

Inductiveness of types and Normalization of terms

Ki Yung Ahn

September 14, 2011

1 Introduction: dividing the problem space of typed formal reasoning systems

My proposed thesis is:

The two important concepts in typed formal reasoning systems (or, typed programming languages), *inductiveness of types* (i.e., well defined as logical propositions) and *normalization of terms* (i.e., canonical values and total functions), can be and must be considered separately.

We can categorize the problem space of typed formal reasoning systems (e.g., LCF, HOL, Coq, Twelf) by the inductiveness of types and normalization of terms. There are four combinations (IND , IND_{\perp} , REC , REC_{\perp}) in this two dimensional space as shown in Figure 1. The upper two (IND , IND_{\perp}) categorize inductive types and the lower two (REC , REC_{\perp}) categorize recursive types. The left two (IND , REC) only allow total functions and the right two (IND_{\perp} , REC_{\perp}) allow partial functions. These four fragments are related by inclusion: the upper ones are included in the lower ones ($IND \subset REC$, $IND_{\perp} \subset REC_{\perp}$) and the left ones are included in the right ones ($IND \subset IND_{\perp}$, $REC \subset REC_{\perp}$). And, as a consequence of transitivity, it is the case that $IND \subset REC_{\perp}$. Note, IND is the smallest and REC_{\perp} is the largest of the four divisions of the problem space. As we move to a smaller space, we are less expressive, but we have more properties guaranteed. As we move to a larger space, we are more expressive but have fewer properties guaranteed. For instance, in IND , we can not use general recursion to express terms, but have a normalization guarantee; whereas, in IND_{\perp} , we can use general recursion but no longer have a normalization guarantee.

An ideal general purpose formal reasoning system should be able to shift gears smoothly between these four spaces. If we can smoothly move around from one space to another, we can take advantage of the benefits of both the smaller space and the larger space. That is, we can express terms in flexible ways allowed by the larger space, yet also depend upon the desired properties guaranteed by the smaller space. For example, consider the following

	normalizing terms	possibly non-normalizing terms
inductive types	IND	IND _⊥
recursive types (possibly non-inductive)	REC	REC _⊥

Figure 1: The four problem spaces categorized by inductiveness and normalization

IND consists of the normalizing terms with inductive types. Terms of the simply typed lambda calculus, extended with finite ground types (e.g., unit, boolean) or (more generally) inductive types with primitive recursion (e.g., natural number, list, tree) also belong to IND.


IND_⊥ consists of possibly non-normalizing terms with inductive types. That is, both normalizing and non-normalizing terms with inductive types are in IND_⊥. Thus, any term belonging to IND also belongs to IND_⊥ (i.e., $\text{IND} \subset \text{IND}_{\perp}$) by definition. A typical way of extending a language, to break the IND containment, is to add unrestricted *general recursion* to the language. For example, the simply typed lambda calculus extended with general recursion (i.e., by adding a Y-combinator) have non-normalizing (or, non-terminating) terms, which then belong to IND_⊥ but not IND.

REC consists of normalizing terms of recursive types, which are possibly not inductive. That is, some recursive types correspond to inductive types, while others do not. So, IND is contained in REC (i.e., $\text{IND} \subset \text{REC}$) by definition. Terms of System F belong to REC, since System F is powerful enough to encode all recursive types, while remaining strongly normalizing. Although the fact that there exist sound languages contained in REC, such languages have often been overlooked in the design of formal reasoning systems.

REC_⊥ consists of possibly non-normalizing terms with recursive types, which are possibly not inductive. Unlike languages with inductive types, languages that allow unrestricted use of recursive types can have non-normalizing terms, even without unrestricted general recursion. For example, the simply typed lambda calculus extended with recursive types can have non-normalizing terms (i.e., can express non-terminating computations). Note, it is much easier to introduce non-termination in the presence of recursive types (where one doesn't need general recursion) than in the presence of inductive types (where one needs general recursion to introduce non-termination). For this reason, REC has often been neglected and REC_⊥ has mainly been considered for reasoning about languages with recursive types.

diagram of function compositions, which commute: That is, $f = g \circ f' \circ h$.

$$\begin{array}{ccc} A_1 & \xrightarrow{f} & A_2 \\ h \downarrow & & \uparrow g \\ B_1 & \xrightarrow{f'} & B_2 \end{array}$$

Let this diagram lie in **REC**. That is, all the functions (f, h, g, f') are total and all the types (A_1, A_2, B_1, B_2) are recursive types, which may or may not be inductive. Then I expect the following property to hold: If A_1 and A_2 are inductive, then f is in **IND**. 

3-1

The motivation for the diagram above and its related property is the following. We want to define f , which may be definable in **IND** but the definition is likely to be annoying since we have limited expressiveness in **IND**. It may be easier and more natural to define f' in **REC**, which is a function over non-inductive recursive types B_1 and B_2 , where the meaning of f' corresponds to f . Then, we would be able to define f as a composition of f' with (hopefully simple) mapping functions, g and h . Later on, we will introduce a well known example of Normalization by Evaluation (§2.4.3), which directly correspond to this diagram.

More generally, I expect that a function f in **REC** can also be considered to be in **IND** when domain and range of f are inductive types. We can also write this in the style of typing judgement as follows:

$$\text{REC-IND}(\rightarrow) \frac{\Gamma \vdash_{\text{REC}} f : A \rightarrow B \quad \Gamma \vdash_{\text{IND}} A : \star \quad \Gamma \vdash_{\text{IND}} B : \star}{\Gamma \vdash_{\text{IND}} f : A \rightarrow B}$$

Note, $A \rightarrow B$ is a inductive type when A and B are inductive. Thus, even more generally, I expect any term e in **REC** can also be considered to be in **IND** when its type is inductive:

$$\text{REC-IND} \frac{\Gamma \vdash_{\text{REC}} e : T \quad \Gamma \vdash_{\text{IND}} T : \star}{\Gamma \vdash_{\text{IND}} e : T}$$

The typing rule for going the other way (i.e., considering a term of **IND** as a term of **REC**) is trivial by the inclusion relation between **IND** and **REC**:

$$\text{IND-REC} \frac{\Gamma \vdash_{\text{IND}} e : T}{\Gamma \vdash_{\text{REC}} e : T}$$

3-2

There would be similar properties between other pairs of the four fragments, which are in an inclusion relation (e.g., **IND** and **IND_⊥**, **IND_⊥** and **REC_⊥**). In my thesis work, I will identify those properties and categorize prior work by the inclusion relation they depend upon. From my preliminary research, it seems that the relation between **IND** and **IND_⊥** has been most heavily studied and implemented in practice. Various *termination analysis* methods found in proof assistants, which are based on inductive types, can be categorized as relating **IND** and **IND_⊥**. The studies on *strictly positive types* and *monotonicity* relates **IND** and **REC**. Although there exists some theoretical studies relating **REC** and **REC_⊥**, few formal reasoning system make use of those studies in practice. Thus, in my dissertation, I will put more focus on relating **REC** and **REC_⊥** among the other inclusion relations. More detailed plans for my dissertation work will be presented in §4, after reviewing the background literature on inductive types and recursive types in §2 and summarizing our preliminary work in §3.

2 Background: Recursive types, Inductive types, and Normalization

4-1



The literature which studies recursive types and inductive types can be categorized into two different paradigms: the recursive type paradigm (types in programs), and the inductive type paradigm (types in logic).

The recursive type paradigm is a syntactic approach. It considers any type definition, constructed using a well formed syntax, as defining a valid type. The recursive type paradigm originated in the programming language community, where the primary interest was type safety. From this perspective, types are viewed as safety properties to be preserved throughout the execution of programs, which are possibly non-terminating. We discuss this perspective on recursive types in §2.1.

The inductive type paradigm is a semantic approach. It admits only the types that have simple and well-behaved interpretations (e.g., sets). The types are built up from basic types, whose interpretations are trivial (e.g., finite sets), using well-understood connectives and induction principles, so that all definable types have well-behaved interpretations by construction. The inductive type paradigm originated in the constructive mathematics community, where the primary interest was logical consistency. In this perspective, types are viewed as propositions inhabited by proofs. **A proof is a normal form** which has the proposition as its type. Thus, only the normalizing programs, which produce normal forms interpreted as sound proofs, can be well typed. We discuss this perspective of inductive types in §2.2.

4-2



There exist certain classes of recursive types, which cannot be admitted as valid types in the inductive type paradigm. One would naturally question “when do recursive types coincide with inductive types?” And, “how should recursive types be treated when they do not correspond to inductive types?”

The literature attempting to answer these questions is rich, and with many more questions than answers, remains an area of active research. This document discusses the relationship between recursive types and inductive types in §2.3, and exhibits some examples of recursive types that are not inductive types in §2.4. The general scope of the proposed thesis is to answer questions about how and when can programs involving recursive types in a programming language, can also be interpreted as proofs in a logic.

4-3

2.1 Recursive types



In the recursive type paradigm, a recursive type operator μ is provided for defining recursive types. The typing rules for μ allow a recursive type to be unrolled (i.e., the full type can be substituted for recursive occurrences in its body). This substitution can be used as many times as necessary. For example, a recursive type definition for lists containing elements of type A is $\mu \alpha. \text{Unit} + (A \times \alpha)$, which represents the solution of X for the recursive type equation $X = \text{Unit} + (A \times (X))$. That is, a list is either an empty list, represented by the value of type Unit , or a non empty list, represented by a pair value of type A (head) and type α (tail). Note, the type variable α , bound by μ , occurs where the recursive list type is

equi-recursive types	iso-recursive types
$\text{equi-roll} \frac{\Gamma \vdash e : T[\mu\alpha.T/\alpha]}{\Gamma \vdash e : \mu\alpha.T}$	$\text{iso-roll} \frac{\Gamma \vdash e : T[\mu\alpha.T/\alpha]}{\Gamma \vdash \text{roll } e : \mu\alpha.T}$
$\text{equi-unroll} \frac{\Gamma \vdash e : \mu\alpha.T}{\Gamma \vdash e : T[\mu\alpha.T/\alpha]}$	$\text{iso-unroll} \frac{\Gamma \vdash e : \mu\alpha.T}{\Gamma \vdash \text{unroll } e : T[\mu\alpha.T/\alpha]}$

Figure 2: Typing rules for equi-recursive types and iso-recursive types

expected (i.e., tail of the list). We are allowed to unroll the list type as many times as we need (going downwards):

$$\begin{aligned}
& \mu\alpha.\text{Unit} + (A \times \alpha) \\
& \text{Unit} + (A \times (\mu\alpha.\text{Unit} + (A \times \alpha))) \\
& \text{Unit} + (A \times (\text{Unit} + (A \times (\mu\alpha.\text{Unit} + (A \times \alpha))))) \\
& \text{Unit} + (A \times (\text{Unit} + (A \times (\text{Unit} + (A \times (\mu\alpha.\text{Unit} + (A \times \alpha)))))) \\
& \vdots
\end{aligned}$$

Conversely, we can roll the list type the other way (going upwards). The typing rules for unrolling and rolling recursive types are based purely on syntactic substitution (see Figure 2), regardless of whether a recursive type definition actually has a well behaved interpretation. In particular, it is well known that we can construct non-terminating terms with unrestricted use of recursive types, even without introducing unrestricted general recursion into the programming language. When our interest is not strong normalization but only type safety, having non-termination in the programming language is not a problem, as long as we can show type safety of the language. General purpose programming languages, which are Turing complete, should be able to express non-terminating computations anyway. Therefore, typed lambda calculi extended with recursive types have been studied as theoretic models for understanding general purpose programming languages. These studies include functional languages (e.g. ML, Haskell), object-oriented languages (e.g. Java), and imperative languages (e.g. Algol 68 [59]).

There are two styles of typing rules for recursive types (Figure 2): *equi-recursive types* and *iso-recursive types*. The typing rules for equi-recursive types allow implicit rolling unrolling of recursive types. That is, equi-recursive types are not syntax directed, and the implementation of the type system should infer where to apply the rolling and unrolling of recursive types. The typing rules for iso-recursive types are syntax directed. The explicit term syntax, **roll** and **unroll**, guides exactly where to apply the rolling and unrolling rules. Thus, type checking does not become any more complicated by adding iso-recursive types into the language. However, we need an additional reduction rule, such as **unroll** (**roll** e) $\rightarrow e$, for the extra term syntax introduced by adding iso-recursive types.

System F [31, 53] and its related extensions, such as F_ω , are particularly interesting when studying the recursive type paradigm, since it is possible to embed all recursive type definitions inside these systems. Such embeddings are, of course, less expressive than having

the recursive type operator μ as a primitive language construct, since languages like F and F_ω are strongly normalizing. Embeddings of recursive types in such languages restrict certain *uses* of these recursive types, while being able to *define* all of them. More specifically, such embeddings amount to supporting arbitrary use of rolling, but restricting the use of unrolling. For functional programmers, this would amount to the free use of data constructors to construct values, but restricted use of pattern matching to destruct existing values. Such a pattern of terminating computation over recursive types is called *iteration* in contrast to primitive recursion. We will discuss further on iteration and recursion over non-inductive types in §3.

Another way of retaining strong normalization in typed lambda calculi, in the presence of recursive types, is to limit the recursive type definitions to only the well-behaved ones [41–43]. This alternative approach is closely related to the paradigm of inductive types. We discuss further the relationship between the recursive types and inductive types in §2.3.

2.2 Inductive types

In the inductive type paradigm, types must have well-behaved interpretations (i.e., sets). Thus, types should be built from well-understood types using well-understood connectives, so that all types have well-behaved interpretations by construction. Martin-Löf’s Intuitionistic Type Theory [39] is the representative system using this paradigm. So, we often refer to the inductive type paradigm as the Martin-Löf paradigm. In Martin-Löf’s type theory, finite sets (or, finite types) are given, and we can build other types by a dependent function connective (Π -type), a dependent pair connective (Σ -type), and a well-founded induction principle (W -type). This paradigm is suitable for designing type systems of formal proof assistants, which support logical reasoning by interpreting types as propositions and programs of those types as proofs (a.k.a. Curry-Howard correspondence).

Note, Martin-Löf’s type theory [39] is a very powerful system supporting transfinite induction (W -type) and dependent types (Π -type, Σ -type). The inductive type paradigm naturally incorporates dependent types by interpreting them as indexed families of sets.

2.3 Approaches to relating recursive types and inductive types

Not all recursive types are inductive types. That is, there are recursive types, which cannot be interpreted as types in the inductive type paradigm. For example, The recursive type $\mu\alpha.\alpha \rightarrow \alpha$ is a classic example of a recursive type without a well defined meaning as a set. Types whose definitions involve the function space over the type being defined often have this problem. Such types are often called *reflexive types*. More specifically, recursive type definitions involving function spaces, which mention the recursive type being defined in the domain of a function space (i.e., left-hand side of \rightarrow), do not correspond to inductive types. We will see more examples of such non-inductive recursive types in §2.4.

Knowing that non-inductive recursive types exist, one would naturally question

Question (1): when do recursive types coincide with inductive types, and

Question (2): how should recursive types be treated when they don't.

There have been many studies around these questions, and the proposed thesis will hopefully answer some aspects of these questions.

Question (1): When do recursive types coincide with inductive types?

A widely accepted answer to this question is that when the recursive type is *strictly positive*. A recursive type is strictly positive when the recursive type variable does not appear free on the left-hand side of the \rightarrow (i.e., in the domain of the function space), but only on the right-hand side of the \rightarrow (i.e., in the range of the function space). For sum types and product types, both of their components should be strictly positive. Non-recursive types are strictly positive by default. Dybjer [27] showed that any strictly positive type definitions using single recursive variable (i.e., only one μ appears in a type definition) can be represented using W -types. Abbott, Altenkirch, and Ghani [2, 3] generalized Dybjer's work [27] to strictly positive type definitions using arbitrary number of recursive type variables (i.e., many μ can appear in a type definition). Gambino and Hyland [29] have generalized these results [2, 3, 27] to dependently typed setting. Coquand and Paulin [21] developed a similar construction based on Calculus of Constructions (CC, or CoC) [22], but without relating the inductive definitions with W -types.

However, the notion of strict positivity does not generalize well to richer families of datatypes in languages more expressive than System F. Strict positivity is a syntactic condition, which makes good sense for recursive types in the context of System F (and, of course, in more simple languages like the simply typed lambda calculus) where recursive type variables bound by μ have kind \star (i.e., range over types rather than type constructors). In more expressive languages like F_ω , it becomes unclear how we should generalize the syntactic condition of strict positivity, since we can also have recursive type operators for type constructors (of arbitrary kinds) as well as for types (of kind \star). That is, we have a family of recursive type operators μ_κ indexed by kind κ , which can express richer families of recursive types. For example, the μ operator for types corresponds to μ_\star . In order to have $\mu_\star \alpha_0. T$ be well kinded, the bound variable should be kinded: $\alpha_0 : \star$, the body of the μ should be kinded: $T : \star$, and as a result the complete μ -expression is kinded as $(\mu_\star \alpha_0. T) : \star$.

For type constructors of kind $\star \rightarrow \star$, which take one type argument to produce a type, we have the $\mu_{\star \rightarrow \star}$ operator to define recursive type constructors, or families of recursive types indexed by a type. In order to have $\mu_{\star \rightarrow \star} \alpha_1. F$ be well kinded, the bound variable should be kinded: $\alpha : \star \rightarrow \star$, the body of the μ should be kinded: $F : \star \rightarrow \star$, and as a result the complete μ -expressions should be kinded: $(\mu_{\star \rightarrow \star} \alpha_1. F) : \star \rightarrow \star$.

Note, it is not enough to restrict $\alpha_1 : \star \rightarrow \star$ to be used in strictly positive positions since it represents the type constructor defined by the recursive definition, which takes a type and produces a type. In the case of μ_\star , where the type variable α_0 ranges over simple types, it is okay to consider that α_0 corresponds to well-behaved inductive types, provided that all the ground types in the language (e.g., `Unit`) correspond to well-behaved inductive types. For the type variable α_1 which is bound by $\mu_{\star \rightarrow \star}$ (more generally, by μ_κ where κ is not \star), which

ranges over type constructors, we cannot make the same argument as we did for α_0 , since we don't really have ground type constructors to serve as a base case argument.

Matthes [42] gives a remarkably good answer by relating extensions of System F with inductive types and fixed-point types (i.e., recursive types) using a notion of *monotonicity*, and later generalizes the notion of monotonicity to an extension of System F_ω for type constructors of rank 2 [43]. Instead of requiring syntactic constraints on $\mu\alpha.T$, only a *monotonicity witness*, which is a term of type $\forall\alpha.\forall\beta(\alpha \rightarrow \beta) \rightarrow T \rightarrow T[\beta/\alpha]$, is required [42]. The primitive recursion over any recursive type (of rank 1, or kind \star) is guaranteed to terminate, once we can provide a monotonicity witness for that type. Matthes [43] generalizes the notion of monotonicity up to rank 2 type constructors, and poses the open question whether the notion of monotonicity could generalize further to type constructors of rank higher than 2.

Although monotone types are very good penalization of strictly positive datatypes and positive datatypes, sharing the same computational behavior in regards to primitive recursion, it is not the definitive answer for types being inductive. Being a monotone type makes it a good candidate for being an inductive type, but it is not a sufficient condition. The inductive type paradigm requires set theoretic interpretation of types to view them as propositions. Some positive, but not strictly positive, datatypes do not have set theoretic interpretations, although all positive datatypes are monotone. For instance, The recursive type $\mu\alpha.(\alpha \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Bool}$ is positive, but not strictly positive. This type, when interpreted as a set theoretic proposition, asserts an isomorphism between the powerset of powerset of α and α itself, which is a set theoretic nonsense. Whether and how monotone data constructors of higher-rank can be understood inside the inductive type paradigm is (to my knowledge) an open question.

Question (2): How should non-inductive recursive types be treated?

Some recursive types do not correspond to inductive types. That is, there exist recursive types, which cannot be interpreted as sets. Mendler [45] showed that reflexive types can introduce non-terminating computation even without having unrestricted general recursion in the language. Constable and Mendler [18] suggested an approach of interpreting non-inductive recursive types as Scott domains, and the function space over such recursive types as partial functions over the Scott domains. Scott domains are mathematical models, developed by Scott [56], for the languages capable of expressing non-terminating computations such as the untyped lambda calculus and languages with unrestricted general recursion. For example, in the type for infinitely branching trees $\mu\alpha.T + (\mathbb{N} \rightarrow \alpha)$, where the leaves contain values of type T , they use the usual arrow (\rightarrow) for total functions, and in the reflexive type $\mu\alpha.\alpha \rightsquigarrow \alpha$, which represents the semantics of the untyped lambda calculus, they use a different arrow (\rightsquigarrow) for partial functions.

Although interpreting non-inductive recursive types as Scott domains is indeed a sound interpretation, it is, in a way, an over-approximation. Non-termination is not a characteristics of non-inductive types in general. Recall that the inductive type paradigm is all about insisting that types have simple interpretations as sets (more generally, indexed families of sets). When we are faithful to the design principles of inductive types, we get strong normal-


ization of the language as a consequence. However, not all strongly normalizing languages belong to the inductive type paradigm. There exist well known strongly normalizing languages, which do not have set theoretic interpretations. Reynolds and Plotkin [54] proved the non-existence of a set theoretic interpretation for System F using the encodings of non-strictly positive datatypes. Sets cannot be interpretations for some System F types, but neither are Scott domains satisfactory, since all functions are total in System F. Therefore, it is an over-approximation to categorize all the function spaces over non-inductive recursive types as partial functions of Scott’s domain theory.

Therefore, I strongly believe that a better approach is to separate concerns about termination of programs from concerns about the inductiveness of types definitions (recall Figure 1 discussed in §1). These two properties need not always be linked by design: inductiveness does require termination, but non-inductiveness does not imply non-termination. Let us first observe how certain desired properties are guaranteed for inductive types, and recognize that the same strategy can apply to recursive types.

We should recognize that the type *definition* itself does not guarantee types to be inductive (i.e., interpreted as sets). We also need some restrictions on how we *use* the terms of those types. When we allow unrestricted general recursion in the language, types cannot be interpreted as sets, regardless of whether they are inductive or not. In the presence of unrestricted general recursion, all types need to be interpreted as Scott domains, in order to handle non-termination. Note, the type definitions, which look like inductive types, (or possibly inductive definitions) become truly inductive type definitions only when we stick to principled recursion schemes, which are known to guarantee termination (e.g., primitive recursion, structural recursion). In other words, we must rely on principled *use* of inductive types to guarantee desired properties such as termination and the Curry-Howard correspondence (i.e., types are propositions and programs are proofs), as well as the inductiveness of their *definitions*.

Similarly, principled *uses* of recursive types can guarantee certain desired properties such as termination. Although the termination property (or strong normalization) of recursive types has been observed in various contexts [5, 6, 9, 14, 24, 26, 28, 30, 40, 41, 44–46, 61], it has not been put to use in a systematic way of handling termination separate from inductiveness in any language design we are aware of. There have been largely two approaches to handling non-termination in formal reasoning systems: using coinductive types and modeling types as domains. Neither of these two approaches handle all possible combinations of termination and inductiveness in a systematic way. The coinductive types approach are limited to strictly positive types, just as inductive types are. The types as domains approach has the problem of over-approximating non-termination as we discussed earlier.


Many proof assistants (e.g. Cog, Agda, Epigram) support coinductive types as well as inductive types. Coinductive types (or, greatest fixpoint types), which are dual constructions of inductive types (or, least fixpoint types), can have values that are possibly infinite (e.g., infinite lists). We can model certain class of non-terminating computations by principled corecursion schemes over such infinite structures (or, coductive values). Nevertheless, coinductive types are limited to strictly positive datatypes, just as inductive types are. That

is, even with coinductive types, we cannot directly express and reason about all recursive types in general, especially the non-inductive types. 

The types as domains approach originates from Scott's observation that types used in programs are different from types used in logic [55, 57]. The systems [33, 48, 49] supporting Scott's Logic for Computable Functions (LCF) model types as domains to reason about possibly non-terminating programs. The functions in LCF are partial functions defined over Scott domains. **However, LCF is not designed for reasoning about types in programs coincide with types in logic, or totality of functions defined by principled recursion schemes.** In other words, using the four problem spaces we defined in §1, we can say that Scott's domain theory, or LCF, does not distinguish REC from REC_\perp .

We will discuss some more details on the termination property of non-inductive recursive types in §3.

2.4 Examples of non-inductive recursive types

In this section, I introduce three example programs which make use of non-inductive recursive types: Scott numerals (§2.4.1), Higher-Order Abstract Syntax (§2.4.2), and Normalization by Evaluation (§2.4.3). Here, and in the other parts of this document, we will use the *strictly positive types* as an example of inductive recursive types for simplicity, rather than examining the more general concept of monotonicity. We can categorize non-inductive recursive types, that is non-strictly positive datatypes, into two categories: (1) positive (but not strictly positive) datatypes, and (2) negative datatypes. The type for the Scott numerals is an example of a (not strictly) positive datatype (a.k.a. covariant recursive types), and the others are examples of negative datatypes (a.k.a. contravariant types). 

2.4.1 Scott numerals

There are multiple ways of encoding the natural numbers in lambda calculi. The Church numerals are the most well known of those encodings. An alternate encoding in the untyped lambda calculus is one proposed by Scott. In 1993, Abadi, Cardelli, and Plotkin [1] wrote a note on assigning a type to the constructors of the Scott numerals in an extended System F (extended with covariant recursive types, i.e., positive datatypes).

You can observe the different encodings of zero (0), successor (`succ`), and the primitive conditional operations: `case` (for the Scott numerals), and `zero?` (for the Church numerals) in Figure 4. The expression `zero? n` reduces to `T` when `n` reduces to the Church numeral 0, and `F` otherwise, where $T \stackrel{\text{def}}{=} \lambda x. \lambda y. x$ and $F \stackrel{\text{def}}{=} \lambda x. \lambda y. y$ are the Church encodings of the boolean values, True and False. The expression `case n a f`, reduces to `a`, when `n` reduces to the Scott numeral 0, and `f n'` otherwise, where `n'` is the predecessor of `n`.

Let us start the discussion in the context of untyped lambda calculus (see the upper row of Figure 4).

The normal form for the Church numeral of value `n` is $\lambda x. \lambda f. f^n x$, where $f^n x$ is an abbreviation for `n` applications of `f` to `x`. For instance, the normal form for the Church numeral

Church numerals in untyped λ calculus

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \lambda f. \lambda x. x \\ \text{succ} &\stackrel{\text{def}}{=} \lambda n. \lambda f. \lambda x. f (n f x) \\ \text{zero?} &\stackrel{\text{def}}{=} \lambda n. n (\lambda x. \text{F}) \text{T} \end{aligned}$$

Scott numerals in untyped λ calculus

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \lambda x. \lambda y. x \\ \text{succ} &\stackrel{\text{def}}{=} \lambda n. \lambda x. \lambda y. y n \\ \text{case} &\stackrel{\text{def}}{=} \lambda n. \lambda a. \lambda f. n a f \end{aligned}$$

Church numerals in System F

$$\begin{aligned} N &\stackrel{\text{def}}{=} \forall \beta. (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \\ 0 &\stackrel{\text{def}}{=} \Lambda \beta. \lambda f : (\beta \rightarrow \beta). \lambda x : \beta. x \\ &: N \\ \text{succ} &\stackrel{\text{def}}{=} \lambda n : N. \Lambda \beta. \lambda f : (\beta \rightarrow \beta). \lambda x : \beta. f (n \beta f x) \\ &: N \rightarrow N \\ \text{zero?} &\stackrel{\text{def}}{=} \lambda n : N. \Lambda \beta. n \beta (\text{T} \beta) (\lambda x : \beta. \text{F} \beta) \\ &: N \rightarrow B \end{aligned}$$

Scott numerals in System $F^{+\mu(\text{equi})}$

$$\begin{aligned} S &\stackrel{\text{def}}{=} \mu \alpha. \forall \beta. \beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \\ 0 &\stackrel{\text{def}}{=} \Lambda \beta. \lambda x : \beta. \lambda y : (S \rightarrow \beta). x \\ &: S \\ \text{succ} &\stackrel{\text{def}}{=} \lambda n : S. \Lambda \beta. \lambda x : \beta. \lambda y : (S \rightarrow \beta). y n \\ &: S \rightarrow S \\ \text{case} &\stackrel{\text{def}}{=} \lambda n. \lambda a. \lambda f. n \alpha a f \\ &: S \rightarrow \forall \beta. \beta \rightarrow (S \rightarrow \beta) \rightarrow \beta \end{aligned}$$

Figure 3: The Church numerals and the Scott numerals in the untyped and typed λ calculi

3 is $\lambda x. \lambda f. f(f(f x))$. The normal form for the Scott numeral of value $n > 0$ is $\lambda x. \lambda y. y n'$, where n' is the normal form for the Scott numeral of value $n - 1$ (i.e., predecessor of n). For instance, the normal form for the Scott numeral 3 is $\lambda x. \lambda y. y(\lambda x. \lambda y. y(\lambda x. \lambda y. y(\lambda x. \lambda y. x)))$.

One advantage of the Scott numerals over the Church numerals is the existence of the predecessor function $\lambda n. n 0 \text{id}$, where $\text{id} \stackrel{\text{def}}{=} \lambda x. x$, which reduces in a constant number of steps when applied to a Scott numeral in normal form. This is not the case for the predecessor function for Church numerals, which needs a number of reduction steps linearly proportional to the value of the applied argument.

Now, let us shift our discussion to similar encodings in a typed calculi that are powerful enough to assign types to each of the different encodings of natural numbers (see the bottom row of Figure 4).

On one hand, we can assign types to the Church numerals in System F (without any extensions). The type N for the Church numerals is $\forall \beta. (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$, which is an impredicative encoding (i.e., β can be instantiated with N itself) of the natural number type. The boolean type (B) and its values (T and F) appearing in the definition of **zero?** are defined as: $B \stackrel{\text{def}}{=} \forall \beta. \beta \rightarrow \beta \rightarrow \beta$, $\text{T} \stackrel{\text{def}}{=} \Lambda \beta. \lambda x : \beta. \lambda y : \beta. x$, and $\text{F} \stackrel{\text{def}}{=} \Lambda \beta. \lambda x : \beta. \lambda y : \beta. y$.

On the other hand, we cannot assign proper types to the Scott numerals in System F. We can assign types to the Scott numerals only when we have an extended System F with positive datatypes. This extra power required for the type system, in order to type the Scott numerals, is due to the ability to define a constant time predecessor.¹

¹More generally, predecessor-like functions (e.g., a tail function for lists) of constant reduction steps are

```

{-# LANGUAGE RankNTypes #-}

module Scott where

data Scott = Scott
    (forall b . b          -- return this if its zero
     -> (Scott -> b)       -- how to proceed given the predecessor?
     -> b)

z    = Scott n  where n z s = z

s x = Scott n  where n :: b -> (Scott -> b) -> b
              n z s = s x

scottCase :: Scott -> a -> (Scott -> a) -> a
scottCase (Scott n) a f = n a f

pred n = scottCase n z id

pred2 (Scott n) = n z id

```

Figure 4: Scott numerals in Haskell

The type S for Scott Numerals is defined to be $\mu\alpha.\forall\beta.\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$, which is a positive, but not strictly positive, datatype. Note, α appears in a double negative position, thus, positive. To make this clear, let us explicitly parenthesize the \rightarrow , which is right associative, as follows: $\mu\alpha.\forall\beta.\beta \rightarrow ((\alpha \rightarrow \beta) \rightarrow \beta)$. Then, we can observe that $((\alpha \rightarrow \beta) \rightarrow \beta)$ is in a positive position since it appears to the right of \rightarrow . The subterm $(\alpha \rightarrow \beta)$ is in a negative position, since it is to the left of \rightarrow , and the variable α is also in negative position inside the negative subterm $(\alpha \rightarrow \beta)$. Thus, the variable α is in a positive (or, covariant) position.

Figure 4 may be helpful for understanding Scott numerals, if you are familiar with Haskell or any other similar functional language.

2.4.2 Higher-Order Abstract Syntax

An important example that uses a negative datatype is Higher-Order Abstract Syntax (HOAS) [17, 47, 51]. HOAS is a technique used to model an object-language with a syntactic form which is binding construct (the syntactic form binds a variable in the scope of another

not known to be definable in System F. This is a characteristic that distinguishes iteration from (primitive) recursion. I will discuss this further in §3.

term). The technique uses a data structure in the host-language (or, meta-language) which employs a meta-language function embedded in the data structure to encode the binding construct.

For instance, the recursive type definition of the HOAS for the untyped lambda calculus, which is the most simple HOAS, can be defined as $\mu\alpha.(\alpha \rightarrow \alpha) + (\alpha \times \alpha)$. Note, α appears in a negative position, left of \rightarrow in $(\alpha \rightarrow \alpha)$. In this example the left summand $(\alpha \rightarrow \alpha)$ is used to encode the binding lambda, and the right summand $(\alpha \times \alpha)$ is used to encode the binary application. In Haskell we might encode this negative datatype as follows:

```
data Term = Lam (Term -> Term) | App (Term, Term)
```

When we use HOAS, we get capture avoiding substitution over the object-language terms, as a simple homomorphism. That is, HOAS lifts the burden of writing substitution functions in programming language implementations, and the burden of proving substitution lemmas in the mechanized metatheories of programming languages. For this reason, HOAS is used in implementations of interpreters and partial evaluators [15, 23, 58], and in mechanized theories [4, 16, 25, 35].


2.4.3 Normalization by Evaluation

Another well-known use of contravariant recursive types (or, negative datatypes) appear in the work on Normalization by Evaluation (NbE) [11]. The idea of NbE is to use an evaluator, or an interpreter, (i.e., denotational semantics) for normalization. More specifically, NbE works by evaluating a syntactic term into a value in the semantic domain, and then reifying the resulting value of the evaluation back into a normalized syntactic term. The recursive type representing any interesting semantic domain involving functions is always a negative datatype, since the semantic domain would contain its function space (i.e., $D \supseteq [D \rightarrow D]$). Recently, NbE has been popularized as an implementation technique for dependently typed languages [7, 8, 37], some of which are specifically studied in the context of Martin-Löf's type theory. Note, such implementations using NbE rely on recursive type definitions for semantic domains, which are not admissible in Martin-Löf's type theory. An amusing irony: to believe in the soundness of this kind of implementations for the Martin-Löf's type theory, we also have to believe in the soundness of certain use of recursive types, which are outside the scope of the Martin-Löf paradigm.

3 Preliminary work: Mendler style Iteration over Negative Datatypes

Our preliminary work [9] is about a Mendler style iteration over negative datatypes using syntactic inverses. In this section, I will introduce the concept of iteration and recursion (§3.1), and iteration in Mendler style (§3.2). Then, I will give a summary of the preliminary results (§3.3), related work (§3.4), and future work (§3.5) I plan to work on.



3.1 Iteration, (primitive) recursion, and induction


In this subsection, we discuss datatypes in regards to iteration, recursion, and induction. I will first overview the inclusion relation between the types related to each  see concepts (§3.1.1), and contrast the difference between iteration and primitive recursion on natural numbers (§3.1.2). Then, I will introduce folds (§3.1.3), which are implementations of iteration in functional language, over inductive datatypes other than natural numbers.

14-1

3.1.1 Inclusion relations of the types relevant to each concept

14-2

Induction and recursion: All types having well-founded  induction are also normalizing under primitive recursion. However, there exists types normalizing under primitive recursion, which do not  well-founded induction principle with set theoretic interpretations.

A proof by  induction can be realized by primitive recursion [52]. That is, the computational content of a proof term using an induction principle is no more than a primitive recursive function. Pfenning and Paulin-mohring [52] showed that extending the Calculus of Constructions by inductive types and their induction principles does not alter the set of functions in its computational fragment, F_ω . In short, inductive types are normalizing under primitive recursion. However, not all primitive recursive functions have their logical counterparts of inductive predicates. As we mentioned earlier in §2.3, not all types normalizing under primitive recursion have set theoretic interpretations.

Recursion and iteration: All types normalizing under primitive recursion are also normalizing under iteration. However, there exists types normalizing under iteration but not under primitive recursion. Negative datatypes are not normalizing under primitive recursion. In fact, Mendler observed that negative datatypes are not normalizing even without any recursion at the term level. The following is Mendler's observation transcribed into Haskell:

```
data T = C (T -> ())
p (C f) = f
w t = (p t) t
```

```
selfapp = w (C w)  -- corresponds to (\x.xx) (\x.xx) in untyped lambda calc
```

The inductive datatype T is negative, and p and w are well typed non-recursive functions. Note, we did not use any recursion above, yet `selfapp` diverges: $w (C w) \rightsquigarrow (p (C w)) (C w) \rightsquigarrow w (C w) \rightsquigarrow \dots$. Another more interesting example, which shows that negative datatypes can encode non-termination, is the HOAS for untyped lambda calculus:

```
data Exp = Lam (Exp -> Exp) | App (Exp, Exp)
```

This datatype can model arbitrary terms in the untyped lambda calculus, some of which are diverging.

I have mentioned several times in this document that iteration can ensure normalization for negative datatypes as well as positive datatypes, but have not yet introduced what iteration is. In the remainder of this subsection, I will introduce the concept of iteration in comparison to primitive recursion. I will first start with the most simple case of natural numbers, and then other inductive types such as lists and trees. The normalization property of iteration over recursive types in general, including negative datatypes, will be discussed in the next subsection (§3.2) when we compare two different styles of forming catamorphsim, which is another name for iteration in the context of functional languages.

3.1.2 Primitive recursion and iteration on natural numbers

Primitive recursion: The primitive recursion on natural numbers, in the tradition of Church's System T [32], can be defined by the three reduction rules as follows:

$$\begin{array}{c} \text{Pr-0} \frac{}{\text{Pr } 0 \ e_0 \ e_2 \rightarrow e_0} \\ \text{Pr-s} \frac{}{\text{Pr } (\text{S } n) \ e_0 \ e_2 \rightarrow e_2 \ n \ (\text{Pr } n \ e_0 \ e_2)} \\ \text{Pr-ctx} \frac{e \rightarrow e'}{\text{Pr } e \ e_0 \ e_2 \rightarrow \text{Pr } e' \ e_0 \ e_2} \end{array}$$

The primitive recursion operator, or recursor, **Pr** has three arguments: the first argument is the natural number to recurse on; the second argument is the resulting expression when the value of the first argument is zero; and the third argument is an expression expecting two arguments, which we use when the value of the first argument is non-zero.

The first rule **Pr-0** defines the reduction when the first argument is zero (0), simply reducing to e_0 .

The second rule **Pr-s** defines the reduction when the first argument is in the successor form ($\text{S } n$). The result of the reduction is e_2 applied to the predecessor n and the result of the primitive recursion over the predecessor ($\text{Pr } n \ e_0 \ e_2$).

The third rule **Pr-ctx** defines the reduction when the first argument is not in canonical form (i.e., either 0 or $\text{S } n$). It is a self explanatory context rule.

Iteration: We can formulate the iteration on natural numbers in a similar fashion to the primitive recursion as follows:

$$\begin{array}{c} \text{It-0} \frac{}{\text{It } 0 \ e_0 \ e_1 \rightarrow e_0} \\ \text{It-s} \frac{}{\text{It } (\text{S } n) \ e_0 \ e_1 \rightarrow e_1 \ (\text{It } n \ e_0 \ e_1)} \\ \text{It-ctx} \frac{e \rightarrow e'}{\text{It } e \ e_0 \ e_1 \rightarrow \text{It } e' \ e_0 \ e_1} \end{array}$$

The three reduction rules for the iteration operator, or iterator, **It** are very similar to the definition of the recursor **Pr**. The only difference from primitive recursion is that iteration does not have direct access to the predecessor. Note, e_1 only expects one argument ($\text{It } n \ e_0 \ e_1$), which is the result of the iteration over the predecessor, in the second rule **it-s**.

Comparison of recursion and iteration: We can calculate the predecessor of n using the recursor by $\text{Pr } n \ 0 \ (\lambda x. \lambda y. x)$ in constant time, provided that n is in canonical form.

Calculating the predecessor using the iterator is not as simple as using the recursor, since we can no longer directly refer to the predecessor in iteration. It is known that the iterator It has the same computational power as the recursor Pr , provided that we have pairs in the language [10]. We can define Pr and It in terms of each other. Defining It in terms of Pr is trivial:

$$\text{It } n \ e_0 \ e_1 \stackrel{\text{def}}{=} \text{Pr } n \ e_0 \ (\lambda x. e_1) \quad \text{where } x \text{ does not appear free in } e_1$$

Conversely, we can define Pr in terms of It using pairs, storing the predecessor in the first element and the result of the iteration in the second element, as follows:

$$\text{Pr } n \ e_0 \ e_2 \stackrel{\text{def}}{=} \pi_2(\text{It } n \ \langle e_0, 0 \rangle \ (\lambda y. \langle e_2 (\pi_1 y) (\pi_2 y), S(\pi_1 y) \rangle))$$

However, the number of reduction steps required for calculating the predecessor is not constant when we use this encoding of the recursor, which is defined in terms of the iterator. Using this encoding of the recursor, the number of reduction steps for calculating the predecessor is linear to the value of the given natural number n .

The observation that computational power of primitive recursion and iteration is the same but efficiency differs holds for inductive datatypes more generally. I will shortly introduce the iteration for other datatypes, which is also called folds in the context of function programming, in §3.1.3. However, for non-inductive datatypes, especially for negative datatypes, this observation no longer holds. In fact, iteration needs to be formulated in a different style since popular style of formulating folds only generalize to limited class of inductive types (i.e., not even all inductive datatypes, not to mention of non-inductive datatypes such as negative datatypes). I will introduce two different styles of formulating iteration in §3.2.

3.1.3 Iterators, or folds, for other inductive datatypes

We have discussed iteration over natural numbers so far. Similarly, we can define iteration for other inductive datatypes. In functional languages, the functions implementing iteration are called folds. The following Haskell code is the definitions of folds for several datatypes:

```
data Nat    = Zero | Succ Nat
data List p = Nil  | Cons p (List p)
data Tree p = Leaf p | Node (Tree p) (Tree p)
data Blah p = Con1 p | Con2 Int p | Con3 p (Blah p) | Con4 (Blah p) p

foldNat :: a -> (a->a) -> Nat -> a
foldNat v f Zero      = v
foldNat v f (Succ n) = f (foldNat v f n)

foldList :: a -> (p->a->a) -> List p -> a
foldList v f Nil      = v
foldList v f (Cons x xs) = f x (foldList v f xs)
```



```

foldTree :: (p->a) -> (a->a->a) -> Tree p -> a
foldTree fL fN (Leaf x)      = fL x
foldTree fL fN (Node t1 t2) = fN (foldTree fL fN t1) (foldTree fL fN t2)

foldBlah :: (p->a) -> (Int->p->a) -> (p->a->a) -> (a->p->a) -> Blah p -> a
foldBlah f1 f2 f3 f4 (Con1 x)    = f1 x
foldBlah f1 f2 f3 f4 (Con2 n x) = f2 n x
foldBlah f1 f2 f3 f4 (Con3 x b) = f3 x (foldBlah f1 f2 f3 f4 b)
foldBlah f1 f2 f3 f4 (Con3 b x) = f4 (foldBlah f1 f2 f3 f4 b) x

```

The function `foldNat` for the natural number type `Nat` is basically the same definition to the iterator `lt` we discussed earlier, except that the natural number we recurse on passed into the last argument rather than the first argument. The first argument `v` is the answer when the last argument is zero (`Zero`), and the second argument `f` is the function to be applied to answer of the fold over the predecessor (`foldNat v f n`) when the last argument is non-zero (`Succ n`).

The function `foldList` is the fold for the list type `List`. Since `List` have two constructors, like `Nat` does, `foldList` also have two arguments before the list argument to fold over. The difference from `foldNat` is that the second argument `f` is a binary function, rather than unary function, for the non-empty list case, which combines the head element (`x`) with the answer of the fold over the tail (`foldList v f xs`).

The function `foldTree` is the fold for the binary tree type `Tree`. Since there are two constructors `Leaf` and `Node`, we have two arguments `fL` and `fN` to handle each constructor. The `fL` is a unary function applied to the value inside `Leaf`. The `fN` is a binary function combines the answers of the folds over the left and right children of the `Node`.

You can see that the type signature of the fold functions become larger as the number and arity of the data constructors become larger. The fold for `Blah` type (`foldBlah`) have four arguments (`f1`, `f2`, `f3`, and `f4` before it takes the `Blah` argument to fold over. Each of those argument has the appropriate type to handle the values inside each of the constructor.

All of these folds are normalizing, provided that supplied arguments are normalizing. Why? It is because they are structurally recursive on its last argument. The recursive call is always on the recursive subcomponents. For `Nat`, the recursive call is on the predecessor; for `List`, the recursive call is on the tail; for `Tree`, the recursive calls are on the children in the definition of their folds. When datatypes are well founded, which is the case for inductive datatypes, they is guaranteed to terminate by structural recursion.

Let us see some functions we can define in terms of folds. For example, we can define a sum function for lists (`sumList`) and a length function for lists `lenList` as follows:

```

sumList = foldList 0 (\x a->x+a)
lenList = foldList 0 (\x a->1+a)

```

The following illustrates some of the reduction steps calculating the sum of all the elements in an integer list using `sumList`:

```

    sumList (Cons 1 (Cons 2 (Cons 3 Nil)))
  ~>* foldList 0 (\x a->x+a) (Cons 1 (Cons 2 (Cons 3 Nil)))
  ~>* (\x a->x+a) 1 (foldList 0 (\x a->x+a) (Cons 2 (Cons 3 Nil)))
  ~>* (\x a->x+a) 1 5
  ~>* 1+5
  ~>* 6

```

We will revisit this example when we discuss catamorphism.

3.2 Conventional (Squiggol style) vs. Mendler style

Another name for iteration, in the context of functional programming, is catamorphism. Catamorphism is a generalization of folds. So far, we have seen how to formulate folds individually for each inductive datatype. Although we can informally observe that those folds are based on the same concept and have similar properties, we have not yet formally related them, or unified into a common interface. To formally describe the general concept of iteration (a.k.a. catamorphism), or unify the interface of folds, we first tear down an inductive type into two levels, a datatype fixpoint, which ties the knot of the recursion in a datatype, and a base datatype, which describes the shape of the datatype including where the recursion occurs. Such a two level type definitions correspond to iso-recursive types, whereas plain recursive definitions in the previous subsection correspond to equi-recursive types. The two level types are commonly used in both conventional and Mendler style catamorphism.

The conventional (or, Squiggol style) catamorphism (§3.2.2) is a generalization of folds (§3.1.3). The Mendler style catamorphism (§3.2.4) is another way of formulating catamorphism. The Mendler style catamorphism is considerably more expressive than the conventional catamorphism.

3.2.1 Two level types are iso-recursive types

Let us see how we can define the datatypes in the example of folds (§3.1.3).

The natural number datatype `Nat` can be split into two levels as follows:

```

newtype Mu f = Roll (f (Mu f)) -- datatype fixpoint
unRoll (Roll e) = e             -- reduction rule for unRoll . Roll composition

data N r = Z | S r              -- base datatype for Nat

type Nat = Mu N                 -- Nat defined in terms of Mu and N

zero  = Roll Z                  -- data constructor corresponding to Zero
succ n = Roll (S n)             -- data constructor corresponding to Succ

```

The datatype fixpoint `Mu` correspond to the μ binder of the recursive type we discussed earlier in §2.1. The data constructor `Roll :: f (Mu f) -> Mu f` and the destructor function `unRoll :: Mu f -> f (Mu f)` correspond to `roll` and `unroll` appearing in the typing rules

iso-roll and iso-unroll in §2.1. We can understand plain version of `Nat` in §3.1.3 as equi-recursive types, and the two level type version using `Mu` as iso-recursive types needing explicit use of `Roll` and `unRoll`. The data constructors for `Nat` should now be encoded using `Roll` and the data constructors of the base datatype. When we need to destruct (i.e., case match over) the values of `Nat`, we should use the deconstructor function `unRoll`. For instance, the case match in the plain version of `Nat`:

```
case n      of { Zero -> e0; Succ n -> e1 }
```

should be written as follows in the two level version:

```
case (unRoll n) of { Z      -> e0; S n      -> e1 }
```

In the two level type version, pattern matching over `Mu` (i.e., unrolling) is considered special since it is the only place where we tie the recursive knot. All the other pattern matches are over the non-recursive base datatypes.

We will shortly see that, this change of representation to iso-recursive types helps us understand the essence of iteration (in contrast to recursion) more clearly and intuitively in regards to programming language constructs. Before we continue the discussion on iteration, or catamorphisms, let us see couple more examples of the two level type definitions for other inductive datatypes.

We can define two level definition of the `List` datatype, using the same datatype fixpoint `Mu`, as follows:

```
data L p r = N | C p r -- base datatype for List

type List p = Mu (L p) -- List defined in terms of Mu and L

nil      = Roll N      -- data constructor corresponding to Nil
cons x xs = Roll (C x xs) -- data constructor corresponding to Cons
```

We can define two level definition of the `Tree` datatype similarly:

```
data T p r = L p | C p r -- base datatype for List

type Tree p = Mu (T p)

leaf x      = Roll (L x)      -- data constructor corresponding to Leaf
node t1 t2 = Roll (N t1 t2) -- data constructor corresponding to Node
```

More complex inductive datatypes can be defined in two level in a similar fashion.

3.2.2 Conventional (or, Squiggol style) catamorphism

To generalize from folds to conventional catamorphism, we factor out a mapping function (`fmap`), which handles recursive calls to recursive subcomponents, and a combining function (`phi :: f a -> a`), which combines the non-recursive components with the answers of processing the recursive subcomponents. The definition of the conventional catamorphism `cata` is as follows:

```
cata :: Functor f => (f a->a) -> Mu f -> a
cata phi (Roll x) = phi (fmap (cata phi) x)
```

Alternatively, the same definition in point-free style is:

```
cata :: Functor f => (f a->a) -> Mu f -> a
cata phi = phi . fmap (cata phi) . unRoll
```

The definition of `cata` captures the essence of structural recursion on inductive datatypes. Each time `cata` deepens ~~is~~ its recursive call one `Roll` is discharged by pattern matching (or, `unRoll`). Since the values of inductive datatypes consists of finite number of data constructors, each of which is encoded by one `Roll` and a data constructor of the base, `cata` is guaranteed to normalize, provided that `fmap` and `phi` function is non-recursive and does add more `Roll` constructor in its result.

The overloaded function `fmap :: Functor f => (a -> b) -> f a -> f b` should naturally define where to apply the recursive call in the base datatype. We should think that the definition for the `fmap` function is naturally derived from the datatype definition, rather than a user defined function. That is, `fmap` describes an inherent property of the inductive datatype. For instance, the definition of `fmap` for base `N` for natural numbers is defined as follows:

```
instance Functor N where
  fmap h Z      = Z
  fmap h (S x) = S (h x)
```

Note, we do nothing for recursive case `Z`, and the function `h` is applied to the predecessor position `x` for the recursive case `(S x)`.

Similarly, the `fmap` for bases `L` and `T` for lists and trees is defined as follows:

```
instance Functor (L p) where
  fmap h N      = N
  fmap h (C x xs) = C x (h xs)

instance Functor (T p) where
  fmap h (L x)      = L x
  fmap h (N t1 t2) = N (h t1) (h t2)
```

Note, the function `h` is applied to the recursive subcomponents, that is, to the tail position for lists and to the children position for trees.

Once we have seen how `fmap` is defined for each datatype, we can have better understanding of `cata`. Let us focus our attention back to the definition of `cata`.

```
cata :: Functor f => (f a->a) -> Mu f -> a
cata phi (Roll x) = phi (fmap (cata phi) x)
```

In the definition of `cata`, the first argument to `fmap` is `(cata phi)`, which is passed into `h` in the body of `fmap`. Thus, `fmap (cata phi) :: f (Mu f) -> f a` is a function that maps recursive subcomponents in `a` into answers of applying the catamorphism to those recursive subcomponents. After processing the subcomponents into answers, the combining function `phi :: f a -> a` combines the base structure containing the answers of processing the recursive subcomponents into the final answer.

We can define specific functions by supplying the user defined combining function into `phi`. For example, we can define the function `sumList`, which sums up all the elements in an integer list, as follows:

```
sumList = cata phi
  where
    phi N          = 0
    phi (C x ans) = x + ans
```

The following illustrates some of the reduction steps calculating the sum of all the elements in an integer list using `sumList`:

```
sumList (Roll(C 1 (Roll(C 2 (Roll(C 3 (Roll N)))))))
~>* cata phi (Roll(C 1 (Roll(C 2 (Roll(C 3 (Roll N)))))))
~>* phi (fmap (cata phi) (C 1 (Roll(C 2 (Roll(C 3 (Roll N)))))))
~>* phi (C 1 5)
~>* 1+5
~>* 6
```

The limitation of conventional (or, Squiggol style) catamorphsim is that it only works for regular inductive datatypes. That is, the limitation of `cata` comes in two dimensions: The definition of `cata` generalize neither for non-regular datatypes (including nested datatypes) nor for non-inductive datatypes (including negative datatypes).

Firstly, the conventional catamorphism does not generalize well to non-regular datatypes. The datatype we have seen so far, while introducing folds and `cata` are all regular datatypes. There exist many examples of non-regular datatypes in functional programming including nested datatypes and GADTs. One example of a nested datatype is the powerlist datatype defined as follows:

```
data Powl a = Nil | Cons a (Powl (a,a))
```

Note, the recursion is non-regular in the sense that the recursion (`Powl (a,a)`) is different from (`Powl a`). The definition of `cata` won't generalize to nested datatypes in a trivial way. There has been several approaches [12, 34, 38] to extend folds or conventional catamorphisms for nested datatypes.

Secondly, the conventional catamorphism does not generalize well to non-inductive datatypes, especially for negative datatypes.

3.2.3 Mendler style catamorphism

The functional programming community has traditionally focused on families of combinators that work well in Hindley-Milner languages, characterized by folds, or more generally (conventional) catamorphism, which we have been discussed so far. On the other hand, the Mendler style combinators were originally developed in the context of the Nuprl [19] type system. Nuprl made extensive use of g polymorphism and dependent types. General type checking in Nuprl was done by interactive theorem proving – not by type inference. The conventional catamorphism is widely known, especially on the list type (e.g., `foldr` in Haskell standard library). The conventional catamorphism has been more often used in functional programming than the Mendler style catamorphism, but it does not generalize easily to non-regular datatypes such as GADTs, or nested datatypes. The Mendler style catamorphism, being free from the two limitations of the conventional style combinators, is considerably more expressive than the conventional (or, Squiggol school [13] style) catamorphism.

Here, we briefly introduce Mendler style catamorphism and focus on its characteristics on non-inductive datatypes, in particular, negative datatypes. For further details, including its characteristics on non-regular inductive datatypes, you may refer to our conference paper [9] on Mendler style iteration and recursion combinators.

3.2.4 Definition of a Mendler style catamorphism

The definition of a Mendler style catamorphism is the following:

```
mcata :: (forall r . (r -> a) -> (f r -> a)) -> Mu f -> a
mcata phi (Roll x) = phi (mcata phi) x
```

Although we defined `Mu` as a newtype and `mcata` as a function in Haskell, you should consider them as an information hiding abstraction. The rules of the game are that you are only allowed to construct values using the `Roll` constructor (as in `zero`, `succ`, `nil` and `cons`), but you are not allowed to deconstruct those values by pattern matching against `Roll` (or, by using the selector function `unRoll`).

Note, `mcata` does not require `Functor` class in its type signature. The Mendler catamorphism `mcata` lifts the restriction that the base type be a functor, but still maintains the strict termination behavior of `cata`. This restriction is lifted by using two devices.

- The combining function `phi` becomes a function of 2 arguments rather than 1. The first argument is a function that represents an abstract recursive caller, the second

the conventional base structure that must be combined into an answer. The abstract recursive caller allows the programmer to direct where recursive calls must be made. The `Functor` class requirement is lifted, because no call to `fmap` is required in the definition of `mcata`:

```
mcata phi (Roll x) = phi (mcata phi) x
```

- The second device uses higher-rank polymorphism to insist that the abstract caller, with type $(r \rightarrow a)$, and the base structure, with type $(f\ r)$, work over an abstract type, denoted by (r) .

```
mcata :: (forall r. (r -> a) -> (f r -> a)) -> Mu f -> a
```

3.2.5 Mendler style catamorphism over inductive datatypes

The intuitive reasoning behind the termination property of `mcata` for all inductive datatypes is that (1) `mcata` strips off one `Roll` constructor each time it is called, and (2) `mcata` only recurses on the direct subcomponents (e.g., tail of a list) of its argument (because the type of the abstract recursive caller won't allow it to be applied to anything else). Once we observe these two properties, it is obvious that `mcata` always terminates since those properties imply that every recursive call to `mcata` decreases the number of `Roll` constructors in its argument.²

Writing the list length example in Mendler style will give clearer intuition explained above. The following is a side-by-side definition of the list length function in general recursion style (left) and in Mendler style (right).

<pre>data List p = N C p (List p) len :: List p -> Int len N = 0 len (C x xs) = 1 + len xs</pre>	<pre>data L p r = N C p r type List p = Mu (L p) nil = Roll N cons x xs = Roll (C x xs) lenm :: List p -> Int lenm = mcata phi where phi len N = 0 phi len (C x xs) = 1 + len xs</pre>
---	---

In the definition of `lenm`, we name the first argument of `phi`, the abstract recursive caller, to be `len`. We use this `len` exactly where we would recursively call the recursive function in the general recursion style (`len` on the left).

However, unlike the general recursion style, it is not possible to call `len :: r -> Int` on anything other than the tail `xs :: r`. Using general recursion, we could easily err (by mistake or by design) causing length to diverge, if we wrote its second equation as follows: `len (C x xs) = 1 + len (C x xs)`.

²We assume that the values of inductive types are always finite. We can construct infinite values (or, co-inductive values) in Haskell exploiting laziness, but we exclude such infinite values from our discussion in this work.

We cannot encode such diverging recursion in Mendler style because `len :: r -> Int` requires its argument to have the parametric type `r`, while `(C x xs) :: L p r` has more specific type than `r`.

3.2.6 Mendler style catamorphism over negative datatypes

Let us revisit the negative inductive datatype `T` (from §3.1.1) from which we constructed a diverging computation. We can define a two level version of `T`, let us name it `T_m`, as follows:

```
data TBase r = C_m (r -> ())
type T_m = Mu TBase
```

If we can write two functions `p_m :: T_m -> (T_m -> ())`, and `w_m :: T_m -> ()`, corresponding to `p` and `w` from §3.1.1, we would be able to reconstruct the same diverging computation. The function

```
w_m x = (p_m x) x
```

is easy since it is just a non-recursive function. The function `p_m` is problematic. By the rules of the game, we cannot pattern match on `Roll` (or use `unRoll`) so we must resort to using one of the combinators, such as `mcata`. However, it is not possible to write `p_m` in terms of `mcata`. Here is an attempt (seemingly the only one possible) that fails:

```
p_m :: T_m -> (T_m -> ())
p_m = mcata phi
  where
    phi :: (r -> (T_m -> ())) -> TBase r -> (T_m -> ())
    phi _ (C_m f) = f
```

We write the explicit type signature for the combining function `phi` (even though the type can be inferred from the type of `mcata`), to make it clear why this attempt fails to type check. The combining function `phi` take two arguments. The recursive caller (for which we have used the pattern `_`, since we don't intend to call it) and the base structure `(C_m f)`, from which we can extract the function `f :: r -> ()`. Note that `r` is an abstract (universally quantified) type, and the result type of `phi` requires `f :: T_m -> ()`. The types `t` and `T_m` can never match, if `r` is to remain abstract. Thus, `p_m` fails to type check.

There is a function, with the right type, that you can define:

```
pconst :: T_m -> (T_m -> ())
pconst = mcata phi
  where
    phi g (C f) = const ()
```

Not surprisingly, given the abstract pieces composed of the recursive caller `g :: r -> ()`, the base structure `(C f) :: TBase r`, and the function we can extract from the base structure `f :: r -> ()`, the only function that returns a unit value (modulo extensional equivalence) is, in fact, the constant function returning the unit value.


```

data FooF r = Noo | Coo (r -> r) r
type Foo = Mu FooF
noo        = Roll Noo
coo f xs   = Roll (Coo f xs)

lenFoo :: Foo -> Int
lenFoo = mcata phi where
  phi len Noo          = 0
  phi len (Coo f xs)   = 1 + len (f xs)

```

Figure 5: An example of a total function `lenFoo` using `mcata` over a negative datatype `Foo`

```

type Mu f = forall a . (forall r . (r -> a) -> f r -> a) -> a

mcata :: (forall r . (r -> a) -> f r -> a) -> Mu f -> a
mcata phi r = r phi

roll :: f (Mu f) -> Mu f
roll r phi = phi (mcata phi) r

```

Figure 6: F_ω encoding of `Mu` and `mcata` in Haskell

This illustrates the essence of how Mendler catamorphism guarantees normalization even in the presence of negative occurrences in the inductive type definition. By quantifying over the recursive type parameter of the base datatype (e.g. `r` in `TBase r`), it prevents an embedded function with a negative occurrence from flowing into any outside terms (especially terms embedding that function).

Given these restrictions, the astute reader may ask, are types with embedded function with negative occurrences good for anything at all? Can we ever call such functions? A simple example which uses an embedded function inside a negative inductive datatype is illustrated in Figure 5. The datatype `Foo` (defined as a fixpoint of `FooF`) is a list-like data structure with two data constructors `Noo` and `Coo`. The data constructor `Noo` is like the `nil` and `Coo` is like the `cons`. Interestingly, the element (with type `Foo->Foo`) contained `Coo` is a function that transforms a `Foo` value into another `Foo` value. The function `lenFoo`, defined with `mcata`, is a length like function, but it recurses on the transformed tail, `(f xs)`, instead of the original tail `xs`. The intuition behind the termination of `mcata` for this negative datatype `Foo` is similar to the intuition for positive datatypes. The embedded function `f :: r -> r` can only apply to the direct subcomponent of its parent, or to its sibling, `xs` and its transformed values (e.g. `f xs`, `f (f xs)`, ...), but no larger values that contains `f` itself. We illustrate a general proof on termination property of `mcata` in Figure 6.

3.3 Iteration over negative datatypes using syntactic inverses

Although we can define some simple functions such as `pconst` and `lenFoo` with `mcata`, the functions we can define with `mcata` are rather limited. In the functional programming community, variations of catamorphism over datatypes with embedded functions, including negative datatypes, has been studied to write more useful total functions. Our contribution is that we have shown it is also possible to formulate such a variation of catamorphism in Mendler style as well, and proved its termination property. We will introduce our development of `msfcata`, namely the Mendler style Sheard-Fegaras catamorphism, using a case study on formatting Higher-Order Abstract Syntax (HOAS).

3.3.1 Formatting HOAS

To lead up to the Mendler style solution to formatting HOAS, we first review some earlier work on turning expressions, expressed in Higher-Order Abstract Syntax (HOAS) [17, 51], into strings. The most simple HOAS datatype definition (of the untyped lambda calculus) in Haskell is: `data Exp = Lam (Exp -> Exp) | App Exp Exp`. We want to format a term of `Exp` into a string. For example, `Lam(\x->(Lam(\y->App x y)))` can be formatted into `(\x->(\y->(x y)))`. A solution to this problem was suggested by Fegaras and Sheard [28]. They were studying yet another abstract recursion scheme described by Paterson [50] and Meijer and Hutton [44] that could only be used if the combining function `phi` had a true inverse. This seemed a bit limiting, so Fegaras and Sheard introduced the idea of a syntactic inverse (or, a placeholder). The syntactic inverse was realized by augmenting the `Mu` type with a second constructor, which we call here `Rec`. This augmented datatype fixpoint `Rec` has the same structure as `Mu`, but with an additional data constructor as follows:

```
data Rec f a = Roll' (f (Rec f)) | Inverse a
unRoll' (Roll' e) = e
```

The algorithm worked, but, the augmentation introduces junk (i.e., the values constructed by `Inverse` is only an intermediate placeholder to calculate the answer later on, but can never be a valid input value). Washburn and Weirich[61] eliminated the junk by exploiting parametricity. It is a coincidence that Mendler style iteration/recursion schemes also use the same technique, parametricity, for a different purpose, to guarantee termination. Fortunately, these two approaches work together without getting in each other's way.

3.3.2 A general recursive implementation for open HOAS

The recursive datatype `Exp_g` in Figure 7 is an open HOAS. By *open*, we express that `Exp_g` has a data constructor `Var_g`, which enables us to introduce free variables. The constructor `Lam_g` holds an embedded function of type `(Exp_g -> Exp_g)`. This is called a shallow embedding, since we use functions in the host language, Haskell, to represent lambda abstractions in the object language `Exp_g`. For example, using the Haskell lambda expressions, we can construct some `Exp_g` representing lambda expressions as follows:

```

data Exp_g = Lam_g (Exp_g -> Exp_g) | App_g Exp_g Exp_g | Var_g String

showExp_g :: Exp_g -> String
showExp_g e = show' e vars where
  show' (App_g x y) = \vs      -> "(" ++ show' x vs ++ " "
                                ++ show' y vs ++ ")"
  show' (Lam_g z)   = \(v:vs) -> "(" ++ v ++ "->"
                                ++ show' (z (Var_g v)) vs ++ ")"
  show' (Var_g v)   = \vs      -> v

data ExpF r = Lam (r -> r) | App r r
type Exp' a = Rec ExpF a
type Exp = forall a . Exp' a
lam e      = Roll' (Lam e)
app f e    = Roll' (App f e)

showExp :: Exp -> String
showExp e = msfcata phi e vars where
  phi :: ([String] -> String) -> r -> (r -> ([String] -> String))
        -> ExpF r -> ([String] -> String)
  phi inv show' (App x y) = \vs      -> "(" ++ show' x vs ++ " "
                                ++ show' y vs ++ ")"
  phi inv show' (Lam z)   = \(v:vs) -> "(" ++ v ++ "->"
                                ++ show' (z (inv (const v))) vs ++ ")"

vars :: [String]
vars = [ [i] | i <- ['a'..'z'] ] ++ [ i:show j | j<-[1..], i<-['a'..'z'] ]

```

Figure 7: msfcata example: String formatting function for Higher-Order Abstract Syntax (HOAS)

```

k_g    = Lam_g (\x -> Lam_g (\y -> x))
s_g    = Lam_g (\x -> Lam_g (\y -> Lam_g (\z -> App_g x z 'App_g' App_g y z)))
skk_g  = App_g s_g k_g 'App_g' k_g
w_g    = Lam_g (\x -> x 'App_g' x)

```

Since we can build any untyped lambda expressions with `Exp_g`, even the problematic self application expression `w_g`, it is not possible to write a terminating evaluation function for `Exp_g`. However, there are many functions that recurse over the structure of `Exp_g`, and when they terminate produce something useful. One of them is the string formatting function `showExp_g` defined in Figure 7.

Given an expression (`Exp_g`) and a list of fresh variable names (`[String]`), the function `show'` (defined in the `where` clause of `showExp_g`) returns a string (`String`) that represents the given expression. To format an application expression (`App_g x y`), we simply recurse over each of the subexpressions `x` and `y`. To format a lambda expression, we take a fresh name `v` to represent the binder and we recurse over `(z (Var_g v))`, which is the application of the embedded function (`z :: Exp_g -> Exp_g`) to a variable expression (`Var_g v :: Exp_g`) constructed from the fresh name. Note, we had to create a new variable expression to format the function body since we cannot look inside the function values of Haskell. To format a variable expression (`Var_g v`), we only need to return its name `v`. The local function is `show'` (and hence also `showExp_g`), is total as long as the function values embedded in the `Lam_g` constructors are total.

With `showExp_g` we can format and print out the terms `k_g`, `s_g`, `skk_g` and `w_g` as follows:

```

> putStrLn (showExp_g k_g)
(\a->(\b->a))

> putStrLn (showExp_g s_g)
(\a->(\b->(\c->((a c) (b c)))))

> putStrLn (showExp_g skk_g)
(((\a->(\b->(\c->((a c) (b c))))) (\a->(\b->a)))
(\a->(\b->a)))

> putStrLn (showExp_g w_g)
(\a->(a a))

```

Note that `show'` is not structurally inductive in the `Lam_g` case. The recursive argument `(z (Var_g v))`, in particular `Var_g v`, is not a subexpression of `(Lam_g z)`. Thus the recursive call to `show'` may not terminate. This function terminated only because the embedded function `z` was well behaved, and the argument we passed to `z`, `(Var_g v)`, was well behaved. If we had applied `z` to the expression `(Lam_g (\x->x))` in place of `Var_g v`, or `z` itself had been divergent, the recursive call would have diverged. If `z` is divergent, then obviously `show' (z x)` diverges for all `x`. More interestingly, suppose `z` is not divergent (perhaps something as simple as the identity function) and `show'` was written to recurse on `(Lam_g (\x->x))`, then what happens?

```
show' (Lam_g z) (v: vs) = "(\\\"++v++\"->\"
                        ++ show' (z (Lam_g (\\x->x)) vs ++)"
```

The function is no longer total. To format `(z (Lam_g (\\x->x)))` in the recursive call, it loops back to the `Lam_g` case again, unless `z` is a function that ignores its argument. This will form an infinite recursion, since this altered `show'` forms yet another new `Lam_g (\\x->x)` expression and keeps on recursing.

3.3.3 A Mendler style solution for closed HOAS

Our exploration of the code in Figure 7 illustrates three potential problems with the general recursive approach.

- The embedded functions may not terminate.
- In a recursive call, the arguments to an embedded function may introduce a constructor with another embedded function, leading to a non terminating cycle.
- We got lucky, in that the answer we required was a `String`, and we happened to have a constructor `Var_g :: String -> Exp_g`. In general we may not be so lucky.

In Figure 7, we defined `Exp_g` in anticipation of our need to write a function `showExp_g :: Exp_g -> String`, by including a constructor `Var_g :: String -> Exp_g`. Had we anticipated another function `f :: Exp_g -> Int` we would have needed another constructor `C :: Int -> Exp_g`. Clearly we need a better solution. The solution is to generalize the kind of the datatype from `Exp_g :: *` to `Exp :: * -> *`, and add a universal inverse.

```
data Exp a    = App (Exp a) (Exp a)
              | Lam (Exp a -> Exp a)
              | Inv a

countLam :: Exp Int -> Int
countLam (Inv n) = n
countLam (App x y) = countLam x + countLam y
countLam (Lam f) = countLam(f (Inv 1))
```

Generalizing from `countLam` we can define a function from `Exp` to any type. How do we lift this kind of solution to the Mendler style? Fegaras and Sheard[28] proposed moving the general inverse from the base type to the datatype fixpoint. Later this approach was refined by Washburn and Weirich[61] to remove the junk introduced by that augmentation (i.e. things like `App (Inv 1) (Inv 1)`).

We use the same inverse augmented datatype fixpoint appearing in Washburn and Weirich[61]. Here, we call it `Rec`. The inverse augmented datatype fixpoint `Rec` is similar to the standard datatype fixpoint `Mu`. The difference is that `Rec` has an additional type index `a` and an additional data constructor `Inverse :: a -> Rec a i`, corresponding to the universal

inverse. The data constructor `Roll'` and the projection function `unRoll'` correspond to `Roll` and `unRoll` of the normal fixpoint `Mu`. As usual we restrict the use of `unRoll'`, or pattern matching against `Roll'`.

We illustrate this in the second part of Figure 7. As usual, we define `Exp' a` as a fixpoint of the base datatype `ExpF` and define shorthand constructors `lam` and `app`. Using the shorthand constructor functions, we can define some lambda expressions:

```
k    = lam (\x -> lam (\y -> x))
s    = lam (\x -> lam (\y -> lam (\z -> app x z 'app' app y z)))
skk  = app s k 'app' k
w    = lam (\x -> x 'app' x)
```

However, there is another way to construct `Exp'` values that is problematic. Using the constructor `Inverse`, we can turn values of arbitrary type `t` into values of `Exp' t`. For example, `Inverse True :: Exp' Bool`. This value is junk, since it does not correspond to any lambda term. By design, we wish to hide `Inverse` behind an abstraction boundary. We should never allow the user to construct expressions such as `Inverse True`, except for using them as placeholders for intermediate results during computation.

We can distinguish pure expressions that are inverse-free from expressions that contain inverse values by exploiting parametricity. The expressions that do not contain inverses have a fully polymorphic type. For instance, `k`, `s`, `skk` and `w` are of type `(Exp' a)`. The expressions that contain `Inverse` have more specific type (e.g., `(Inverse True) :: (Exp' Bool)`). Therefore, we define the type of `Exp` to be `forall a . Exp' a`. Then, expressions of type `Exp` are guaranteed to be inverse-free. Using parametricity to sort out junk introduced by the inverse is the key idea of Washburn and Weirich[61], and the inverse augmented fixpoint `Rec` is the key idea of Fegaras and Sheard[28]. The contribution we make in this work is putting together these ideas in Mender-style setting. By doing so, we are able define recursion combinators over types with negative occurrences, which have well understood termination properties enforced by parametricity.

- The combining function `phi` becomes a function of 3 arguments. An abstract inverse, an abstract recursive caller, and a base structure.

```
msfcata phi (Roll' x)    = phi Inverse (msfcata phi) x
msfcata phi (Inverse z) = z
```


- For inverse values, return the value inside `Inverse` as it is.
- We use higher-rank polymorphism to insist that the abstract inverse function, with type `(a -> r a)`, the abstract recursive caller function, with type `(r a -> a)`, and the base structure, with type `(f (r a))`, only work over an abstract type constructor, denoted by `(r)`.

```
msfcata :: (forall r. (a -> r a) ->
                    (r a -> a) ->
                    f (r a) -> a) -> (forall a. Rec f a) -> a
```

- Note, the abstract recursive type `r` is parameterized by the answer type `a` because the augmented datatype fixpoint `Rec` is parameterized by the answer type `a`.

Also, note, the second argument of `msfcata`, the object being operated on, has the higher-rank polymorphic type `(forall a . Rec f a)`, insisting the input value to be inverse-free by enforcing `a` to be abstract.

In Figure 7, using `msfcata`, it is easy to define `showExp`, the string formatting function for `Exp`, as in Figure 7. The `App` case is similar to the general recursive implementation. The body of `phi` is almost textually identical to the body of `show'` in the general recursive solution, except we use the inverse expression `inv (const v)` to create an abstract `r` value to pass to the embedded function `z`. Note, `const v` plays exactly the same roll as `(Var_g v)` in `show'`.

Does `msfcata` really guarantee termination? To prove this we need to address the first two of the three potential problems described at the beginning of §3.3.3. The first problem (embedded functions may be partial) is addressed using  equality analysis. The second problem (cyclic use of constructors as arguments to embedded functions) is addressed by the same argument we used in §3.2.6. The abstract type of the inverse doesn't allow it to be applied to constructors, they're not abstract enough. Just as we couldn't define `p_m` (in §3.2.6) we can't apply `z` to things like `(Lam(\x->x))`.

We provide an embedding of `msfcata` into the strongly normalizing language F_ω . This constitutes a proof that `msfcata` terminates for all inductive datatypes, even those with negative occurrences.

Figure 8 is the F_ω encoding of the inverse augmented datatype `Rec` and its catamorphism `msfcata`. We use the sum type to encode `Rec` since it consists of two constructors, one for the inverse and the other for the recursion. The newtype `Id` wraps answer values inside the inverse. The catamorphism combinator `msfcata` unwraps the result (`unId`) when `x` is an inverse. Otherwise, `msfcata` runs the combining function `phi` over the recursive structure `(\f->f(phi Id))`. The utility function `lift` abstracts a common pattern, useful when we define the shorthand constructors (`lam` and `app`).

Figure 9 is the F_ω encoding of the sum type `(:+:)` and its constructors (or injection functions) `inL` and `inR`. The case expression `caseSum` for the sum type is just binary function application. In the F_ω encoding, they could be omitted (i.e., `caseSum x f g` simplifies to `x f g`). But, we choose to write in terms of `caseSum` to make the definitions easier to read.

In Figure 10, we define both an inductive datatype for HOAS (`Exp`), and the string formatting function (`showExp`), with these F_ω encodings We can define simple expressions using the shorthand constructors and print out those expressions using `showExp`. For example,

```
> putStrLn (showExp (lam(\x->lam(\y->x)))
(\a->(\b->a)))
```

31-1

```

type Rec f r a = (r a) :+: (((r a -> a) -> f (r a) -> a) -> a)

newtype Id x = Id { unId :: x }

msfcata :: (forall r . (a -> r a) -> (r a -> a) -> f (r a) -> a)
        -> (forall a . Rec f Id a) -> a
msfcata phi x = caseSum x unId (\ f -> f (phi Id))

lift :: ((Id a -> a) -> f (Id a) -> a) -> Rec f Id a -> Id a
lift h x = caseSum x id (\ x -> Id(x h))

```

Figure 8: F_ω encoding of Rec and msfcata

```

type a :+: b = forall c . (a -> c) -> (b -> c) -> c
inL :: a -> (a :+: b)
inL a = \ f g -> f a
inR :: b -> (a :+: b)
inR b = \ f g -> g b
caseSum :: (a :+: b) -> (a -> c) -> (b -> c) -> c
caseSum x f g = x f g

```

Figure 9: F_ω encoding of the sum type

```

data ExpF x = App x x | Lam (x -> x)
type Exp' a = Rec ExpF Id a
type Exp = forall a . Exp' a
app :: Exp' a -> Exp' a -> Exp' a
app x y = inR (\h -> h unId (App (lift h x) (lift h y)))
lam :: (Exp' a -> Exp' a) -> Exp' a
lam f = inR (\h -> h unId (Lam (\x -> lift h(f(inL x)))) )

showExp :: Exp -> String
showExp e = msfcata phi e vars where
  phi inv show' (App x y) = \vs ->
    ("++ show' x vs ++" "++ show' y vs ++")
  phi inv show' (Lam z) = \ (v:vs) ->
    "(\\\"++v++\"->\"++ show'(z (inv (const v))) vs ++)"

```

Figure 10: HOAS string formatting example in F_ω .

3.4 Related Work

Since formal proof systems based on inductive type paradigm such as Coq do not support negative datatypes, the application of HOAS in such systems (e.g. [16]) are often based on Weak HOAS [25, 35] to avoid the use of negative datatypes. However there have been some work on iteration, recursion, and induction over HOAS.

Our preliminary work is about iteration over HOAS, and more generally over negative datatypes. Despeyroux et al. [26] introduced a primitive recursion on HOAS in a modal λ -calculus. Since their work can analyze inside functions (or, inside the λ -binder), their system can express total functions that analyze the subcomponents of the application (e.g., parallel reduction). Such functions are not expressible in our preliminary work [9] and in the work by Fegaras and Sheard [28] and Meijer and Hutton [44], which our preliminary work is based on. However, their study of primitive recursion over HOAS is in the context of simple types, but not parameterized datatypes nor indexed datatypes. The Mendler style catamorphism work well with parameterized and indexed datatypes. Later, Despeyroux and Leleu [24] tried to extended primitive recursion over HOAS in the presence of dependent types. Despeyroux and Leleu [24] were able to prove type safety of their system, but have not proved normalization yet.

Induction principles over HOAS seem to been studied in various contexts, but I need to further background literature search to list and categorize the related work on induction over HOAS.

3.5 Future Work

I am thinking of two follow-up work on our preliminary work. First is extending the Mendler style catamorphism to dependent types (§3.5.1). Second is searching for a Mendler style recursion combinator that guarantee termination for negative datatypes (§3.5.2).

3.5.1 Extending the Mendler style catamorphism to dependent types

Consider the following dependently typed program which shows that every natural number is either even or odd:

```
data Nat where                -- inductive definition of natrual numbers
  Zero : Nat
  Succ : Nat -> Nat

data Either (a:Type) (b:Type) where -- the sum type
  Left  : a -> Either a b
  Right : b -> Either a b

data Even (n:Nat) where      -- inductive definition of
  Even0 : Even Zero          -- the evenness property,
  EvenS : Odd n -> Even (Succ n) -- mutually recursive with Odd
```

```

data Odd (n:Nat) where                -- inductive definition of
  OddS : Even n -> Odd (Succ n)      -- the oddness property

evenOrOdd : (n:Nat) -> Either (Even n) (Odd n)
evenOrOdd Zero      = Left CZero
evenOrOdd (Succ n) = case evenOrOdd n of
  Left p  -> Right (OddS p)
  Right p -> Left (EvenS p)

```

The function `evenOrOdd` takes a natural number `n` and returns either a proof that `n` is even or a proof that `n` is odd. Except for the dependency using the value `n` in the return type, the recursion pattern has the form of a catamorphism. It is an open question whether the Mendler style catamorphism naturally extends to dependent types as it does to indexed types. Assuming that we were able write a dependent version of the Mendler style catamorphism, say `mcataD`, then we would be able to write `evenOrOdd` in terms of `mcataD` as follows:

```

data N r = Z | S r
type Nat = Mu N

evenOrOdd = mcataD phi where
  phi eoo Z      = Left CZero
  phi eoo (S n) = case eoo n of
    Left p  -> Right (OddS p)
    Right p -> Left (EvenS p)

```

Recall that, in Mendler style, we encode a datatype (e.g., `Nat`) as a fixpoint (e.g., `Mu`) of base functor (e.g., `N`).

The main problem here is that the Mendler style recursion combinators use parametricity to abstract the type of the argument value, but the return type of the function depends on the argument value. In the second equation of the `phi` function above, `(S n) :: N r`, and therefore `n :: r`, where `r` is abstract. But, what should be the type of the abstract recursive caller `eoo`? It would look something like `eoo :: (n:r) -> Either (Even n) (Odd n)`. We can already see that this does not type check since `Even :: Nat -> Type` and `Odd :: Nat -> Type` but `n :: r`. Recall, we cannot unify `r` with any specific type. Thus `(Even n)` and `(Odd n)` are ill-typed (or ill-kinded). We see that it is hard to use a value without unveiling the details of its type. This problem is analogous to the problem of using abstract types for parameterized modules. We want to encode the type of modules to be abstract types, but we also want to know certain instances of the parameterized modules have share same parameter since we want a reasonable separate compilation scheme. Translucent sums [36] were suggested to solve this problem when type checking modules. Here, we also need a translucency in the sense that we want the type of the argument to be abstract when implementing the runtime behavior in the function definition to limit the dangerous

34-1

recursion, but we want to know the type of the argument in the type signature of the function due to the true value dependency on the argument.

My proposed attempt here tries to implement translucency using the features of the Trellys language. We do not know yet whether this is possible, or we need to add new feature to support translucency. My preliminary thoughts on the type signature for the dependent Mendler style catamorphism is the following:

```
mcataD : (forall (r:Type) . [tr: r->Mu f] -> [tfr: f r->Mu f]
        -> [pr: tr = id ] -> [pfr: tfr = Roll ]
        -> ((z:r) -> a (tr z)) -> (y:f r) -> a (tfr y))
        -> (x:Mu f) -> a x
```

This dependent version of the Mendler style catamorphism combinator has four additional arguments for the `phi` function (`tr`, `tfr`, `pr`, and `pfr`), when compared to `mcata`. Note, `pr` has an equality proof involving `tr` and `pfr` has an equality proof involving `tfr`.

Here, we use an interesting feature found in the Trellys language: we require those four arguments to be *erasable arguments* by annotating them with square brackets. Erasable arguments can only be used for type checking purposes, but have no effect on the runtime behavior of the function. The first two arguments, `tr` and `tfr`, are type casting functions that turn the abstract types `r` and `f r` into a concrete inductive datatype `Mu f`. Since these type casting functions (`tr` and `tfr`) and their equality property proofs (`pr` and `pfr`) break parametricity of `r`, we should limit their use in the runtime definition of the `phi` function, but only allow their use in the type signatures and type casting purposes.

Another interesting feature of Trellys we require in this proposed approach is the *heterogeneous equality* in the equality types of `pr` and `pfr`. Note, the left- and right-hand sides of `tr = id` and `tfr = Roll` have different types (e.g., `tfr:f r->Mu f` and `Roll:f (Mu f)->Mu f`). These heterogeneous equality makes it possible to type check the `evenOrOdd` example. Consider the case branch `Left p -> Right (OddS p)`. Since `ooo n : a (tr n)`, `Left p : a (tr n)` and `Right (OddS p) : a (Roll (S (tr n)))`. Since the final return type of `phi` must be `a (trf (S n))`, we should show that `a (Roll (S (tr n))) = a (trf (S n))`. Since `tr = id`, the left-hand side is equivalent to `a (Roll (S n))`. Since `trf = Roll`, the right-hand side is also equivalent to `a (Roll (S n))`.

Note, the tentative approach discussed above is only a preliminary thought (which may or may not work) and we might end up with a better approach. I started with this tentative approach to understand the problem better, but not expecting this approach leads to complete success. The advantage of using existing language features, in contrast to inventing new features or language constructs, is that we do not need to worry about breaking the soundness and consistency of the type system, provided that the properties of the language features we rely on are well-studied.

3.5.2 Mendler style recursion combinator for negative datatypes

In §3.4, I mentioned related work on primitive recursion for HOAS using modal types by Despeyroux and Leleu [24], Despeyroux et al. [26]. Their work was not in Mendler style. I

suspect that it may be possible to refine the Mendler style histomorphism, which is a Mendler style recursion combinator capable of encoding course of values, to guarantee termination for negative datatypes. To discover such a recursion combinator, I will try to encode the ideas of Despeyroux et al. [26] in a Mendler style setting.

The Mendler style histomorphism is proven to guarantee termination for regular inductive datatypes [60]. For non-regular inductive datatypes (e.g., `Pow1` in §3.2.2), the termination property of the Mendler style histomorphism is left out as a conjecture strongly believed to be true [43].³ For negative datatypes, I recently found a counterexample that the Mendler style histomorphism is not normalizing for negative datatypes. You look up the counterexample in our conference paper [9].

I have omitted the discussion on histomorphism in this document in order to avoid too much digression into details on course of values recursion combinators, and, instead, focus on iteration on negative datatypes such as HOAS. So, I am closing the discussion of future work on this subject. You can find further details on the Mendler style histomorphism in our conference paper [9] and Vene’s thesis [60].

4 Plans for dissertation work

My dissertation will contain four parts. The first two parts will review and summarize the literature organized from the viewpoint of my thesis, and the later two parts will contain mostly original contributions.

Part I is the introduction and background. This part will consist of chapters extending §1 (introduction) and §2 (background) of this document. That is, first, further details on what is already in this document, second, additional background material from an extended literature search, and third, introduction to the issues to be discussed in later parts.

Part II discusses current ideas from the literature on normalization. Strong normalization is an important component of logical consistency, so understanding normalization is a necessary component of the thesis. The current literature deals mostly with positive datatypes, thus this part will consist of chapters discussing normalization of both strictly positive datatypes and non-strictly positive datatypes.

Part III is on normalization of negative datatypes. This part will consist of chapters extending §3 of this document, where we discussed our preliminary results using Mendler style iteration. Further details will be added. In addition, this part will have chapters addressing some of the open questions and issues related to normalization of negative datatypes (e.g., interaction with dependent types, course of values recursion).

Part IV explores the design of languages that can shift gears between different fragments of the problem spaces (`IND`, `IND⊥`, `REC`, `REC⊥`). I will extend lambda calculi with recursive types, and then extend them with facilities that track which fragment the terms belong to, and what conditions can make the terms shiftable from one fragment to another.

³Matthes [43] left the proof on course of values recursion as an open question in his work on monotonicity. It may have already been proven, but I haven’t yet encountered one yet.

4.1 Outline and Timeline

Part 1 (introduction and background): This part will be an extended literature search, adding to the material from the introduction (§1) and the background (§2) sections of this document. A tentative list of additional subjects I am planning to survey include:

- Examples of formal reasoning system designs that relate IND and IND_\perp (e.g., the bar type [20] in Nuprl).
- More generally, how *extensional type theory* (in contrast to *intensional type theory*), can be more flexible when tracking non-terminating computations using the type system. Extensional type theory does not distinguish computational equality from propositional equality. Type checking become undecidable when non-terminating computation is allowed while type checking.
- *Observational type theory*, which claims to take advantage of the benefits from both intensional and extensional type theories.
- Libraries that track partiality as effects or monads in formal reasoning systems (e.g., Agda, Epigram), which lie in IND
- Various termination analysis methods, and whether those methods are known to reduce to more primitive ways of ensuring termination (e.g., primitive recursion or structural recursion)
- More details and examples on the use of monotonicity, especially for non-strictly positive datatypes (e.g., Matthes' thesis mentions that there exists monotone but not positive datatypes due to Ulrich Berger).

I plan to finish the writing of Part I of the thesis in November 2011, this fall.

Part II (positive datatypes) Although this part is also a background literature search, I decided to separate this material from Part I and add more detail on positive datatypes for several reasons.

Firstly, strong normalization is necessary for logical consistency. So understanding current approaches is necessary for part IV.

Secondly, a detailed understanding of how normalization is ensured for positive datatypes, supports comparing and contrasting with how normalization is ensured for negative datatypes later in Part III.

Thirdly, it is one of my theses that normalization and inductiveness (or, logical interpretation of types) are indeed separate concerns of logical consistency. By reviewing the current literature on positive datatypes (where the two ideas are conflated), we can begin to unravel the knot that currently binds them together.

Computationally, all positive datatypes behave the same regardless of whether they are strictly positive or not. Primitive recursion supports normalization of both strictly positive datatypes and non-strictly positive datatypes (more generally, monotone datatypes).

However, logically, not all positive datatypes can be accepted as propositions, depending on what is considered an acceptable logic. In the inductive paradigm, where types must have set theoretic interpretations, strictly positive datatypes are valid types, but not all positive datatypes are considered to be valid types in general. For instance, when we interpret types as sets and \rightarrow as function space over sets, the positive, but not strictly positive, datatype $\mu\alpha.(\alpha \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Bool}$ asserts an isomorphism between the powerset of powerset of α and α itself, which is a set theoretic nonsense.

Although monotonicity is a more general principle than the syntactic condition of positivity, positivity is nice approximation since monotonicity witness for positive datatypes can be simply derived automatically. For non-positive monotone datatypes, I have not seen work on automatic derivation of their monotonicity witnesses. So, I think a practical system would use the syntactic conditions like positivity by default, and to be more flexible, it can have a mechanism to check and accept monotonicity witness manually provided by the user.

I plan to finish the first pass of writing Part II in December 2011, this winter.

Part III (negative datatypes) This part will report the preliminary results I have already finished. Additional writing will be necessary.

I will add more detailed descriptions to the material reported in this document (§3) on Mendler style iteration for negative datatypes, and a more detailed summary of the literature on primitive recursion and induction principles proposed for HOAS. I plan to write these chapters in January 2012. Considerable text in this area exists from published papers, which needs to be recast in the style of a thesis.

As my research progresses, I will address some of the open questions and issues related to normalization of negative datatypes (e.g., interaction with dependent types, course of values recursion). I will write additional chapters on those results. Since this is speculative work to be done, I cannot predict a confident timeline for this, but my initial plan is to finish writing this in March 2012.

Part IV (language design) This part is on the high level design of logical languages (or, formal reasoning systems) that can shift gears between the four fragments of \mathbf{IND} , \mathbf{IND}_\perp , \mathbf{REC} , and \mathbf{REC}_\perp . Although the ideal design would cover all of the four fragments, I will only focus on four pairs of fragments to avoid accidental complexity. That is, I will illustrate four typed lambda calculi that cover \mathbf{IND} - \mathbf{IND}_\perp , \mathbf{IND} - \mathbf{REC} , \mathbf{REC} - \mathbf{REC}_\perp , and \mathbf{IND}_\perp - \mathbf{REC}_\perp .

The highlight of the design of the type system for these 4 different calculi is going to be how to represent and track the side conditions that determines when a term in a larger fragment can be seen as a term in a smaller fragment (i.e., a term in \mathbf{IND}_\perp as a term in \mathbf{IND} , a term in \mathbf{REC} as a term in \mathbf{IND} , a term in \mathbf{REC}_\perp as a term in \mathbf{REC} , a term in \mathbf{REC}_\perp as a term in \mathbf{IND}_\perp).

In addition to developing these calculi, there will be a chapter of case studies that demonstrates the usefulness of the developed calculi. In the case study chapter, I will formulate the examples that work over more than one fragment (e.g., Normalization by Evaluation) inside one of these calculi.

Since this part is again speculative work to be done, I cannot make a confident timeline for this, but my initial plan is to complete the writing for Part III in May 2012.

4.2 Publication goals and methods for research

Since Part I and II are survey on existing work by literature search, results for publication will come out from working on Part III and IV.

I am planning to work on a journal version of our preliminary work. I have two improvements to make in mind from our conference version. First, I will have better context discussions and clear use of vocabularies in the journal version. I was less clear on distinguishing iteration from recursion at the point when we were writing the conference version. Secondly, I will discuss further on related work. Due to the space limitations, the conference version lacks the detailed discussion of the related work in comparison to our results.

As mentioned in §3.5, I am searching for more general and more expressive Mendler style iteration and recursion for negative datatypes. The result this work will be another material for publication. In particular, there are two generalization in consideration.

First is to generalize Mendler style iteration to dependently typed setting. Mendler style iteration rely on parametricity, but dependent types makes it hard to rely on parametricity. The method I am currently trying is to use erasable arguments and heterogeneous equality, in order to make the dependent type indices be observable only at the type level, but remain abstract at the value level (see §3.5). I will try applying this method on examples other than even/odd datatype to see if this method makes sense for other examples too. After trying out several examples and become more confident that this may be a general enough method, I will formally describe a calculi with dependent types, erasable argument, and heterogeneous equality, and start proving the termination behavior of the Mendler style iteration in that calculus,

Second is to formulate Mendler style recursion, rather than just iteration, which guarantee totality. As we have seen in §3.4, primitive recursion for HOAS has been studied using modal types. However, those work are for simple types and not in Mendler style. In our preliminary work, we have been successful in formulating Fegaras-Shared catamorphism in Mendler style, which has only been studied in conventional style previously. Similarly, I will try to formulate primitive recursion for HOAS in Mendler style.

Lastly, the calculi to be developed in Part IV is yet another topic for publication. I will start from four calculi that is known to be type safe in each of the four fragment (IND , IND_\perp , REC , REC_\perp), where the two calculi for IND and REC should be normalizing calculi. Among the four calculi, the calculus for REC_\perp is the most flexible calculi allowing all the recursive types, which will be the basis for the other three calculi. The other three calculi would have additional restrictions that restrict certain formation and use of recursive types and recursive control structures. Then, I will try to design a type system that bridge between the four pairs

we focus on, which preserves the original property of the individual calculi and yet possible to shift gears between the two calculi of the different fragments. I would also need to show that the logic described by the calculus for **IND** or any pair involving **IND** is consistent. To prove the consistency of our developed calculus, I will study the literature on proving logical consistency and try to apply the proof strategies to the calculi to be developed.

Acknowledgements

Bob Harper's and Hugo Herbelin's OPLSS 2011 lectures⁴ and Alexandre Miquel's Types summer school 2005 lecture notes⁵ has been a great guide in writing background section.

References

- [1] Martín Abadi, Luca Cardelli, and Gordon D. Plotkin. Types for the scott numerals. Obtained from authors, February 1993.
- [2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In Josep Daz, Juhani Karhumki, Arto Lepist, and Donald Sannella, editors, *Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 87–95. Springer Berlin / Heidelberg, 2004.
- [3] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3 – 27, 2005. ISSN 0304-3975. doi: DOI:10.1016/j.tcs.2005.06.002. URL <http://www.sciencedirect.com/science/article/pii/S0304397505003373>. Applied Semantics: Selected Topics.
- [4] Andreas Abel. Normalization for the simply-typed lambda-calculus in twelf. *Electr. Notes Theor. Comput. Sci.*, 199:3–16, 2008. URL <http://dx.doi.org/10.1016/j.entcs.2007.11.009>.
- [5] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *LNCS*, pages 54–69. Springer, 2003.
- [6] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1-2):3 – 66, 2005. ISSN 0304-3975.
- [7] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for martin-Löf type theory with one universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, 2007. URL <http://dx.doi.org/10.1016/j.entcs.2007.02.025>.

⁴<http://www.cs.uoregon.edu/Activities/summerschool/summer11/curriculum.html>

⁵<http://www.cse.chalmers.se/research/group/logic/Types/tutorials.html>

- [8] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for martin-lof type theory with typed equality judgements. In *LICS*, pages 3–12. IEEE Computer Society, 2007. URL <http://doi.ieeecomputersociety.org/10.1109/LICS.2007.33>.
- [9] Ki Yung Ahn and Tim Sheard. A hierarchy of Mendler style recursion combinators: Taming datatypes with negative occurrences. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '11, 2011.
- [10] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system t revisited. *Theor. Comput. Sci.*, 411:1484–1500, March 2010. ISSN 0304-3975. doi: <http://dx.doi.org/10.1016/j.tcs.2009.11.014>. URL <http://dx.doi.org/10.1016/j.tcs.2009.11.014>.
- [11] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In Gilles Kahn, editor, *IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
- [12] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:11–2, 1999.
- [13] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.
- [14] Frédéric Blanqui. Inductive types in the Calculus of Algebraic Constructions. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003*, volume 2701, Valencia, Espagne, 2003. URL <http://hal.inria.fr/inria-00105617/en/>.
- [15] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19: 509–543, September 2009. ISSN 0956-7968. doi: 10.1017/S0956796809007205. URL <http://portal.acm.org/citation.cfm?id=1630623.1630626>.
- [16] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.
- [17] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [18] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. Technical Report TR 85-659, Dept Computer Science, Cornell University, January 1985. URL <http://hdl.handle.net/1813/6499>. reflexive recursive types as Scott domains and functions over domains as partial functions.

- [19] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.
- [20] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theor. Comput. Sci.*, 121:89–112, December 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90085-8. URL <http://dl.acm.org/citation.cfm?id=170005.170020>.
- [21] T. Coquand and C. Paulin. Inductively defined types. In *Proceedings of the international conference on Computer logic*, pages 50–66, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-52335-9. URL <http://portal.acm.org/citation.cfm?id=88278.88291>.
- [22] Thierry Coquand and Gérard Huet. The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France, May 1986.
- [23] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2001. ISBN 3-540-41739-7. URL http://dx.doi.org/10.1007/3-540-44716-4_22.
- [24] Joëlle Despeyroux and Pierre Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *Informal proceedings of the FLoC’99 IMLA*, June 1999.
- [25] Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, London, UK, 1995. Springer-Verlag.
- [26] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote, editor, *TLCA*, volume 1210 of *LNCS*, pages 147–163. Springer, 1997. ISBN 3-540-62688-3.
- [27] Peter Dybjer. Representing inductively defined sets by wellorderings in martin-lf’s type theory. *Theoretical Computer Science*, 176(1-2):329 – 335, 1997. ISSN 0304-3975. doi: DOI:10.1016/S0304-3975(96)00145-4. URL <http://www.sciencedirect.com/science/article/pii/S0304397596001454>.
- [28] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’96, pages 284–294. ACM, 1996.
- [29] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*,

- volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003. ISBN 3-540-22164-6. URL http://dx.doi.org/10.1007/978-3-540-24849-1_14.
- [30] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 193–217. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992. URL http://www.cs.kun.nl/~herman/BRABasInf_RecTyp.ps.gz.
 - [31] Jean-Yves Girard. Une extension de l’interprétation de Gödel à l’analyse, et son application à l’élimination des coupures dans l’analyse et la théorie des types. In *Proceedings 2nd Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971. North-Holland.
 - [32] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(3–4):280–287, 1958.
 - [33] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1979. The second author is listed on the cover as Arthur J. Milner, which is clearly a mistake.
 - [34] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proc. of 2nd Workshop on Generic Programming*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ. July 2000.
 - [35] Honsell, Miculan, and Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2001.
 - [36] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997.
 - [37] Andres Löb, Conor McBride, and Wouter Swierstra. Simply easy! an implementation of a dependently typed lambda calculus, 2007.
 - [38] Clare E. Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.
 - [39] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
 - [40] Matthes. Non-strictly positive fixed points for classical natural deduction. *ANALSPAL: Annals of Pure and Applied Logic*, 133, 2005.
 - [41] Ralph Matthes. Monotone fixed-point types and strong normalization. In *In Proceedings of CSL 1998, Lecture Notes in Computer Science. Submitted*, pages 1076–1993. IEEE Press, 1998.

- [42] Ralph Matthes. Monotone (co)inductive types and positive fixed-point types. *Information Theories and Applications*, 33(4–5):309–328, 1999. URL <ftp://ftp.tcs.informatik.uni-muenchen.de/pub/matthes/publ/fics98.ps.gz>.
- [43] Ralph Matthes. Monotone inductive and coinductive constructors of rank 2. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 600–614, London, UK, 2001. Springer-Verlag. ISBN 3-540-42554-3. URL <http://portal.acm.org/citation.cfm?id=647851.737260>.
- [44] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 324–333. ACM, 1995.
- [45] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.
- [46] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.
- [47] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *SLP*, pages 379–388, 1987.
- [48] Robin Milner. Implementation and applications of scott’s logic for computable functions. In *Proceedings of ACM conference on Proving assertions about programs*, pages 1–6, New York, NY, USA, 1972. ACM. doi: <http://doi.acm.org/10.1145/800235.807067>. URL <http://doi.acm.org/10.1145/800235.807067>.
- [49] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
- [50] Ross Paterson. Control structures from types. Unpublished draft, 1993.
- [51] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208. ACM, 1988.
- [52] Frank Pfenning and Christine Paulin-mohring. Inductively defined types in the calculus of constructions. pages 209–228. Springer-Verlag, 1990.
- [53] John C. Reynolds. Towards a theory of type structure. In *Paris colloquium on programming*, volume 19 of *LNCS*. Springer-Verlag, 1974.
- [54] John C. Reynolds and Gordon Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105(1):1–29, 1993. proof that set theoretic model for System F cannot exist. non-existence.

- [55] Dana S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Manuscript, 1969.
- [56] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, September 1976.
- [57] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1&2):411–440, 1993.
- [58] Eijiro Sumii and Naoki Kobayashi. Online type-directed partial evaluation for dynamically-typed languages. *Computer Software*, 17:38–62, 1999.
- [59] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, editors. *Report on the Algorithmic Language Algol 68*. Springer, Berlin, 1969.
- [60] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis (Diss. Math. Univ. Tartuensis 23), Dept. of Computer Science, Univ. of Tartu, August 2000.
- [61] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 249–262. ACM, 2003.

1-1

Sep 20, 2011 8:38 AM, James Hook

Are these the only two important concepts? Simplify. Loose the parentheticals.

1-2

Sep 20, 2011 8:38 AM, James Hook

Abrupt start. Unmotivated.

1-3

Sep 22, 2011 9:12 AM, James Hook

Starts abruptly and without motivation.

1-4

Sep 22, 2011 9:12 AM, James Hook

What is this? Are there examples?

1-5

Sep 22, 2011 9:12 AM, James Hook

Justify such a bold claim. Why shouldn't an ideal system dodge all these issues by avoiding them? Model checking? Acl2? Maude? I happen to agree with you, but this is a point you must support.

3-1

Sep 22, 2011 9:12 AM, James Hook

Why should I care about this example? Can you find a well motivated example instead? Is there something I can't do in coq or Isabelle that I will be able to do in this system?

3-2

Sep 22, 2011 9:12 AM, James Hook

It is good to plow new ground, but you still need to motivate it. Is the reason the other two quadrants have been studied? What have we missed?

4-1

Sep 22, 2011 9:12 AM, James Hook

That. You have a lot of which/that issues. I only marked this one.

4-2

Sep 22, 2011 9:12 AM, James Hook

Why normal? Are all proofs cut free? Gentzen's main theorem establishes that proofs in the systems he studied are equal to cut free proofs, not that all proofs are cut free. This statement seems very strange.

4-3

Sep 22, 2011 9:12 AM, James Hook

This is your most informative Thais-like statement.

10-1

Sep 22, 2011 9:12 AM, James Hook

Examples would be very helpful in this section.

10-2

Sep 22, 2011 9:12 AM, James Hook

Do you define this?

13-1

Sep 22, 2011 9:12 AM, James Hook

Are

13-2

Sep 22, 2011 9:12 AM, James Hook

I really like section 2. The writing and scholarship in this section is much better than your earlier attempts. Section 1 needs to be brought up to this level.

14-1

Sep 22, 2011 9:12 AM, James Hook

Of the

14-2

Sep 22, 2011 9:12 AM, James Hook

Founded

14-3

Sep 22, 2011 9:12 AM, James Hook

Number agreement. Perhaps principles? Or a well-founded

15-1

Sep 22, 2011 9:12 AM, James Hook

Given the umlaut (two dots) the e is not correct. Oe is an encoding of o umlaut.

15-2

Sep 22, 2011 9:12 AM, James Hook

Has

19-1

Sep 22, 2011 9:12 AM, James Hook

A

20-1

Sep 22, 2011 9:12 AM, James Hook

The

31-1

Sep 22, 2011 9:12 AM, James Hook

Undefined term

33-1

Sep 22, 2011 9:12 AM, James Hook

What are inside functions?

33-2

Sep 22, 2011 9:12 AM, James Hook

S

34-1

Sep 22, 2011 9:12 AM, James Hook

Compiler folks call control join points phi functions. I would avoid this term in this context. Elsewhere you called these combining functions.

38-1

Sep 22, 2011 9:12 AM, James Hook

This sounds very ambitious. I want you to reflect on these issues, but not to black hole.

This draft shows more focused thought than your earlier work.

39-1

Sep 22, 2011 9:12 AM, James Hook

S

39-2

Sep 22, 2011 9:12 AM, James Hook

The

39-3

Sep 22, 2011 9:12 AM, James Hook

Relies