B}\,(\text{app})}{\vdash \lambda x.xx : A \to B}\,(\text{abs})$$

Figure 13: Guarded principle type derivation for ω.

However, when the propagation of guards is interrupted by a constructor, the promise of productivity allows more expressive self-applications to continue unimpeded. For example, the only way to justify the outer self-application of the Y combinator is for f to remove a guard. Therefore, fixed-point combi-

nators are still typeable, but with a restricted type: they are only safe to use on productive functions.

$$\dfrac{\dfrac{}{f : \bullet X \to X \vdash f : \bullet X \to X}\,(\text{var}) \qquad \dfrac{\dfrac{\dfrac{A \equiv \bullet A \to X}{x : \bullet A \vdash x : \bullet(\bullet A \to X)}\,(\text{var})}{x : \bullet A \vdash x : \bullet\bullet A \to \bullet X}\,(\kappa) \qquad \dfrac{\dfrac{}{x : \bullet A \vdash x : \bullet A}\,(\text{var})}{x : \bullet A \vdash x : \bullet\bullet A}\,(\iota)}{x : \bullet A \vdash xx : \bullet X}\,(\text{app})}{f : \bullet X \to X,\, x : \bullet A \vdash f(xx) : X}\,(\text{app})}{f : \bullet X \to X \vdash \lambda x.f(xx) : \bullet A \to X}\,(\text{abs})$$

(Lemma 1)

$$\dfrac{\dfrac{A \equiv \bullet A \to X}{f : \bullet X \to X \vdash \lambda x.f(xx) : \bullet A \to X}\,(1) \qquad \dfrac{\dfrac{\dfrac{A \equiv \bullet A \to X}{f : \bullet X \to X \vdash \lambda x.f(xx) : \bullet A \to X}\,(1)}{f : \bullet X \to X \vdash \lambda x.f(xx) : A}\,(\equiv)}{f : \bullet X \to X \vdash \lambda x.f(xx) : \bullet A}\,(\iota)}{f : \bullet X \to X \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : X}\,(\text{app})}{\vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\bullet X \to X) \to X}\,(\text{abs})$$

Figure 14: Guarded principle type derivation for the Y combinator.

3.6 Implementation

In practice, choosing between the ordinary rules and the guard operations at each step of a type derivation is unwieldy. Instead, it's computationally preferable to annotate each type with a natural number of guards. This way, the guard operations are bundled together with the unique derivation rule determined by term syntax. As described earlier, this is effectively the subject construction property for guarded recursive types: the quotient map on types that removes their guards extends to act on derivations, formalising the obvious correspondence between these rules and the rules for cosimple types.

Therefore, one natural way to perform type inference is to first synthesize a cosimple type, and then attempt to annotate it with guards. We have implemented this algorithm in Rust, explicitly solving these annotation constraints as a linear optimization problem. Although almost all of these techniques are

$$\frac{}{\Gamma,\, x : \bullet_\alpha\, X,\, \Delta \vdash\ x : \bullet_{(\alpha+\beta)}\, X}\ (\text{var})$$

$$\frac{\Gamma,\, x : \bullet_\alpha\, A \vdash\ s : \bullet_\beta\, B}{\Gamma \vdash\ \lambda x.s : \bullet_\gamma(\bullet_\alpha\, A \to \bullet_\beta\, B)}\ (\text{abs})$$

$$\frac{\Gamma \vdash\ p : \bullet_\gamma(\bullet_\alpha\, A \to \bullet_\beta\, B) \qquad \Gamma \vdash\ q : \bullet_{\gamma+\alpha}\, A}{\Gamma \vdash\ pq : \bullet_{(\gamma+\beta)}\, B}\ (\text{app})$$

Figure 15: Derivation rules for the guarded recursive type system.

fairly standard, the application of imperative practices to high-level type theory is somewhat unusual. But since the cosimple types make such extensive use of cyclic definitions, an imperative approach is favourable for its flexibility handling low-level data structures. In particular, our implementation repeatedly employs the New Type Idiom[4] to provide type-safe graph representations.

3.7 Consistency

Under the Curry-Howard isomorphism, the inhabitance problem of each type system induces an interesting formal logic. Just as propositional logic simplifies intuitionistic logic, one reductive assumption we could make is that all guarded types are inhabited, since they might be employing recursive self-reference. We define an assignment on some guarded comagma is a mapping from each element to a truth value, intended to model a possible complete set of simultaneously inhabited types. In order to ensure that these models accurately represent type inhabitance, we require them to preserve functional structure.

$$\begin{aligned} \mathcal{A}[\![X \to Y]\!] &= \mathcal{A}[\![X]\!] \Rightarrow \mathcal{A}[\![Y]\!] \\ \mathcal{A}[\![\bullet\, Z]\!] &= \top \end{aligned}$$

Figure 16: The laws of assignments.

If we had permitted unguarded cyclic types, these model laws could lead to re-

cursive constraints such as $\mathcal{A}[\![A]\!] = \neg\mathcal{A}[\![A]\!] \vee \mathcal{A}[\![B]\!]$. Instead, properly guarded types have a much broader class of models, closely resembling the way inductive assignments are extended from an arbitrary map on variables. Formally, assignments can always be extended along injective type substitutions, so any choice of n disjoint type variables gives rise to 2^n assignments.

We will show that these are sound models, meaning that an inhabited type is satisfied by every assignment. By contrapositive, it follows that any type with an unsatisfying assignment — such as a type variable — is uninhabited. Therefore, the internal logic of guarded cosimple types is consistent.

Note: We use an overloaded notation for assignments, treating a context as a conjunction of types, as in $\mathcal{A}[\![\Gamma]\!] = \forall(x : X) \in \Gamma.\ \mathcal{A}[\![X]\!]$.

Claim: $\Gamma \vdash s : S \implies \mathcal{A}[\![\Gamma \to S]\!]$.

Proof. By structural induction on the type derivation:

Variable Case:

Here, $(s : S) \in \Gamma$.

By weakening, $\mathcal{A}[\![\Gamma]\!] \Rightarrow \mathcal{A}[\![S]\!]$.

Therefore, $\mathcal{A}[\![\Gamma \to S]\!]$.

Abstraction Case:

Here, $s \equiv \lambda x.t$ and $S \equiv A \to B$ with $\Gamma,\ x : A \vdash t : B$.

By the inductive hypothesis, $\mathcal{A}[\![(\Gamma \cdot A) \to B]\!]$.

This is equivalent to $\mathcal{A}[\![\Gamma \to S]\!]$ by definition.

Application Case:

Here, $s \equiv pq$ with $\Gamma \vdash p : Q \to S$ and $\Gamma \vdash q : Q$ for some Q.

By the inductive hypothesis, $\mathcal{A}[\![\Gamma \to Q \to S]\!]$ and $\mathcal{A}[\![\Gamma \to Q]\!]$.

Assume $\mathcal{A}[\![\Gamma]\!]$, since $\mathcal{A}[\![\Gamma \to S]\!] \equiv \mathcal{A}[\![\Gamma]\!] \Rightarrow \mathcal{A}[\![S]\!]$.

Then, $\mathcal{A}[\![Q \to S]\!]$ and $\mathcal{A}[\![Q]\!]$, so $\mathcal{A}[\![S]\!]$.

Necessitation Case:

Here, $S \equiv \bullet\, T$.

Trivially, $\mathcal{A}[\![\bullet\, T]\!]$ holds.

Therefore, $\mathcal{A}[\![\Gamma \to S]\!]$.

Distribution Case:

Here, $S \equiv \bullet\, A \to \bullet\, B$.

Trivially, $\mathcal{A}[\![\bullet\, B]\!]$ holds.

Therefore, $\mathcal{A}[\![\Gamma \to S]\!]$.

□

4 Conclusion

4.1 Future Work

The main result we still hope to achieve is a formal proof of term normalisation. The typeability of conatural numbers, infinite lists, and other coinductive data structures clearly demonstrates that terms are not required to ever achieve a normal form. Instead, these terms intuitively seem to converge towards an infinite normal form, allowing for the evaluation of an arbitrary large redex-free prefix. Based on limited experimental testing, it seems plausible that the guarded recursion annotation system successfully enforces weak head normalisation, where a weak head normal form is any term that doesn't begin with a

redex.

Importantly, since type derivations act on all subterms, this would immediately imply a much stronger result: hereditary weak head normalisation. This property seems very similar to the informal behavior we wish to capture, where any redex can eventually be eliminated, or at least pushed further down the syntax tree. In fact, the intuition from earlier of convergence towards an infinite normal form is made precise by the construction of Lévy-Longo trees[3], defined as the infinite unrolling of weak head normalisation.

Another promising direction for future research is the integration of these principles into more complex type systems. These ideas can easily be retrofit with basic extensions such as explicit sum and product types, or incorporated into more advanced systems with polymorphism and dependent types. However, these direct translations are not yet as idiomatic as they could be.

4.2 Applications

Although this paper frequently refers to recursion productivity as a more expressive condition than the acyclicity requirement of the simply typed lambda calculus, these guarded recursive types do not achieve any superior arithmetic capabilities. Instead, they enable the expression of more complex processes. Whereas Turing machines conceptualize programs as existing in a meaningless, unobservable intermediate state unless and until they halt with some finite result, processes are all intermediate state, characterised by their ongoing communications with the outside world.

A prominent example of process communication is the higher or lower game. On each turn, the guesser must guess a natural number and the attestor must attest to a trichotomous comparison with a number of their choice. The game

ends when they attest to equality, and the guesser wins. In the Design and Analysis of Algorithms course, we discussed an asymptotically optimal strategy for the guesser as shown.

```
type Range = (Int, Maybe Int)

midpoint :: Range -> Int
midpoint (low, Just high) = (low + high) `div` 2
midpoint (low, Nothing) = 2 * low

guesser :: (Int, Maybe Int) -> Int
guesser (low, high) = bisect (attestor mid)
  where
    mid = midpoint (low, high)
    bisect EQ = mid
    bisect LT = guesser (low, Just mid)
    bisect GT = guesser (mid, high)
```

Figure 17: Bisection algorithm for the higher or lower game.

Two major factors stand out about this solution. First, it's recursive definition is not productive. No matter how much we trust the inner workings of our code, at the type level, guesser uses bisect at its head and vice versa, resulting in an unguarded self-reference. Second, the types used to represent this problem do not accurately portray the underlying constraints. Int and Range values are unconstrained, leaving the guesser vulnerable to manipulation by a contradictory attestor. However, these violations are immediately noticeable, and become very straightforward to address in a setting where the guesser can report errors.

Even equipped with a state of the art proof assistant, there are still some nuances to explore regarding the classic playground bluff: an attestor that always says higher. The difficulty lies in conveying this constraint to the guesser in a way that is useful without fundamentally altering the game: providing an explicit upper bound is too strong, but specifying that something doesn't happen is too weak. The usual solution is the Merely monad, which can conceal the identity of

the upper bound, informally representing a promise that somewhere an upper bound exists.

In practice though, this game and many others are often played with a lot less trust. With cosimple types, we have another option: the Delay monad, a resource for safely encapsulating non-termination. This monad augments each type with a self-referential constructor Later, which can be used as a meaningless guard for arbitrarily recursive definitions. This ensures that unrolling still makes progress, even if that progress is only accumulating an increasing delay.

```
data Delay X = Now X | Later (Delay X)

instance Monad Delay where
  return = Now

  Now x >>= f = Now (f x)
  Later x >>= f = Later (x >>= f)
```

Figure 18: Haskell definition for the Delay monad.

References

[1] James Garson. "Modal Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Spring 2024. Metaphysics Research Lab, Stanford University, 2024.

[2] Robert Harper. *Natural and Co-Natural Numbers*. https://www.cs.cmu.edu/~rwh/pfpl/supplements/natconat.pdf. 2023.

[3] Giuseppe Longo. "Set-theoretical models of -calculus: theories, expansions, isomorphisms". In: *Annals of Pure and Applied Logic* 24.2 (1983), pp. 153–188. DOI: 10.1016/0168-0072(83)90030-1.

[4] *New Type Idiom - Rust By Example*. https://doc.rust-lang.org/rust-by-example/generics/new_types.html.