Fall 12-16-2014

The Nax Language: Unifying Functional Programming and Logical Reasoning in a Language based on Mendler-style Recursion Schemes and Term-indexed Types

Ki Yung Ahn
Portland State University

10.15760/etd.2086

The Nax Language :

Unifying Functional Programming and Logical Reasoning

in a Language based on Mendler-style Recursion Schemes

and Term-indexed Types

by

Ki Yung Ahn

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Tim Sheard, Chair
James Hook
Mark P. Jones
Andrew Tolmach
Douglas V. Hall

Portland State University
2014

ABSTRACT

Two major applications of lambda calculi in computer science are functional programming languages and mechanized reasoning systems (or, proof assistants). According to the Curry–Howard correspondence, it is possible, in principle, to design a unified language based on a typed lambda calculus for both logical reasoning and programming. However, the different requirements of programming languages and reasoning systems make it difficult to design such a unified language useful for both purposes. Programming languages usually extend lambda calculi with programming-friendly features (e.g., recursive datatypes, general recursion) for supporting the flexibility to model various computations, while sacrificing logical consistency. Logical reasoning systems usually extend lambda calculi with logic-friendly features (e.g., induction principles, dependent types) for paradox-free inference over fine-grained properties, while being more restrictive in modeling computations.

In this dissertation, we design and implement a language called Nax that balances between the benefits of both. Nax accepts all recursive datatypes, thus, allowing the same flexibility of defining recursive datatypes as functional languages. Nax supports a number of Mendler-style recursion schemes that can express various kinds of recursive computations and also grantee termination. Nax supports term-indexed types to support specifications of fine-grained properties. In addition, Nax supports a conservative extension of Hindley–Milner type inference.

The theoretical contributions of this dissertation include theories for Mendler-style recursion schemes and term-indexed types, which we developed to establish strong normalization and logical consistency of Nax.

DEDICATION

To the Logos from the beginning of the universe, who was with the creator and the creator himself, the source of all beings, and the light of life shines in the darkness.

ACKNOWLEDGMENTS

I would like to thank my advisor Tim Sheard, who has always been patient and open to experimenting with new ideas, the members of the dissertation committee for their useful feedback, and special thanks to Andrew M. Pitts and Marcelo Fiore for the discussions greatetly improved my theiss research during the visit to Cambridge, UK, and also to Gabor Grief, who gave feedback on my thesis related research. Many thanks to my parents, Rev. Sam Whan Kim, and friends in Myungsung Presbyterian Church for their love and prayers from the other side of the Pacific; the young adults' group in Oregon Eden Presbyterian Church for being a warm and dynamic community; and, to coach Ruth Aquino and friends in the Lloyd Center Ice Rink, where I chilled out in the evenings.

Table of Contents

List of Tables

List of Figures

Part I

Prelude

Chapter 1

INTRODUCTION

In this dissertation, we contribute to answering the question: "how does one build a seamless system where programmers can both write (functional) programs and formally reason about those programs?" We set the scope of our research by clarifying what we mean by *programming* and *formal reasoning*, and how they are related (Section 1.1). We outline the motivations our research by describing the gap between typical design choices made in functional programming languages and formal reasoning systems (Section 1.2), and assert the thesis (Section 1.3). We introduce the preliminary concepts of Mendler-style recursion schemes and term-indexed types (Section 1.4), and highlight the contributions of this dissertation (Section 1.5). We close this chapter by providing an overview of the chapter organization (Section 1.6).

1.1 PROGRAMMING AND FORMAL REASONING

In this dissertation, *programming* refers to writing programs in modern functional programming languages (e.g., Haskell, ML) that support (recursive) datatypes, higher-order functions, (parametric) polymorphism, and type inference. When we refer to *functional programming languages*, *functional languages*, or *programming languages*, we mean such languages. We do not consider so-called dynamically typed functional languages (e.g., Lisp, Erlang) in this dissertation.

Formal reasoning or *logical reasoning* refers to the construction of proofs with

dependently-typed formal reasoning systems (e.g., Coq, Agda) that support (inductive) datatypes, higher-order functions, polymorphism (i.e., type arguments to type constructors), and dependent types (i.e., term arguments to type constructors). In referring to *formal reasoning systems*, *logical reasoning systems*, or, simply *reasoning systems*, we mean such systems.

Functional languages and reasoning systems are closely related. Proofs in reasoning systems are similar in structure to programs in functional languages. For example, the Haskell program *id*, which computes a value of type A when given a value of type A, and, the Agda proof *id*, which proves that proposition A is true when given a proof of A, are very similar:

Haskell	Agda
$id :: A \to A$	$id : A \to A$
$id = \lambda x \to x$	$id = \lambda x \to x$

Such similarities are not accidental but intended by the design of reasoning systems, based on the observation that proofs correspond to programs and propositions correspond to types. This observation is called the Curry–Howard correspondence [50]. In the following subsections, we explain a few preliminary concepts necessary for understanding the Curry–Howard correspondence

We assume that readers are familiar with programming in functional languages, basic concepts of lambda calculi (e.g., β-reduction, normal forms) and type theory (i.e., familiar with describing typing rules in the inference rule format). Moreover, we assume that the readers understand basic concepts of logic (e.g., axioms, inference, and proofs) but are not necessarily familiar with formal reasoning systems.

1.1.1 The Curry–Howard correspondence

In the late 1960s, Howard [50] observed that intuitionistic natural deduction, which is a proof system for a formal logic, and a typed lambda calculus, which is a

model of computation, are directly related. This relationship, known as the Curry–Howard correspondence, is established as follows:

- A proposition in natural deduction corresponds to a type in lambda calculus.
- A proof for a proposition corresponds to a term of that type.
- Simplification of proofs corresponds to computation, that is, simplification of terms.

Once these are established, we can formalize the logic within one unified language system by internally witnessing proofs as terms (or, programs) because we consider *"propositions as types"*.[1] In contrast, in a more traditional approach, where one represents *"propositions as terms"*,[2] one needs a meta-language other than the object language for logic, which is an extension of lambda calculus with logical constants and connectives, in order to construct proofs.

We explain this correspondence with a very simple version of intuitionistic natural deduction and a simply-typed lambda calculus (Figure 1.1).[3] There are many variations of formal logic based on natural deduction (and their corresponding typed lambda calculi) depending on the set of logical connectives, constants, and quantifiers they support. We show an example of intuitionistic natural deduction with implication ($\to$) and falsity ($\bot$), as well as its corresponding simply-typed

[1] Coq and Agda are based on the Curry–Howard correspondence.

[2] HOL family is based on this more traditional approach.

[3] Readers who are familiar with the literature on natural deduction might notice that the left column in Figure 1.1 is not in the same style as the natural deduction formalized by Gentzen [35, 36]. It has styles similar to both natural deduction and sequent calculus. The context Γ is part of the judgement syntax as in sequent calculus. Instead of the syntactic structural rules (such as weakening, contraction, permutation) of sequent calculus, we simply rely on $A^x \in \Gamma$ to use the hypothetical propositions from Γ. We choose to formalize natural deduction this way to emphasize the structural similarities to the typical formalization of typing rules of lambda calculi.

Traditionally, in *Hilbert-style deduction*, a logic is formalized by a minimal set of inference rules (e.g., modus ponens $\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$) and a set of axiom schemes (e.g., $A \to A$ and $A \to (B \to A)$ where the meta-variables A and B can be instantiated to arbitrary propositions). *Natural deduction*, in contrast, is another style of formalizing logic mostly by using a set of inference rules and a minimal (often empty) set of axiom schemes.

Intuitionistic natural deduction		Typing rules of STLC
$\dfrac{A^x \in \Gamma}{\Gamma \vdash A}$	(Ax)	$\dfrac{x : A \in \Gamma}{\Gamma \vdash A}$
$\dfrac{\Gamma, A^x \vdash B}{\Gamma \vdash A \to B}$	$(\to_{\mathrm{I}})$	$\dfrac{\Gamma, x : A \vdash B}{\Gamma \vdash \lambda x.t : A \to B}$
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$(\to_{\mathrm{E}})$	$\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B}$
$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}$	$(\bot_{\mathrm{E}})$	$\dfrac{\Gamma \vdash t : \bot}{\Gamma \vdash \mathtt{elim}_{\bot}\ t : A}$

Figure 1.1: Intuitionistic natural deduction with implication and falsity, and its corresponding simply-typed lambda calculus.

lambda calculus (STLC) in Figure 1.1. The implication and the falsity in natural deduction correspond to the function type ($\to$) and the void type ($\bot$) in STLC. In the inference rules (Ax) and ($\to_{\mathrm{I}}$), x on A^x is a meta-tag for distinguishing between multiple possible occurrences of A in Γ (e.g., $\Gamma = A^{x_1}, B^{x_2}, A^{x_3}$).

Note that each typing rule (in the right column) has exactly the same structure as its corresponding inference rule (in the left column), with the exception of the variables and terms appearing on the left-hand side of the colon (e.g., x in $x : A$ and t in $t : A \to B$). Therefore, a type-correct term, which is justified by the derivation following the typing rules of lambda calculus, captures the structure of its corresponding proof by natural deduction. For instance, $(\lambda x.x)$ is a term that captures the structure of a proof for $A \to A$. For this reason, such type-correct terms are called proof terms, proof objects, or simply proofs in reasoning systems.

We explained that proofs and propositions in natural deduction correspond to terms and types in lambda calculus. Lastly, we need to show the correspondence

$$\dfrac{\dfrac{\dfrac{}{A^{x_1} \in A^{x_1}, A^{x_2}}\,(\mathrm{Ax})^{x_1}}{A^{x_1}, A^{x_2} \vdash A}\,(\to_{\mathrm{I}})^{x_2}}{A^{x_1} \vdash A \to A}\,(\to_{\mathrm{I}})^{x_1}}{\vdash A \to A \to A} \qquad \dfrac{\dfrac{\dfrac{}{x_1 : A \in x_1 : A, x_2 : A}\,(\mathrm{Ax})}{x_1 : A, x_2 : A \vdash x_1 : A}\,(\to_{\mathrm{I}})}{x_1 : A \vdash \lambda x_2.x_1 : A \to A}\,(\to_{\mathrm{I}})}{\vdash \lambda x_1.\lambda x_2.x_1 : A \to A \to A}$$

$$\dfrac{\dfrac{\dfrac{}{A^{x_2} \in A^{x_1}, A^{x_2}}\,(\mathrm{Ax})^{x_2}}{A^{x_1}, A^{x_2} \vdash A}\,(\to_{\mathrm{I}})^{x_2}}{A^{x_1} \vdash A \to A}\,(\to_{\mathrm{I}})^{x_1}}{\vdash A \to A \to A} \qquad \dfrac{\dfrac{\dfrac{}{x_2 : A \in x_1 : A, x_2 : A}\,(\mathrm{Ax})}{x_1 : A, x_2 : A \vdash x_2 : A}\,(\to_{\mathrm{I}})}{x_1 : A \vdash \lambda x_2.x_2 : A \to A}\,(\to_{\mathrm{I}})}{\vdash \lambda x_1.\lambda x_2.x_2 : A \to A \to A}$$

Figure 1.2: Typing derivations (right) for terms of type $A \to A \to A$ in STLC and their corresponding proofs (left) in natural deduction.

between simplification of proofs and simplification of terms (i.e., computation) in order to establish the Curry–Howard correspondence between natural deduction and lambda calculus.

There can be multiple proofs for the same proposition, and, correspondingly, there can be more than one term of the same type. Some of these terms are closely related while others are rather independent. For example, $\lambda x_1.\lambda x_2.x_1$ and $\lambda x_1.\lambda x_2.x_2$ are rather independent terms that inhabit the same type $A \to A \to A$. The typing derivations for these terms (right column) and their corresponding proofs (left column) are illustrated in Figure 1.2.

On the other hand, there are closely related terms of the same type. For example, $(\lambda x_1.x_2)x_3$ reduces to x_2 by a β-reduction step (i.e., $(\lambda x_1.x_2)x_3 \longrightarrow_\beta x_2$). A term such as x_2, which cannot (β-)reduce any further, is called a (β-)*normal form* or a (β-)*normal term*. Prawitz [80] established a notion of reduction and normalization for natural deduction. One can reduce a proof when there are consecutive uses of introduction and elimination rules. An introduction rule introduces a certain form of proposition in the conclusion (below the horizontal bar), and, an elimination rule uses propositions of that form in the premises (above the horizontal bar). The rules ($\to_{\mathrm{I}}$) and ($\to_{\mathrm{E}}$) are introduction and elimination

rules, respectively, for implication ($\rightarrow$). So, we need to show the correspondence between reduction of proofs over implication in the natural deduction and reduction over type-correct terms in the lambda calculus, in order to establish the Curry–Howard correspondence.

Figure 1.3 illustrates that the reduction of a proof involving consecutive uses of ($\rightarrow_{\mathrm{I}}$) and ($\rightarrow_{\mathrm{E}}$) corresponds to the β-reduction of well-typed terms.[4] To reduce the proof, we replace the uses of A^x in the premise of $(\rightarrow_{\mathrm{I}})^x$ with derivation D', which deduces A from the original context Γ, so that we can remove A^x from the left-hand sides of the turnstile ($\vdash$) throughout the proof. The change of subscripts from $\mathcal{D}_{[\Gamma, A^x]}$ to $\mathcal{D}_{[\Gamma]}$ before and after the reduction denotes that we consistently remove A^x from the context (i.e., left-hand sides of $\vdash$). We leave it as an exercise for the readers to construct a corresponding proof for the reduction $(\lambda x_1.x_2)x_3 \longrightarrow_\beta x_2$ from the previous paragraph (*hint*: Start with $\Gamma = x_2 : A, x_3 : B$).

We illustrated the Curry–Howard correspondence between a minimal intuitionistic logic and its corresponding lambda calculus in Figure 1.1. That is, a proposition, its proof, and simplification of proofs (or proof reduction) correspond to a term, its type, and simplification of terms (or, computation), respectively. The first and second pieces of the correspondence — a proposition and its proof corresponding to a term and its type — are quite evident from the structural similarity between inference rules and typing rules. The last piece of the correspondence between proof reduction and computation needs further analysis of the introduction and elimination rules. We illustrated the correspondence between proof reduction over implication and β-reduction, which describes computation over function applications, in Figure 1.3. The correspondence between proof reduction over falsity and computation over the void type hold vacuously. There is no proof reduction

[4] Introduction and elimination rules for ($\rightarrow$) used consecutively in the opposite order corresponds to η-reduction (i.e., $(\lambda x.t\ x) \longrightarrow_\eta t$ where x does not appear free in t). In this dissertation, we only consider β-reduction.

Reducing a proof involving consecutive uses of $(\to_{\mathrm{I}})$ and $(\to_{\mathrm{E}})$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A^x, \Gamma' \vdash A}\,(\mathrm{Ax})^x}{\vdots}\;\mathcal{D}_{[\Gamma, A^x]}}{\dfrac{\Gamma, A^x \vdash B}{\Gamma \vdash A \to B}\,(\to_{\mathrm{I}})^x} \qquad \dfrac{\mathcal{D}'_{[\Gamma]}}{\Gamma \vdash A} \qquad (\to_{\mathrm{E}})}{\Gamma \vdash B} \quad \longrightarrow \quad \dfrac{\dfrac{\mathcal{D}'_{[\Gamma, \Gamma']}}{\Gamma, \Gamma' \vdash A}}{\vdots}\;\mathcal{D}_{[\Gamma]}}{\Gamma \vdash B}$$

Reduction of a well-typed term along with its typing derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : A, \Gamma' \vdash x : A}\,(\mathrm{Ax})}{\vdots}\;\mathcal{D}_{[\Gamma, x:A]}}{\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\to_{\mathrm{I}})} \qquad \dfrac{\mathcal{D}'_{[\Gamma]}}{\Gamma \vdash s : A} \qquad (\to_{\mathrm{E}})}{\Gamma \vdash (\lambda x.t)\, s : B} \quad \longrightarrow_{\beta} \quad \dfrac{\dfrac{\mathcal{D}'_{[\Gamma, \Gamma']}}{\Gamma, \Gamma' \vdash s : A}}{\vdots}\;\mathcal{D}_{[\Gamma]}}{\Gamma \vdash t[s/x] : B}$$

Figure 1.3: Reduction of a proof involving introduction and elimination rules for implication, and its corresponding β-reduction of a well-typed term.

and computation over $\bot$ because it lacks the introduction rule — it only has the elimination rule ($\bot_{\mathrm{E}}$).

1.1.2 Logical consistency and strong normalization

A logic is *consistent* when not all propositions are provable. *Logical consistency*, is absolutely necessary for a logic to be meaningful, that is, to be able to justify true propositions and refute false propositions. A standard way of showing logical consistency is to find a sound model[5] for the logic, and show that there exists a proposition whose interpretation in the model is a falsity value.

For instance, to show that the logic described in Figure 1.1 is consistent, we would construct a sound model such that the meaning of $\bot$ is a falsity value. Using the Curry–Howard correspondence, we can construct a model for the logic using the syntactic structure of its corresponding lambda calculus. For instance, we can define interpretation $[\![A]\!]$ for proposition A as the set of terms of type A in lambda calculus. In a model following such interpretations, the empty set would be the falsity value. Then, we can show that there exist no closed terms in $[\![\bot]\!]$, which implies that $\vdash \bot$ cannot be proved. In Chapter 2, we construct models for several lambda calculi based on this idea of interpreting propositions (or, types) as sets of well-typed terms. Given that the scope of our research includes reasoning systems that are based on the Curry–Howard correspondence, most of our descriptions will be in terms of lambda calculi, without mentioning their corresponding natural deduction counterparts.

In the construction of such models of lambda calculus, we typically assume strong normalization.[6] In the models we construct in Chapter 2, the interpretation

[5] In this dissertation, we mean *logic* in a proof theoretic sense. A model is sound with respect to logic described by a proof system when any provable proposition in the logic is interpreted as a truth value in the model.

[6] Strong normalization is a property that all well-typed terms reduce to their normal form regardless of the reduction strategy (i.e., choice of which redex to reduce first).

$[\![A]\!]$ for type A is inductively defined as follows:

- Base case: normal forms of type A are in $[\![A]\!]$
- Inductive case: if $t' \in [\![A]\!]$ and $t \longrightarrow_\beta t'$ then $t \in [\![A]\!]$.

That is, interpretations of types are (β-)equivalence classes of their well-typed normal forms. We show that each of the calculi in Chapter 2 are logically consistent by demonstrating that there is no closed term in the interpretation of the void type.

When we admit diverging (or, non-terminating) terms in a lambda calculus, the definition of type interpretations becomes more complicated and we cannot establish the Curry–Howard correspondence, because those diverging terms cannot be considered as proofs in general. In fact, it is well-known that functional languages are logically inconsistent if we try to view diverging terms (i.e., non-terminating programs) in functional languages as proofs using a naive Curry–Howard correspondence. For example, the non-terminating program defined as $loop = loop$ in Haskell can inhabit arbitrary types (even $\forall a.a$, which is the polymorphic encoding of the void type). Intuitively, such non-terminating programs correspond to logical fallacies of circular reasoning (i.e., arguing something by assuming the same thing). Therefore, reasoning systems are designed to be strongly normalizing, unlike functional languages. Moreover, we limit the scope of our research to strongly normalizing languages because one of the design goals of our language system is achieving logical consistency under the Curry–Howard correspondence.

1.1.3 Datatypes and recursion schemes

Both functional languages and reasoning systems support other language constructs in addition to those supported by the lambda calculus. Among those constructs, datatypes and recursion are the most common and the most essential. Datatype definitions in both programming languages and reasoning systems have

Datatypes in functional languages	Datatypes in reasoning systems
Datatypes may involve diverging computations (e.g., functions defined by general recursion)	Curry–Howard correspondence must holds for all datatypes
Type constructors may have type arguments	Type constructors may have term arguments (or, term indices) as well as type arguments

Figure 1.4: Comparison of datatypes in functional languages and datatypes in reasoning systems.

the form of disjoint sums (over several data constructors) of products (of the argument types for each data constructor). For example, in Haskell, we can define a datatype (*Diagram*) that defines a diagram, which is either empty (*Empty*), a point (*Point*) defined by a single coordinate, a line segment (*LineSeg*) defined by two coordinates, or a triangle (*Triangle*) defined by three coordinates, as follows:

```
data Diagram = Empty
             | Point Coord
             | LineSeg Coord Coord
             | Triangle Coord Coord Coord
```

In type theory, we can understand the above datatype as a sum of products

$$\mathit{Diagram} \triangleq \mathit{Unit} + \mathit{Coord} + (\mathit{Coord} \times \mathit{Coord}) + (\mathit{Coord} \times \mathit{Coord} \times \mathit{Coord})$$

where $+$ and $\times$ are binary operators for sums and products and *Unit* is the identity for products. Non-recursive datatypes, excluding dependent types (i.e., types indexed by terms), in reasoning systems can be understood in the same manner. It is well-known that the Curry–Howard correspondence holds between type operators, sum ($+$), and product ($\times$), and, the logical connectives, disjunction ($\vee$), and conjunction ($\wedge$),

In Figure 1.4, we summarize the different characteristics between the datatypes in functional languages and the datatypes in reasoning systems. Our approach

on datatypes is to find a common middle ground balancing between the desired properties of functional languages (few restrictions on datatype definitions) and reasoning systems (logical consistency, term indices). When we exclude general recursion from functional languages and exclude dependent types from reasoning systems, non-recursive datatypes in functional languages and reasoning systems coincide. However, for datatypes that are defined recursively (or inductively), the situation is more subtle — they do not coincide even when we disregard general recursion and dependent types.

Datatypes defined in terms of themselves are called *recursive datatypes* in functional languages and *inductive datatypes* in reasoning systems. Recursive datatypes in functional languages have few restrictions. Any syntactically valid[7] datatype definition is admitted as a valid type. In contrast, inductive datatypes in reasoning systems have additional restrictions. Only those datatypes for which the Curry–Howard correspondence hold are admitted. Some recursive datatypes, admitted as valid in functional languages, are not admitted as valid inductive datatypes in reasoning systems. For example, the Haskell datatype T below would not be admitted in a reasoning system.

$$
\begin{array}{l}
\textbf{data}\ T\ a = C\ (T \to a) \quad \texttt{--}\ \text{datatype recursive on a contravariant position} \\
w\ ::\ T\ a \to a \qquad \texttt{--}\ \text{an encoding of the untyped } (\lambda x.x\ x) \text{ in a typed language} \\
w = \lambda x \to \textbf{case}\ x\ \textbf{of}\ C\ f \to f\ x \\
\textbf{fix}\ ::\ (a \to a) \to a \quad \texttt{--}\ \text{an encoding of } (\lambda f.(\lambda x.f(x\ x))\,(\lambda x.f(x\ x))) \\
\textbf{fix} = \lambda f \to (\lambda x \to f(w\ x))\,(C(\lambda x \to f(w\ x)))
\end{array}
$$

Surprisingly (if you hadn't known), we can encode the well known general recursive combinator **fix** (a.k.a. Y-combinator) using a datatype recursive on a contravariant position (i.e., left-hand side of $\to$), without using any recursion at the

[7] More accurately, what we really mean is *well-kinded* rather than syntactically valid. For those who are not familiar with the use of kinds in type systems, we discuss them in later chapters, such as in Section 2.3 where we describe System F_ω.

term-level. Datatypes that are recursive only over covariant[8] positions are called *positive datatypes*, and, datatypes that are recursive over one or more contravariant positions are called *negative datatypes*. Negative datatypes are not admitted in reasoning systems because they might cause non-termination, as illustrated above. Recall that the Curry–Howard correspondence is established only when proofs are normalizing.

A process for designing reasoning systems can be summarized as follows: start from a strongly normalizing and logically consistent calculus, add language extensions, and then theoretically justify that those extensions do not break normalization and consistency. Datatypes and their recursion schemes are one of the most common and significant extensions. Each recursive datatype admitted under the Curry–Howard correspondence comes with a recursion scheme in the calculus or an induction principle in the logic.[9] The reduction step for a recursion scheme in the calculus should follow from the Curry–Howard correspondence over the recursive types, just as β-reduction follows from the Curry–Howard correspondence over function types.

There are two approaches for ensuring normalization of datatypes and their recursion schemes. One is to restrict datatype definitions (i.e., formation rules) and the other is to restrict the use of datatypes (i.e., elimination rules).

The former approach, also known as the *conventional*[10] approach, is used in most reasoning systems (e.g., Coq, Agda) and studies on terminating recursion schemes in functional languages following the Squiggol school of constructive programming [12, 13, 43]. The conventional approach restricts the definition of

[8]Type arguments without $\rightarrow$ are by default in covariant positions. Right-hand sides of $\rightarrow$ are covariant and a contravariant position of a contravariant position (e.g., A in $(A \rightarrow B) \rightarrow B$) is covariant as well.

[9] Induction principles provided in reasoning systems are dependently-typed. Disregarding dependent types, these induction principles are computationally equivalent to recursion schemes such as primitive recursion, course-of-values recursion, or lexicographic recursion. Since we do not consider dependent types, we will only discuss those non-dependently-typed recursion schemes.

[10] We adopted the word "*conventional*" from the literature on Mendler style (e.g., [3]).

datatypes and the key mechanisms of checking termination of recursion schemes are based on size decreasing arguments in an untyped setting. We rely on the assumption that recursive values contained inside a data constructor are always smaller than the value which contains them. This assumption holds because of the positivity restriction on datatype definitions. However, measuring size by structural containment does not always hold for negative datatypes.

The latter non-conventional approach, known as the *Mendler-style* approach, puts no restrictions on datatype definitions, but instead, carefully restricts the use of data values. In particular, the decomposition (i.e., elimination, or, pattern matching) of recursive values is restricted. In this approach, theoretical development for termination of recursion schemes is type based. Instead of treating datatypes as primitive language constructs, datatypes and their recursion schemes are embedded into a typed lambda calculus, which is proven to be strongly normalizing and logically consistent. Then, there is no extra theoretical burden for establishing the Curry–Howard correspondence for datatypes because the datatypes are encoded using the basic primitives of the lambda calculus, for which we already know that the Curry–Howard correspondence hold.s In addition, there is no need for any extra mechanism, other than type checking, to ensure termination of the recursion schemes. We adopt this non-conventional approach for our language system design.

Further details on conventional approach and Mendler-style approach for terminating recursion schemes are discussed in Section 1.4 and Chapter 3. From now on, we develop our discussions from the perspective of recursive datatypes. That is, when we need to describe inductive datatypes, we consider them as recursive datatypes with additional restrictions on their formation.

1.2 MOTIVATION

Since the discovery of the Curry–Howard correspondence, logicians and programming language researchers have dreamed of building a system in which one can both write programs (i.e., model computation) and formally reason about (i.e., construct proofs of) the properties (i.e., types) of those programs.

However, building a practical system that unifies programming and formal reasoning based on the Curry–Howard correspondence remains an open research problem. The gap between the conflicting design goals of typed functional programming languages, such as ML and Haskell, and formal reasoning systems, such as Coq and Agda, is still wide. As discussed in the previous section, one of the difficulties is that datatypes admitted in functional languages and those admitted in reasoning systems do not coincide completely.

- Programming languages are designed to achieve computational expressiveness, for which they often sacrifice logical consistency. Programmers should be able to conveniently express all possible computations, regardless of whether those computations have a logical interpretation (by the Curry–Howard correspondence).
- Formal reasoning systems are designed to achieve logical consistency, for which they often sacrifice computational expressiveness. Users expect that it is only possible to prove true propositions but impossible to prove falsity. They are willing to live with the difficultly (or even inability) to express certain computations within the reasoning system for achieving logical consistency.

Consequently, the recursion schemes of programming languages and formal reasoning systems differ considerably. Programming languages provide unrestricted general recursion to conveniently express computations that may or may not terminate. Formal reasoning systems provide induction principles for sound reasoning,

or, from the computational viewpoint, principled recursion schemes that can only express terminating computations.

The two different design goals for programming languages and reasoning systems are reflected in the design of their type systems, especially regarding datatypes and recursion schemes. Programming languages place few restrictions on the definition of datatypes. Programmers can express computations over a wide variety of datatypes. In reasoning systems based on conventional approach, additional restrictions are enforced on datatype definition — only positive datatypes are accepted. In addition, most functional programing languages have a clearly distinguish between terms and types (i.e., terms do not appear in types). In reasoning systems, terms can appear in types for specifying fine-grained properties involving values at the term-level (e.g., size invariants of data structures).

This dissertation explores a sweet spot where one can benefit from the advantages of both programming languages and formal reasoning systems. That is, we design a unified language system called Nax that is logically consistent while being able to conveniently express many useful computations. We do this by placing few restrictions on datatype definitions, as is done in programming languages, but also provide a rich set of non-conventional recursion schemes that always terminate. These non-conventional recursion schemes are known as *Mendler-style recursion schemes*.[11] Another major design choice in Nax is supporting (non-dependent) *term indices* in types, a middle ground between polymorphic types and dependent types.

In the following section, we clarify what we mean by the sweet spot between programming languages and reasoning systems, and assert the thesis.

[11] We introduce the concepts of conventional and non-conventional recursion schemes in Section 1.4.

1.3 THESIS

We characterize the sweet spot of language design for unifying programming and reasoning by supporting the following four features:

(1) **A convenient programming** style supported by the major constructs of modern functional programming languages: parametric polymorphism, recursive datatypes, recursive functions, and type inference,

(2) **An expressive logic** that can specify fine-grained program properties using types, and terms that witness the proofs of these properties (Curry–Howard correspondence),

(3) **A small theory** based on a minimal foundational calculus that is expressive enough to support programming features, expressive enough to embed propositions and proofs about programs, and logically consistent to avoid paradoxical proofs in the logic, and

(4) **A simple implementation** that keeps the trusted base small.

Our thesis is that a language design based on *Mendler-style recursion schemes* and *term-indexed types* can lead to a system that supports these four features.

The following chapters support the thesis as follows: Mendler-style recursion schemes support (1) because they are based on parametric polymorphism and are well-defined over a wide range of datatypes. Term-indexed types support (2), because they can statically track program properties. For instance the size of data structures can be tracked by using a natural number term in their types. To support (3), we design several foundational calculi, each of which extends a well known polymorphic lambda calculus with term-indexed types. Moreover, Mendler-style recursion schemes support (4) because their termination is type-based — no need for an auxiliary termination checker.

1.4 MENDLER-STYLE RECURSION AND TERM-INDEXED TYPES

We summarize the preliminary concepts of Mendler-style recursion schemes (Section 1.4.2) and term-indexed types (Section 1.4.3). Further details and the historical background of each of these concepts appears in subsequent chapters (see Section 1.6 for an overview of chapter organization).

1.4.1 Restriction on recursive types for normalization

Logical reasoning systems establish the Curry–Howard correspondence assuming normalization. So, one challenge in the successful design of reasoning systems is how to restrict recursion so that all well-typed terms have normal forms. The two different design choices to this end are shown in Figure 1.5, in contrast to the unrestricted general recursion in functional languages. The conventional approach restricts the formation of recursive types (i.e., the restriction is in datatype definition), whereas the Mendler-style approach restricts the elimination of the values of recursive types (i.e., restrict pattern matching).

Recursive types in functional programming languages. Let us start with a review of the theory of recursive types used in functional programming languages.

Just as we can capture the essence of unrestricted general recursion at the term level by using a fixpoint operator (usually denoted by Y or **fix**), we can capture the essence of recursive types by the using a recursive type operator μ at the type-level. The rules for the formation (μ-form), introduction (μ-intro), and elimination (μ-elim) of the recursive type operator μ are described in Figure 1.5. We also need a reduction rule (unIn-In) that relates In, the data constructor for recursive types, and unIn, the destructor for recursive types, at the term-level.

The recursive type operator μ described in Figure 1.5, is already powerful enough to express non-terminating programs, even without introducing the general

Unrestricted general recursion in functional languages	kinding: $(\mu\text{-form})\dfrac{\Gamma \vdash F : * \to *}{\Gamma \vdash \mu F : *}$ typing: $(\mu\text{-intro})\dfrac{\Gamma \vdash t : F(\mu F)}{\Gamma \vdash \mathsf{In}\ t : \mu F}$ $\quad (\mu\text{-elim})\dfrac{\Gamma \vdash t : \mu F}{\Gamma \vdash \mathsf{unIn}\ t : F(\mu F)}$ reduction: $(\mathsf{unIn}\text{-}\mathsf{In})\dfrac{}{\mathsf{unIn}\ (\mathsf{In}\ t) \rightsquigarrow t}$
A conventional recursion scheme	kinding: $(\mu\text{-form}^+)\dfrac{\Gamma \vdash F : * \to * \quad \mathsf{positive}(F)}{\Gamma \vdash \mu F : *}$ typing: (μ-intro) and (μ-elim) same as functional language $(\mathbf{It})\dfrac{\Gamma \vdash t : \mu F \quad \Gamma \vdash \varphi : FA \to A}{\Gamma \vdash \mathbf{It}\ \varphi\ t : A}$ reduction: (unIn-In) same as functional language $(\mathbf{It}\text{-}\mathsf{In})\dfrac{}{\mathbf{It}\ \varphi\ (\mathsf{In}\ t) \rightsquigarrow \varphi\ (\mathsf{map}_F\ (\mathbf{It}\ \varphi)\ t)}$
A Mendler-style recursion scheme	kinding: (μ-form) same as functional language typing: (μ-intro) same as functional language $(\mathbf{mit})\dfrac{\Gamma \vdash t : \mu F \quad \Gamma \vdash \varphi : \forall X.(X \to A) \to FX \to A}{\Gamma \vdash \mathbf{mit}\ \varphi\ t : A}$ reduction: $(\mathbf{mit}\text{-}\mathsf{In})\dfrac{}{\mathbf{mit}\ \varphi\ (\mathsf{In}\ t) \rightsquigarrow \varphi\ (\mathbf{mit}\ \varphi)\ t}$

Figure 1.5: Two different approaches to terminating recursion schemes (in contrast to unrestricted general recursion in functional languages).

recursive *term* operator **fix** into the language. We illustrate this below.[12] First, here is a short reminder of how a fixpoint at the term-level works. The typing rule and the reduction rule for **fix** can be given as follows:

$$\text{typing: } \frac{\Gamma \vdash f : A \to A}{\mathbf{fix}\, f : A} \qquad \text{reduction : } \mathbf{fix}\, f \rightsquigarrow f(\mathbf{fix}\, f)$$

We can actually implement **fix** using μ as follows (using Haskell-like syntax):

$\mathbf{data}\ T\ a\ r = C\ (r \to a)$ -- a non-recursive datatype

$w\ :\ \mu(T\ a) \to a$ -- an encoding of the untyped $(\lambda x.x\ x)$ in a typed language

$w = \lambda x.\, \mathbf{case}\ \mathsf{unIn}\ x\ \mathbf{of}\ C\ f \to f\ x$

$\mathbf{fix}\ :\ (a \to a) \to a$ -- an encoding of $(\lambda f.(\lambda x.f(x\ x))\,(\lambda x.f(x\ x)))$

$\mathbf{fix} = \lambda f.(\lambda x.f(w\ x))\,(\mathsf{In}(C(\lambda x.f(w\ x))))$

Thus, we need to alter the rules for μ in someways to guarantee termination. One way is to restrict the rule μ-form and the other way is to restrict the rule μ-elim. The design of principled recursion combinators (e.g., **It** for the former and **mit** for the latter) follows from the choice of the rule to restrict..

Positive (recursive) datatypes and negative (recursive) datatypes. Positive datatypes are recursive on only covariant positions. For example, $\mu\, T_2$, where $\mathbf{data}\ T_2\ r\ =\ C_2\,(Bool\ \to\ r)$, is a positive datatype since the recursive argument r in the base structure T_2 appears only in the covariant position. Recursive datatypes that have no function arguments are by default positive datatypes. For instance, the natural number datatype $\mu\, N$, where $\mathbf{data}\, N\, r = S\, r \mid Z$, is a positive datatype.

Negative datatypes have recursion in contravariant positions. Note that $\mu(T\ a)$ in the example above is a negative datatype because the recursive argument r in

[12] This is essentially the same example we discussed in Section 1.1.3, but this time using μ.

the base structure T appears in the contravariant position. Another example of a negative datatype is $\mu\, T'$, where **data** $T'\ r = C'\,(r \to r)$ because r in T' appears in both contravariant and covariant positions. We say that r is in a negative position because $(r \to a)$ is analogous to $(\neg r \wedge a)$ when we think of $\to$ as a logical implication.

Recursive types in conventional approach. In conventional approach, the formation (i.e., datatype definition) of recursive types is restricted, but arbitrary elimination (i.e., pattern matching) over the values of recursive types is allowed. In particular, the formation of negative recursive types is restricted. Only positive recursive types are supported. Thus, in Figure 1.5, we have a restricted version of the formation rule (μ-form$^+$) with an additional condition that F should be positive. The other rules (μ-intro), (μ-elim), and (unIn-In) remain the same as in functional languages. Because we have restricted the recursive types at the type-level and we do not have general recursion at the term-level, the language is indeed normalizing. However, we cannot write interesting (i.e., recursive) programs that involve recursive types, nor can we reason inductively about those programs, unless we have principled recursion schemes that are guaranteed to normalize. One such recursion scheme is called iteration (a.k.a. catamorphism). The typing rules for the conventional iteration **It** are illustrated in Figure 1.5. Note, we have the typing rule (**It**) and the reduction rule (**It**-In) for **It** in addition to the rules for the recursive type operator μ.

Recursive types in Mendler-style approach. In Mendler-style approach, we allow arbitrary formation (i.e., datatype definition) of recursive types, but we restrict the elimination (i.e., pattern matching) over the values of recursive types. The formation rule (μ-form) remains the same as that for functional languages.

That is, we can define arbitrary recursive types, both positive and negative. However, we no longer have the elimination rule (μ-elim). That is, we are not allowed to pattern match against the values of recursive types freely, as we do for values of non-recursive datatypes using case expressions. We can only pattern match over the values of recursive types using Mendler-style recursion combinators. The rules for the Mendler-style iteration combinator **mit** are illustrated in Figure 1.5. Note that there are no rules for unIn in the Mendler-style approach. The typing rule (μ-elim) is replaced by (**mit**) and the reduction rule (unIn-In) is replaced by (**mit**-In). More precisely, the typing rule **mit** is both an elimination rule for recursive types and a typing rule for the Mendler-style iterator. You can think of the rule (**mit**) as replacing both the elimination rule (μ-elim) and the typing rule for conventional iteration (**It**), but in a safe manner that guarantees normalization.

1.4.2 Justification of Mendler style as a design choice.

We choose to base our approach to the design of a seamless synthesis of both logical reasoning and programming on Mendler style. It restricts elimination (i.e., pattern matching) over values of recursive types rather than restricting the formation (i.e., datatype definition) of recursive types (a more conventional approach). This design choice enables our language system to include all datatype definitions that are used in functional programming languages.

Functional programming promotes "functions as first class values". It is natural for both pass functions as arguments and embedding functions into (recursive) datatypes. There exist many interesting and useful examples in functional programming involving negative datatypes. In Section 3.9, we illustrate that the Mendler-style recursion scheme, which we discovered, can be used for expressing interesting examples involving negative datatypes.

Recall that the motivation of this thesis research is to search for an answer to the question "how does one build a seamless system where programmers can both

write programs and formally reason about those programs?" Mendler style is a promising approach because all recursive types (both positive and negative) are definable and the recursion schemes over those types are normalizing.

Under the Curry–Howard correspondence, to formally reason about a program, the logic needs to refer to the type of the program because the type interpreted as a proposition describes the property of the program. Since the Mendler-style approach does not restrict recursive datatype definitions, we can directly refer to the types of programs that use negative recursive datatypes. One may object that it is possible to indirectly model negative recursive types in conventional style, via alternative equivalent encodings, which map negative recursive types into positive ones. But, such encodings do not align with our motivation towards a seamless unified system for both programming and reasoning. It is undesirable to require programmers to significantly change their programs just to reason about them. If the change is unavoidable, it should be kept small. That is, the changed program should syntactically resemble the original program, which programmers usually write in functional programming languages. In Chapter 3, we show a number of example programs written in Mendler style that look closer to programs written using general recursion than programs written in conventional style.

1.4.3 Term-indexed types, type inference, and datatypes

One of the most frequently asked questions about our design choices for Nax regarding term-indexed types, is "why not dependent types?" Our answer is that a moderate extension of the polymorphic calculus is a better candidate than a dependently-typed calculus as the basis for a practical programming system. Language designs based on indexed types can benefit from existing compiler technology and type inference algorithms for functional programming languages. In addition, theories for term-indexed datatypes are simpler than theories for full-fledged dependent datatypes because term-indexed datatypes can be encoded as functions

(using Church-like encodings).

The implementation technology for functional programming languages based on polymorphic calculi is quite mature. There exist industrial-strength implementations such as the Glasgow Haskell Compiler (GHC), whose intermediate core language is an extension of F_ω. Our term-indexed calculi described in Part III are closely related to F_ω by an index-erasure property. The hope is that our implementation can benefit from these technologies.

Type inference algorithms for functional programming languages are often based on certain restrictions of Curry-style polymorphic lambda calculi. These restrictions are designed to avoid higher-order unification during type inference. We develop a conservative extension of Hindley–Milner type inference for Nax (Chapter 8). This is possible because Nax is based on our term-indexed calculi (Part III). Dependently-typed languages, however, are often based on bidirectional type checking, which requires annotations on top level definitions, unlike the Hindley–Milner type inference.

In dependent type theories, datatypes are usually supported as primitive constructs with axioms, rather than as functional encodings (e.g., Church encodings). One can give functional encodings for datatypes in a dependent type theory, but one soon realizes that the induction principles (or, dependent eliminators) for those datatypes cannot be derived within the pure dependent calculi [39]. So, dependently-typed reasoning systems support datatypes as primitives. For instance, Coq is based on the Calculus of Inductive Constructions, which extends the Calculus of Constructions [26] with dependent datatypes and their induction principles.

In contrast, in polymorphic type theories, all imaginable datatypes within the calculi have functional encodings (e.g., Church encodings). For instance, F_ω need not introduce datatypes as primitive constructs since F_ω can embed all imaginable datatypes including non-regular recursive datatypes with type indices.

Another reason to use term-indexed calculi over dependent type theories is to extend the application of Mendler-style recursion schemes, which are well-understood in the context of F_ω. Researchers have thought about (though not published)[13] a dependently-typed Mendler-style primitive recursion that is well-defined for positive datatypes (i.e., datatypes that have a map) but not for negative (or mixed-variant) datatypes. In our term-indexed calculi, we can embed Mendler-style recursion schemes (just as we embedded them in F_ω) that are well-defined for negative datatypes as well.

1.5 CONTRIBUTIONS

This dissertation makes contributions in several areas.

1. It organizes and expands the realm of *Mendler-style recursion schemes* (Part II)
2. It establishes meta-theories for *term-indexed types* (Part III),
3. It designs a practical language (with an implementation) *in the sweet spot* between programming and logical reasoning (Part IV), and
4. It identifies several interesting open problems related to the three aforementioned areas.

1.5.1 Contributions related to Mendler style

We organize a hierarchy of Mendler-style recursion schemes in two dimensions. The first dimension is the abstract operations they support. For instance, the Mendler-style iteration (**mit**) supports a single abstract operation, the recursive call. All other Mendler-style recursion schemes support the recursive call and an additional set of abstract operations. The second dimension is over the kind of the datatypes they operate over. For example, `Nat` has kind $*$, while `Vec` has kind

[13] Tarmo Uustalu described this on a whiteboard when we met with him at the University of Cambridge in 2011. We discuss this in the related work chapter (Section 9.3).

$* \to \mathtt{Nat} \to *$. Each recursion scheme is actually a family of recursion combinators sharing the same term definition (i.e., uniformly defined) but with different type signatures at each kind.

We expand the realm of Mendler-style recursion schemes in several ways. First, we report on a new recursion scheme **msfit**, which is useful for negative datatypes. Second, we study the termination behaviors of Mendler-style recursion schemes. Some recursion schemes (e.g., **mit**, **msfit**) always terminate for any recursive type, whereas others (e.g., **mcvpr**) terminate only for certain classes of recursive types. Third, we extend all Mendler-style recursion schemes to be expressive over term-indexed types. The Mendler style has been studied in the context of F_ω (and several extensions) which can express *type*-indexed types. To extend Mendler-style recursion schemes to be expressive over *term*-indexed types, we report on several theories for calculi (F_i and Fix_i) that support term indices. This is another important facet of our contribution.

We provide examples that illustrate scenarios in which each recursion scheme is useful in Chapter 3. The most interesting example among them is the type-preserving evaluator for a simply-typed Higher-Order Abstract Syntax (HOAS) in Section 3.9.3, which involves negative datatypes with indices. This example is our novel discovery, which reports that a type-preserving evaluator for a simply-typed HOAS can be expressed within F_ω.

In addition, we develop a better understanding of some existing Mendler-style recursion schemes. For instance, the existence of Mendler-style course-of-values recursion (**mcvpr**) is reported in the literature, but the calculus that can embed **mcvpr** was unknown hitherto. We embed Mendler-style course-of-values recursion into Fix_i (or into Fix_ω [3] when we do not consider term-indices).

1.5.2 Contributions to the theory of term-indexed types

Mendler-style recursion schemes have been studied in the context of polymorphic lambda calculi. For instance, Abel, Matthes, and Uustalu [4] embedded Mendler-style iteration (**mit**) into F_ω and Abel and Matthes [3] embedded Mendler-style primitive recursion (**mpr**) into Fix_ω. These calculi support type-indexed types.

To extend the realm of Mendler-style recursion schemes in order to include term-indexed types, we extended F_ω and Fix_ω to support term indices. In Part III, we present our new calculi F_i (Chapter 4) that extends F_ω with term indices, and Fix_i (Chapter 5) that extends Fix_ω with term indices. These calculi have an erasure property that states that well-typed terms in each calculus are also well-typed terms (when erased) in the underlying calculus. For instance, any well-typed term in F_i is also a well-typed term in F_ω, and there are no additional well-typed terms in F_i that are not well-typed in F_ω.

Our new calculi F_i and $\mathsf{Fix}_i$are strongly normalizing and logically consistent as we show by using the erasure properties. That is, the strong normalization and logical consistency of F_i and Fix_i are inherited from F_ω and Fix_ω. Since F_i and Fix_i are strongly normalizing and logically consistent, Mendler-style recursion schemes that can be embedded into these calculi are adequate for logical reasoning as well as for programming.

1.5.3 Contributions towards the Nax language design

We design and implement a prototypical language Nax that explores the sweet spot between programming oriented systems and logic oriented systems. The language features supported by Nax provide the advantages of both programming oriented systems and logic oriented systems. Nax supports both term- and type-indexed datatypes, rich families of Mendler-style recursion combinators, and a conservative

extension of Hindley–Milner type inference. We designed Nax so that its foundational theory and implementation framework could be kept simple.

Term- and type-indexed datatypes can express fine-grained program properties via the Curry–Howard correspondence, as in logic-oriented systems. Although not as flexible as full-fledged dependent types, indexed datatypes can still express program invariants such as stack-safe compilation (Section 7.2) and size invariants on data structures. Index types can simulate much of what dependent types can do using singleton types. Given that Nax has only erasable indices, the foundational theory can be kept simple, and it supports features that have the advantages of programming oriented systems (e.g., type inference, arbitrary recursive datatypes).

Adopting Mendler style provides the merits of both programming oriented systems and logic oriented systems. Since Mendler style is elimination based, one can define all recursive datatypes usually supported in functional programming languages. In addition, programs written using Mendler-style recursion combinators look more similar to the programs written using general recursion programs written in the Squiggol style. Because Nax supports only well-behaved (i.e., strongly normalizing) Mendler-style recursion combinators, it is safe to construct proofs using them. In addition, Mendler-style recursion combinators are naturally well-defined over indexed datatypes, which are essential to express fine-grained program properties. Mendler style provides type based termination, that is, termination is a by-product of type checking. Thus, it makes the implementation framework simple since we do not need extra termination checking algorithms.

The Hindley–Milner-style type inference is familiar to functional programmers. Nax can infer types for all programs that involve only regular datatypes, which are already inferable in Hindley–Milner, without any type annotation. Nax requires programs involving indexed datatypes to annotate their eliminators by index transformers, which specify the relationship between the input type index and the result type. Eliminators of non-recursive datatypes are case expressions and eliminators

of recursive datatypes are Mendler-style recursion combinators.

1.5.4 Contributions identifying open problems

We identify several open problems alongside the contributions mentioned in the previews subsections. We discuss the details of these open problems in Chapter 10. Here, we briefly introduce two of them.

Handling different interpretations of μ in one language system: Nax provides multiple recursion schemes used for describing different kinds of recursive computations over recursive datatypes. These recursion schemes are all motivated by concrete examples, which explains the need for multiple schemes. It is more convenient to express various kinds of recursive computations in Nax by choosing a recursion scheme that fits the structure of the computation, than in systems that provide only one induction scheme. However, there is a theoretical difficulty of handling multiple interpretations of the recursive type operator in one language system.

Recall that we can embed datatypes as functional encodings in our indexed type theory. Recursive datatypes and their recursion schemes in Nax are embedded using Mendler-style encodings. In Mendler style, one encodes the recursive type operator μ and its eliminator (the recursion scheme) as a pair. So, there are several different encodings of μ, one for each recursion scheme. Some recursion schemes subsume others (i.e., the more expressive one can simulate the other).

It would have been easy to describe the theory for Nax if we had one most powerful recursion scheme that subsumes all the others; then it would lead to a single interpretation of μ. Unfortunately, we know of no Mendler-style recursion scheme that subsumes all other recursion schemes we hope to support in Nax. For instance, iteration (**mit**) is subsumed by primitive recursion (**mpr**), but **mpr** does not subsume iteration with a syntactic inverse (**msfit**) or vice versa. There is no

known recursion scheme that can subsume both **msfit** and **mpr**.

We discovered that μ-values, to which **mit** can be applied, can be considered as a superset of $\breve{\mu}$-values, to which **msfit** can be applied. That is, casting from $\breve{\mu}$-values to μ-values is always safe. However, we do not know whether it is safe to cast the other way. We have neither found a casting function from μ-values to $\breve{\mu}$-values nor found a counterexample. We currently plan to support two fixpoints in Nax and allow one way casting. Further discussions can be found in Section 10.2.

Deriving positivity (or monotonicity) from polarized kinds: One can extend the kind syntax of arrow kinds in F_ω with polarities ($p\kappa_1 \to \kappa_2$ where the polarity p is either $+$, $-$, or 0) to track whether a type constructor argument is used in covariant (positive), contravariant (negative), or mixed-variant (both positive and negative) positions. Whether it is possible to derive monotonicity (i.e., the existence of a map) for a type constructor from its polarized kind, without examining the type constructor definition remains an open problem.

We identified a useful application for a solution to this open problem. We discovered an embedding of Mendler-style course-of-values recursion in a polarized system for positive (or monotone) type constructors. That is, once you can show the existence of a map for a datatype, course-of-values recursion always terminates. However, in a practical language system, it is not desirable to burden users with the manual derivation for every datatype on which they might want to perform course-of-values recursion. If the type system can automatically categorize datatypes that have maps from their polarized kinds, this burden can be alleviated.

1.6 CHAPTER ORGANIZATION

This dissertation consists of five parts: Part I (Prelude), Part II (Mendler style), Part III (Term-indexed λ-calculi), Part IV (Nax language), and Part V (Postlude). The three parts in the middle describe the three steps of our approach. First,

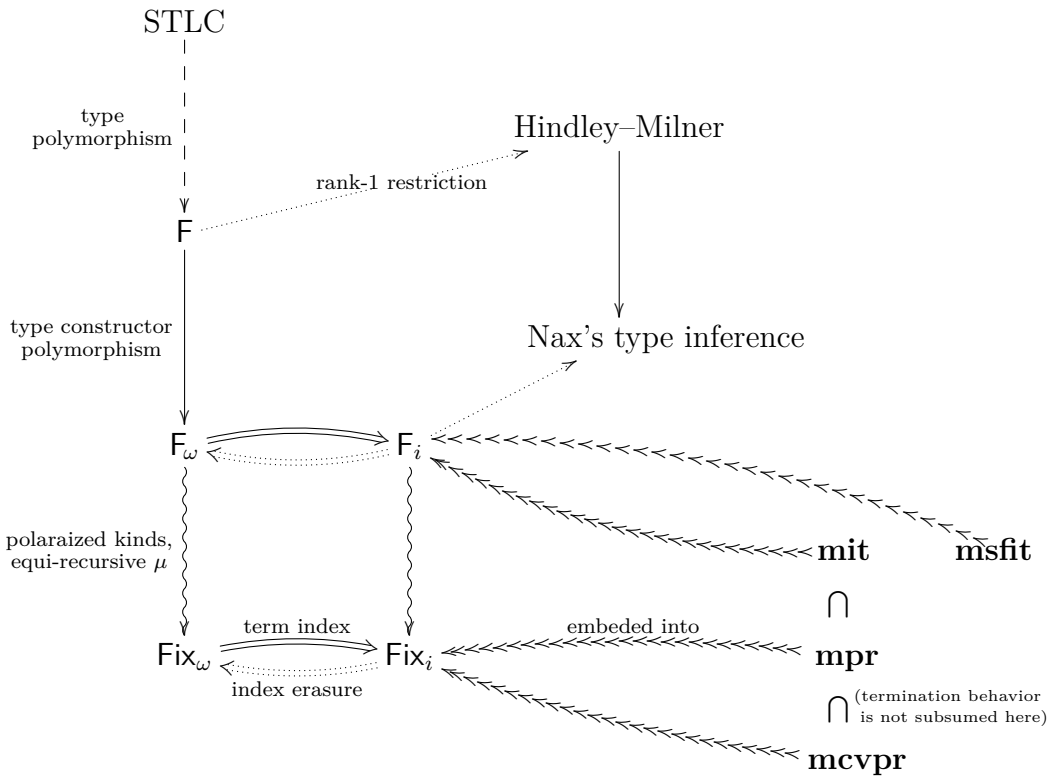


Figure 1.6: Summary of the relationships among key concepts.

we explore new ideas about Mendler-style recursion schemes driven by concrete examples using Haskell (with some GHC extensions). Second, we develop theories (i.e., lambda calculi) for term-indexed datatypes to prove that the Mendler-style recursion schemes are well-defined over indexed datatypes and have the expected termination behavior. Third, we design a language system with practical features, which implements our ideas and is based on the theory we developed. Figure 1.6 summarizes the organization of key concepts throughout the dissertation.

Part I (Prelude) comprises Chapter 1 (which you are currently reading) and Chapter 2 which reviews the theory of several well-known typed lambda calculi: the simply-typed lambda calculus (STLC) (Section 2.1), System F (Section 2.2), System F_ω (Section 2.3), and the Hindley–Milner type system (Section 2.4).

In Sections 2.1-2.3, we review strong normalization proofs (using saturated sets) for each of the three calculi: STLC (no polymorphism), System F (polymorphism over types), and System F_ω (polymorphism over type constructors).

Each proof extends the normalization proof of the previous calculus. We will use the strong normalization of System F_ω to show that our term-indexed lambda calculi in Part III are strongly normalizing. Readers familiar with strong normalization proofs of these calculi may skip or quickly skim over these sections. It is worth noticing two stylistic choices in our formalization of System F and F_ω: (1) terms are in Curry style and (2) typing contexts are divided into two parts (one for type variables and the other for term variables). This choice prepares readers for our formalization of the term-indexed calculi in Part III, which involves Curry-style terms and typing contexts divided into two parts.

In Section 2.4, we review the type inference algorithm for the Hindley–Milner type system (Section 2.4). The Hindley–Milner type system (HM) is a restriction of System F, which makes it possible to infer types without any type annotation on terms. Later in Part IV Chapter 8, we formulate a conservative extension of HM, which restricts the term-indexed calculus System F_i (Chapter 4) in a similar manner.

Part II (Mendler style) introduces the concept of Mendler-style recursion schemes (Chapter 3) using examples written in Haskell (with some GHC extensions). The readers of Chapter 3 need no background knowledge on typed lambda calculi but only some familiarity with functional programming. We explain the concepts of a number of Mendler-style recursion schemes, their termination properties, and the relationships among the recursion schemes. We also provide semi-formal proofs of termination for some of the recursion schemes (**mit** and **msfit**) by embedding them into the F_ω fragment of Haskell. More formal and general proofs, by embedding the schemes into our term-indexed lambda calculi, are given later

in Part III.

Mendler-style recursion schemes discussed in Chapter 3 include iteration (**mit**), iteration with syntactic inverse (**msfit**), primitive recursion (**mpr**), course-of-values iteration (**mcvit**), and course-of-values recursion (**mcvpr**). Of these, **msfit** was discovered while writing this dissertation. There are more Mendler-style recursion schemes that are not discussed in Chapter 3 — we give pointers to them in our related work chapter (Chapter 9 of Part V).

Part III (term-indexed lambda calculi) describes the developments of theories for term-indexed types. We formalize two term-indexed lambda calculi, which extends their underlying polymorphic calculi that support type indices only. System F_i (Chapter 4) extends System F_ω with term indices and System Fix_i (Chapter 5) extends System Fix_ω [3] with term indices.

We prove these term-indexed calculi to be strong normalizing and logical consistent of using their index erasure properties. The index erasure property of a term-indexed calculus projects a typing in the term-index calculi into its underlying polymorphic calculus from which the term-indexed calculus extends. That is, all well-typed terms in F_i and Fix_i are also well-typed terms in F_ω and Fix_ω.

By embedding Mendler-style recursion schemes into our term-indexed lambda calculi, we prove that those schemes are well-defined and terminate over term-indexed datatypes. For instance, **mit** and **msfit** can be embedded into System F_i, and, **mpr** and **mcvpr** can be embedded into System Fix_i.

Part IV (the Nax language) consists of three chapters. First, we introduce the features of Nax (Chapter 6) in a tutorial format using small Nax code snippets as examples. Next, we discuss the design principles of the type system (Chapter 7) by comparing it with two other systems: Haskell's datatype promotion and Agda. In Chapter 7 we develop larger and more practical examples, a type-preserving

interpreter and a stack-safe compiler. Lastly, we discuss type inference in Nax (Chapter 8), which is a conservative extension of the Hindley–Milner type system (HM). That is, any program whose type is inferable in HM, can also have its type inferred in Nax without any annotation. Programs involving term- or type-indexed datatypes, which are not supported in HM, need some annotation for their types to be inferred in Nax. These annotations are only required on three syntactic entities (datatype declarations, case expressions, and Mendler-style recursion combinators).

Part V (Postlude) closes the dissertation by summarizing related work (Chapter 9), future work (Chapter 10), and conclusions (Chapter 11).

Chapter 2

POLYMORPHIC TYPE SYSTEMS

In this chapter, we review the simply-typed lambda calculus (Section 2.1), a non-polymorphic type system, and a series of well-known polymorphic type systems: System F (Section 2.2), System F_ω (Section 2.3), and the Hindley–Milner type system (Section 2.4). We review them because F_i (Chapter 4), Fix_i (Chapter 5), and the Nax language (Chapter 7) in later chapters are extensions of these systems.

We assume the reader has some familiarity with lambda calculi, at least with the untyped lambda calculus. Readers with an expert understanding on polymorphic type systems and encodings of datatypes in such systems may skip this chapter and continue directly to the following chapters.

One of the purposes of this chapter is illustrating the strong normalization theorem for less common formulations of the polymorphic lambda calculi. System F and System F_ω are more often formulated in Church style and with a single typing context. Here, we illustrate them in Curry style and their typing rules with two typing contexts, because our indexed type theories, System F_i and System Fix_i, in Part III are formulated in such ways. Another purpose of this chapter is to familiarize the readers with functional encodings of datatypes in polymorphic type systems (see Section 2.2.1 and Section 2.3.1).

2.1 SIMPLY-TYPED LAMBDA CALCULUS

We illustrate two styles of the simply-typed lambda calculus (STLC) in Figure 2.1. The left column of the figure illustrates the Church-style STLC and the right

Church-style

term syntax

$$\begin{array}{rrl} t, s & ::= & x & \text{variable} \\ & | & \lambda(x : A).t & \text{abstraction} \\ & | & t\ s & \text{application} \end{array}$$

type syntax

$$\begin{array}{rrl} A, B & ::= & A \to B & \text{arrow type} \\ & | & \iota & \text{ground type} \end{array}$$

typing context

$$\begin{array}{rcl} \Gamma & ::= & \cdot \\ & | & \Gamma, x : A \quad (x \notin \mathrm{dom}(\Gamma)) \end{array}$$

typing rules $\boxed{\Gamma \vdash t : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Abs}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : A \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B}$$

reduction rules $\boxed{t \longrightarrow t'}$

$$\textsc{RedBeta}\ \frac{}{(\lambda(x : A).t)\ s \longrightarrow t[s/x]}$$

$$\textsc{RedAbs}\ \frac{t \longrightarrow t'}{\lambda(x : A).t \longrightarrow \lambda(x : A).t'}$$

$$\textsc{RedApp1}\ \frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s}$$

$$\textsc{RedApp2}\ \frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

Curry-style

term syntax

$$\begin{array}{rcl} t, s & ::= & x \\ & | & \lambda x.t \\ & | & t\ s \end{array}$$

type syntax

$$\begin{array}{rcl} A, B & ::= & A \to B \\ & | & \iota \end{array}$$

typing context

$$\begin{array}{rcl} \Gamma & ::= & \cdot \\ & | & \Gamma, x : A \quad (x \notin \mathrm{dom}(\Gamma)) \end{array}$$

typing rules $\boxed{\Gamma \vdash t : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Abs}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B}$$

reduction rules $\boxed{t \longrightarrow t'}$

$$\textsc{RedBeta}\ \frac{}{(\lambda x.t)\ s \longrightarrow t[s/x]}$$

$$\textsc{RedAbs}\ \frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'}$$

$$\textsc{RedApp1}\ \frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s}$$

$$\textsc{RedApp2}\ \frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

Figure 2.1: Simply-typed lambda calculus in Church style and Curry style

column illustrates the Curry-style STLC.

A term can be either a variable, an abstraction (a.k.a. lambda term), or an application. The distinction between the two styles comes from whether the abstraction has a type annotation in the term syntax. Abstractions in Church style have the form $\lambda(x : A).t$ with a type annotation A on the variable x bound in t. Abstractions in Curry style have the form $\lambda x.t$ without any type annotation. The differences in typing rules and reduction rules between the two styles follow from this distinction.

A type can be either an arrow type or a ground type. The type syntax is exactly the same in both styles. Arrow types are types for functions. For instance, abstractions have arrow types. We need ground types as a base case for the inductive definition of types. Otherwise, if there were no ground types, we would not be able to populate types.[1] Here, we choose to include only the simplest ground type, ι, which is also known as the void type. Note that there does not exist any closed term of type ι. It is only possible to construct terms of type ι when we have a bound variable, whose type is either ι or an arrow type that eventually returns ι, in the typing context.

When using the STLC to model a programming language (with simple types), a richer set of ground types (e.g., unit, boolean, natural numbers), rather than the void type alone, are provided. In such versions of the STLC, one must extend the term syntax by providing normal terms (or, constants) for those ground types (e.g., true and false for booleans) and eliminators (e.g., if-then-else expression for booleans) that can examine the normal terms. Later on, we shall see that polymorphic type systems such as System F (Section 2.2) and System F_ω (Section 2.3) are expressive enough to encode those ground types without introducing them as primitive constructs of the calculi. Having the void type as a ground type is enough

[1] If we allow infinite types, then it is possible to populate types without ground types. There exist exotic lambda calculi with infinite types, but these are rather uncommon.

to motivate polymorphic type systems, without complicating the term syntax of the STLC.

Typing rules are the rules to derive (or prove) typing judgments. A typing judgment $\Gamma \vdash t : A$ means that the term t has type A under the typing context Γ. We say t is well-typed (or, t is a well-typed term) under Γ when we can derive (or prove) a typing judgment $\Gamma \vdash t : A$ for some A according to the typing rules. There are just three typing rules — one typing rule for each item of the term syntax. Therefore, the typing rules of the STLC are syntax-directed in both styles. That is, there is exactly one rule to choose for the typing derivation by examining the shape of the term.

The reduction rules in Figure 2.1 describes β-reduction for the STLC. The RedBeta rule describes the key concept of β-reduction, the β-redex. A β-redex is an application of an abstraction to another term. The other three rules describe the idea that a redex may appear in subterms even though the term itself is not a redex. The reduction rules of the STLC are virtually the same as the reduction rules of the untyped lambda calculus. Note that reduction rules are not deterministic. There is no preferred order when there is more than one redex in a term. For instance, when there are redexes in both t and s in the application $(t \;\; s)$, one may apply either of the two rules RedApp1 and RedApp2.

We first discuss two important properties of the STLC, subject reduction and strong normalization, hold in both Curry style and Church style (Section 2.1.1). Then, we motivate the discussion of polymorphic type systems by reviewing the limitations of the STLC (Section 2.1.2).

2.1.1 Strong normalization

We discuss two important properties of the STLC, which hold in both Church style and Curry style — *subject reduction* (a.k.a. type preservation) and *strong*

normalization. Since we focus on strong normalization, we will be rather brief on the proof of subject reduction and elaborate in more detail on the proof of strong normalization.

Subject reduction states that reduction preserves types.

Theorem 2.1.1 (subject reduction)**.** $\dfrac{\Gamma \vdash t : A \qquad t \longrightarrow t'}{\Gamma \vdash t' : A}$

That is, when a well-typed term takes a reduction step, then the reduced term has the same type as the well-typed term. We can prove subject reduction by induction on the derivation of the reduction rules. The only interesting case is the RedBeta rule. Proving all the other rules is simply done by applying the induction hypothesis. Proving the RedBeta rule amounts to proving the substitution lemma:

Lemma 2.1.1 (substitution)**.** $\dfrac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash t[s/x] : B}$

Proof of the substitution lemma is a straightforward induction on the derivation of the typing judgement.

As an aside, when people use the STLC to model a programming language, they usually consider another property called *progress*, which states that well-typed terms are either *values* or can take an evaluation step. Values are terms that meet certain syntactic criteria, i.e., those terms that are meant to represent "final answers", or terms that are done evaluating. We do not further discuss progress in this dissertation.

An *evaluation* is a reduction strategy (i.e., a certain subset of the reduction relation which computes a value, hence the name *evaluation*), which is often deterministic. In such a setting, type safety is usually defined to be subject reduction together with progress — all well-typed terms are either fully evaluated (i.e., they are values), or they can take a step to another well-typed term. However, in a calculus considering reductions of terms to normal forms, rather than evaluations

to values, type safety is just subject reduction since normal terms are irreducible by definition.

Strong normalization. When we consider terms of the STLC as proofs in a propositional logic using the Curry–Howard correspondence, strong normalization is another important property of the STLC. Strong normalization states that every well-typed term reduces to a normal form, no matter what reduction strategy is followed.

To prove strong normalization of the STLC, we use the following proof strategy. We first define the set of strongly normalizing terms, which may or may not be well-typed, and show that all well-typed terms belong to this set. For each type, we define a distinct set of terms called the interpretation of that type. We show that the interpretation of every type belongs to the set of normalizing terms.

The discussion below on strong normalization uses the Curry-style term syntax, but this proof strategy also works well for the Church-style STLC.[2] In fact, this strategy originates from Girard's strong normalization proof for System F using reducibility candidates [40], and later rephrased using Tait's saturated sets [87]. In particular, we adopt the notation used in the paper by Abel and Matthes [3], which includes a strong normalization proof for an extension of F_ω using saturated sets.

The strong normalization proofs for System F (Section 2.2) and System F_ω (Section 2.3) in this chapter are also based on this strategy using saturated sets. As the language increases in complexity, we gradually increase the complexity of the interpretation of types in those systems.

The set of strongly normalizing terms (SN) can be defined using a straight

[2]This proof strategy generalizes well to more complicated systems such as System F, System F_ω, and even to dependently-typed calculi such as the Calculus of Constructions[38].

forward inductive definition:

$$\frac{s_1, \ldots, s_n \in \mathsf{SN}}{x\ s_1\ \cdots\ s_n \in \mathsf{SN}} \qquad \frac{t \in \mathsf{SN}}{\lambda x.t \in \mathsf{SN}} \qquad \frac{t' \in \mathsf{SN} \quad t \longrightarrow t'}{t \in \mathsf{SN}}$$

That is, variables and applications of a variable to strongly normalizing terms are in SN, abstractions are in SN when their bodies are in SN, and terms that reduce to SN are also in SN. Relying on the fact that normal order reduction (i.e., reduce the outermost leftmost redex first) always leads to a normal form if a normal form exists, we can alter the last rule of the inductive definition above to be more syntactic, which defines the same set SN, as follows:

$$\frac{s_1, \ldots, s_n \in \mathsf{SN}}{x\ s_1\ \cdots\ s_n \in \mathsf{SN}} \quad \frac{t \in \mathsf{SN}}{\lambda x.t \in \mathsf{SN}} \quad \frac{t[s/x]\ s_1\ \cdots\ s_n \in \mathsf{SN} \quad s \in \mathsf{SN}}{(\lambda x.t)\ s\ s_1\ \cdots\ s_n \in \mathsf{SN}}$$

A set $\mathcal{A}$ is saturated when it is closed under adding strongly normalizing neutral terms, and when it is closed under strongly normalizing weak head expansion:

$$\frac{s_1, \ldots, s_n \in \mathsf{SN}}{x\ s_1\ \cdots\ s_n \in \mathcal{A}} \qquad \frac{t[s/x]\ s_1\ \cdots\ s_n \in \mathcal{A} \quad s \in \mathsf{SN}}{(\lambda x.t)\ s\ s_1\ \cdots\ s_n \in \mathcal{A}}$$

There is a sort of cleverness in this definition of saturated. A set is saturated when the terms it contains are either variables, or "come from" other terms in the saturated set using these two rules (neutral terms and weak head expansion). We can easily observe that SN is a saturated set by definition. We can get the first and last part of the inductive definition for SN when $\mathcal{A} = \mathsf{SN}$. We can define an arrow operation ($\rightarrow$), which given two saturated sets, defines a third saturated set as follows:

$$\mathcal{A} \rightarrow \mathcal{B} = \{t \in \mathsf{SN} \mid t\ s \in \mathcal{B} \text{ for all } s \in \mathcal{A}\}$$

It is known that $\mathcal{A} \rightarrow \mathcal{B}$ is saturated when both $\mathcal{A}$ and $\mathcal{B}$ are saturated [87].

We interpret types as saturated subsets of SN (i.e., subsets of SN that are saturated) as in Figure 2.2. We interpret the void type as the minimal saturated set ($\bot$), which is saturated from the empty set. We choose the symbol $\bot$ since

Interpretation of types as saturated sets of normalizing terms:

$$\begin{aligned} [\![\iota]\!] &= \bot \qquad\qquad \text{(the minimal saturated set)} \\ [\![A \to B]\!] &= [\![A]\!] \to [\![B]\!] \end{aligned}$$

Interpretation of typing contexts as sets of valuations (ρ):

$$[\![\Gamma]\!] = \{\ \rho \in \mathrm{dom}(\Gamma) \to \mathsf{SN} \ \mid\ \rho(x) \in [\![\Gamma(x)]\!] \text{ for all } x \in \mathrm{dom}(\Gamma)\ \}$$

Interpretation of terms as terms themselves whose free variables are substituted according to the given valuation (ρ):

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![\lambda x.t]\!]_\rho &= \lambda x.[\![t]\!]_\rho \qquad (x \notin \mathrm{dom}(\rho)) \\ [\![t\ s]\!]_\rho &= [\![t]\!]_\rho\ [\![s]\!]_\rho \end{aligned}$$

Figure 2.2: Interpretation of types, typing contexts, and terms of the STLC for the proof of strong normalization

saturated sets form a complete lattice under the subset relation as the partial order. We may denote SN as $\top$ since it is the maximal element of the lattice. Note that $\bot$, or $[\![\iota]\!]$, does not include any abstraction (λ-terms) since ι is not the type of a function. Arrow types ($A \to B$) are interpreted as the saturated-set arrow over the interpretations of the domain type and the range type ($[\![A]\!] \to [\![B]\!]$).

We interpret a typing context (Γ) as a set of valuations (ρ). For every variable binding in the typing context ($x : A \in \Gamma$), a valuation should map the variable (x) to a term that belongs to the interpretation of its desired type ($[\![A]\!]$). That is, if $x : A \in \Gamma$ then any $\rho \in [\![\Gamma]\!]$ should satisfy the proposition that $\rho(x) \in [\![A]\!]$.

The proof of strong normalization amounts to proving the following theorem:

Theorem 2.1.2 (soundness of typing)**.** $\dfrac{\Gamma \vdash t : A \qquad \rho \in [\![\Gamma]\!]}{[\![t]\!]_\rho \in [\![A]\!]}$

Proof. We prove this by induction on the typing derivation ($\Gamma \vdash t : A$).

For variables, it is trivial to show that $\dfrac{\Gamma \vdash x : A \qquad \rho \in [\![\Gamma]\!]}{[\![x]\!]_\rho \in [\![A]\!]}$.
Because of the VAR rule, $x : A \in \Gamma$. Thus, $[\![x]\!]_\rho = \rho(x) \in [\![\Gamma(x)]\!] = [\![A]\!]$.

For abstractions, we need to show that $\dfrac{\Gamma \vdash \lambda x.t : A \to B \qquad \rho \in [\![\Gamma]\!]}{[\![\lambda x.t]\!]_\rho \in [\![A \to B]\!]}$.
Since $[\![\lambda x.t]\!]_\rho = \lambda x.[\![t]\!]_\rho$ and $[\![A \to B]\!] = \{t \in \mathsf{SN} \mid t\ s \in [\![B]\!] \text{ for all } s \in [\![A]\!]\}$, what we need to show is equivalent to the following:

$$\frac{\Gamma \vdash \lambda x.t : A \to B \qquad \rho \in [\![\Gamma]\!]}{\lambda x.[\![t]\!]_\rho \in \{t \in \mathsf{SN} \mid t\ s \in [\![B]\!] \text{ for all } s \in [\![A]\!]\}}$$

By induction, we know that: $\dfrac{\Gamma, x : A \vdash t : B \qquad \rho' \in [\![\Gamma, x : A]\!]}{[\![t]\!]_{\rho'} \in [\![B]\!]}$.
Since this holds for all $\rho' \in [\![\Gamma, x : A]\!]$, it also holds for a particular ρ', where $\rho' = \rho[x \mapsto s]$ for any $s \in [\![A]\!]$. So, $[\![t]\!]_{\rho[x \mapsto s]} = ([\![t]\!]_\rho)[s/x] \in [\![B]\!]$ for any $s \in [\![A]\!]$. Since saturated sets are closed under normalizing weak head expansion, $(\lambda x.[\![t]\!]_\rho)\ s \in [\![B]\!]$ for any $s \in [\![A]\!]$. Therefore, $\lambda x.[\![t]\!]_\rho$ is obviously in the set, which we wanted it to be in, i.e., $\lambda x.[\![t]\!]_\rho \in \{t \in \mathsf{SN} \mid t\ s \in [\![B]\!] \text{ for all } s \in [\![A]\!]\}$.

For applications, we need to show that $\dfrac{\Gamma \vdash t\ s : B \qquad \rho \in [\![\Gamma]\!]}{[\![t\ s]\!]_\rho \in [\![B]\!]}$.
By induction we know that

$$\frac{\Gamma \vdash t : A \to B \qquad \rho \in [\![\Gamma]\!]}{[\![t]\!]_\rho \in [\![A \to B]\!]} \qquad \frac{\Gamma \vdash s : A \qquad \rho \in [\![\Gamma]\!]}{[\![s]\!]_\rho \in [\![A]\!]}$$

Then, it is straightforward to see that $[\![t\ s]\!]_\rho \in [\![B]\!]$ by definition of $[\![A \to B]\!]$.

□

Corollary 2.1.1 (strong normalization)**.** $\dfrac{\Gamma \vdash t : A}{t \in \mathsf{SN}}$

Once we have proved the soundness of typing with respect to interpretation, it is easy to see that the STLC is strongly normalizing, even for open terms (i.e., terms with free variables), by giving a trivial interpretation such that $\rho(x) = x$ for all $x \in \mathrm{dom}(\Gamma)$. Note that $[\![t]\!]_\rho = t \in [\![A]\!] \subset \mathsf{SN}$ under the trivial interpretation.

2.1.2 Motivations for polymorphic type systems

A limitation of the STLC is that a variable (x) can be given only one type binding ($x : A$) in a given context (Γ). Thus a variable term can have only one type.

It is possible to give many typings for terms other than variables in Curry style (e.g., the abstraction $\lambda x.x$ in the previous subsection), a type for variable (x) is uniquely determined once the context (Γ) is determined. This becomes inconvenient when we want to abstract over functions that can be given multiple types, such as the identity function ($\lambda x.x$). That is, when we have a variable x_{id} that stands for ($\lambda x.x$) and have a context Γ such that $x_{\mathsf{id}} : A \to A \in \Gamma$ for some A, we cannot apply this x_{id} to arguments of differing types within the same context Γ. Bucause of this limitation, most typed functional languages are based on a polymorphic lambda calculus, which has a richer notion of types than the STLC. Polymorphic lambda calculi supports polymorphic types, such as $\forall X.X \to X$ for the type of the identity function, which capture the idea that the type variable X can be instantiated to any type, in each occurrence of the identity function (x_{id}).

We will introduce several well-known polymorphic lambda calculi and discuss their properties in the following subsections.

2.2 SYSTEM F

System F [40] extends the type syntax of the STLC with type variables (X) and forall types ($\forall X.B$), which enable us to express polymorphic types (see Figure 2.3). However, System F does not have a dedicated syntax for ground types, such as the void type ι in the STLC. In System F, we can populate types from forall types such as $\forall X.X$. This type is, in fact, an encoding of the void type. We shall see that a large class of datatypes are encodable in System F (Section 2.2.1)

Unlike in the STLC, not all types constructed by the type syntax of System F make sense. Since we have type variables in System F, we need to make sure that types are well-kinded. That is, we should make sure that all the type variables appearing in types are properly bound by universal quantifiers ($\forall$). For instance, consider the two types $\forall X.X$ and $\forall X.X'$. Under the empty kinding context, $\forall X.X$ is well-kinded since X is bounded by $\forall$, but $\forall X.X'$ is ill-kinded since X' is an unbound type variable. The kinding rules determine whether a type is well-kinded. In the kinding rules and typing rules, the kinding context (Δ) keeps track of the bound type variables. The complete syntax, kinding rules, and typing rules of System F are illustrated in Figure 2.3. The left column describes the Church-style System F and the right column describes the Curry-style System F. The reduction rules are shown separately in Figure 2.4.

As in the STLC, the term syntax for abstractions differs between the two styles. The Church-style System F has type annotations in abstractions but the Curry-style System F does not. Furthermore, the Church-style System F has additional syntax for type abstractions and type applications. The syntax for type abstractions ($\Lambda X.t$) makes it explicit that the type of the term should be generalized to a forall type. The syntax for type applications ($t[A]$) makes it explicit that the type

Church style

term syntax

$$
\begin{array}{rll}
t, s ::= & x & \text{variable} \\
| & \lambda(x : A).t & \text{abstraction} \\
| & t\ s & \text{application} \\
| & \Lambda X.t & \text{type abstraction} \\
| & t[A] & \text{type application}
\end{array}
$$

type syntax

$$
\begin{array}{rll}
A, B ::= & X & \text{variable type} \\
| & A \to B & \text{arrow type} \\
| & \forall X.B & \text{forall type}
\end{array}
$$

kinding & typing contexts

$$
\begin{array}{rl}
\Delta ::= & \cdot \\
| & \Delta, X \qquad (X \notin \mathrm{dom}(\Delta)) \\
\Gamma ::= & \cdot \\
| & \Gamma, x : A \qquad (x \notin \mathrm{dom}(\Gamma))
\end{array}
$$

kinding rules $\boxed{\Delta \vdash A}$

$$\textsc{TVar}\ \frac{X \in \Delta}{\Delta \vdash X}$$

$$\textsc{TArr}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\textsc{TAll}\ \frac{\Delta, X \vdash B}{\Delta \vdash \forall X.B}$$

typing rules $\boxed{\Delta; \Gamma \vdash t : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}$$

$$\textsc{Abs}\ \frac{\Delta \vdash A \qquad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda(x : A).t : A \to B}$$

$$\textsc{App}\ \frac{\Delta; \Gamma \vdash t : A \to B \qquad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash t\ s : B}$$

$$\textsc{TyAbs}\ \frac{\Delta, X; \Gamma \vdash t : B}{\Delta; \Gamma \vdash \Lambda X.t : \forall X.B}\ (X \notin \mathrm{FV}(\Gamma))$$

$$\textsc{TyApp}\ \frac{\Delta; \Gamma \vdash t : \forall X.B \qquad \Delta \vdash A}{\Delta; \Gamma \vdash t[A] : B[A/X]}$$

Curry style

term syntax

$$
\begin{array}{rll}
t, s ::= & x & \\
| & \lambda x.t & \\
| & t\ s &
\end{array}
$$

type syntax

$$
\begin{array}{rll}
A, B ::= & X & \\
| & A \to B & \\
| & \forall X.B &
\end{array}
$$

kinding & typing contexts

$$
\begin{array}{rl}
\Delta ::= & \cdot \\
| & \Delta, X \qquad (X \notin \mathrm{dom}(\Delta)) \\
\Gamma ::= & \cdot \\
| & \Gamma, x : A \qquad (x \notin \mathrm{dom}(\Gamma))
\end{array}
$$

kinding rules $\boxed{\Delta \vdash A}$

$$\textsc{TVar}\ \frac{X \in \Delta}{\Delta \vdash X}$$

$$\textsc{TArr}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\textsc{TAll}\ \frac{\Delta, X \vdash B}{\Delta \vdash \forall X.B}$$

typing rules $\boxed{\Delta; \Gamma \vdash t : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}$$

$$\textsc{Abs}\ \frac{\Delta \vdash A \qquad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{App}\ \frac{\Delta; \Gamma \vdash t : A \to B \qquad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash t\ s : B}$$

$$\textsc{TyAbs}\ \frac{\Delta, X; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X.B}\ (X \notin \mathrm{FV}(\Gamma))$$

$$\textsc{TyApp}\ \frac{\Delta; \Gamma \vdash t : \forall X.B \qquad \Delta \vdash A}{\Delta; \Gamma \vdash t : B[A/X]}$$

Figure 2.3: System F in Church style and Curry style.

Reduction rules for the Church-style System F

$$\textsc{RedBeta}\ \frac{}{(\lambda(x:A).t)\ s \longrightarrow t[s/x]} \qquad \textsc{RedTy}\ \frac{}{(\Lambda X.t)[A] \longrightarrow t[A/X]}$$

$$\textsc{RedAbs}\ \frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'} \qquad \textsc{RedTyAbs}\ \frac{t \longrightarrow t'}{\Lambda X.t \longrightarrow \Lambda X.t'}$$

$$\textsc{RedApp1}\ \frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s} \qquad \textsc{RedTyApp}\ \frac{t \longrightarrow t'}{t[A] \longrightarrow t'[A]}$$

$$\textsc{RedApp2}\ \frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

Reduction rules for the Curry-style System F

$$\textsc{RedBeta}\ \frac{}{(\lambda x.t)\ s \longrightarrow t[s/x]}$$

$$\textsc{RedAbs}\ \frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'}$$

$$\textsc{RedApp1}\ \frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s}$$

$$\textsc{RedApp2}\ \frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

Figure 2.4: Reduction rules of System F.

of the term should be instantiated to a specific type from a forall type. On the contrary, the Curry-style System F has neither type abstractions nor type applications in the term syntax. So, types are implicitly generalized and instantiated in Curry style. The differences in typing rules and reduction rules between the two styles follow from this difference in the term syntax.

The typing rules Var, Abs, and App are essentially the same as in the STLC except that we carry around the kinding context (Δ) along with the typing context (Γ). What are new in System F are the typing rules for type abstractions (TyAbs) and type applications (TyApp), which enable us to introduce forall types and instantiate forall types to a specific type. In Church style, the use of these two rules TyAbs and TyAbs are guided by the term syntax of type abstractions ($\Lambda X.t$) and type applications ($t[A]$). So, the typing rules of the Church-style System F are syntax-directed. In Curry style, on the other hand, there are no term syntax to guide the use of the rules TyAbs and TyApp. So, the typing rules of the Curry-style System F are not syntax-directed.

The reduction rules for the Church-style System F include all the reduction rules for the Church-style STLC. In addition, there are three more reduction rules (RedTy, RedTyAbs, and RedTyApp) involving type abstractions and type applications.

The reduction rules for the Curry-style System F are exactly the same as the reduction rules for the Curry-style STLC (Figure 2.1) since the terms of the Curry-style System F are identical to the terms of the Curry-style STLC.

2.2.1 Encoding datatypes in System F

System F is powerful enough to encode a fairly large class of datatypes within its type system. Encodings of well-known datatypes are listed in Table 2.1. In System F, we can encode non-recursive datatypes that are either simply typed (e.g., void, unit, and booleans) or parametrized (e.g., pairs and sums). More

interestingly, we can also encode recursive datatypes that are either simply typed (natural numbers) or parametrized (lists). All of these datatypes are classified as *regular datatypes*.[3] All regular datatypes that are not mutually recursive are encodable in System F. Encodings of mutually recursive datatypes seem to require more expressive type systems such as System F_ω (Section 2.3).

Church [22] devised an encoding for natural numbers in the untyped lambda calculus, based on the idea that the natural number n is represented by a higher-order function $(\lambda x_s.\lambda x_z.x_s^n\ x_z)$, which applies the first argument (x_s) n times to the second argument (x_z). Such an encoding of natural numbers is called Church numerals, after Alonzo Church. More generally, term encodings of the objects of datatypes based on similar ideas are called Church encodings. Church encodings were developed for the untyped λ-calculus. They cannot be well-typed in the STLC.

In System F, these Church-encoded terms can be well-typed by encoding the datatype as a polymorphic type of System F, as illustrated in Table 2.1. Such encodings for datatypes are called impredicative encodings since they rely on the impredicative polymorphism[4] of System F.

Encodings of types, constructors, and eliminators for well-known datatypes are listed in Table 2.1. We use the Curry-style System F since the constructors and the eliminators are exactly the same as the Church encodings in the untyped lambda calculus. If we were to use the Church-style System F, we would need to adjust the constructors and the eliminators by adding type abstractions and type applications in appropriate places. For example, the constructor for *Unit* would be $\texttt{Unit} = \Lambda X.\lambda x : X.x$ and the eliminator would be $\lambda(x : \mathit{Unit}).\Lambda X.x[X]\ x'$.

[3] We discuss the concept of regular datatypes, in contrast to non-regular datatypes, in Section 2.3.1.

[4] In System F, polymorphic type variables in a polymorphic type can be instantiated with the same polymorphic type itself. This self-referential property is called impredicativity. For instance, $\forall X.X \to X$ can be instantiated to $(\forall X.X \to X) \to (\forall X.X \to X)$ where X is instantiated with $\forall X.X \to X$,

void	encoding of type	$\mathit{Void} = \forall X.X$
	constructor	
	eliminator	$\lambda x.x$
unit	encoding of type	$\mathit{Unit} = \forall X.X \to X$
	constructor	$\texttt{Unit} = \lambda x.x$
	eliminator	$\lambda x.\lambda x'.x\ x'$
booleans	encoding of type	$\mathit{Bool} = \forall X.X \to X \to X$
	constructors	$\texttt{True} = \lambda x_1.\lambda x_2.x_1, \quad \texttt{False} = \lambda x_1.\lambda x_2.x_2$
	eliminator	$\lambda x.\lambda x_1.\lambda x_2.x\ x_1\ x_2$ (**if** x **then** x_1 **else** x_2)
pairs	encoding of type	$A_1 \times A_2 = \forall X.(A_1 \to A_2 \to X) \to X$
	constructor	$\texttt{Pair} = \lambda x_1.\lambda x_2.\lambda x'.x'\ x_1\ x_2$
	eliminator	$\lambda x.\lambda x'.x\ x'$
		(by passing appropriate values to x', we get
		$\mathit{fst} = \lambda x.x(\lambda x_1.\lambda x_2.x_1)$, $\mathit{snd} = \lambda x.x(\lambda x_1.\lambda x_2.x_2)$)
sums	encoding of type	$A_1 + A_2 = \forall X.(A_1 \to X) \to (A_2 \to X) \to X$
	constructors	$\texttt{Inl} = \lambda x.\lambda x_1.\lambda x_2.x_1\ x, \quad \texttt{Inr} = \lambda x.\lambda x_2.\lambda x_2.x_2\ x$
	eliminator	$\lambda x.\lambda x_1.\lambda x_2.x\ x_1\ x_2$
		(**case** x **of** $\{\texttt{Inl}\ x' \to x_1\ x'$; $\texttt{Inr}\ x' \to x_2\ x'\}$)
natural numbers	encoding of type	$\mathit{Nat} = \forall X.(X \to X) \to X \to X$
	constructors	$\texttt{Succ} = \lambda x.\lambda x_s.\lambda.x_z.x_s(x\ x_s\ x_z)$,
		$\texttt{Zero} = \lambda x_s.\lambda x_z.x_z$
	eliminator	$\lambda x.\lambda x_z.\lambda x_s.x\ x_s\ x_z$ (iteration on natural num.)
lists	encoding of type	$\mathit{List}\ A = \forall X.(A \to X \to X) \to X \to X$
	constructors	$\texttt{Cons} = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\ x_a\ (x\ x_c\ x_n)$,
		$\texttt{Nil} \ = \lambda x_c.\lambda x_n.\lambda x_n$
	eliminator	$\lambda x.\lambda x_c.\lambda x_n.x\ x_c\ x_n$ ($\mathit{foldr}\ x_z\ x_c\ x$ in Haskell)

Table 2.1: Church encodings of regular datatypes can be well-typed in System F.

Constructors produce objects of a datatype. Nullary constructors (a.k.a. constants) are objects by themselves. For example, `Unit` (or, $\lambda x.x$) is a unit object, `True` (or, $\lambda x_1.\lambda x_2.x_1$) is a boolean object, `Zero` (or, $\lambda x_s.\lambda x_z.x_z$) is a natural number, and `Nil` (or, $\lambda x_c.\lambda x_n.\lambda x_n$) is a list. That is,

$$\vdash \texttt{Unit} : \mathit{Unit} \qquad \vdash \texttt{True} : \mathit{Bool} \qquad \vdash \texttt{Zero} : \mathit{Nat} \qquad \vdash \texttt{Nil} : \forall X_a.\mathit{List}\, X_a$$

where *Unit* is a shorthand notation (or, type synonym) for $\forall X.X \to X$, *Bool* is for $\forall X.X \to X \to X$, and so on, as described in Figure 2.1. Other (non-nullary) constructors expect some arguments in order to produce objects. For example, `Pair` expects two arbitrary arguments to produce a pair, `Succ` expects a natural number argument to produce another natural number, and `Cons` expects a new element and a list as arguments to produce another list. That is,

$$\vdash \texttt{Pair} : \forall X_1.\forall X_2.X_1 \to X_2 \to X_1 \times X_2 \qquad \vdash \texttt{Succ} : \mathit{Nat} \to \mathit{Nat}$$
$$\vdash \texttt{Cons} : \forall X_a.X_a \to \mathit{List}\, X_a \to \mathit{List}\, X_a$$

where $X_1 \times X_2$, *Nat*, and $\mathit{List}\, X_a$ are shorthand notations for encodings of the datatypes, as described in Figure 2.1.

We can deduce the number of constructors for a datatype and the types of those constructors from the impredicative encoding of the datatype. The general form for the encodings of the simply-typed datatypes is:

$$D = \forall X.A_1 \to \cdots \to A_n \to X \qquad \text{where} \quad A_i = A_{i1} \to \cdots \to A_{ik} \to X$$

From the encoding of type above, we can deduce the following facts:

- n is the number of constructors,
- k is the arity of the ith constructor, and
- the type of the ith constructor is $A_i[D/X]$.

Note, D is a shorthand notation for the entire encoding of the type. So, $A_i[D/X]$ expands to $A_i[(\forall X.A_1 \to \cdots \to A_n \to X)/X]$. Here, the type variable X in A_i is

substituted by a polymorphic type $D = (\forall X. \cdots)$. Recall that X in A_i comes from the variable universally quantified in D. In System F, we are able to substitute the universally quantified type variable X with the very polymorphic type D, within which X is universally quantified. For this ability of self-instantiation referring to itself, we say "System F is *impredicative*". Impredicative encodings of datatypes rely on this impredicative nature (or, impredicativity) of System F.

Similarly, the general form for the encodings of the parametrized datatypes is $D\,X_1 \cdots X_k = \forall X.A_1 \to \cdots \to A_n \to X$. Then, the number of constructors is n and the type of the ith constructor is $\forall X_1. \cdots \forall X_n.A_i[D\,X_1 \cdots X_k/X]$.

Eliminators consume objects of a datatype for computation. An eliminator for a datatype expects an object of the datatype as its first argument followed by arguments of computations to be performed for each of the constructors. For instance, the eliminator for void ($\lambda x.x$) expects only one argument since void has no constructor, the eliminator for unit ($\lambda x.\lambda x'.x\ x'$) expect two arguments since unit has one constructor, and the eliminator for booleans ($\lambda x.\lambda x_1.\lambda x_2.x\ \ x_1\,x_2$) expect three arguments since there are two boolean constructors.

Eliminators examine the shape of the object (i.e., by which constructor it is constructed) in order to perform the computation that corresponds to the shape of the object. For instance, the eliminator for booleans amounts to the well-known if-then-else expression. For recursive types, computations are performed recursively because some of their constructors would expect recursive arguments. For instance, note that ($x\ x_s\ x_z$) appearing in the definition of `Succ` coincides with the body of the eliminator for natural numbers. Eliminators for recursive types are also known as iterators or folds.

The impredicative encoding of a datatype specifies what is needed to eliminate an object of the datatype. Recall the general form for the encodings of the simply-typed datatypes:

$$D = \forall X.A_1 \to \cdots \to A_n \to X \qquad \text{where} \quad A_i = A_{i1} \to \cdots \to A_{ik} \to X$$

We can understand this encoding as follows:

> To compute the result of type X from an object of type D, we need n small computations, whose types are $A_1, \ldots, A_n$. When the object is constructed by ith constructor, we use the ith small computation, whose type is A_i, that is, $A_{i1} \to \cdots \to A_{ik} \to X$. This small computation gathers all the k arguments supplied to the ith constructor for the object construction, in order to compute the result from those arguments.

For constants, the eliminator simply returns the argument being passed to handle the constant, as it is. For example, the unit eliminator $(\lambda x.\lambda x'.x\ x')$ will return what is passed into x'. That is,

$$(\lambda x.\lambda x'.x\ x')\ \texttt{Unit}\ s \longrightarrow (\lambda x'.\texttt{Unit}\ x')\ s \longrightarrow \texttt{Unit}\ s \longrightarrow s$$

since $\texttt{Unit} = \lambda x.x$. Similarly, the boolean eliminator $(\lambda x.\lambda x_1.\lambda x_2.x\ x_1\ x_2)$ simply returns x_1 when x is `True` and returns x_2 when x is `False`, owing to the definition of $\texttt{True} = \lambda x_1.\lambda x_2.x_1$ and $\texttt{False} = \lambda x_1.\lambda x_2.x_2$.

For non-nullary constructors, the argument being passed to the eliminator to handle the constructor must be a function that collects the arguments used for the object construction. The pair eliminator $(\lambda x.\lambda x'.x\ x')$ expects the argument x' be of type $X_1 \to X_2 \to X$ where X is the result type you want to compute. For example, you may pass an addition function $(\mathit{Nat} \to \mathit{Nat} \to \mathit{Nat})$ to x' to compute the sum of the first element and the second element of a pair of natural numbers $(\mathit{Nat} \times \mathit{Nat})$. We can define selector functions *fst* and *snd* for pairs by providing an appropriate argument for x' as described in Table 2.1.

The key idea behind Church encodings is that objects are defined by how they will be eliminated. That is, the Church encoded objects are, in fact, eliminators. Readers familiar with lambda calculi may have noticed that all the eliminators in

Table 2.1 are η-expansions of the identity function. The formulation of eliminators in Table 2.1 is simply to emphasize how many arguments each eliminator expects.

2.2.2 Subject reduction and strong normalization

We discuss two important properties of System F, which hold in both Church style and Curry style — *subject reduction* (a.k.a. *type preservation*) and *strong normalization*.

Subject reduction

The subject reduction theorem for System F can be stated as follows:

Theorem 2.2.1 (subject reduction)**.** $\dfrac{\Delta; \Gamma \vdash t : A \qquad t \longrightarrow t'}{\Delta; \Gamma \vdash t' : A}$

We can prove subject reduction for System F in a similar fashion to the proof of subject reduction for the STLC, by induction on the derivation of the reduction rules.

In Church style, proof for all other cases except for the rules RedBeta and RedTy are simply done by applying the induction hypothesis. Since the typing rules in Church style are syntax-directed, there is no ambiguity for which typing rule should be used in the derivation for a certain judgment. For the RedBeta case, we use the substitution lemma. For proving the RedTy case, we use the type substitution lemma. The substitution lemma and the type-substitution lemma are stated below:

Lemma 2.2.1 (substitution)**.** $\dfrac{\Delta; \Gamma, x : A \vdash t : B \qquad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash t[s/x] : B}$

Lemma 2.2.2 (type substitution)**.** $\dfrac{\Delta, X; \Gamma \vdash t : B \qquad \Delta \vdash A}{\Delta; \Gamma \vdash t[A/X] : B[A/X]}$ $(X \notin \mathrm{FV}(\Gamma))$

In Curry style, the most interesting case is the RedBeta rule, where we use the substitution lemma. The other rules simply apply the induction hypothesis.

There is a small complication in the proof, compared to the proof in Church style, since the typing rules are not syntax-directed. Although we have fewer rules to consider than in the Church-style System F, we need to deal with the ambiguity of which rule to apply in order to obtain a typing judgement. The ambiguity is due to the rules TyAbs and TyApp.

An alternative way to prove subject reduction for the Curry-style System F is by translating the subject reduction property of the Curry-style System F into the subject reduction property of the Church-style System F. That is, we extract a Church-style term from a typing derivation in Curry style. It is not difficult to see that each typing derivation in Curry style corresponds to a unique Church-style term, and, that a reduction step in Curry style corresponds to one or more reduction steps in Church style.[5]

Strong Normalization

To prove strong normalization of System F, we use the same proof strategy as in the proof of strong normalization of the STLC in Section 2.1.1. That is, we interpret types as saturated sets of normalizing terms, which may or may not be well-typed. The interpretation of types, contexts, and terms of System F are illustrated in Figure 2.5. Since we have type variables, we need a type valuation (ξ) that maps the type variables to interpretations of types. So, the interpretation of types are indexed by the type valuation (ξ), and the interpretation of terms are indexed by the pair of term and type valuations ($\xi; \rho$). A type valuation ξ is a function from $\mathrm{dom}(\Delta)$, the set of type variables bound in Δ, to SAT, the set of all saturated sets.

Any type interpretation is a saturated set. Since ξ maps a type variable to a saturated set, $[\![X]\!]_\xi \in \mathsf{SAT}$. We know $[\![A \to B]\!]_\xi \in \mathsf{SAT}$ since saturated sets are

[5] This correspondence between reduction steps in two styles is not always one step to one step. For instance, the reduction rules RedTyAbs and RedTyApp in Church style correspond to zero reduction step in Curry style.

Interpretation of types as saturated sets of normalizing terms whose free type variables are substituted according to the given type valuation (ξ):

$$\begin{aligned}
[\![X]\!]_\xi &= \xi(X) \\
[\![A \to B]\!]_\xi &= [\![A]\!]_\xi \to [\![B]\!]_\xi \\
[\![\forall X.B]\!]_\xi &= \bigcap_{\mathcal{A} \in \mathsf{SAT}} [\![B]\!]_{\xi[X \mapsto \mathcal{A}]} \qquad (X \notin \mathrm{dom}(\xi))
\end{aligned}$$

Interpretation of kinding and typing contexts as sets of type valuations and term valuations (ξ and ρ):

$$\begin{aligned}
[\![\Delta]\!] &= \mathrm{dom}(\Delta) \to \mathsf{SAT} \\
[\![\Delta; \Gamma]\!] &= \{\xi; \rho \mid \xi \in [\![\Delta]\!], \rho \in [\![\Gamma]\!]_\xi\} \\
[\![\Gamma]\!]_\xi &= \{\rho \in \mathrm{dom}(\Gamma) \to \mathsf{SN} \mid \rho(x) = [\![\Gamma(x)]\!]_\xi \text{ for all } x \in \mathrm{dom}(\Gamma)\}
\end{aligned}$$

Interpretation of terms as terms themselves whose free variables are substituted according to the given pair of type and term valuation (ξ;ρ):

$$\begin{aligned}
[\![x]\!]_{\xi;\rho} &= \rho(x) \\
[\![\lambda x.t]\!]_{\xi;\rho} &= \lambda x.[\![t]\!]_{\xi;\rho} \qquad (x \notin \mathrm{dom}(\rho)) \\
[\![t\ s]\!]_{\xi;\rho} &= [\![t]\!]_{\xi;\rho}\ [\![s]\!]_{\xi;\rho}
\end{aligned}$$

Figure 2.5: Interpretation of types, kinding and typing contexts, and terms of System F for the proof of strong normalization.

closed under the arrow operation ($\rightarrow$), as we mentioned in Section 2.1.1. $[\![\forall X.B]\!]_\xi \in \mathsf{SAT}$ since it is known that saturated sets are closed under set indexed intersection.

The proof of strong normalization amounts to proving the following theorem:

Theorem 2.2.2 (soundness of typing)**.** $\dfrac{\Delta; \Gamma \vdash t : A \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![A]\!]_\xi}$

Proof. We prove by induction on the typing derivation ($\Delta; \Gamma \vdash t : A$).

Case (VAR) It is trivial to show that $\dfrac{\Delta; \Gamma \vdash x : A \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![x]\!]_{\xi;\rho} \in [\![A]\!]_\xi}$.
We know that $x : A \in \Gamma$ from the VAR rule. So, $[\![x]\!]_{\xi;\rho} = \rho(x) \in [\![\Gamma(x)]\!]_\xi = [\![A]\!]_\xi$.

Case (ABS) We need to show that $\dfrac{\Delta; \Gamma \vdash \lambda x.t : A \rightarrow B \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![\lambda x.t]\!]_{\xi;\rho} \in [\![A \rightarrow B]\!]_\xi}$.
By induction, we know that $\dfrac{\Delta; \Gamma, x : A \vdash t : B \qquad \xi'; \rho' \in [\![\Delta; \Gamma, x : A]\!]}{[\![t]\!]_{\xi';\rho'} \in [\![B]\!]_\xi}$.
Since this holds for all $\xi'; \rho' \in [\![\Delta; \Gamma, x : A]\!]$, it also holds for particular $\xi'; \rho'$ such that $\xi' = \xi$ and $\rho' = \rho[x \mapsto s]$ for any $s \in [\![A]\!]'_\xi = [\![A]\!]_\xi$. Since saturated sets are closed under normalizing weak head expansion, $(\lambda x.[\![t]\!]_{\xi;\rho})\ s \in [\![B]\!]_\xi$ for any $s \in [\![A]\!]_\xi$. Therefore, $\lambda x.[\![t]\!]_{\xi;\rho}$ is obviously in the desired set,

$$[\![\lambda x.t]\!]_{\xi;\rho} = \lambda x.[\![t]\!]_{\xi;\rho} \in \{t \in \mathsf{SN} \mid t\ s \in [\![B]\!] \text{ for all } s \in [\![A]\!]\} = [\![A \rightarrow B]\!]_\xi$$

Case (APP) We need to show that $\dfrac{\Delta; \Gamma \vdash t\ s : B \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t\ s]\!]_{\xi;\rho} \in [\![B]\!]_\xi}$.

By induction we know that

$$\frac{\Delta; \Gamma \vdash t : A \rightarrow B \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![A \rightarrow B]\!]_\xi} \qquad \frac{\Delta; \Gamma \vdash s : A \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![s]\!]_{\xi;\rho} \in [\![A]\!]_\xi}$$

Then, it is straightforward to see that $[\![t\ s]\!]_{\xi;\rho} \in [\![B]\!]_\xi$ by the definition of $[\![A \rightarrow B]\!]_\xi$.

Case (TYABS) We need to show that $\dfrac{\Delta; \Gamma \vdash t : \forall X.B \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![\forall X.B]\!]_\xi}$

By induction, we know that

$$\frac{\Delta, X; \Gamma \vdash t : B \qquad \xi'; \rho' \in [\![\Delta, X; \Gamma]\!]}{[\![t]\!]_{\xi';\rho'} \in [\![B]\!]_{\xi'}}\ (X \notin \mathrm{FV}(\Gamma))$$

Since this holds for all $\xi'; \rho' \in [\![\Delta, X; \Gamma]\!]$ where $X \notin \mathrm{FV}(\Gamma)$, it also holds for particular subset such that $\xi' = \xi[X \mapsto \mathcal{A}]$ and $\rho' = \rho$ for all $\mathcal{A} \in \mathsf{SAT}$. That is,

$$[\![t]\!]_{\xi[X\mapsto\mathcal{A}];\rho} \in [\![B]\!]_{\xi[X\mapsto\mathcal{A}]} \quad \text{for all } \mathcal{A} \in \mathsf{SAT}$$

From $X \notin \mathrm{FV}(\Gamma)$, we know that $[\![t]\!]_{\xi[X\mapsto\mathcal{A}];\rho} = [\![t]\!]_{\xi;\rho}$ because ρ is independent of that to which X maps. So, we know that

$$[\![t]\!]_{\xi;\rho} \in [\![B]\!]_{\xi[X\mapsto\mathcal{A}]} \quad \text{for all } \mathcal{A} \in \mathsf{SAT}$$

By set theoretic definition, this is exactly what we wanted to show:

$$[\![t]\!]_{\xi;\rho} \in \bigcap_{\mathcal{A}\in\mathsf{SAT}} [\![B]\!]_{\xi[X\mapsto\mathcal{A}]} = [\![\forall X.B]\!]_{\xi}$$

Case (TyApp) We need to show that $\dfrac{\Delta; \Gamma \vdash t : B[A/X] \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![B[A/X]]\!]_{\xi}}$

By induction, we know that $\dfrac{\Delta; \Gamma \vdash t : \forall X.B \qquad \xi'; \rho' \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi';\rho'} \in [\![\forall X.B]\!]_{\xi'}}$.

Since this holds for all $\xi'; \rho' \in [\![\Delta, \Gamma]\!]$, it also holds for $\xi' = \xi$ and $\rho' = \rho$.

Thus, $[\![t]\!]_{\xi;\rho} \in [\![\forall X.B]\!]_{\xi} = \bigcap_{\mathcal{A}\in\mathsf{SAT}} [\![B]\!]_{\xi[X\mapsto\mathcal{A}]} \subseteq [\![B]\!]_{\xi[X\mapsto[\![A]\!]_{\xi}]} = [\![B[A/X]]\!]_{\xi}$.

□

Corollary 2.2.1 (strong normalization)**.** $\dfrac{\Delta; \Gamma \vdash t : A}{t \in \mathsf{SN}}$

Once we have proved the soundness of typing with respect to interpretation, it is easy to see that System F is strongly normalizing by giving a trivial term interpretation $\rho(x) = x$ for all the free variables. Note that $[\![t]\!]_{\xi;\rho} = t \in [\![A]\!]_{\xi} \subset \mathsf{SN}$ under the trivial interpretation.

2.3 SYSTEM F_ω

System F_ω [41] extends the type syntax of System F with lambda types and application types (see Figure 2.6). Lambda types ($\lambda X^\kappa.F$) and application types ($F\ G$), at the type-level, are analogous to lambda terms and applications at the term level. Type constructors are like functions, but at the type-level. Type constructors are categorized by kinds, just as terms are categorized by types. Type constructors of kind $*$ are just *types*, and do not expect any arguments. Type constructors that expect an argument have arrow kinds ($\kappa \to \kappa'$). A type constructor of kind $\kappa \to \kappa'$ expects another type constructor of kind κ as an argument to produce yet another type constructor of kind κ', just as a function of type $A \to B$ expects another term of type A as an argument, to produce yet another term of type B. By convention, A and B stand for types (i.e., type constructors of kind $*$), while F and G stand for type constructors or arbitrary kinds.

We can think of System F as a restriction of System F_ω, where we only allow types of kind $*$. That is, all the type variables appearing in well-kinded types in System F are of kind $*$. Since there exists only one kind ($*$) in System F, the kinding rules of System F only needs to ensure that type variables are bound in the context.

Since the kind structure of System F_ω is richer than the kind structure of System F, we need to keep track of the kind of the type variables in the kinding context (Δ). So, the kinding context is extended by a type variable annotated by its kind (X^κ). The kinding rules of System F_ω keep track of the kinds of type constructors as well as ensuring that the type variables are bound in Δ.

The kinding rules - TVar, TArr, and TAll - for the type syntax inherited from System F are similar to their counterparts in System F, except for this additional kinding annotation. The kinding rules TLam and TApp state when the extensions (lambda types and application types) to System F are well-kinded.

The typing rules of System F_ω are almost identical to the typing rules of System F, except for one new rule Conv. The Conv rule supports conversion between equivalent types.

- In the STLC, types are equal when they are syntactically identical.
- In System F, types are equal when they are α-equivalent (i.e., up to change of bound type variable names). For example, $\forall X.X$ and $\forall X'.X'$ are considered to be same types in System F.
- In System F_ω, we expect a richer notion of equality which incorporates the notion of β-equivalence at the type-level, since the type syntax of System F_ω has the structure of the STLC at the type-level. For instance, we want $(\lambda X^*.X)A = A$.

The equality rules over the type constructors of System F_ω are illustrated in Figure 2.7. The EqTBeta rule describes the essence of β-equivalence. Other rules describe the structural nature of equality (EqTVar, EqTArr, EqTAll, EqTLam, EqTApp) and transitivity of equality (EqTTrans).

The syntax, kinding rules, and typing rules of System F_ω are illustrated in Figure 2.6. We consider only the Curry-style term syntax for System F_ω. Since lambda binders exist at both the term- and the type-levels in System F_ω, we also have a choice of either Church style (kind annotations on lambda types) or Curry style (no kind annotations on lambda types) for the type syntax. We consider only the Church-style type syntax with explicit kind annotations.

The reduction rules of System F_ω are almost identical to the reduction rules of System F since the term syntax of System F_ω is almost identical to the term syntax of System F. Reduction rules are defined only on the structure of terms, usually ignoring types.

term syntax	$t, s ::= x$	variable
	$\mid \lambda(x : A).t$	abstraction
	$\mid t\ s$	application
type syntax	$F, G, A, B ::= X$	variable type
	$\mid A \to B$	arrow type
	$\mid \forall X^\kappa.B$	forall type
	$\mid \lambda X^\kappa.F$	lambda type
	$\mid F\ G$	application type
kind syntax	$\kappa ::= \kappa \to \kappa'$	arrow kind
	$\mid *$	star kind

kinding rules $\boxed{\Delta \vdash F : \kappa}$

$$\textsc{TVar}\frac{X^\kappa \in \Delta}{\Delta \vdash X : \kappa} \qquad \textsc{TArr}\frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$$

$$\textsc{TAll}\frac{\Delta, X^\kappa \vdash B : *}{\Delta \vdash \forall X^\kappa.B : *} \qquad \textsc{TLam}\frac{\Delta, X^\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X^\kappa.F : \kappa \to \kappa'}$$

$$\textsc{TApp}\frac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\ G : \kappa'}$$

typing rules $\boxed{\Delta; \Gamma \vdash t : A}$

$$\textsc{Var}\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A} \qquad \textsc{Abs}\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{App}\frac{\Delta; \Gamma \vdash t : A \to B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash t\ s : B} \qquad \textsc{TyAbs}\frac{\Delta, X^\kappa; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X^\kappa.B}\ (X \notin \mathrm{FV}(\Gamma))$$

$$\textsc{TyApp}\frac{\Delta; \Gamma \vdash t : \forall X^\kappa.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t : B[G/X]} \qquad \textsc{Conv}\frac{\Delta; \Gamma \vdash t : A \quad \Delta \vdash A = A' : *}{\Delta; \Gamma \vdash t : A'}$$

reduction rules $\boxed{t \longrightarrow t'}$

$$\textsc{RedBeta}\frac{}{(\lambda x.t)\ s \longrightarrow t[s/x]} \qquad \textsc{RedAbs}\frac{t \longrightarrow t'}{\lambda x.t \longrightarrow \lambda x.t'}$$

$$\textsc{RedApp1}\frac{t \longrightarrow t'}{t\ s \longrightarrow t'\ s} \qquad \textsc{RedApp2}\frac{s \longrightarrow s'}{t\ s \longrightarrow t\ s'}$$

Figure 2.6: Syntax, kinding rules, typing rules, and reduction rules of System F_ω.

$$\textsc{EqTBeta}\ \frac{\Delta, X^\kappa \vdash F : \kappa \to \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\ G = F[G/X] : \kappa'} \qquad \textsc{EqTVar}\ \frac{X^\kappa \in \Delta}{\Delta \vdash X = X : \kappa}$$

$$\textsc{EqTArr}\ \frac{\Delta \vdash A = A' : * \qquad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *} \qquad \textsc{EqTAll}\ \frac{\Delta, X^\kappa \vdash B = B' : *}{\Delta \vdash \forall X^\kappa.B = B' : *}$$

$$\textsc{EqTLam}\ \frac{\Delta, X^\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X^\kappa.F = \lambda X^\kappa.F' : \kappa \to \kappa'}$$

$$\textsc{EqTApp}\ \frac{\Delta \vdash F = F' : \kappa \to \kappa' \qquad \Delta \vdash G = G' : \kappa}{\Delta \vdash F\ G = F'\ G' : \kappa'}$$

$$\textsc{EqTTrans}\ \frac{\Delta \vdash F = F' : \kappa \qquad \Delta \vdash F' = F'' : \kappa}{\Delta \vdash F = F'' : \kappa'}$$

Figure 2.7: Type constructor equality rules of System F_ω.

2.3.1 Encodings of datatypes in System F_ω

In System F_ω we can encode all the datatypes encodable in System F (see Section 2.2.1) and more. In addition to the obvious type constructors, one can encode indexed types, nested types, and even fixpoint operators over type constructors.

- *Type constructors* for polymorphic datatypes can be encoded using lambda types that abstract over types.
- *Non-regular datatypes*, or *nested datatypes*, can be encoded using forall types that are polymorphic over type constructors.
- With higher-kinded type constructors, we can even encode the recursive type operator μ in System F_ω by abstracting over non-star type constructors.

This additional expressive power comes from two different uses of type-level constructs other than types of kind $*$.

- *Higher-kinded polymorphism* is the ability to universally quantify over both type constructors of arbitrary kinds.
- *Type constructors of higher kinds* or *higher-kinded type constructors* are type constructors that expect type constructors as their arguments.

In fact, we combine these two to define a family of kind-indexed recursive type operators μ_κ using both higher-kinded type constructors and higher-kinded polymorphism.

Type constructors for polymorphic datatypes expect other types as arguments to produce a datatype. We can encode these type constructors in System F_ω. For example, the shorthand notations (or, type synonyms) in Section 2.2.1, such as $(\times)$ for pair types and $(+)$ for sum types, can be encoded as as follows:[6]

$$(\times) = \lambda X_1^*.\lambda X_2^*.(X_1 \to X_2 \to X) \to X \qquad : * \to * \to *$$
$$(+) = \lambda X_1^*.\lambda X_2^*.(X_1 \to X) \to (X_2 \to X) \to X \qquad : * \to * \to *$$

Type constructors for polymorphic recursive datatypes are encodable as well. For instance, we can encode the constructor *List* for the polymorphic list datatype:

$$\mathit{List} = \lambda X_a^*.\forall X^*.(X_a \to X \to X) \to X \to X \qquad : * \to *$$

In System F, type constructors, such as $(\times)$, $(+)$, and *List*, are meta-level concepts (or, shorthand notations, macros) that cannot be encoded within the type system of System F. In System F_ω, these datatype constructors are encodable as type constructors, which are ordinary constructs of System F_ω.

Higher-kinded datatype constructors that expect not only types but also type constructors of arbitrary kinds as arguments are encodable in System F_ω as well. For example, we can encode *Flip*, which flips the order of the first and second arguments of a binary type constructor (i.e., $(\mathit{Flip}\ F)\ A_1\ A_2 = F\ A_2\ A_1$), and *Comp*, which composes two unary type constructors (i.e., $(\mathit{Comp}\ F_1\ F_2)\ A = F_1\ (F_2\ A)$),

[6]Here, we used a Haskell-ish notation of turning a infix binary operator into a prefix binary operator by surrounding the operator in parenthesis (e.g., $(+)\ X_1\ X_2 = X_1 + X_2$). I also annotated the kinds of the type constructors after the colon (:).

as follows:

$$Flip = \lambda X_f^{*\to *\to *}.\lambda X_1^*.\lambda X_2^*.X_f\, X_2\, X_1 \quad : (* \to * \to *) \to * \to * \to *$$

$$Compose = \lambda X_f^{*\to *}.\lambda X_g^{*\to *}.\lambda X^*.X_f\,(X_g\, X) \quad : (* \to *) \to (* \to *) \to * \to *$$

Higher-kinded polymorphism is the ability to universally quantify over type constructors as well as types. That is, we can have $\forall X^{\kappa}.B$ where κ is not kind $*$. We can encode *non-regular (recursive) datatypes* in System F_ω using higher-kinded polymorphism.

We mentioned that we can encode *regular (recursive) datatypes* in Systerm F (Section 2.2.1), but have not discussed what regular datatypes are. A representative example of a regular datatype is the polymorphic list type $(\forall X_a.List\, X_a)$. We say that the polymorphic list type is regular since its recursive component, the tail, has exactly the same type. That is, for any non-empty list of type $List\, A$, its tail must be of type $List\, A$. Many other well-known recursive datatypes are also regular (e.g., binary trees).

But, one can imagine a non-regular twist to the regular polymorphic list type by insisting the recursive components (i.e., tails) have different type arguments from the list they are part of. For instance, we may insist that a list-like datatype of type $(Powl\, A)$ must have its tail be of type $(Powl\,(A \times A))$. That is, if the first element is an integer (e.g., 1), then the second element must be a pair of integers (e.g., $(2, 3)$), and the third elment must be a pair of pair of integers (e.g., $((4, 5), (6, 7))$), and so on. We can depict an example of this list-like datatype with three elements as: $[1, (2, 3), ((4, 5), (6, 7))]$. This is a representative example of a non-regular datatype called powerlists. Such datatypes are also called *nested datatypes* since the type constructor is applied to ever-increasing complex arguments (here they are nested, but one can imagine even richer kinds of complexity) as we step further inside the recursive components.

We can encode the type constructor *Powl* for powerlists using higher-kinded

polymorphism of System F_ω, as follows (cf. encoding of *List*):

$$\begin{array}{rlllll} \mathit{Powl} = \lambda X_a^*.\forall X^{*\to *} & .(X_a \to X(X_a \times X_a) & \to X\, X_a) & \to X\, X_a & \to X\, X_a \\ \mathit{List} = \lambda X_a^*.\forall X^* & .(X_a \to X & \to X) & \to X & \to X \end{array}$$

Unlike the encoding of *List*, where X is polymorphic over types of kind $*$, the universally quantified variable X in the encoding of *Powl* is polymorphic over constructors of kind $* \to *$. Intuitively, X in the list encoding corresponds to $\mathit{List}\, X_a$ (i.e., the type constructor *List* applied to its uniform argument X_a), and, X in the powerlist encoding corresponds to *Powl* without being applied to its argument so that it may be applied to a non-regular argument (e.g., $X(X_a \times X_a)$). See Section 3.7 for more examples and discussions on non-regular datatypes.

The recursive type operator μ builds a recursive type (μF) from a non-recursive base structure ($F : * \to *$). Theories on recursive datatypes are often formulated in terms of the recursive type operator μ, which satisfies the property that $\mu F = F(\mu F)$ for any $F : * \to *$. A recursive datatype (μF) is built from its base structure (F) by applying the recursive operator. For example, the natural number datatype can be built from the base structure $F = \lambda X_r^*.X_r + \mathit{Unit}$. Intuitively, we can understand this base structure as a specification for natural numbers: a natural number is either a successor of a recursive object (X_r) or zero encoded as the unit object (*Unit*). From this base structure, we can define $\mathit{Nat} = \mu(\lambda X_r^*.X_r + \mathit{Unit})$. Let us write down the desired property of μ for *Nat*.

$$\begin{aligned} \mu(\lambda X_r^*.X_r + \mathit{Unit}) &= (\lambda X_r^*.X_r + \mathit{Unit})(\mu(\lambda X_r^*.X_r + \mathit{Unit})) \\ \mathit{Nat} &= (\lambda X_r^*.X_r + \mathit{Unit})\, \mathit{Nat} \\ \mathit{Nat} &= \mathit{Nat} + \mathit{Unit} \end{aligned}$$

The simplified last equation looks very similar to the recursive datatype definitions for unary natural numbers in functional languages, such as Haskell:

$$\textbf{data}\ \mathit{Nat} = \texttt{Succ}\ \mathit{Nat} \mid \texttt{Zero}$$

See Chapter 3 for more Haskell examples on recursive datatypes and μ.

Although recursive datatypes are encodable in System F (Section 2.2.1), extensions of System F with μ have been studied since one can reason about the properties of recursive datatypes more uniformly by factoring out the recursion at the type-level as the fixpoint μ. In System F_ω, we can encode μ using higher-kinded type constructors and higher-kinded polymorphism as follows:

$$\mu = \lambda X_f^{*\to *}.\forall X'^{*}.(\forall X_r^{*}.(X_r \to X') \to X_f\, X_r \to X') \to X' \;:\; (* \to *) \to *$$

Let us intuitively derive above the encoding of μ starting from the impredicative encoding of natural numbers:

$$\begin{aligned} Nat &= \forall X^{*}.(X \to X) \to X \to X \\ &\cong \forall X^{*}.(X \to X) \to (\mathit{Unit} \to X) \to X \qquad (\because \mathit{Unit} \to X \cong X) \\ &\cong \mu(\lambda X_r^{*}.\, X_r + \mathit{Unit}) \qquad \text{(to show)} \end{aligned}$$

We want to show that the impredicative encoding of natural numbers is equivalent to the natural number type defined using μ. We need to turn the impredicative encoding of natural numbers into a non-recursive base structure by abstract away the recursive component, which is the underlined part below. That is, we replace the underlined X with a new variable X_r:

$$\forall X^{*}.(\underline{X} \to X) \to (\mathit{Unit} \to X) \to X$$
$$\forall X^{*}.(X_r \to X) \to (\mathit{Unit} \to X) \to X$$

Recall that $X_r + \mathit{Unit} = \forall X^{*}.(X_r \to X) \to (\mathit{Unit} \to X) \to X$. Also, recall that the idea behind the impredicative encoding is that we can eliminate an object of the datatype into an arbitrary result type X. If we are to encode datatypes constructed by μ, we apply this idea of impredicative encoding in two layers: for the base structure and for μ. We already know how to encode the base structure; with the encoding above, we can eliminate in order to obtain an arbitrary result

type X. For μ, we introduce yet another variable X' so that we can eliminate in order to obtain an arbitrary result type X'. Thus, the encoding for the natural number type constructed using μ would be of the following form:

$$\forall X'^{*}.(\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots (X_r + \mathit{Unit}) \to X') \to X'$$

Since the recursive type contains the base structure, we would be able to eliminate the recursive type, given that we know how to eliminate the base structure ($(X_r + \mathit{Unit}) \to X'$). However, this is not yet complete because we do not know how to eliminate X_r. So, we require that we should also know how to eliminate X_r, as follows:

$$\forall X'^{*}.(\forall X_r^{*}.(X_r \to X') \to (X_r + \mathit{Unit}) \to X') \to X'$$

We can derive the encoding for μ (repeated below) so that $\mu(\lambda X_r^{*}.X_r + \mathit{Unit})$ is equivalent to above.

$$\mu = \lambda X_f^{*\to *}.\forall X'^{*}.(\forall X_r^{*}.(X_r \to X') \to X_f\, X_r \to X') \to X' \ :\ (* \to *) \to *$$

Note that X_r is also universally quantified in $(\forall X_r^{*}.(X_r \to X') \to X_f\, X_r \to X')$ locally. See Chapter 3 for an intuitive explanation for why X_r should be universally quantified.

The (data) constructor for the recursive type operator μ is called In and the eliminator is called **mit**. The encodings of In and **mit** as Curry-style terms are as follows:

$$\mathsf{In} = \lambda x_r.\lambda x_\varphi.x_\varphi\,(\mathbf{mit}\ x_\varphi)\,x_r \qquad\qquad \mathbf{mit} = \lambda x_\varphi.\lambda x_r.x_r\,x_\varphi$$

These (μ, In, and **mit**) are, in fact, encodings for Mendler-style iteration, which will be discussed in Section 3.10.

A kind-indexed family of recursive type operators μ_κ: The recursive type operator $\mu : (* \to *) \to *$ discussed so far can only construct (non-mutually

recursive) regular datatypes. For example,

$$Nat = \mu(\lambda X^*.X + Unit)$$
$$List = \lambda X_a^*.\mu(\lambda X^*.(X_a \times X) + Unit)$$

More generally, there is a family of recursive type operators $\mu_\kappa : (\kappa \to \kappa) \to \kappa$ for each kind κ. The μ, which we discussed above, is $\mu_* : (* \to *) \to *$. We can construct *Powl*, which is a non-regular datatype, using another recursive typer operator $\mu_{*\to*} : ((* \to *) \to (* \to *)) \to (* \to *)$ as follows (cf. *List*).

$$Powl = \mu_{*\to*}(\lambda X^{*\to*}.\lambda X_a^*.(X_a \times X(X_a \times X_a)) + Unit)$$
$$List = \lambda X_a^*.\mu_*(\lambda X^*.(X_a \times X) + Unit)$$

Note the difference on where X_a is bound in the definitions of *Powl* and *List*. The encodings of μ_* and $\mu_{*\to*}$ in System F_ω are shown below:

$$\mu_* = \lambda X_f^{*\to*}.\forall X'^*.(\forall X_r^*.(X_r \to X') \to (X_f\, X_r \to X')) \to X'$$
$$\mu_{*\to*} = \lambda X_f^{(*\to*)\to(*\to*)}.\lambda X_a^*.$$
$$\forall X'^{*\to*}.\Big(\forall X_r^{*\to*}.(\forall X_a^*.X_r\, X_a \to X'\, X_a) \to$$
$$(\forall X_a^*.X_f\, X_r\, X_a \to X'\, X_a)\Big) \to X'\, X_a$$

The general form for the encoding of μ_κ is as follows:

$$\mu_\kappa = \lambda X_f^{\kappa\to\kappa}.\lambda \vec{X}^{\vec{\kappa}}.\forall X'^{*\to*}.\Big(\forall X_r^{\kappa\to\kappa}.(\forall \vec{X}^{\vec{\kappa}}.X_r\, \vec{X} \to X'\, \vec{X}) \to$$
$$(\forall \vec{X}^{\vec{\kappa}}.X_f\, X_r\, \vec{X} \to X'\, \vec{X})\Big) \to X'\, \vec{X}$$

where $\vec{X}$ denotes a sequence of n variables such that $n = 0$ when $\kappa = *$; otherwise, $n = |\vec{\kappa}|$ when $\kappa = \vec{\kappa} \to * = \kappa_1 \to \cdots \to \kappa_n \to *$.[7] That is, we can simply erase all the $\lambda \vec{X}^{\vec{\kappa}}$, $\forall \vec{X}^{\vec{\kappa}}$, and $\vec{X}$ from above when $\kappa = *$; otherwise, $\lambda \vec{X}^{\vec{\kappa}}$ stands for $\lambda X_1^{\kappa_1}.\cdots.\lambda X_n^{\kappa_n}$, $\forall \vec{X}^{\vec{\kappa}}$ stands for $\forall X_1^{\kappa_1}.\cdots.\forall X_n^{\kappa_n}$, and $F\, \vec{X}$ stands for $F\, X_1 \cdots X_n$ when $\kappa = \vec{\kappa} \to * = \kappa_1 \to \cdots \to \kappa_n \to *$.

[7] κ always end up with $*$ when it is an arrow kind since $\to$ is right associative by convention.

The (data) constructor for the recursive type operator μ_κ is called In_κ and the eliminator is called $\mathbf{mit}_\kappa$. The encodings of In_κ and $\mathbf{mit}_\kappa$ as Curry-style terms are exactly the same as for In and $\mathbf{mit}$ for the star kind:

$$\mathsf{In}_\kappa = \lambda x_r.\lambda x_\varphi.x_\varphi\,(\mathbf{mit}\ x_\varphi)\,x_r \qquad\qquad \mathbf{mit}_\kappa = \lambda x_\varphi.\lambda x_r.x_r\,x_\varphi$$

These (μ_κ, In_κ, and $\mathbf{mit}_\kappa$) are, in fact, encodings for Mendler-style iteration in F_ω, which will be discussed in Section 4.2.

2.3.2 Strong normalization

Here, we will take the *subject reduction* (Theorem 2.3.1) (a.k.a. *type preservation*) for granted,[8] and focus our discussion on the *strong normalization* of System F_ω.

Theorem 2.3.1 (subject reduction)**.** $\dfrac{\Delta;\Gamma \vdash t : A \qquad t \longrightarrow t'}{\Delta;\Gamma \vdash t' : A}$

To prove strong normalization of System F, we use the same proof strategy as in the proof of strong normalization of System F (Section 2.2.2). That is, we interpret types as saturated sets of normalizing terms as we did for System F. However, we need to generalize the interpretation of types to the interpretation of type constructors.

In the strong normalization proof of System F, we had a complete lattice $(\mathsf{SAT}, \subseteq)$. We generalize from $(\mathsf{SAT}, \subseteq)$, which is for kind $*$ only, to $(\mathsf{SAT}_\kappa, \sqsubseteq_\kappa)$ for an arbitrary kind κ, as follows:

- The set SAT_κ is a generalization of SAT such that

$$\mathsf{SAT}_* = \mathsf{SAT}$$
$$\mathsf{SAT}_{\kappa\to\kappa'} = \mathsf{SAT}_\kappa \to \mathsf{SAT}_{\kappa'} \qquad \text{(i.e., functions from } \mathsf{SAT}_\kappa \text{ to } \mathsf{SAT}'_\kappa\text{)}.$$

[8] The proof for subject reduction of System F_ω is similar to the proof for the subject reduction of System F.

Interpretation of kinds as pointwise generalization of SAT

$$[\![\kappa]\!] = \mathsf{SAT}_\kappa$$

Interpretation of type constructors as function spaces over saturated sets of normalizing terms whose free type variables are substituted according to the given type constructor valuation (ξ):

$$\begin{aligned}
[\![X]\!]_\xi &= \xi(X) \\
[\![A \to B]\!]_\xi &= [\![A]\!]_\xi \to [\![B]\!]_\xi \\
[\![\forall X^\kappa.B]\!]_\xi &= \bigcap_{\mathcal{F} \in [\![\kappa]\!]} [\![B]\!]_{\xi[X \mapsto \mathcal{F}]} && (X \notin \mathrm{dom}(\xi)) \\
[\![\lambda X^\kappa.F]\!]_\xi &= \boldsymbol{\lambda}(\mathcal{G} \in [\![\kappa]\!]).[\![F]\!]_{\xi[X \mapsto \mathcal{G}]} && (X \notin \mathrm{dom}(\xi)) \\
[\![F\ G]\!]_\xi &= [\![F]\!]_\xi([\![G]\!]_\xi)
\end{aligned}$$

Interpretation of kinding and typing contexts as sets of type constructor valuations and term valuations (ξ and ρ):

$$\begin{aligned}
[\![\Delta]\!] &= \{\xi \in \mathrm{dom}(\Delta) \to \bigcup_\kappa [\![\kappa]\!] \mid \xi(x) \in [\![\Delta(x)]\!] \text{ for all } x \in \mathrm{dom}(\Delta)\} \\
[\![\Delta; \Gamma]\!] &= \{\xi; \rho \mid \xi \in [\![\Delta]\!], \rho \in [\![\Gamma]\!]_\xi\} \\
[\![\Gamma]\!]_\xi &= \{\rho \in \mathrm{dom}(\Gamma) \to \mathsf{SN} \mid \rho(x) = [\![\Gamma(x)]\!]_\xi \text{ for all } x \in \mathrm{dom}(\Gamma)\}
\end{aligned}$$

Interpretation of terms as terms themselves whose free variables are substituted according to the given pair of type constructor and term valuations (ξ;ρ):

$$\begin{aligned}
[\![x]\!]_{\xi;\rho} &= \rho(x) \\
[\![\lambda x.t]\!]_{\xi;\rho} &= \lambda x.[\![t]\!]_{\xi;\rho} && (x \notin \mathrm{dom}(\rho)) \\
[\![t\ s]\!]_{\xi;\rho} &= [\![t]\!]_{\xi;\rho}\ [\![s]\!]_{\xi;\rho}
\end{aligned}$$

Figure 2.8: Interpretation of type constructors, kinding and typing contexts, and terms of System F_ω for the proof of strong normalization.

- The relation $\sqsubseteq_\kappa$ is a pointwise generalization of $\subseteq$ such that

$$\mathcal{A} \sqsubseteq_* \mathcal{A}' = \mathcal{A} \subseteq \mathcal{A}'$$
$$\mathcal{F} \sqsubseteq_{\kappa\to\kappa'} \mathcal{F}' = \mathcal{F}(\mathcal{G}) \sqsubseteq_{\kappa'} \mathcal{F}'(\mathcal{G}) \text{ for all } \mathcal{G} \in \mathsf{SAT}_\kappa$$

It is easy to see that $(\mathsf{SAT}_\kappa, \sqsubseteq_\kappa)$ forms a complete lattice by induction on κ. For kind $\star$, this is obvious since we already know that $(\mathsf{SAT}, \subseteq)$ forms a complete lattice. For an arrow kind $\kappa \to \kappa'$, we know that $(\mathsf{SAT}_{\kappa'}, \sqsubseteq_\kappa)$ forms a complete lattice by induction. It is easy to see that for every two element $\mathcal{F}_1, \mathcal{F}_2 \in \mathsf{SAT}\kappa', \sqsubseteq_\kappa$ there exist a greatest lower bound $(\mathcal{F}_1 \wedge \mathcal{F}_2)$ and a greatest upper bound $(\mathcal{F}_1 \vee \mathcal{F}_2)$, defined pointwisely as follows:

$$(\mathcal{F}_1 \wedge \mathcal{F}_2)(\mathcal{G}) = \mathcal{F}_1(\mathcal{G}) \wedge \mathcal{F}_2(\mathcal{G}) \text{ for all } \mathcal{G} \in \mathsf{SAT}_\kappa$$
$$(\mathcal{F}_1 \vee \mathcal{F}_2)(\mathcal{G}) = \mathcal{F}_1(\mathcal{G}) \vee \mathcal{F}_2(\mathcal{G}) \text{ for all } \mathcal{G} \in \mathsf{SAT}_\kappa$$

The top and bottom elements of an arrow kind $\bot_{\kappa\to\kappa'}$ are also defined pointwisely. Let $\bot_{\kappa\to\kappa'}$ be the constant function that always returns $\bot_{\kappa'}$ (the bottom element at κ', and, let $\top_{\kappa\to\kappa'}$ be the constant function that always returns $\top_{\kappa'}$ (the top element of the lattice at κ'). It is easy to see that $\bot_{\kappa\to\kappa'}$ and $\top_{\kappa\to\kappa'}$ are the bottom and top elements at kind $\kappa \to \kappa'$ by definition of $\sqsubseteq_{\kappa\to\kappa'}$.

Then, we can give an interpretation of kind κ as SAT_κ. That is, $[\![\kappa]\!] = \mathsf{SAT}_\kappa$. An interpretation of a type constructor of kind κ should be a member of $[\![\kappa]\!]$, i.e., SAT_κ. The interpretation of kinds, type constructors, contexts, and terms of System F_ω are illustrated in Figure 2.8.

We use the Curry-style System F_ω to present the strong normalization proof. It is more convenient to interpret terms in Curry style since the Curry-style terms syntax is simpler than the Church-style term syntax. It is more convenient to interpret type constructors in Curry style since the kind annotation makes it clear how to interpret the bound type variable X in forall types and lambda types (i.e., for X^κ choose from $[\![\kappa]\!]$).

The proof of strong normalization amounts to proving the following theorem:

Theorem 2.3.2 (soundness of typing)**.** $\dfrac{\Delta; \Gamma \vdash t : A \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![A]\!]_{\xi}}$

Proof. We prove by induction on the typing derivation ($\Delta; \Gamma \vdash t : A$).

The cases for Var, Abs, and App are pretty much the same as the strong normalization proof for System F. The cases for TyAbs and TyApp is almost the same as the strong normalization proof for System F, except that the type variable can be of some kind κ other than just the star kind. We need to consider one more rule Conv, which is new in System F_ω. Let us elaborate on the three cases of TyAbs and TyApp, and Conv.

Case (TyAbs) We need to show that $\dfrac{\Delta; \Gamma \vdash t : \forall X.B \qquad \xi; \rho \in [\![\Delta; \Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![\forall X^\kappa.B]\!]_{\xi}}$

By induction, we know that

$$\frac{\Delta, X^\kappa; \Gamma \vdash t : B \qquad \xi'; \rho' \in [\![\Delta, X; \Gamma]\!]}{[\![t]\!]_{\xi';\rho'} \in [\![B]\!]_{\xi'}} \ (X \notin \mathrm{FV}(\Gamma))$$

Since this holds for all $\xi'; \rho' \in [\![\Delta, X^\kappa; \Gamma]\!]$ where $X \notin \mathrm{FV}(\Gamma)$, it also holds for particular subset such that $\xi' = \xi[X \mapsto \mathcal{F}]$ and $\rho' = \rho$ for all $\mathcal{F} \in [\![\kappa]\!]$. That is,

$$[\![t]\!]_{\xi[X\mapsto\mathcal{F}];\rho} \in [\![B]\!]_{\xi[X\mapsto\mathcal{F}]} \quad \text{for all } \mathcal{F} \in [\![\kappa]\!]$$

From $X \notin \mathrm{FV}(\Gamma)$, we know that $[\![t]\!]_{\xi[X\mapsto\mathcal{F}];\rho} = [\![t]\!]_{\xi;\rho}$ because ρ is independent of that to which X maps to. So, we know that

$$[\![t]\!]_{\xi;\rho} \in [\![B]\!]_{\xi[X\mapsto\mathcal{F}]} \quad \text{for all } \mathcal{F} \in [\![\kappa]\!]$$

By set theoretic definition, this is exactly what we wanted to show:

$$[\![t]\!]_{\xi;\rho} \in \bigcap_{\mathcal{F}\in[\![\kappa]\!]} [\![B]\!]_{\xi[X\mapsto\mathcal{F}]} = [\![\forall X^\kappa.B]\!]_{\xi}$$

Case (TyApp) We need to show that $\dfrac{\Delta;\Gamma \vdash t : B[G/X] \quad \xi;\rho \in [\![\Delta;\Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![B[G/X]]\!]_\xi}$.

By induction, we know that $\dfrac{\Delta;\Gamma \vdash t : \forall X^\kappa.B \quad \xi';\rho' \in [\![\Delta;\Gamma]\!]}{[\![t]\!]_{\xi';\rho'} \in [\![\forall X^\kappa.B]\!]_{\xi'}}$.

Since this holds for all $\xi';\rho' \in [\![\Delta,\Gamma]\!]$, it also holds for $\xi' = \xi$ and $\rho' = \rho$. Then, we are done: $[\![t]\!]_{\xi;\rho} \in [\![\forall X^\kappa.B]\!]_\xi = \bigcap_{\mathcal{G}\in[\![\kappa]\!]} [\![B]\!]_{\xi[X\mapsto\mathcal{G}]} \subseteq [\![B]\!]_{\xi[X\mapsto[\![G]\!]_\xi]} = [\![B[G/X]]\!]_\xi$.

Case (Conv) We need to show that $\dfrac{\Delta;\Gamma \vdash t : A' \quad \xi;\rho \in [\![\Delta;\Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![A']\!]_\xi}$

By induction we know that $\dfrac{\Delta;\Gamma \vdash t : A \quad \xi;\rho \in [\![\Delta;\Gamma]\!]}{[\![t]\!]_{\xi;\rho} \in [\![A]\!]_\xi}$

If we can show that $[\![A]\!]_\xi = [\![A']\!]_\xi$, we are done. To show that $[\![A]\!]_\xi = [\![A']\!]_\xi$, we use the soundness of type constructor equality lemma (Lemma 2.3.1). □

Corollary 2.3.1 (strong normalization)**.** $\dfrac{\Delta;\Gamma \vdash t : A}{t \in \mathsf{SN}}$

Lemma 2.3.1 (soundness of type equality)**.** $\dfrac{\Delta \vdash F = F' : \kappa \quad \xi \in [\![\Delta]\!]}{[\![F]\!]_\xi = [\![F']\!]_\xi}$

Proof. The only interesting case is the EqTBeta rule. The EqTVar is trivial and all other rules are handled by induction. Let us elaborate on the EqTBeta case.

Case (EqTBeta) We need to show that

$$\frac{\Delta \vdash (\lambda X^\kappa.F)\ G = F[G/X] : \kappa' \quad \xi \in [\![\Delta]\!]}{[\![(\lambda X^\kappa.F)\ G]\!]_\xi = [\![F[G/X]]\!]_\xi}$$

By applying the soundness of kinding lemma (Lemma 2.3.2) to the premises, we know that

$$\frac{\Delta, X^\kappa \vdash F : \kappa \to \kappa' \quad \xi' \in [\![\Delta, X^\kappa]\!]}{[\![F]\!]_{\xi'} \in [\![\kappa']\!]} \quad \text{and} \quad \frac{\Delta \vdash G : \kappa \quad \xi \in [\![\Delta]\!]}{[\![G]\!]_\xi \in [\![\kappa]\!]}$$

Since it should hold for arbitrary ξ', it should also hold for a particular ξ' such that $\xi' = \xi[X \mapsto \mathcal{G}]$ for any $\mathcal{G} \in [\![\kappa]\!]$. Therefore, we can rewrite the left-hand side of the conclusion, which is what we wanted to show, into the right-hand side as

follows:

$$\begin{aligned}
[\![(\lambda X^{\kappa}.F)\ G]\!]_{\xi} &= [\![(\lambda X^{\kappa}.F)]\!]_{\xi}([\![G]\!]_{\xi}) \\
&= (\boldsymbol{\lambda}(\mathcal{G} \in [\![\kappa]\!]).[\![F]\!]_{\xi[X \mapsto \mathcal{G}]})([\![G]\!]_{\xi}) \\
&= [\![F]\!]_{\xi[X \mapsto [\![G]\!]_{\xi}]} \\
&= [\![F[G/X]]\!]_{\xi}
\end{aligned}$$

□

System F_{ω} has a richer kind structure than System F, which has kind $(*)$ only. So, the interpretation of type constructors would only be well-defined when the type constructors are well-kinded. For example, the interpretation of a type constructor application $[\![F\ G]\!]_{\xi}$ would only make sense when $[\![F]\!]_{\xi} \in [\![\kappa \to \kappa']\!]$ and $[\![G]\!]_{\xi} \in [\![\kappa]\!]$ for some κ and κ'. The soundness of kinding lemma below states the property that well-kinded type constructors indeed have a well-defined interpretation.

Lemma 2.3.2 (soundness of kinding)**.** $\dfrac{\Delta \vdash F : \kappa \qquad \xi \in [\![\Delta]\!]}{[\![F]\!]_{\xi} \in [\![\kappa]\!]}$

Proof. We prove by induction on the kinding judgment.

Case (TVAR) Straightforward by definition of $[\![\Delta]\!]$.

$[\![X]\!]_{\xi} = \xi(X) \in [\![\kappa]\!]$ since $\xi(X) \in [\![\kappa]\!]$ for any $\xi \in [\![\Delta]\!]$ when $X^{\kappa} \in [\![\Delta]\!]$.

Case (TARR) By induction, straightforward.

Case (TALL) We need to show that $\dfrac{\Delta \vdash \forall X^{\kappa}.B : * \qquad \xi \in [\![\Delta]\!]}{[\![\forall X^{\kappa}.B]\!]_{\xi} \in [\![*]\!]}$.

By induction, we know that $\dfrac{\Delta, X^{\kappa} \vdash B : * \qquad \xi' \in [\![\Delta, X^{\kappa}]\!]}{[\![B]\!]_{\xi'} \in [\![*]\!]}$.

Since it should hold for any ξ', it also holds for $\xi' = \xi[X \mapsto \mathcal{G}]$ for any $\mathcal{G} \in [\![\kappa]\!]$.

Therefore, $[\![\forall X^{\kappa}.B]\!]_{\xi} = \bigcap_{\mathcal{G} \in [\![\kappa]\!]} [\![B]\!]_{\xi[X \mapsto \mathcal{G}]} \in [\![*]\!]$.

Case (TLAM) We need to show that $\dfrac{\Delta \vdash \lambda X^{\kappa}.B : * \qquad \xi \in [\![\Delta]\!]}{[\![\forall X.B]\!]_{\xi} \in [\![*]\!]}$.

By induction, we know that $\dfrac{\Delta, X^{\kappa} \vdash F : \kappa' \qquad \xi' \in [\![\Delta, X^{\kappa}]\!]}{[\![F]\!]_{\xi'} \in [\![\kappa']\!]}$.

Since it should hold for any ξ, it also holds for $\xi' = \xi[X \mapsto \mathcal{G}]$ for any $\mathcal{G} \in [\![\kappa]\!]$.

Therefore, $[\![\lambda X^{\kappa}.F]\!]_{\xi} = \boldsymbol{\lambda}(\mathcal{G} \in [\![\kappa]\!]).[\![F]\!]_{\xi[X \mapsto \mathcal{G}]} \in [\![\kappa \to \kappa']\!]$.

Case (TAPP) By induction, straightforward. □

2.4 THE HINDLEY–MILNER TYPE SYSTEM

Hindley [48] demonstrated the existence of a unique principal type scheme for every object in a combinatory logic. Milner [66] rediscovered this fact in the setting of a polymorphic lambda calculus. He was devising an algorithm, called algorithm W, which infers a most general type scheme (a.k.a. principal type scheme) for a Curry-style lambda term. Damas [28] (a student of Milner) published detailed theories about Milner's polymorphic lambda calculus and the type inference algorithm W. This type system for Milner's polymorphic lambda calculus [28, 29, 66] is also known as the Hindley–Milner type system (HM), the Damas–Hindley–Milner type system (DHM), or let-polymorphic type system.

The syntax of Milner's polymorphic lambda calculus and its typing rules are illustrated in Figure 2.9. The type inference algorithm W is illustrated in Figure 2.10. We discuss each of these figures separately — the syntax in Section 2.4.1, the typing rules in Section 2.4.2 and Section 2.4.3, and the inference algorithm in Section 2.4.4. We provide two equivalent sets of typing rules (we prove this in Section 2.4.3). The declarative typing rules (Section 2.4.2) are suited for reasoning about the soundness of typing. The syntax-directed typing rules (Section 2.4.3) are suited for reasoning about the properties of the type inference algorithm W (Section 2.4.4).

Term	$t, s \ ::= \ x \mid \lambda x.t \mid t\ s \mid \mathbf{let}\ x = s\ \mathbf{in}\ t$
Type (or, monotype)	$A, B \ ::= \ A \to B \mid \iota \mid X$
Type scheme (or, polytype)	$\sigma \ ::= \ \forall X.\sigma \mid A$
Typing context	$\Gamma \ ::= \ \cdot \mid \Gamma, x : \sigma \quad (x \notin \mathrm{dom}(\Gamma))$

Type scheme ordering (or, generic instantiation) $\boxed{\sigma \sqsubseteq \sigma'}$

$$\textsc{GenInst}\ \frac{X'_1, \ldots, X'_m \notin \mathrm{FV}(\forall X_1 \ldots X_n.A)}{\forall X_1 \ldots X_n.A \ \sqsubseteq \ \forall X'_1 \ldots X'_m.\, A[B_1/X_1] \cdots [B_n/X_n]}$$

Declarative typing rules

$\boxed{\Gamma \vdash t : \sigma}$

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B} \qquad \textsc{Let}\ \frac{\Gamma \vdash s : \sigma \quad \Gamma, x : \sigma \vdash t : B}{\Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : B}$$

$$\textsc{Inst}\ \frac{\Gamma \vdash t : \sigma \quad \sigma \sqsubseteq \sigma'}{\Gamma \vdash t : \sigma'} \qquad \textsc{Gen}\ \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall X.\sigma}\ (X \notin \mathrm{FV}(\Gamma))$$

Syntax-directed typing rules

$\boxed{\Gamma \vdash^{\mathsf{s}} t : A}$

$$\textsc{Var}_s\ \frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq A}{\Gamma \vdash^{\mathsf{s}} x : A} \qquad \textsc{Abs}_s\ \frac{\Gamma, x : A \vdash^{\mathsf{s}} t : B}{\Gamma \vdash^{\mathsf{s}} \lambda x.t : A \to B}$$

$$\textsc{App}_s\ \frac{\Gamma \vdash^{\mathsf{s}} t : A \to B \quad \Gamma \vdash^{\mathsf{s}} s : A}{\Gamma \vdash^{\mathsf{s}} t\ s : B} \qquad \textsc{Let}_s\ \frac{\Gamma \vdash^{\mathsf{s}} s : A \quad \Gamma, x : \overline{\Gamma}(A) \vdash^{\mathsf{s}} t : B}{\Gamma \vdash^{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : B}$$

$\overline{\Gamma}(A) = \forall \vec{X}.A$ where $\vec{X} = \mathrm{FV}(A) \setminus \mathrm{FV}(\Gamma)$

Figure 2.9: Milner's polymorphic lambda calculus.

2.4.1 Syntax

The syntax of terms includes the usual Curry-style terms (x, $\lambda x.t$, and $t\ s$) and let-terms ($\mathbf{let}\ x = s\ \mathbf{in}\ t$). A let term, $\mathbf{let}\ x = s\ \mathbf{in}\ t$, is semantically equivalent to $(\lambda x.t)\, s$. That is, $\mathbf{let}\ x = s\ \mathbf{in}\ t$ is a syntactic sugar for $(\lambda x.t)\, s$ when we think about reduction.[9] However, a let-term ($\mathbf{let}\ x = s\ \mathbf{in}\ t$) is assigned a significantly different type than its semantic equivalent ($(\lambda x.t)\, s$). The typing rules support the introduction of a polymorphic type scheme for x into the typing context when typing the let-term's body t. We will discuss many further details of typing let-terms (the Let rule) when we explain the typing rules.

The syntax of types (or, monotypes) includes all the types in the STLC ($A \to B$ and ι) and type variables (X). The syntax of type schemes (or, polytypes) are similar to the polymorphic types of System F, but universal quantification must appear only at the top level. Syntactically, type schemes are either universal quantifications over other type schemes ($\forall X.\sigma$) or (mono)types (A). Typing contexts (Γ) keep track of each term variable and its associated type scheme ($x : \sigma$).

The ordering between two type schemes $\sigma \sqsubseteq \sigma'$, defined in Figure 2.9, means that σ is more general than or equivalent to σ'. The ordering relation $\sqsubseteq$ comes from Damas and Milner [29], and is also known as generic instantiation — σ' is called a generic instance of σ when $\sigma \sqsubseteq \sigma'$. The shorthand notation $\forall X_1 \ldots X_n.A$ stands for consecutive universal quantification of n variables. For instance, $\forall X_1\, X_2\, X_3.A$ is a shorthand for $\forall X_1.\forall X_2.\forall X_3.A$.

Two type schemes σ and σ' are equivalent when $\sigma \sqsubseteq \sigma'$ and $\sigma' \sqsubseteq \sigma$. This coincides with α-equivalence (e.g., $\forall X.X \to X$ is equivalent to $\forall X'.X' \to X'$). In fact, we can derive α-equivalence as a special case of the type scheme ordering rule GenInst (Figure 2.9), where $n = m$ and $B_i = X'_i$ for each i from 1 to n.

[9] The reduction rules for the terms of HM are exactly the same as the reduction rules for Curry-style terms in the previous sections, once we desugar all the let terms.

The usual instantiation (i.e., substitution of quantified variables with monotypes) is a special case of generic instantiation. For example, consider the instantiations of $\forall X_1\, X_2.X_1 \to X_2$ below:

$$(\forall X_1\, X_2.X_1 \to X_2) \sqsubseteq (\forall X_2.\iota \to X_2) \sqsubseteq (\iota \to \iota)$$

Here, we instantiate the quantified X_1 with ι, and then, instantiate the quantified X_2 with ι. In such cases of $\sigma \sqsubseteq \sigma'$, we can call σ' an instance (as well as a generic instance) of σ. For example, (1) $\iota \to \iota$ is an instance of $\forall X_2.\iota \to X_2$ and an instance of $\forall X_1\, X_2.X \to X_2$; and (2) both $\iota \to \iota$ and $\forall X_2.\iota \to X_2$ are instances of $\forall X_1\, X_2.X_1 \to X_2$.

The relation $\sqsubseteq$ is more than α-equivalence and instantiation, since the type scheme ordering rule allows quantifying newly introduced variables in σ', which do not appear free in σ. For example, consider the two generic instances of $\forall X.X \to X$ below:

$$\forall X.X \to X \sqsubseteq (X' \to X') \to (X' \to X')$$
$$\forall X.X \to X \sqsubseteq \forall X'.(X' \to X') \to (X' \to X')$$

The former, $(X' \to X') \to (X' \to X')$, is an instance of $\forall X.X \to X$ instantiating X to $(X' \to X')$. However, the latter, $\forall X'.(X' \to X') \to (X' \to X')$, is not an instance but a generic instance of $\forall X.X \to X$ because the newly introduced variable X' is universally quantified.

There is a difference between the (mono)type $(X' \to X') \to (X' \to X')$, where X' is free, and the type scheme $\forall X'.(X' \to X') \to (X' \to X')$, where X' is universally quantified. A function of the monomorphic type $(X' \to X') \to (X' \to X')$ can only be applied to functions of the same type in one program, but a function of the polymorphic type scheme $\forall X'.(X' \to X') \to (X' \to X')$ can be applied to functions of many different types in one program. For example, consider a typing

context Γ with four term-variables such that[10]

$$\mathit{square} : \quad \texttt{int} \to \texttt{int} \quad \in \Gamma \qquad Id_{\to}^{mon} : \quad (X' \to X') \to (X' \to X') \quad \in \Gamma$$

$$\mathit{revstr} : \texttt{string} \to \texttt{string} \in \Gamma \qquad Id_{\to}^{poly} : \forall X'.(X' \to X') \to (X' \to X') \in \Gamma$$

The four function names, with the types assigned as above, are available in the context. Under this typing context Γ, it is possible to apply $Id_{\to}^{mon}$, the monomorphic identity function over endofunctions, to either *square* or *revstr* (as we do below), as long as we do not try to apply $Id_{\to}^{mon}$ to both of them in the same program.[11] For example, we note the different types for each application, and consider the different type that $Id_{\to}^{mon}$ must have inside each term.

$$\Gamma \vdash (Id_{\to}^{mon}\ \mathit{square}) : \texttt{int} \to \texttt{int}$$

$$\Gamma \vdash (Id_{\to}^{mon}\ \mathit{revstr}) : \texttt{string} \to \texttt{string}$$

It is impossible to derive a type for a program that applies $Id_{\to}^{mon}$ to both *square* and *revstr* in one program, since there is no solution for the inconsistent equations $X' = \texttt{int}$ and $X' = \texttt{string}$.

$$\Gamma \vdash \ldots (Id_{\to}^{mon}\ \mathit{square}) \ldots$$
$$\ldots (Id_{\to}^{mon}\ \mathit{revstr}) \ldots \ : \text{ this is a type error}$$

On the other hand, we can apply $Id_{\to}^{poly}$, the polymorphic identity function over endofunctions, to both *square* and *revstr* in the same program, since the universally quantified type variable X' can be instantiated to many different types including `int` and `string`.

$$\Gamma \vdash (Id_{\to}^{poly}\ \mathit{square}) : \texttt{int} \to \texttt{int}$$

$$\Gamma \vdash (Id_{\to}^{poly}\ \mathit{revstr}) : \texttt{string} \to \texttt{string}$$

[10] For an intuitive explanation, we assume `int` and `string` to be existing ground types although our formal definition of HM in Figure 2.9 only has the void type ι as the ground type for simplicity.

[11] A program is just a term, but it sounds like a more practical example.

$\Gamma \vdash \ldots (Id_{\rightarrow}^{poly}\ square) \ldots$

$\ldots (Id_{\rightarrow}^{poly}\ revstr) \ldots$: this can be type correct

To discover why the first must be ill-typed, and the second can be well-typed we must look at the details of the typing rules.

2.4.2 Declarative typing rules

The declarative typing rules deduce a type scheme for a given term under a typing context ($\Gamma \vdash t : \sigma$). The type scheme (σ) deduced for the given term (t) under the typing context (Γ) may not be unique. For example,

$\cdot \vdash \lambda x.x : \iota \rightarrow \iota$

$\cdot \vdash \lambda x.x : X \rightarrow X$

$\cdot \vdash \lambda x.x : (X \rightarrow X) \rightarrow (X \rightarrow X)$

$\cdot \vdash \lambda x.x : \forall X.X \rightarrow X$

$\cdot \vdash \lambda x.x : \forall X.(X \rightarrow X) \rightarrow (X \rightarrow X)$

$\cdot \vdash \lambda x.x : \forall X_1 X_2.(X_1 \rightarrow X_2) \rightarrow (X_1 \rightarrow X_2)$

$\vdots$

This is expected since terms of HM are Curry style. Recall that the uniqueness of typing does not hold for lambda calculi with Curry-style terms.

The first three declarative rules – Var, Abs, and App – in Figure 2.9 are fairly standard. The Var rule deduces the type scheme of a variable according to the type scheme binding of the variable in the typing context. Note that the type schemes deduced by the rules Abs and App are restricted to the form of (mono)types[12] since the domain and range of function ($\rightarrow$) types are restricted to (mono)types.

[12]Recall that (mono)types are subset of type schemes.

The Let rule can introduce polymorphic type schemes into the typing context (we discuss more about this shortly, in the next page). The most interesting rules are the Inst rule and the Gen rule.

- **The Inst rule** deduces a generic instance (σ') of any type scheme (σ). The Inst rule is essential when variables with polymorphic type schemes appear in the rules Abs and App. For instance, when t is a variable with a polymorphic type scheme in Γ, we need to instantiate the type scheme into a type since Abs and App are restricted to deduce (mono)types. A typical usage (where $\Gamma = x' : \forall X'.X' \to X'$) of the Inst rule is illustrated below:

$$\textsc{Abs}\dfrac{\textsc{Inst}\dfrac{\textsc{Var}\dfrac{x' : \forall X'.X' \to X' \in \Gamma, x : X}{\Gamma, x : X \vdash x' : \forall X'.X' \to X'}}{\Gamma, x : X \vdash x' : X'' \to X''}}{\Gamma \vdash \lambda x.x' : X \to (X'' \to X'')}$$

- **The Gen rule** deduces a generalization (i.e., universal quantification) of a type scheme, as long as the quantified variable does not appear free in the typing context. The Gen is essential for the Let rule to be useful. For instance, consider that s is a function that may be polymorphic, such as the identity function $\lambda x'.x'$. We want to bind this function in a let term, $\textbf{let}\ x = \lambda x'.x'\ \textbf{in}\ t$, and use x as a polymorphic function in t (i.e., extend the typing context with $x : \forall X.X \to X$). However, the Abs rule can only deduce a function type without any universal quantification, such as $\Gamma \vdash \lambda x'.x' : X \to X$. Here, we can use the Gen rule to generalize $X \to X$ to $\forall X.X \to X$, provided that X does not appear free in the typing context Γ, as below:

$$\textsc{Let}\dfrac{\textsc{Gen}\dfrac{\textsc{Abs}\dfrac{\vdots}{\Gamma \vdash \lambda x'.x' : X \to X}}{\Gamma \vdash \lambda x'.x' : \forall X.X \to X} \qquad \Gamma, x : \forall X.X \to X \vdash t : B}{\Gamma \vdash \textbf{let}\ x = \lambda x'.x'\ \textbf{in}\ t : B}$$

The soundness of typing is obvious once we observe that HM is a restriction of the Curry-style System F (i.e., if $\Gamma \vdash t : \sigma$ in HM, then $\Delta; \Gamma \vdash t : \sigma$ in System F). The terms in HM are exactly the same as the terms in the Curry-style System F, if we consider the let-term as a syntactic sugar. Both types (or, monotypes) and type schemes (or, polytypes) in HM are restrictions of types in System F. The declarative typing rules (of Figure 2.9) are also a restriction of the typing rules in System F. The rules Var, Abs, App, and Gen in HM are virtually the same as their counterparts in System F.[13] Thus, we only need ensure that the Let rule and the Inst rule in HM are admissible in System F.

A single derivation step of Let in HM corresponds to two derivation steps in System F involving the App and Abs rules. Let us start from the Let rule in HM, quoted verbatim from Figure 2.9:

$$\textsc{Let}\frac{\Gamma \vdash s : \sigma \qquad \Gamma, x : \sigma \vdash t : B}{\Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : B}$$

Recall that a let-term, $\mathbf{let}\ x = s\ \mathbf{in}\ t$ is semantically equivalent to $(\lambda x.t)\ s$. We first desugar the let-term into an application of an abstraction $(\lambda x.t)$ to the body of the local definition (s). Then, we can simply apply the App rule and the Abs rule in System F, as below:

$$\textsc{App}\frac{\textsc{Abs}\ \dfrac{\Delta \vdash \sigma \qquad \Delta; \Gamma, x : \sigma \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B} \qquad \Delta; \Gamma \vdash s : \sigma}{\Delta; \Gamma \vdash (\lambda x.t)\ s}$$

A singe derivation step of Inst in HM corresponds to multiple derivation steps in System F involving the rules TyAbs and TyApp rules. Since the Inst rule refers to the generic instantiation relation $\sqsubseteq$, the shape of σ and σ' in the Inst rule must match the left- and right-hand sides of $\sqsubseteq$ in the generic instantiation rule, as below:

[13] The names of corresponding rules in HM and System F are the same (Var, Abs, App), except for the Gen rule. The Gen rule in HM corresponds to the TyAbs rule in System F.

$$\textsc{Inst}\frac{\Gamma \vdash t : \sigma \qquad \sigma \sqsubseteq \sigma'}{\Gamma \vdash t : \sigma'} \quad \text{where} \quad \begin{array}{rl} \sigma = & \forall X_1 \ldots X_n.A \\ \sigma' = & \forall X'_1 \ldots X'_m.\, A[B_1/X_1] \cdots [B_n/X_n] \end{array}$$

$$\text{such that} \quad \frac{X'_1, \ldots, X'_m \notin \mathrm{FV}(\forall X_1 \ldots X_n.A)}{\forall X_1 \ldots X_n.A \;\sqsubseteq\; \forall X'_1 \ldots X'_m.\, A[B_1/X_1] \cdots [B_n/X_n]}$$

The generic instantiation from σ to σ' in the Inst rule above can be understood as having two phases: instantiation from σ to $A[B_1/X_1] \cdots [B_n/X_n]$ and generalization from $A[B_1/X_1] \cdots [B_n/X_n]$ to σ'. The instantiation phase, from $\forall.X_1 \ldots X_n.A$ to $A[B_1/X_1] \cdots [B_n/X_n]$, can be broken down to n small steps of instantiation — each step instantiates one of the quantified variables $(X_1 \ldots X_n)$. The generalization phase, from $A[B_1/X_1] \cdots [B_n/X_n]$ to $\forall X'_1 \ldots X'_m.A[B_1/X_1] \cdots [B_n/X_n]$, can be broken down into m small steps of generalization — each step universally quantifies one of the newly introduced variables $(X'_1 \ldots X_m)$ from the instantiation phase. The Curry-style System F has rules, which correspond exactly to these small steps (see Figure 2.3 in Section 2.2). The TyApp rule captures the small steps in the instantiation phase. The TyAbs rule captures the small steps in the generalization phase. Therefore, we can translate the Inst rule in HM into consecutive applications of the TyApp rule followed by consecutive applications of the TyAbs rule in System F, as below:

$$\textsc{TyApp}\frac{\Delta;\Gamma \vdash t : \forall X_1 \ldots X_n.A \qquad \Delta \vdash \forall X_1 \ldots X_n.A}{\Delta;\Gamma \vdash t : \forall X_2 \ldots X_n.A[B_1/X_1]}$$

$$\textsc{TyApp}\frac{\begin{array}{c}\vdots \qquad\qquad\qquad\qquad \Delta \vdash \forall X_n.A[B_1/X_1] \cdots [B_{n-1}/X_{n-1}] \end{array}}{\Delta;\Gamma \vdash t : A[B_1/X_1] \cdots [B_n/X_n]}$$

$$\textsc{TyAbs}\frac{}{\Delta;\Gamma \vdash t : \forall X'_m.\, A[B_1/X_1] \cdots [B_n/X_n]} \; (X'_m \notin \mathrm{FV}(A[B_1/X_1] \cdots [B_n/X_n]))$$

$$\vdots$$

$$\textsc{TyAbs}\frac{\Delta;\Gamma \vdash t : \forall X'_2 \ldots X'_m.\, A[B_1/X_1] \cdots [B_n/X_n]}{\Delta;\Gamma \vdash t : \forall X'_1 \ldots X'_m.\, A[B_1/X_1] \cdots [B_n/X_n]} \; (X'_1 \notin \mathrm{FV}\left(\begin{smallmatrix} \forall X'_2 \ldots X'_m. \\ A[B_1/X_1] \cdots [B_n/X_n] \end{smallmatrix}\right))$$

2.4.3 Syntax-directed typing rules

The syntax-directed typing rules [23] deduce a type, rather than a type scheme, for a given term under a typing context ($\Gamma \vdash t : A$). These rules are syntax-directed, since for each syntactic category of terms, there is only one typing rule that can apply.

The syntax-directed typing rules are based on the observation that the Inst and Gen in the declarative typing rules are only necessary at the Var and Let rules, respectively. That is, we only need to apply the Inst rule to the conclusion of the Var rule, and, we only need to apply the Gen rule to the first premise ($\Gamma \vdash s : \sigma$) of the Let rule. The Var_s rule can be understood as a merging of Var and Inst into one rule. The Abs_s rule and the App_s rule remain the same as their counterparts in the declarative typing rules. The Let_s rule can be understood as a merging of the Let and the Gen into one rule.

The notation $\overline{\Gamma}(A)$ appearing in the rule Let_s is the generalization closure of the type A with respect to Γ. That is, $\overline{\Gamma}(A)$ generalizes A over all the free type variables occurring in A, except the free types variables occurring in Γ. The free type variables of Γ are defined as $\text{FV}(\Gamma) = \bigcup_{x:\sigma \in \Gamma} \text{FV}(\sigma)$.

The syntax-directed typing rules are sound (Theorem 2.4.1) and complete (Theorem 2.4.2) with respect to the declarative typing rules.

We will simply sketch the key ideas for the proof of the soundness of $\vdash_s$ since the soundness is rather obvious. All we need to do is transform any given derivation for $\vdash_s$ into a derivation for $\vdash$, which is straightforward.

Theorem 2.4.1 ($\vdash_s$ is sound with respect to $\vdash$)**.** $\dfrac{\Gamma \vdash_s t : A}{\Gamma \vdash t : A}$

Proof. Recall that the Var_s rule can be understood as a merging of Var and Inst. Thus, we can transform any derivation step using the Var_s rule into two steps of derivation: using the Var rule and then applying the Inst rule to the conclusion

of the VAR rule.

The ABS_s rule and the APP_s rules are mapped to the ABS rule and the APP rule, respectively.

Recall that the LET_s rule can be understood as a merging of LET and GEN. We can transform any derivation step using the LET_s rule into a series of GEN rules applied to the first premise of the LET rule, and then applying the LET rule. Since the definition of the closure $\overline{\Gamma}(A)$ appearing in the LET_s rule generalizes only the free type variables of A, which do not appear free in Γ, the condition $X \notin \text{FV}(\Gamma)$ appearing in the GEN rule holds. □

The completeness of $\vdash_{\mathsf{s}}$ is stated below. Note that the completeness of $\vdash_{\mathsf{s}}$ must be stated in terms of generalization closure and of the type scheme ordering relation ($\overline{\Gamma}(A) \sqsubseteq \sigma$) since the syntax-directed rules can only deduce types, not type schemes. The syntax-directed rule $\vdash_{\mathsf{s}}$ is complete in the sense that for any given term we can always deduce a type A such that the closure of A is more general than the type scheme σ deduced from the declarative typing rules.

Theorem 2.4.2 ($\vdash_{\mathsf{s}}$ is complete with respect to $\vdash$)**.** $\dfrac{\Gamma \vdash t : \sigma}{\exists A.\ \Gamma \vdash_{\mathsf{s}} t : A \ \wedge\ \overline{\Gamma}(A) \sqsubseteq \sigma}$

Proof. We prove this by induction on the derivation of $\Gamma \vdash t : \sigma$. Let us consider the cases by the last rule applied (i.e., root of the derivation).

When the last rule is VAR, we know that $x : \sigma \in \Gamma$. We choose A in the VAR_s rule to be an instance of σ, instantiating each universally quantified variable with a fresh variable, which neither appears free in σ nor Γ. We further restrict A to satisfy $\overline{\Gamma}(A) \sqsubseteq \sigma$. For example, when $\sigma = \forall X_1.\forall X_2.X_1 \to X_2 \to X$, we choose $A = X_1' \to X_2' \to X$ where $X_1', X_2' \notin \text{FV}(\Gamma)$. If $X \in \text{FV}(\Gamma)$, then $\overline{\Gamma}(A) = \forall X_1'.\forall X_2'.X_1' \to X_2' \to X$, which is α-equivalent to σ, therefore, $\overline{\Gamma}(A) \sqsubseteq \sigma$. Otherwise, if $X \notin \text{FV}(\Gamma)$, then $\overline{\Gamma}(A) = \forall X_1'.\forall X_2'.\forall X.X_1' \to X_2' \to X$; so, $\overline{\Gamma}(A) \sqsubseteq \sigma$ still holds.

When the last rule is ABS, it is straightforward by induction.

When the last rule is App, it is straightforward by induction.

When the last rule is Let (**let** $x = s$ **in** t), we know by induction that there exists A' and B' such that $\Gamma \vdash_{\mathsf{s}} s : A' \wedge \overline{\Gamma}(A') \sqsubseteq \sigma$ and $\Gamma, x : \sigma \vdash_{\mathsf{s}} t : B' \wedge \overline{\Gamma, x : \sigma}(B') \sqsubseteq B$. The case for Let would be complete if we could show that $\Gamma, x : \overline{\Gamma}(A') \vdash_{\mathsf{s}} t : B' \wedge \overline{\Gamma, x : \overline{\Gamma}(A')}(B') \sqsubseteq B$. Instead we can only show $\Gamma, x : \sigma \vdash_{\mathsf{s}} t : B' \wedge \overline{\Gamma, x : \sigma}(B') \sqsubseteq B$. We use Lemma 2.4.1 to prove $\Gamma, x : \overline{\Gamma}(A') \vdash_{\mathsf{s}} t : B'$ from $\Gamma, x : \sigma \vdash_{\mathsf{s}} t : B'$, and we use Lemma 2.4.2 to prove $\overline{\Gamma, x : \overline{\Gamma}(A')}(B') \sqsubseteq B$ from $\overline{\Gamma, x : \sigma}(B') \sqsubseteq B$ and the transitivity of $\sqsubseteq$. These two lemmas are introduced directly following the proof of this theorem.

When the last rule is Inst, it is straightforward by induction and transitivity of $\sqsubseteq$.

When the last rule is Gen, we know by induction that there exists A such that $\overline{\Gamma}(A) \sqsubseteq \sigma$. We also know that $X \notin \mathrm{FV}(\overline{\Gamma}(A))$ by the definition of generalization closure. This step follows from a proof by contradiction argument. If it were the case that $X \in \overline{\Gamma}(A)$, then it should be the case that $X \in \mathrm{FV}(\Gamma)$ by the definition of generalization closure. This contradicts the side condition of the Gen rule: $X \notin \mathrm{FV}(\Gamma)$. Recall that generic instantiation allows quantifying type variables that do not appear free in the original type scheme. Thus, $\overline{\Gamma}(A) \sqsubseteq \forall X.\sigma$ by definition of $\sqsubseteq$. □

Lemma 2.4.1 (generalizing typing context is safe)**.** $\dfrac{\Gamma \vdash_{\mathsf{s}} t : A \qquad \Gamma' \sqsubseteq \Gamma}{\Gamma' \vdash_{\mathsf{s}} t : A}$ *where*

$\Gamma' \sqsubseteq \Gamma$ *when for any* $x : \sigma \in \Gamma$*, there exists* $x : \sigma' \in \Gamma'$ *such that* $\sigma' \sqsubseteq \sigma$*.*

Proof. This is an intuitively obvious property since assuming more general type schemes for variables only makes it possible to deduce all the judgments of $\vdash_{\mathsf{s}}$ and more, but no less, by transitivity of $\sqsubseteq$ over type schemes. We will simply give a proof for the base case, the Var rule, which illustrates this intuition. Other cases are straightforward by induction on the derivation of $\vdash_{\mathsf{s}}$.

When $\Gamma \vdash^{\mathsf{s}} x : A$, we know that there exists $x, \sigma' \in \Gamma'$ such that $\sigma' \sqsubseteq \sigma$. By the $\textsc{Var}_s$ rule, we can deduce any A' for x such that $\Gamma' \vdash^{\mathsf{s}} x : A'$ and $\sigma' \sqsubseteq A'$. By transitivity of $\sqsubseteq$, $\sigma' \sqsubseteq \sigma \sqsubseteq A$. Therefore, $\Gamma' \vdash^{\mathsf{s}} x : A$. □

Lemma 2.4.2 (closure of a more general typing context is more general)**.**

$$\frac{\Gamma' \sqsubseteq \Gamma}{\overline{\Gamma'}(A) \sqsubseteq \overline{\Gamma}(A)} \textit{ for any } A.$$

Proof. It is obvious once we observe that $\mathrm{FV}(\Gamma') \subseteq \mathrm{FV}(\Gamma)$. To show this, it suffices to show that $(\subseteq)$ relation holds pointwise on the type schemes in the context. That is, $\mathrm{FV}(\sigma') \sqsubseteq \mathrm{FV}(\sigma)$ when $x : \sigma' \in \Gamma'$ and $x : \sigma \in \Gamma$. Note that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$ by definition of $\sqsubseteq$ over contexts. From the assumption $\Gamma' \sqsubseteq \Gamma$, we know that $\sigma' \sqsubseteq \sigma$ when $x : \sigma' \in \Gamma'$ and $x : \sigma \in \Gamma$. Thus, we only need to show that $\sigma' \sqsubseteq \sigma$ implies $\mathrm{FV}(\sigma') \subseteq \mathrm{FV}(\sigma)$, which is not difficult to observe from the definition of $\sqsubseteq$ over type schemes ($\textsc{GenInst}$ in Figure 2.9). □

2.4.4 The type inference algorithm W

Damas and Milner [29] presented the type inference algorithm W (Figure 2.10) and proved its soundness and completeness with respect to the declarative typing rules. Here, we show the soundness and completeness of the type inference algorithm, W (Figure 2.10), with respect to the syntax-directed typing rules. Each rule of the type inference algorithm W has a similar structure to the corresponding syntax-directed rule. The type inference algorithm has additional details of explicitly managing fresh type variable introduction and substitution.

The unification of A_1 and A_2 succeeds when there exists a substitution S such that $SA_1 = SA_2$. When the unification succeeds, we write $\mathrm{unify}(A_1, A_2) \rightsquigarrow S$, where the resulting substitution S is a unifier of A_1 and A_2. Furthermore, S is a most general unifier [47, 81] whose domain is a subset of $\mathrm{FV}(A_1) \cup \mathrm{FV}(A_2)$. That is, for any unifier S' such that $S'A_1 = S'A_2$ and $\mathrm{dom}(S') \subseteq \mathrm{FV}(A_1) \cup \mathrm{FV}(A_2)$, there exists a substitution R such that $S' = R \circ S$ and $\mathrm{dom}(R) \subseteq \mathrm{dom}(S)$. The

$$\textsc{Var}_W \frac{\begin{array}{c} x : \forall X_1 \ldots X_n.A \in \Gamma \\ X'_1, \ldots, X'_n \text{ fresh} \end{array}}{W(\Gamma, x) \rightsquigarrow (\emptyset, A[X'_1/X_1] \cdots [X'_n/X_n])}$$

$$\textsc{Abs}_W \frac{\begin{array}{c} X \text{ fresh} \\ W((\Gamma, x : X), t) \rightsquigarrow (S_1, A) \end{array}}{W(\Gamma, \lambda x.t) \rightsquigarrow (S_1, S_1 X \to A)}$$

$$\textsc{App}_W \frac{\begin{array}{c} W(\Gamma, t) \rightsquigarrow (S_1, A_1) \\ W(S_1\Gamma, s) \rightsquigarrow (S_2, A_2) \\ X \text{ fresh} \\ \text{unify}(S_2 A_1, A_2 \to X) \rightsquigarrow S_3 \end{array}}{W(\Gamma, t\ s) \rightsquigarrow (S_3 \circ S_2 \circ S_1, S_3 X)}$$

$$\textsc{Let}_W \frac{\begin{array}{c} W(\Gamma, s) \rightsquigarrow (S_1, A_1) \\ W((S_1\Gamma, x : \overline{S_1\Gamma}(A_1)), t) \rightsquigarrow (S_2, A_2) \end{array}}{W(\Gamma, \textbf{let } x = s \textbf{ in } t) \rightsquigarrow (S_2 \circ S_1, A_2)}$$

Figure 2.10: The type inference algorithm W.

composition of two substitutions is defined as $(S_2 \circ S_1)A = S_2(S_1(A))$.

Proposition 2.4.1. $\dfrac{\Gamma \vdash_s t : A}{S\Gamma \vdash_s t : SA}$

Proposition 2.4.2. $S(\overline{\Gamma}(A)) = \overline{S\Gamma}(SA)$

Theorem 2.4.3 (Soundness of W)**.** $\dfrac{W(\Gamma, t) \rightsquigarrow (S, A)}{S\Gamma \vdash_s t : A}$

Proof. By induction on the syntax of the term t.

case (x) Obvious, by definition of $\sqsubseteq$.

case $(\lambda x.t)$ We need to show that $S_1\Gamma \vdash_s \lambda x.t : S_1 X \to A$.

By induction, we know that $S_1(\Gamma, x : X) \vdash_s t : A$.

We know that $S_1\Gamma, x : S_1 X \vdash_s t : A$, since $S_1(\Gamma, x : X) = S_1\Gamma, x : S_1 X$.

By $\textsc{Abs}_s$ rule, we have $S_1\Gamma \vdash_s \lambda x.t : S_1 X \to A$.

case $(t\ s)$ We need to show that $(S_3 \circ S_2 \circ S_1)\Gamma \vdash_s t\ s : S_3 X$.

By induction, we know that $S_1\Gamma \vdash_s t : A_1$ and $S_2(S_1\Gamma) \vdash_s s : A_2$. By Proposition 2.4.1, $S_3(S_2(S_1\Gamma)) \vdash_s t : S_3(S_2(A_1))$ and $S_3(S_2(S_1\Gamma)) \vdash_s s : S_3 A_2$.

Due to the property of unification, $S_2 A_1 = A_2 \to X$.

Thus, $S_3(S_2(S_1\Gamma)) \vdash_s t : S_3(A_2 \to X)$ and $S_3(S_2(S_1\Gamma)) \vdash_s s : S_3 A_2$.

That is, $(S_3 \circ S_2 \circ S_1)\Gamma \vdash_s t : S_3 A_2 \to S_3 X$ and $(S_3 \circ S_2 \circ S_1)\Gamma \vdash_s s : S_3 A_2$.

By $\textsc{App}_s$ rule, we have $(S_3 \circ S_2 \circ S_1)\Gamma \vdash_s t\ s : S_3 X$.

case $(\mathbf{let}\ x = s\ \mathbf{in}\ t)$ We need to show that $(S_2 \circ S_1)\Gamma \vdash_s \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2$.

By induction, we have $S_1\Gamma \vdash_s s : A_1$ and $S_2(S_1\Gamma, x : \overline{S_1\Gamma}(A_1)) \vdash_s t : A_2$.

By Proposition 2.4.1, we have $S_2(S_1\Gamma) \vdash_s s : S_2 A_1$.

By Proposition 2.4.2, we have $S_2(S_1\Gamma), x : \overline{S_2(S_1\Gamma)}(S_2 A_1) \vdash_s t : A_2$.

By $\textsc{Let}_s$, we have $S_2(S_1\Gamma) \vdash_s \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2$. Since $S_2(S_1\Gamma) = (S_2 \circ S_1)\Gamma$ by definition, we have $(S_2 \circ S_1)\Gamma \vdash_s \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2$. □

Proposition 2.4.3 ($\textsc{Abs}_s$ inverse)**.** $\dfrac{\Gamma \vdash_s \lambda x.t : A \to B}{\Gamma, x : A \vdash_s t : B}$

Proposition 2.4.4 ($\textsc{App}_s$ inverse)**.** $\dfrac{\Gamma \vdash_s t\ s : B}{\exists A.\, (\Gamma \vdash_s t : A \to B\ \wedge\ \Gamma \vdash_s s : A)}$

Proposition 2.4.5 ($\textsc{Let}_s$ inverse)**.** $\dfrac{\Gamma \vdash_s \mathbf{let}\ x = s\ \mathbf{in}\ t : B}{\exists A.\, \Big(\Gamma \vdash_s s : A\ \wedge\ \Gamma, x : \overline{\Gamma}(A) \vdash_s t : B\Big)}$

Theorem 2.4.4 (Completeness of W)**.**

For any Γ and t, there exist S', where $\mathrm{dom}(S') \subseteq \mathrm{FV}(\Gamma)$, and A' such that

$$\frac{S'\Gamma \vdash_s t : A'}{W(\Gamma, t) \rightsquigarrow (S, A_W)\ \wedge\ \exists R.\, \Big(S'\Gamma = R(S\Gamma)\ \wedge\ R(\overline{S\Gamma}(A_W)) \sqsubseteq A'\Big)}$$

Proof. See Appendix A. □

Corollary 2.4.1 (Completeness of W under closed context)**.**

$$\frac{\Gamma \vdash_s t : A' \qquad \mathrm{FV}(\Gamma) = \emptyset}{W(\Gamma, t) \rightsquigarrow (S, A)\ \wedge\ \exists R.R(\overline{\Gamma}(A)) \sqsubseteq A'}$$

Proof. By Theorem 2.4.4 and the fact that $S'\Gamma = \Gamma$ for any S' when $\mathrm{FV}(\Gamma) = \emptyset$. □

Part II

Mendler style

Chapter 3

MENDLER-STYLE RECURSION SCHEMES

In this chapter, we explore a family of terminating recursion combinators of Mendler style. These combinators behave well over a wide class of recursive datatypes. This chapter is a revised and extended version of the conference paper by Ahn and Sheard [6]. Here, the names of Mendler-style recursion combinators are different from those in the paper to make the names consistent throughout the dissertation: **mit**, **msfit**, **mcvit**, and **msfcvit** correspond to *mcata*, *msfcata*, *mhist*, and *msfhist* in the paper. In addition, we introduce several more families of recursion combinators (**mpr**, **mcvit**, **mcvpr**) and a new example using **msfit** over an indexed datatype,[1] which are not present in the conference paper.

This chapter gives the reader an intuitive understanding of Mendler-style recursion combinators, rather than providing a rigorous formulation of the theories behind Mendler style. The discussions in this chapter are guided by a series of examples (and some semi-formal proofs) written in a certain style of Haskell, which assumes certain conventions (see Section 3.1). Haskell is a *real world* functional programming language [72], which admits a certain level of formality since it is a pure functional language. More rigorous formulations of the background theory used to formalize Mendler style will come in the following chapters (Chapter 4 and Chapter 5).

[1] The type signature of $\mathbf{msfit}_{*\to *}$ and the definition of the $\breve{\mu}_{*\to *}$ datatype is different from the type signature of *msfcata1* and the definition of $\breve{\mu}_{*\to *}$ datatype in the paper. *msfcata1* does type check but its type sigature was too restrictive to write any useful examples. Here, we give a more flexible type signature and defintion for $\mathbf{msfit}_{*\to *}$ and $\breve{\mu}_{*\to *}$ so that we can write useful examples with them.

3.1 INTRODUCTION

The functional programming community has traditionally focused on a style of recursion combinators that works well in Hindley–Milner languages. One well-known combinator is called fold (a.k.a. catamorphism or iteration). We explore a more expressive style called Mendler style. Mendler-style recursion combinators were originally developed in the context of the Nuprl [25] type system. Nuprl made extensive use of dependent types and higher-rank polymorphism. General type checking in Nuprl was done by interactive theorem proving – not by type inference. Mendler-style combinators are considerably more expressive than the conventional combinators of the Squiggol [13] school in two aspects: (1) Mendler-style combinators are well behaved (i.e., they guarantee termination) over a wider range of recursive datatypes and (2) Mendler-style combinators are uniformly defined over non-regular datatypes. An historical perspective on Mendler style is summarized in Section 3.1.2.

Recently, Mendler-style recursion combinators have been studied in the context of modern functional languages with advanced type system features, including higher-rank polymorphism and generalized algebraic datatypes. This chapter extends that work by

- ★ Illustrating that Mendler-style approach applies to useful examples of negative datatypes, through case studies on HOAS (Section 3.9.1 and Section 3.9.3),
- ★ Extending Mendler-style iteration by using the inverse trick (**msfit**) (Section 3.9.1), which was first described by Fegaras and Sheard [32] and later refined by Washburn and Weirich [96] in conventional style,
- ★ Using **msfit** over an indexed datatype to evaluate a simply-typed HOAS (Section 3.9.3), which clearly exemplifies the advantages of Mendler style over conventional style,

- Providing an intuitive explanation of why Mendler-style iteration ensures termination (Section 3.4) even for negative datatypes (Section 3.6). We illustrate a semi-formal proof of termination by encoding **msfit** in the F_ω fragment of Haskell (Figure 3.23 in Section 3.10),
- ★ Providing an intuitive explanation of why Mendler-style course-of-values iteration terminates for positive datatypes (Section 3.5), but may fail to terminate for negative datatypes (Section 3.6), by illustrating a counter-example that obviously fails to terminate,
- Organizing a large class of Mendler-style recursion combinators into an intuitive hierarchy, of increasing generality, that is expressive enough to cover regular datatypes (Section 3.2, Section 3.5), nested datatypes (Section 3.7.1), indexed datatypes (GADTs) (Section 3.7.2), mutually recursive datatypes (Section 3.7.3), and negative datatypes (Section 3.6, Section 3.9.1), and
- Providing a detailed set of examples, all written in Haskell, illustrating two versions (one with general recursion and one with a Mendler-style recursion combinator) side by side, in order to illustrate the usage of each family of recursion combinators.

The ★-items are original contributions, and the others are collective observations of common patterns arising from the study of both previously known combinators and our new combinators.

In this chapter, we demonstrate the Mendler-style combinators in the Glasgow Haskell Compiler [88] (GHC) dialect of Haskell. However, this demonstration depends on a set of conventions, because we want to control the source of non-termination. We assert that all our code fragments conform with our conventions. These conventions include:

1. all values of algebraic datatypes are finite (i.e., do not use laziness to build infinite structures),

2. certain conventions of data abstraction that are not enforced by Haskell (i.e., treating the recursive type operator μ, and the recursion combinators, as primitive constructs, rather than user-defined constructs), and
3. other sources of non-termination are delineated (e.g., not allowed to use general recursion in user-defined datatypes and functions, pattern matching can only be done through the recursion combinators).

Mendler-style combinators operate on types defined in two levels, i.e., two-level types (see Section 3.2). Two-level types are characterized by splitting the definition of a recursive type into a generating functor (or a base datatype) and an explicit application of the appropriate datatype fixpoint operator (μ). There exists an infinite series of datatype fixpoint operators for each different kind (e.g., μ_{*}, $\mu_{*\to *}$). In this chapter, we illustrate the Mendler-style recursion combinators only at the two simplest kinds, $*$ and $* \to *$.

3.1.1 Background - Termination and Negativity

Mendler [64] showed that diverging computations can be expressed using recursive datatypes with negative occurrences of the datatype being defined. No explicit recursion at the value level is required to elicit non-termination. We can illustrate this in Haskell as follows:

$$
\begin{array}{l|l}
\textbf{data}\ T = C\ (T \to ()) & \quad w\ (C\ w) \\
p :: T \to (T \to ()) & \rightsquigarrow (p\ (C\ w))\ (C\ w)) \\
p\ (C\ f) = f & \rightsquigarrow w\ (C\ w) \\
w :: T \to () & \rightsquigarrow (p\ (C\ w))\ (C\ w)) \\
w\ x = (p\ x)\ x & \rightsquigarrow \cdots
\end{array}
$$

On the left is a data definition of the negative datatype T and the non-recursive functions p and q. On the right is a diverging computation ($\rightsquigarrow$ denotes reduction steps).

Note the term $w\ (C\ w) :: T$ diverges even though the functions p and w are non-recursive. The cause of this divergence can be attributed to the "hidden" self application in the term $w\ (C\ w) :: T$. The negative occurrence of T in the datatype definition of T is what enables this self application to be well typed.

For this reason, many systems (e.g., Hagino's CPL [43] and Coq [75]) require all recursive datatypes to be positive (or covariant) in order to ensure normalization. Uustalu and Vene [90] call this style, limiting recursive occurrences to positive positions, the *conventional* style, in contrast to what they name Mendler style [65].

In Mendler style, datatypes are not limited to recurse over positive occurrences, yet functions expressible via iteration (a.k.a. catamorphism) always terminate. This was first reported by Uustalu [89] and Matthes [59], but the search for exciting examples of negative datatypes was postponed (considering it "may have a theoretical value only"[90]). Subsequent work [4, 5, 92, 94], that pioneered Mendler style in practical functional programming also failed to produce good examples that make use of negative datatypes in Mendler style.

In the functional programming community, there are both well-known and useful examples of negative (or mixed-variant) datatypes (e.g., delimited control[82][2]). One of the classic examples is HOAS [21, 76]. A non-standard definition of HOAS in Haskell is:[3] **data** $Exp = Lam\ (Exp \to Exp) \mid App\ Exp\ Exp \mid Var\ String$. We can define a function $showExp :: Exp \to String$ that formats an HOAS expression into a string. For example,

$$showExp\ (Lam\ (\lambda x \to x)) \rightsquigarrow \texttt{"(\textbackslash a->a)"}$$
$$showExp\ (Lam\ (\lambda x \to App\ x\ x)) \rightsquigarrow \texttt{"(\textbackslash a->(a a))"}$$

[2] A Haskell datatype definition for this can be found at
`http://lists.seas.upenn.edu/pipermail/types-list/2004/000267.html`

[3] The datatype *Exp* here is a HOAS-like structure specialized to *String* type. The standard definition of HOAS, which omits the *Var* constructor, makes it more challenging to define *showExp*, as we shall see in Section 3.9.1.

The function *showExp* is total, provided the function values embedded in the *Lam* data constructor are total. We will illustrate that this example (which involves a negative datatype) and many other examples that involve non-regular datatypes and mutually recursive datatypes can all be easily written using Mendler-style recursion combinators, whose termination properties are known. A Detailed case study of how to express this function using our Mendler-style iteration extended with syntactic inverses is presented in Section 3.9.1.

3.1.2 Historical progression

Mendler [64] discovered an interesting way of formalizing primitive recursion, which was later dubbed "Mendler style", while he was formalizing a logic that extended System F with primitive recursion. Interestingly, Mendler did not seem to notice (or maybe just did not bother to mention) that his style of formalizing primitive recursion also guaranteed normalization for non-positive recursive types – Mendler required recursive types to be positive in his extension of System F. A decade later, Matthes [59] and Uustalu [89] noticed that Mendler never used the positivity condition in his proof of strong normalization.

Abel and Matthes [3] generalized Mendler's primitive recursion combinator [64] into a family of combinators that are uniformly defined for type constructors of arbitrary kinds. This was necessary for handling nested datatypes. Their system extends System F_ω (Mendler [64] extends System F). The notion of a kind indexed family of Mendler combinators has now become the norm.

Abel and Matthes [3] proved strong normalization of their language MRec, which extends System F_ω by adding a family of kind-indexed Mendler-style primitive recursion combinators. They showed that MRec has a reduction preserving embedding into a calculus they called Fix_ω. Then, they showed that Fix_ω is strongly normalizing.

Abel, Matthes, and Uustalu [4, 5] studied a kind-indexed family of iteration

combinators, along with examples involving nested datatypes that make use of those combinators. Iteration (a.k.a. catamorphism) is a recursion scheme that has the same computational power as primitive recursion (i.e., both can be defined in terms of each other), but has different algorithmic complexity.

It is strongly believed that primitive recursion is more efficient than iteration. For instance, it is trivial to define a constant time predecessor for natural numbers with primitive recursion, but it is believed impossible to define the constant time predecessor with iteration. The Mendler-style iteration family can be embedded into F_ω in a reduction preserving manner. That is, we can encode the family of Mendler-style iteration combinators into F_ω in such a way that the number of reduction steps of the original and the embedding differ only by a constant number of steps. The primitive recursion family, in contrast, is not believed to have a reduction preserving embedding into F_ω. Abel and Matthes [3] needed a more involved embedding of MRec into Fix_ω, which has a richer structure than F_ω.

Although Matthes, Uustalu, and others were well aware of the fact that the Mendler-style iteration family and the primitive recursion family both normalize for negative recursive types, they did not explore or document actual examples. They postponed "the search for exciting examples of negative recursive types". They stated that the normalization of negative types "may have a theoretical value only"[90]. So, until recently, the study of Mendler-style recursion combinators has focused on examples of positive recursive types possibly with type indices (but not term-indices).

Recently, we developed several new contributions to the study of the Mendler-style recursion schemes [6].[4] These contributions fall into three broad categories:

- discovered a new family of Mendler-style recursion combinators (Section 3.9), which normalizes for negative recursive types and is believed to be more expressive than the Mendler-style iteration family,

[4] This chapter is a revised and extended version of this ICFP paper.

- discovered a counterexample, which proves that some families of Mendler-style recursion combinators do not normalize for negative datatypes but only normalize for positive datatypes (Section 3.6), and
- explored the use of Mendler-style recursion combinators over (almost) term-indexed types (i.e., GADTs) (Section 3.7.2).

3.1.3 Roadmap to a tour of the Mendler-style approach

In this subsection, we give an overview of the Mendler-style approach, to orient the reader to navigate the following sections.

First, we introduce the Mendler-style iteration (**mit**. a.k.a. catamorphism) (Section 3.4) and course-of-values iteration (**mcvit**. a.k.a. histomorphism) (Section 3.5) combinators at kind $*$, that is, for (non-mutually recursive) regular datatypes (Section 3.2). We also give an intuitive explanation why these Mendler-style recursion combinators ensure termination for positive datatypes.

In Section 3.6, we discuss why the Mendler-style iteration (**mit**) ensures termination even for negative datatypes, while the Mendler-style course-of-values iteration (**mcvit**) can only ensure termination for positive datatypes.

Then, we move our focus from non-mutually recursive regular datatypes to more expressive datatypes (Section 3.7), which require recursion combinators at kind $* \to *$. We provide several examples of non-regular datatypes including nested datatypes (Section 3.7.1) and indexed datatypes (GADTs) (Section 3.7.2), which illustrate the use of the Mendler-style iteration (**mit**) and course-of-values iteration (**mcvit**) at kind $* \to *$. We also provide some examples that show how to encode mutually recursive datatypes using indexed datatypes (Section 3.7.3).

In Section 3.8, we introduce the Mendler-style primitive recursion (**mpr**) and course-of-values recursion (**mcvpr**). These two combinators **mpr** and **mcvpr** are equivalent to **mit** and **mcvit**, respectively, in terms of computability, but often lead to more efficient implementations.

In Section 3.9, we introduce a new Mendler-style family (**msfit**), which we discovered, and illustrate its expressiveness over negative datatypes by presenting the case study on formatting HOAS (Section 3.9.1) and evaluating simply-typed HOAS (Section 3.9.3)

Finally, we summarize the properties of Mendler-style recursion combinators in Section 3.10.

All of our results are summarized in Figures 3.1, 3.2, and 3.3. In Figure 3.1, we define the Mendler-style datatype fixpoint operators (i.e., μ_* and $\mu_{*\to *}$). These are datatype definitions in Haskell that take type constructors as arguments. They are used to tie the recursive knot through the generating functor (or base datatype) that they take as an argument.

In Figure 3.2, we provide the types of 8 Mendler-style combinators distributed over the two kinds that we consider, along with the type of a conventional iteration combinator for comparison. The combinators can be organized into a hierarchy of increasing generality. By juxtaposing the types of the combinators, it looks clear where in the hierarchy each combinator appears and how each is related to the others.

In Figure 3.3, we define the combinators themselves, again distributed over two kinds. The definitions of the corresponding combinators at two kinds are textually identical, although they must be given different types at each kind.

In addition to the Mendler-style recursion combinators in Figures 3.2 and 3.3, we introduce Mendler-style primitive recursion (**mpr**) and course-of-values recursion (**mcvpr**) in Section 3.8.

In Figures 3.5, 3.6, 3.9, 3.10, and 3.11, we provide examples[5] selected for each of the combinators $\mathbf{mit}_*$, $\mathbf{mcvit}_*$, $\mathbf{mit}_{*\to *}$, and $\mathbf{mcvit}_{*\to *}$. We provide examples using the **mpr** and **mcvpr** families in Section 3.8. We discuss the remaining combinators of the inverse-augmented fixpoints in Section 3.9.1 and Section 3.9.3,

[5]Some of the examples (Figures 3.5, 3.6, and 3.9) are adopted from [4, 5, 92, 94].

$$
\begin{array}{lllllll}
\textbf{newtype}\ \mu_{*} & (f :: * \to *) & & = & \mathsf{In}_{*} & \{\mathsf{out}_{*} :: f\ (\mu_{*}\ f)\} & \\
\textbf{newtype}\ \mu_{*\to*} & (f :: (* \to *) \to (* \to *)) & i & = & \mathsf{In}_{*\to*} & \{\mathsf{out}_{*\to*} :: f\ (\mu_{*\to*}\ f)\ i\} & \\
\textbf{data} & \breve{\mu}_{*} & (f :: * \to *)\ (a :: *) & = & \mathsf{I\breve{n}}_{*} & \{\breve{\mathsf{o}}\mathsf{ut}_{*} :: f\ (\breve{\mu}_{*}\ f\ a)\} & \mid \mathsf{Inverse}_{*}\ a \\
\textbf{data} & \breve{\mu}_{*\to*} & (f :: (* \to *) \to (* \to *))\ (a :: * \to *)\ i & = & \mathsf{I\breve{n}}_{*\to*} & \{\breve{\mathsf{o}}\mathsf{ut}_{*\to*} :: f\ (\breve{\mu}_{*\to*}\ f\ a)\ i\} & \mid \mathsf{Inverse}_{*\to*}\ (a\ i)
\end{array}
$$

Figure 3.1: Standard (μ) and inverse-augmented ($\breve{\mu}$) datatype fixpoints at kinds $*$ and $* \to *$.

		abstract inverse	abstract unroll	abstract recursive call	combining function	input value	answer
cata	$:: \mathit{Functor}\ f \Rightarrow ($				$(f\ a \to a) \to$	$\mu_{*}\ f$	$\to a$
$\mathbf{mit}_{*}$	$:: (\forall r.$			$(r \to a) \to$	$(f\ r \to a) \to$	$\mu_{*}\ f$	$\to a$
$\mathbf{mit}_{*\to*}$	$:: (\forall r\ i.$			$(\forall i.r\ i \to a\ i) \to$	$(f\ r\ i \to a\ i) \to$	$\mu_{*\to*}\ f\ i$	$\to a\ i$
$\mathbf{mcvit}_{*}$	$:: (\forall r.$		$(r \to f\ r) \to$	$(r \to a) \to$	$(f\ r \to a) \to$	$\mu_{*}\ f$	$\to a$
$\mathbf{mcvit}_{*\to*}$	$:: (\forall r\ i.$		$(\forall i.\ r\ i \to f\ r\ i) \to$	$(\forall i.r\ i \to a\ i) \to$	$(f\ r\ i \to a\ i) \to$	$\mu_{*\to*}\ f\ i$	$\to a\ i$
$\mathbf{msfit}_{*}$	$:: (\forall r.\ (a \to r\ a) \to$			$(r\ a \to a) \to$	$(f\ (r\ a) \to a) \to$	$(\forall a.\breve{\mu}_{*}\ f\ a) \to$	a
$\mathbf{msfit}_{*\to*}$	$:: (\forall r\ i.(\forall i.a\ i \to r\ a\ i) \to$			$(\forall i.r\ a\ i \to a\ i) \to$	$(f\ (r\ a)\ i \to a\ i)) \to$	$(\forall a.\breve{\mu}_{*\to*}\ f\ a\ i) \to$	$a\ i$
$\mathbf{msfcvit}_{*}$	$:: (\forall r.\ (a \to r\ a) \to$	$(\forall a.\ r\ a \to f\ (r\ a)) \to$		$(r\ a \to a) \to$	$(f\ (r\ a) \to a) \to$	$(\forall a.\breve{\mu}_{*}\ f\ a) \to$	a
$\mathbf{msfcvit}_{*\to*}$	$:: (\forall r\ i.(\forall i.a\ i \to r\ a\ i) \to$	$(\forall a\ i.r\ a\ i \to f\ (r\ a)\ i) \to$		$(\forall i.r\ a\ i \to a\ i) \to$	$(f\ (r\ a)\ i \to a\ i)) \to$	$(\forall a.\breve{\mu}_{*\to*}\ f\ a\ i) \to$	$a\ i$

Figure 3.2: Type signatures of recursion combinators. Note the heavy use of higher-rank types.

$$
\begin{array}{lllll}
\textbf{cata} & s\ (\mathsf{In}_*\ x) & = s\ (\mathit{fmap}\ (\textbf{cata}\ s)\ x) & & \\
\textbf{mit}_* & \varphi\ (\mathsf{In}_*\ x) & = \varphi & (\textbf{mit}_*\ \varphi) & x \\
\textbf{mit}_{*\to*} & \varphi\ (\mathsf{In}_{*\to*}\ x) & = \varphi & (\textbf{mit}_{*\to*}\ \varphi) & x \\
\textbf{mcvit}_* & \varphi\ (\mathsf{In}_*\ x) & = \varphi\ \mathsf{out}_* & (\textbf{mcvit}_*\ \varphi) & x \\
\textbf{mcvit}_{*\to*} & \varphi\ (\mathsf{In}_{*\to*}\ x) & = \varphi\ \mathsf{out}_{*\to*} & (\textbf{mcvit}_{*\to*}\ \varphi) & x
\end{array}
$$

$$
\begin{array}{lll}
\textbf{msfit}_* \quad \varphi\ r = \textbf{msfit}\ \varphi\ r\ \textbf{where} & & \\
\quad \textbf{msfit}\ \varphi\ (\breve{\mathsf{In}}_*\ x) & = \varphi\ \mathsf{Inverse}_* \quad (\textbf{msfit}\ \varphi) & x \\
\quad \textbf{msfit}\ \varphi\ (\mathsf{Inverse}_*\ z) & = z &
\end{array}
$$

$$
\begin{array}{lll}
\textbf{msfit}_{*\to*}\ \varphi\ r = \textbf{msfit}\ \varphi\ r\ \textbf{where} & & \\
\quad \textbf{msfit} :: (\forall r\ i.(\forall i.a\ i \to r\ a\ i) & & \\
\qquad \to (\forall i.r\ a\ i \to a\ i) & & \\
\qquad \to (f\ (r\ a)\ i \to a\ i)) \to \breve{\mu}_{*\to*}\ f\ a\ i \to a\ i & & \\
\quad \textbf{msfit}\ \varphi\ (\breve{\mathsf{In}}_{*\to*}\ x) & = \varphi\ \mathsf{Inverse}_{*\to*}\ (\textbf{msfit}\ \varphi) & x \\
\quad \textbf{msfit}\ \varphi\ (\mathsf{Inverse}_{*\to*}\ z) & = z &
\end{array}
$$

$$
\begin{array}{lll}
\textbf{msfcvit}_* \quad \varphi\ r = \textbf{msfcvit}\ \varphi\ r\ \textbf{where} & & \\
\quad \textbf{msfcvit} :: (\forall r.(a \to r\ a) & & \\
\qquad \to (\forall a.r\ a \to f\ (r\ a)) & & \\
\qquad \to (r\ a \to a) & & \\
\qquad \to (f\ (r\ a) \to a) \qquad) \to \breve{\mu}_*\ f\ a \to a & & \\
\quad \textbf{msfcvit}\ \varphi\ (\breve{\mathsf{In}}_*\ x) & = \varphi\ \mathsf{Inverse}_* \quad \breve{\mathsf{out}}_* \quad (\textbf{msfcvit}\ \varphi)\ x & \\
\quad \textbf{msfcvit}\ \varphi\ (\mathsf{Inverse}_*\ z) & = z &
\end{array}
$$

$$
\begin{array}{lll}
\textbf{msfcvit}_{*\to*}\ \varphi\ r = \textbf{msfcvit}\ \varphi\ r\ \textbf{where} & & \\
\quad \textbf{msfcvit} :: (\forall r\ i.(\forall i.a\ i \to r\ a\ i) & & \\
\qquad \to (\forall a\ i.r\ a\ i \to f\ (r\ a)\ i) & & \\
\qquad \to (\forall i.r\ a\ i \to a\ i) & & \\
\qquad \to (f\ (r\ a)\ i \to a\ i) \qquad) \to \breve{\mu}_{*\to*}\ f\ a\ i \to a\ i & & \\
\quad \textbf{msfcvit}\ \varphi\ (\breve{\mathsf{In}}_{*\to*}\ x) & = \varphi\ \mathsf{Inverse}_{*\to*}\ \breve{\mathsf{out}}_{*\to*}\ (\textbf{msfcvit}\ \varphi)\ x & \\
\quad \textbf{msfcvit}\ \varphi\ (\mathsf{Inverse}_{*\to*}\ z) & = z &
\end{array}
$$

Figure 3.3: Definitions of recursion combinators. Note the identical textual definitions of the same operators at different kinds, but with types specialized for that kind.

where we culminate with examples involving HOAS. We have structured each of the examples into two, side by side, parts. On the left, we provide a general recursive version and on the right, a Mendler-style version.

3.2 DEFINING REGULAR RECURSIVE DATATYPES

In the Mendler-style approach, we define recursive datatypes as fixpoints of non-recursive base datatypes. For example, the following are definitions of the natural number type in general recursion style (left) and in Mendler style (right).

$$
\begin{array}{ll}
 & \textbf{data}\ N\ r = Z \mid S\ r \\
\textbf{data}\ Nat & \textbf{type}\ Nat = \mu_*\ N \\
\quad = Z & zero \quad = \mathsf{In}_*\ Z \\
\quad \mid\ S\ Nat & succ\ n = \mathsf{In}_*\ (S\ n)
\end{array}
$$

Note, in Mendler style, we define *Nat* by applying the fixpoint μ_* to the base N. The type argument r in the base N is intended to denote the points of recursion in the recursive datatype. Here, we have only one point of recursion at S, the successor data constructor. Then, we define the shorthand constructors *zero* and *succ* (on the right), which correspond to the data constructors Z and S of the natural number datatype in the general recursive encoding (on the left). We can express the number 2 as $S\ (S\ Z)$ in the general recursive encoding and $succ\ (succ\ zero)$ or $\mathsf{In}_*\ (S\ (\mathsf{In}_*\ (S\ (\mathsf{In}_*\ Z))))$ in the Mendler-style encoding.

We can also define parameterized datatypes, such as lists, in Mendler style, using the same datatype fixpoint μ_*, provided that we consistently order the parameter arguments (p) to come before the type argument that denotes the recursion points (r) in the base datatype definition ($L\ p\ r$):

$$
\begin{array}{ll}
 & \textbf{data}\ L\ p\ r = N \mid C\ p\ r \\
\textbf{data}\ List\ p & \textbf{type}\ List\ p = \mu_*\ (L\ p) \\
\quad = N & nil \quad\quad = \mathsf{In}_*\ N \\
\quad \mid\ C\ p\ (List\ p) & cons\ x\ xs = \mathsf{In}_*\ (C\ x\ xs)
\end{array}
$$

We define *List p* as $\mu_*\ (L\ p)$, which is the fixpoint of the partial application of the base L to the parameter p. We can express the integer list with two elements 1 and 2 as $C\ 1\ (C\ 2\ N)$ in the general recursive encoding, and *cons* 1 (*cons* 2 *nil*) or $\mathsf{In}_*\ (C\ 1\ (\mathsf{In}_*\ (C\ 2\ (\mathsf{In}_*\ N))))$ in the Mendler-style encoding.

3.3 CONVENTIONAL ITERATION FOR REGULAR DATATYPES

The conventional iteration[6] is defined on the very same fixpoint, μ_*, as in Mendler style, provided that the base datatype f is a functor. This, more widely known approach [43], was independently developed at about the same time as Mendler style.

The additional requirement that the base datatype (f) is a functor shows up as a type class constraint (*Functor f*) in the type signature of the conventional iteration combinator **cata**:

cata :: *Functor f* $\Rightarrow (f\ a \to a) \to \mu_*\ f \to a$ (Figure 3.2).

This is necessary because **cata** is defined in terms of *fmap*, which is a method of the *Functor* class:

cata $\varphi\ (\mathsf{In}_*\ x) = \varphi\ (fmap\ (\textbf{cata}\ \varphi)\ x)$ (Figure 3.3).

The combinator **cata** takes a combining function $\varphi :: f\ a \to a$, which assumes the recursive subcomponents (e.g., tail of the list) have already been turned into a value of answer type (a) and combines the overall result.

A typical example of iteration is the list length function. We can define the list length function len_c in conventional style, as in Figure 3.4, which corresponds to the list length function *len* in general recursion style on the left-hand side of Figure 3.5. Of course, we need the functor instance for the base $L\ p$, which properly defines *fmap*, to complete the definition.

The conventional iteration is widely known, especially on the list type, as *foldr*.

[6] Also known as catamorphism. In Haskell-ish words, *foldr* on lists generalized to other datatypes.

$$
\begin{aligned}
&len_c :: List\ p \to Int \\
&len_c = \mathbf{cata}\ \varphi\ \mathbf{where} \\
&\quad \varphi\ N \qquad\quad = 0 \\
&\quad \varphi\ (C\ x\ xslen) = 1 + xslen
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathbf{instance}\ Functor\ (L\ p)\ \mathbf{where} \\
&\quad fmap\ f\ N \qquad\quad = N \\
&\quad fmap\ f\ (C\ x\ xs) = C\ x\ (f\ xs)
\end{aligned}
$$

Figure 3.4: **cata** example: list length function.

$$
\begin{aligned}
&\mathbf{data}\ List\ p \\
&\quad = N \\
&\quad \mid\ C\ p\ (List\ p)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathbf{data}\ L\ p\ r = N \mid C\ p\ r \\
&\mathbf{type}\ List\ p = \mu_*\ (L\ p) \\
&nil \qquad\quad = \mathsf{In}_*\ N \\
&cons\ x\ xs = \mathsf{In}_*\ (C\ x\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&len :: List\ p \to Int \\
&len\ N \qquad\quad = 0 \\
&len\ (C\ x\ xs) = 1 + len\ xs
\end{aligned}
\qquad\qquad
\begin{aligned}
&len_m :: List\ p \to Int \\
&len_m = \mathbf{mit}_*\ \varphi\ \mathbf{where} \\
&\quad \varphi\ len\ N \qquad\quad = 0 \\
&\quad \varphi\ len\ (C\ x\ xs) = 1 + len\ xs
\end{aligned}
$$

Figure 3.5: $\mathbf{mit}_*$ example: list length function.

This conventional iteration is more often used than the Mendler-style iteration, but it does not generalize easily to more exotic datatypes such as nested datatypes and GADTs.

3.4 MENDLER-STYLE ITERATION FOR REGULAR DATATYPES

The Mendler-style iteration combinator $\mathbf{mit}_*$ lifts the restriction that the base type must be a functor, but still maintains the strict termination behavior of **cata**. This restriction is lifted by using two devices.

- The combining function φ becomes a function of two arguments rather than one. The first argument is a function that represents a recursive caller, and the second is the base structure that must be combined into an answer. The

recursive caller allows the programmer to direct where recursive calls must be made. The *Functor* class requirement is lifted, because the definition of $\mathbf{mit}_*$ does not rely on *fmap*:

$$\mathbf{mit}_*\ \varphi\ (\mathsf{In}_*\ x) = \varphi\ (\mathbf{mit}_*\ \varphi)\ x$$

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

$$\mathbf{mit}_* :: (\forall r.(r \to a) \to (f\ r \to a)) \to \mu_*\ f \to a$$

Under what conditions do $\mathbf{mit}_*$ calls always terminate? Although we defined μ_* as a newtype and $\mathbf{mit}_*$ as a function in Haskell, you should consider them as an information hiding abstraction. The rules of the game (which will be enforced by the language design of Nax) require programmers to construct recursive values using the In_* constructor (as in *zero*, *succ*, *nil*, and *cons*), but forbid programmers from deconstructing those values by pattern matching against In_* (or by using the selector function out_*). Whenever you need to decompose values of recursive datatypes, you must do it via $\mathbf{mit}_*$ (or any of the other terminating Mendler-style combinators). To conform to these rules, all functions over positive recursive datatypes, except the trivial ones such as identity and constant functions (which don't inspect their structure), need to be implemented in terms of the combinators described in Figure 3.2. For negative recursive datatypes only the combinators in the iteration family ensure termination.

The intuitive reasoning behind the termination property of $\mathbf{mit}_*$ for all positive recursive datatypes is that (1) $\mathbf{mit}_*$ strips off one In_* constructor each time it is called and (2) $\mathbf{mit}_*$ only recurses, on the direct subcomponents (e.g., tail of a list) of its argument (because the type of the recursive caller won't allow it to be applied to anything else). Once we observe these two properties, it is obvious that $\mathbf{mit}_*$ always terminates since those properties imply that every recursive call to $\mathbf{mit}_*$

decreases the number of In_* constructors in its argument.[7]

The first property is easy to observe from the definition of $\mathbf{mit}_*$ in Figure 3.3, particularly the pattern matching of the second argument with $(\mathsf{In}_*\ x)$. The second property is enforced by the parametricity in the type of the combining function φ of the $\mathbf{mit}_*$ combinator as shown in Figure 3.2,

In Figure 3.5, we redefine the length function (len_m on the right), this time using a Mendler-style iteration. In the definition of len_m, we name the first argument of φ, which is the recursive caller, as *len*. We use this *len* exactly where we would recursively call the recursive function in general recursion style (*len* on the left).

However, unlike general recursion style, it is not possible to call $len :: r \to Int$ 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: $len\ (C\ x\ xs) = 1 + len\ (C\ x\ xs)$.

We cannot encode such diverging recursion in Mendler style because $len :: r \to Int$ requires its argument to have the parametric type r, while $(C\ x\ xs) :: L\ p\ r$ has a more specific type than r. The parametricity enforces weak structural induction.

The scheme of having the combining function φ abstract over the recursive caller *len* is a powerful one. We will reuse this strategy, generalizing φ to abstract over additional arguments, in order to generalize $\mathbf{mit}_*$ to become more expressive.

3.5 MENDLER-STYLE COURSE-OF-VALUES ITERATION FOR REGULAR DATATYPES

Some computations are not easily expressible by iteration, since iteration only recurses on the direct subcomponents (e.g., tail of a list). Terminating recursion schemes on deeper subcomponents (e.g., tail of a tail of a list) requires rather

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

complex encodings in the conventional setting. Functional programmers often write recursive functions using nested pattern matching that recurse on deeper subcomponents exposed by the nested patterns. A typical example is the Fibonacci function:

$$
\begin{aligned}
&\mathit{fib}\ Z &&= 0\\
&\mathit{fib}\ (S\ Z) &&= 1\\
&\mathit{fib}\ (S\ (S\ m)) &&= \mathit{fib}\ (S\ m) + \mathit{fib}\ m
\end{aligned}
$$

Note in the third equation *fib* recurses on both the predecessor $(S\ m)$, which is a direct subcomponent of the argument, and the predecessor of the predecessor m, which is a deeper subcomponent of the argument. Histomorphism [91] captures such patterns of recursion. Histomorphism is also known as the course-of-values iteration. In conventional style, the course-of-values iteration is defined through a co-algebraic construction of an intermediate stream data structure that pairs up the current argument and the results from the previous steps. There are two ways of implementing this. One is a memoizing bottom-up version and the other is a non-memoizing version that repeats the computation of the previous steps. We are not going to show or discuss those implementations here, but the point we want to make is that both versions need to be implemented through co-algebraic construction [92, 94]. The course-of-values iteration expressed in terms of this co-algebraic construction will look very different from its equivalent in general recursion style. One needs to extract both the original arguments and the deep result values from the stream explicitly calling on stream-head and stream-tail operations. However, in Mendler style, we do not need such co-algebraic construction at least for the non-memoizing version.[8]

[8] The Mendler-style historphism combinators implemented here are the non-memoizing ones. Vene [94] suggests how to implement a memoizing Mendler-style histomorphism, which uses co-algebraic construction.

In the Mendler-style course-of-values iteration (**mcvit**), we play the same trick we played in the Mendler-style iteration (**mit**). We arrange for the combining function to take additional arguments (Figures 3.2 and 3.3).

- The combining function φ now becomes a function of 3 arguments. The first argument is a function that represents an abstract unrolling function that projects out the value embedded inside the data constructor In_* by accessing the projection function out_* given in the definition. As in $\mathbf{mit}_*$, the next argument represents a recursive caller, and the last argument represents the base structure that must be combined into an answer.

  $\mathbf{mcvit}_*\ \varphi\ (\mathsf{In}_*\ x) = \varphi\ \mathsf{out}_*\ (\mathbf{mcvit}_*\ \varphi)\ x$

- Again, we use higher-rank polymorphism to insist that the abstract unrolling function, with type $(r \to f\ r)$, the recursive caller function, with type $(r \to a)$, and the base structure, with type $(f\ r)$, only work over an abstract type, denoted by (r).

  $\mathbf{mcvit}_* :: (\forall r.(a \to f\ a) \to (r \to a) \to (f\ r \to a)) \to \cdots$

The Mendler-style course-of-values iteration is much handier than the conventional course-of-values iteration [92]. For example, in Figure 3.6, the definition of the Fibonacci function in general recursion style (left) and the definition in Mendler style (right) look almost identical, particularly when we have unrolled the nested pattern matching in the general recursive definition into a case expression. The only difference between the two is that in Mendler style (left), we pattern match over *out* n in the case expression, while in general recursion style (right) we pattern match over n.

Let us visually relate the definition of $\mathbf{mcvit}_*$ with the second equation of φ in the definition of the Fibonacci function as follows:

$$\begin{array}{lll} \mathbf{mcvit}_*\ \varphi\ (\mathsf{In}_*\ x) = \varphi\ \mathsf{out}_*\ (\mathbf{mcvit}_*\ \varphi) & x \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \end{array}$$

```
data Nat                          data N r = Z | S r
   = Z                            type Nat = μ* N
   | S Nat                        zero     = In* Z
                                  succ n   = In* (S n)

                                  fib_m = mcvit* φ where
fib Z      = 0                      φ out fib Z      = 0
fib (S n) =                         φ out fib (S n) =
  case n of                             case out n of
    Z    → 1                            Z    → 1
    S n' → fib n + fib n'               S n' → fib n + fib n'
```

Figure 3.6: **mcvit**$_*$ example: Fibonacci function.

```
φ out      fib          (S n) = case out n of
                                    Z    → 1
                                    S n' → fib n + fib n'
```

The abstract unrolling function *out* and the recursive caller *fib* stand for the actual arguments out_* and (**mcvit**$_*$ φ), but the higher-rank type of the combining function φ ensures that they are only used in a safe manner. The abstract unrolling function *out* enables us to discharge In_* as many times as we want inside φ.

From the programmer's perspective, out_* is a hidden primitive, hidden by the **mcvit**$_*$ abstraction (i.e., only used within the definition of combinators such as **mcvit**$_*$ but not in the user-defined functions). But, inside the definition of the combining function φ, the programmer can actually access the functionality of out_* through the abstract unrolling function *out*. The higher-rank types limit the use of this abstract unrolling function *out* to values of type r.

In a positive recursive datatype, the only functions with domain r are the abstract unroller and the recursive caller. The programmer can only *whittle down* the r values inside the base structure, of type $(f\ r)$, into smaller structures, of type

$(f\ r)$. The programmer can then decompose these into even smaller r values by pattern matching against the data constructors of the base structure f. However, there is no way to combine any of these decomposed r values to build up larger r values. The only possible use of the decomposed r values is to call the recursive caller, with type $(r \to a)$.

For example, in Figure 3.6, we pattern match over $(out\ n)$, discharging the hidden In_* constructor of n. Note the types inside the $(S\ n')$ pattern matching branch: $n :: r$; $(out\ n) :: (N\ r)$; and $n' :: r$. What can we possibly do with n and n', of type r? The only possible computation is to call $fib :: r \to Int$ on n and n', as we do in $fib\ n + fib\ n'$. It is a type error to call $fib :: r \to Int$ on either $(S\ n) :: N\ r$ or $(S\ n') :: N\ r$. This is why the termination property of $\mathbf{mcvit}_*$ continues to hold for positive datatypes. In Section 5.3, We discuss further when Mendler-style course-of-values recursion is guaranteed to terminate.

For negative datatypes, however, we have additional functions with domain r. Inside the φ function passed to $\mathbf{mcvit}_*$, the embedded functions with negative occurrences will have type r as their domain. These can be problematic, as shown in Figure 3.7, which contains the counterexample to the termination of $\mathbf{mcvit}_*$. In the following section (Section 3.6), we will discuss why the **mcvit** family fails to guarantee termination for negative datatypes while the **mit** family guarantees termination for arbitrary datatypes including negative datatypes.

3.6 MENDLER-STYLE ITERATION AND COURSE-OF-VALUES ITERATION OVER NEGATIVE DATATYPES

Let us revisit the negative recursive datatype T (from Section 3.1.1) from which we constructed a diverging computation. We can define a Mendler-style version of T, called T_m, as follows:

data $TBase\ r = C_m\ (r \to ())$

type $T_m = \mu_*\ TBase$

If we can write two functions, $p_m :: T_m \to (T_m \to ())$ and $u_m :: T_m \to ()$, corresponding to p and w from Section 3.1.1, we can reconstruct the same diverging computation. The function

$$u_m\ x = (p_m\ x)\ x$$

is easy. The function p_m is problematic. By the rules of the game, we cannot pattern match on In_* (or use out_*) and thus we must resort to using one of the combinators, such as $\mathbf{mit}_*$. However, it is not possible to write p_m in terms of $\mathbf{mit}_*$. Here is an attempt (seemingly the only one possible) that fails:

$$\begin{aligned}
&p_m :: T_m \to (T_m \to ()) \\
&p_m = \mathbf{mit}_*\ \varphi\ \mathbf{where} \\
&\qquad \varphi :: (r \to (T_m \to ())) \to TBase\ r \to (T_m \to ()) \\
&\qquad \varphi\ _\ (C_m\ f) = f
\end{aligned}$$

We write the explicit type signature for the combining function φ (even though the type can be inferred from the type of $\mathbf{mit}_*$) to make it clear why this attempt fails to type check. The combining function φ takes two arguments: the recursive caller (for which we have used the pattern _, since we don't intend to call it) and the base structure $(Cm\ f)$, from which we can extract the function $f :: r \to ()$. Note that r is an abstract type (since it is universally quantified in the function argument), and the result type of φ requires $f :: T_m \to ()$. The types r 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:

$$\begin{aligned}
&pconst :: T_m \to (T_m \to ()) \\
&pconst = \mathbf{mit}_*\ \varphi\ \mathbf{where}\ \varphi\ g\ (C\ f) = const\ ()
\end{aligned}$$

Given the abstract pieces composed of the recursive caller $g :: r \to ()$, the base structure $(C\ f) :: TBase\ r$, and the function we can extract from the base structure $f :: r \to ()$, the only function (modulo extensional equivalence) one is able to write is, in fact, the constant function returning the unit value.

This illustrates the essence of how the Mendler-style iteration guarantees normalization even in the presence of negative occurrences in the recursive datatype 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 the following. Are types with embedded functions with negative occurrences good for anything at all? Can we ever call such functions? A simple example that uses an embedded function inside a negative recursive datatype is illustrated in Figure 3.7. 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 $Foo \to Foo$) contained in *Coo* is a function that transforms a *Foo* value into another *Foo* value. The function *lenFoo*, defined by $\mathbf{mit}_*$, 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 $\mathbf{mit}_*$ for this negative datatype *Foo* is similar to the intuition for positive datatypes. The embedded function $f :: r \to r$ 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 contain f itself. In Section 3.10, we illustrate a general proof for the termination of $\mathbf{mit}_*$ (see Figure 3.23).

While all functions written in terms of $\mathbf{mit}_*$ are total, the same cannot be said of function written in terms of $\mathbf{mcvit}_*$. The function *loopFoo* defined by $\mathbf{mcvit}_*$ is a counterexample to totality, which shows that the Mendler-style course-of-values iteration does not always terminate. Try evaluating *loopFoo foo*. It will loop. This function *loopFoo* is similar to *lenFoo*, but has an additional twist. At the very end of the function definition, we recurse on the transformed tail ($f'\ xs$), when we have more than two elements where the first and second elements are named f and f', respectively. Note f' is an element embedded inside the tail xs. Thus,

$$
\begin{aligned}
&\textbf{data}\ \mathit{FooF}\ r = \mathit{Noo} \mid \mathit{Coo}\ (r \to r)\ r \\
&\textbf{type}\ \mathit{Foo} = \mu_*\ \mathit{FooF} \\
&\mathit{noo} \quad\ = \mathsf{In}_*\ \mathit{Noo} \\
&\mathit{coo}\ f\ \mathit{xs} = \mathsf{In}_*\ (\mathit{Coo}\ f\ \mathit{xs}) \\
&\mathit{lenFoo} :: \mathit{Foo} \to \mathit{Int} \\
&\mathit{lenFoo} = \textbf{mit}_*\ \varphi\ \textbf{where} \\
&\quad \varphi\ \mathit{len}\ \mathit{Noo} \quad\ = 0 \\
&\quad \varphi\ \mathit{len}\ (\mathit{Coo}\ f\ \mathit{xs}) = 1 + \mathit{len}\ (f\ \mathit{xs}) \\
&\mathit{loopFoo} :: \mathit{Foo} \to \mathit{Int} \\
&\mathit{loopFoo} = \textbf{mcvit}_*\ \varphi\ \textbf{where} \\
&\quad \varphi\ \mathit{out}\ \mathit{len}\ \mathit{Noo} \quad\ = 0 \\
&\quad \varphi\ \mathit{out}\ \mathit{len}\ (\mathit{Coo}\ f\ \mathit{xs}) = \textbf{case}\ \mathit{out}\ \mathit{xs}\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad \mathit{Noo} \quad\ \to 1 + \mathit{len}\ (f\ \ \mathit{xs}) \\
&\qquad\qquad\qquad\qquad\qquad \mathit{Coo}\ f'\ \_ \to 1 + \mathit{len}\ (f'\ \mathit{xs}) \\
&\mathit{foo} :: \mathit{Foo} \quad \text{-- loops for } \mathit{loopFoo} \\
&\mathit{foo} = \mathit{coo}_0\ (\mathit{coo}_1\ \mathit{noo})\ \textbf{where}\ \mathit{coo}_0 = \mathit{coo}\ \mathit{id} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{coo}_1 = \mathit{coo}\ \mathit{coo}_0
\end{aligned}
$$

Figure 3.7: An example of a total function *lenFoo* over a negative datatype *Foo* defined by $\textbf{mit}_*$, and a counterexample *loopFoo* illustrating that $\textbf{mcvit}_*$ can diverge for negative datatypes.

$(f'\ xs)$ is dangerous since it applies f' to a larger value xs, which contains f'. The abstract type of the unrolling function $(out :: r \to f\ r)$ prevents the recursive caller from being applied to a larger value, but it does not preclude the risk of embedded functions, with negative domains, being applied to larger values that contain the embedded function itself.

3.7 MENDLER-STYLE ITERATION AND COURSE-OF-VALUES ITERATION OVER NON-REGULAR DATATYPES AND MUTUALLY RECURSIVE DATATYPES

We have discussed the Mendler-style iteration and course-of-values iteration over non-mutually recursive datatypes so far. In this section, we discuss these recursion schemes over non-regular datatypes (Section 3.7.1, Section 3.7.2) and mutually recursive datatypes (Section 3.7.3).

3.7.1 Nested datatypes

The datatypes *Nat* and *List*, defined in Section 3.2, are regular datatypes. Non-recursive datatypes (e.g., *Bool*) and recursive datatypes without any type arguments (e.g., *Nat*) are always regular. Among the recursive datatypes with type arguments, those datatypes where all of the recursive occurrences on the right-hand side have exactly the same type argument as those on the left-hand side (in the same order) are considered regular. For example, the list datatype **data** $List\ p = N \mid C\ p\ (List\ p)$ is regular since $(List\ p)$ appearing on right-hand side takes exactly the same argument p as $(List\ p)$ on the left-hand side (**data** $List\ p = \ldots$).

Note every concrete *instantiation* of the list datatype has an equivalent non-parameterized datatype definition. For instance, *List Bool* is equivalent to the following datatype:

$$\textbf{data}\ List_{Bool} = N_{Bool} \mid C_{Bool}\ Bool\ List_{Bool}$$

This instantiation property does *not* hold for nested datatypes.

Type arguments that never change in any recursive occurrences in a datatype definition are called *type parameters*. Type arguments that do change are called *type indices*. Datatypes with only type parameters are always regular. Nested datatypes [10] are non-regular datatypes where type arguments in some of the recursive occurrences in the recursive datatype equation differ from those on the left-hand side of the datatype equation.

Such types can be expressed in Haskell and ML without using GADT extensions. We introduce two well-known examples of nested datatypes, powerlists and bushes. Functions that sum up the elements in those data structures (Figure 3.8 and Figure 3.9). Nested datatypes require us to move from rank-0 Mendler combinators to rank-1 Mendler combinators.[9]

The powerlist datatype is defined as follows (also in Figure 3.8):

$$\textbf{data}\ Powl\ i = N_P \mid C_P\ i\ (Powl\ (i, i))$$

The type argument (i, i) for $Powl$ occurring on the right-hand side is different from i appearing on the left-hand side. Type arguments that occur in variation on the right-hand side, like i, are type indices.

This single datatype equation for $Powl$ relates to a family of datatypes: the tail of an i-powerlist is a (i, i)-powerlist, its tail is a $((i, i), (i, i))$-powerlist, and so on. More concretely,

$$\begin{aligned} ps &= C_P\ 1\ ps' && :: Powl\ Int \\ ps' &= C_P\ (2, 3)\ ps'' && :: Powl\ (Int, Int) \end{aligned}$$

[9] The rank of a kind is defined by these equations: $\text{rank}(*) = 0$ and $\text{rank}(\kappa \to \kappa') = \max(1 + \text{rank}(\kappa), \text{rank}(\kappa'))$. Rank-0 Mendler combinators work on recursive types of kind $*$, whose rank is 0, constructed from base structures of kind $* \to *$, whose rank is 1. Rank-1 Mendler combinators work on recursive type constructors of kind $* \to *$, whose rank is 1, constructed from base structures of kind $(* \to *) \to (* \to *)$, whose rank is 2. We could have called them rank-1 and rank-2 Mendler combinators, matching the rank of the base structure, instead of the rank of the recursive type constructor, but just happen to prefer counting from 0.

$ps'' = C_P\ ((4,5),(6,7))\ N_P :: Powl\ ((Int, Int), (Int, Int))$

The tail of ps is ps', and the tail of ps' is ps''. Note that the shape of elements includes deeper nested pairs as the type indices become more deeply nested.

On the left-hand side of Figure 3.8, we define a function that sums up all the nested elements in a powerlist using general recursion style. This function takes 2 parameters: a function that turns elements into integers and the powerlist itself. The key part in the definition of *psum* is constructing the function $(\lambda(x, y) \rightarrow f\ x + f\ y) :: (i, i) \rightarrow Int$. We must construct this function, on the fly, in order to make the recursive call of *psum* on its tail $xs :: Powl\ (i, i)$. Without this function, the recursive call wouldn't know how to sum up paired elements.

We can specialize *psum*, for instance, for integer powerlists as follows by supplying the identity function:

$sumP :: Powl\ Int \rightarrow Int$

$sumP\ xs = psum\ xs\ id$

Using *sumP*, we can sum up *ps* defined above: $sumP\ ps \rightsquigarrow 28$.

$$\textbf{data}\ Powl\ i = N_P \mid C_P\ i\ (Powl\ (i,i))$$

$$\begin{aligned}
&psum :: Powl\ i \to (i \to Int) \to Int\\
&psum\ N_P \qquad\quad = \lambda f \to 0\\
&psum\ (C_P\ x\ xs) = \lambda f \to\\
&\qquad f\ x + psum\ xs\ (\lambda(x,y) \to f\ x + f\ y)
\end{aligned}$$

$$\textbf{newtype}\ Ret\ i = Ret\ \{\, unRet :: (i \to Int) \to Int\}$$

$$\begin{aligned}
&psum' :: Powl\ i \to Ret\ i\\
&psum'\ N_P \qquad\quad = Ret\ (\lambda f \to 0)\\
&psum'\ (C_P\ x\ xs) = Ret\ (\lambda f \to\\
&\qquad f\ x + psum''\ xs\ (\lambda(x,y) \to f\ x + f\ y))\\
&\quad \textbf{where}\ psum'' :: Powl\ i \to (i \to Int) \to Int\\
&\qquad psum'' = unRet \circ psum'
\end{aligned}$$

$$\textbf{data}\ PowlF\ r\ i = N_P \mid C_P\ i\ (r\ (i,i))$$

$$\textbf{type}\ Powl\ i = \mu_{*\to *}\ PowlF\ i$$

$$\begin{aligned}
&nil_P \qquad\quad = \mathsf{In}_{*\to *}\ N_P\\
&cons_P\ x\ xs = \mathsf{In}_{*\to *}\ (C_P\ x\ xs)
\end{aligned}$$

$$\begin{aligned}
&psum :: Powl\ i \to (i \to Int) \to Int\\
&psum = unRet \circ psum_m
\end{aligned}$$

$$\begin{aligned}
&psum_m :: Powl\ i \to Ret\ i\\
&psum_m = \textbf{mit}_{*\to *}\ \varphi\ \textbf{where}\\
&\quad \varphi :: \forall r\ i'.(\forall i.r\ i \to Ret\ i) \to PowlF\ r\ i' \to Ret\ i'\\
&\quad \varphi\ psum'\ N_P \qquad\quad = Ret\ (\lambda f \to 0)\\
&\quad \varphi\ psum'\ (C_P\ x\ xs) = Ret\ (\lambda f \to\\
&\qquad\qquad f\ x + psum''\ xs\ (\lambda(x,y) \to f\ x + f\ y))\\
&\qquad\quad \textbf{where}\ psum'' :: r\ i \to (i \to Int) \to Int\\
&\qquad\qquad psum'' = unRet \circ psum'
\end{aligned}$$

Figure 3.8: Summing up a powerlist ($Powl$), a nested datatype, expressed in terms of $\textbf{mit}_{*\to *}$.

$$\textbf{data}\ \mathit{Bush}\ i = N_B \mid C_B\ i\ (\mathit{Bush}\ (\mathit{Bush}\ i))$$

$$\begin{aligned}
&\mathit{bsum} :: \mathit{Bush}\ i \to (i \to \mathit{Int}) \to \mathit{Int}\\
&\mathit{bsum}\ N_B \qquad\quad = (\lambda f \to 0)\\
&\mathit{bsum}\ (C_B\ x\ xs) = (\lambda f \to\\
&\qquad\qquad f\ x + \mathit{bsum}\ xs\ (\lambda ys \to \mathit{bsum}\ ys\ f))
\end{aligned}$$

$$\textbf{newtype}\ \mathit{Ret}\ i = \mathit{Ret}\ \{\mathit{unRet} :: (i \to \mathit{Int}) \to \mathit{Int}\}$$

$$\begin{aligned}
&\mathit{bsum}' :: \mathit{Bush}\ i \to \mathit{Ret}\ i\\
&\mathit{bsum}'\ N_B \qquad\quad = \mathit{Ret}\ (\lambda f \to 0)\\
&\mathit{bsum}'\ (C_B\ x\ xs) = \mathit{Ret}\ (\lambda f \to\\
&\qquad\qquad f\ x + \mathit{bsum}''\ xs\ (\lambda ys \to \mathit{bsum}''\ ys\ f))\\
&\qquad \textbf{where}\ \mathit{bsum}'' :: \mathit{Bush}\ i \to (i \to \mathit{Int}) \to \mathit{Int}\\
&\qquad\qquad \mathit{bsum}'' = \mathit{unRet} \circ \mathit{bsum}'
\end{aligned}$$

$$\textbf{data}\ \mathit{BushF}\ r\ i = N_B \mid C_B\ i\ (r\ (r\ i))$$

$$\textbf{type}\ \mathit{Bush}\ i = \mu_{*\to *}\ \mathit{BushF}\ i$$

$$\begin{aligned}
&\mathit{nil}_B \qquad\quad = \mathsf{In}_{*\to *}\ N_B\\
&\mathit{cons}_B\ x\ xs = \mathsf{In}_{*\to *}\ (C_B\ x\ xs)
\end{aligned}$$

$$\begin{aligned}
&\mathit{bsum} :: \mathit{Bush}\ i \to (i \to \mathit{Int}) \to \mathit{Int}\\
&\mathit{bsum} = \mathit{unRet} \circ \mathit{bsum}_m
\end{aligned}$$

$$\begin{aligned}
&\mathit{bsum}_m :: \mathit{Bush}\ i \to \mathit{Ret}\ i\\
&\mathit{bsum}_m = \textbf{mit}_{*\to *}\ \varphi\ \textbf{where}\\
&\quad \varphi :: \forall r\ i'.(\forall i.r\ i \to \mathit{Ret}\ i) \to \mathit{BushF}\ r\ i' \to \mathit{Ret}\ i'\\
&\quad \varphi\ \mathit{bsum}'\ N_B \qquad\quad = \mathit{Ret}\ (\lambda f \to 0)\\
&\quad \varphi\ \mathit{bsum}'\ (C_B\ x\ xs) = \mathit{Ret}\ (\lambda f \to\\
&\qquad\qquad f\ x + \mathit{bsum}''\ xs\ (\lambda ys \to \mathit{bsum}''\ ys\ f))\\
&\qquad \textbf{where}\quad \mathit{bsum}'' :: r\ i \to (i \to \mathit{Int}) \to \mathit{Int}\\
&\qquad\qquad\quad \mathit{bsum}'' = \mathit{unRet} \circ \mathit{bsum}'
\end{aligned}$$

Figure 3.9: Summing up a bush (*Bush*), a recursively nested datatype, expressed in terms of $\textbf{mit}_{*\to *}$.

Before discussing the Mendler-style version, let us take a look at yet another general recursive version of the function $psum'$, which explicitly wraps up the answer values of type $(i \to Int) \to Int$ inside the newtype $Ret\ i$. The relations between the plain vanilla version and the wrapped up version are simply:

$$psum = unRet \circ psum'$$
$$Ret \circ psum = psum'$$

The wrapped up version $psum'$ has the same structure as the Mendler-style version $psum_m$ found on the right-hand side of Figure 3.8. The wrapping of the answer type is for purely technical reasons: to avoid the need for higher-order unification. If we were to work with the unwrapped answer type in Mendler style, the type system would need to unify $(a\ i)$ with $((i \to Int) \to Int)$, which is a higher-order unification, whereas unifying $(a\ i)$ with the wrapped answer type $(Ret\ i)$ is first-order. The type inference algorithm of Haskell (and most other languages) does not support higher-order unification.[10]

The summation function for powerlists in Mendler style is illustrated on the right-hand side in Figure 3.8. First, we give two-level datatype definitions for powerlists. As usual, we define the datatype $Powl$ as a fixpoint of the base $PowlF$. However, an important difference that readers should notice is the use of fixpoint $\mu_{*\to *}$ at kind $* \to *$ bases, instead of μ_*, for the kind $*$ bases inducing regular datatypes. Since we used $\mu_{*\to *}$ to define the recursive datatype, we use $\mathbf{mit}_{*\to *}$, the Mendler-style iteration combinator at kind $* \to *$, to define the function $psum_m$.

The beauty of the Mendler-style approach is that the implementations of the recursion combinators for higher-ranks (or higher-kinds) are *exactly the same* as those for their kind $*$ counterparts. The definitions differ only in their type signatures. As you can see in Figures 3.2 and 3.3, $\mathbf{mit}_{*\to *}$ has a richer type than $\mathbf{mit}_*$,

[10] We may avoid higher-order unification, either by making the Mendler-style combinators language constructs (rather than functions) so that the type system treats them with specialized typing rules or by providing a version of the combinators with syntactic Kan-extension as in [5].

but their implementations are *exactly the same*! This is not the case for the conventional approach. The definition of **cata** won't generalize to nested datatypes in a trivial way. There have been several approaches [11, 49, 57] to extend folds or catamorphisms for nested datatypes in the conventional setting.

We can also define a summation function for bushes in a similar way as the summation function for powerlists. The bush datatype is defined as below (also in Figure 3.9):

$$\textbf{data}\ \mathit{Bush}\ i = N_B \mid C_B\ i\ (\mathit{Bush}\ (\mathit{Bush}\ i))$$

The type argument i for *Bush* is a type index, since the type argument $(\mathit{Bush}\ i)$ occurring on the right-hand side is different from i appearing on the left-hand side. What is intriguing about *Bush* is that the variation of the type index involves itself. Matthes [61] calls such datatypes as *Bush*, *truly nested datatypes*. Here are some examples of bush values:

$$\begin{aligned}
bs &= C_B\ 1\ bs' && :: \mathit{Bush}\ \mathit{Int}\\
bs' &= C_B\ (C_B\ 2\ N_B)\ bs'' && :: \mathit{Bush}\ (\mathit{Bush}\ \mathit{Int})\\
bs'' & && :: \mathit{Bush}\ (\mathit{Bush}\ (\mathit{Bush}\ \mathit{Int}))\\
bs'' &= C_B\ (C_B\ (C_B\ 3\ N_B)\ (C_B\ (C_B\ (C_B\ 4\ N_B)\ N_B)\ N_B))\ N_B
\end{aligned}$$

The tail of bs is bs' and the tail of bs' is bs''. Note that the shape of the elements becomes more deeply nested as we move towards the latter elements. More interestingly, the element type of the bush becomes nested by the bush type itself.

We can define a function that sums up all the nested elements in a bush. Let us first take a look at the function *bsum* in general recursion style, on the left-hand side of Figure 3.9. This function takes 2 parameters: a bush to sum up and a function that turns elements into integers. The key part in the definition of *bsum* is constructing the function $(\lambda ys \rightarrow \mathit{bsum}\ ys\ f) :: \mathit{Bush}\ i \rightarrow \mathit{Int}$. We must construct this function, on the fly, in order to make the recursive call of *bsum* on its tail $xs :: \mathit{Bush}\ (\mathit{Bush}\ i)$. Without this function, the recursive call wouldn't know how to sum up the bushed elements.

We can specialize *bsum*, for instance, for integer bushes as follows by supplying the identity function:

$$sumB :: Bush\ Int \to Int$$
$$sumB\ xs = bsum\ xs\ id$$

Using *sumB*, we can sum up *bs* defined above: $sumB\ bs \rightsquigarrow 10$.

Before discussing the Mendler-style version, let us take a look at yet another general recursive version of the function $bsum'$ that explicitly wraps up the answer values of type $(i \to Int) \to Int$ inside the newtype $Ret\ i$. The relations between the plain vanilla version and the wrapped up version are simply:

$$bsum = unRet \circ bsum'$$
$$Ret \circ bsum = bsum'$$

The wrapped up version $bsum'$ has the same structure as the Mendler-style version $bsum_m$ found on the right-hand side of Figure 3.9. In Mendler style, we define the datatype *Bush* as a fixpoint ($\mu_{*\to *}$) of the base *BushF* and define $bsum_m$ in terms of $\mathbf{mit}_{*\to *}$, similar to the definition of the summation function for powerlists in Mendler style.

The type argument i in both $Powl\ i$ and $Bush\ i$ is a type index that forces us to choose the fixpoint on kind $* \to *$ (and its related recursion combinators). Note in the definition of the base types *PowlF* and *BushF*, we place the index i after the type argument r for the recursion points. This is the convention we use. We always write parameters (p), before the recursion point argument (r), followed by indices (i). Figure 3.10, which we will shortly discuss in Section 3.7.2, contains an example where there are both type parameters and type indices in a datatype ($Vec\ p\ i$).

3.7.2 Indexed datatypes (GADTs)

A recent popular extension to the GHC Haskell compiler is GADTs [84]. In our nested examples, the variation of type indices always occurred in the arguments of

the data constructors. GADTs are indexed datatypes, where the index may vary in the result types of the data constructors. Haskell's normal **data** declaration, which uses an "equation" syntax, makes the assumption that the result types of every constructor are the "same" type with no variation. GHC's GADT datatype extension is more expressive than the usual **data** declaration in equational form. The GHC compiler extends the *datatype* syntax, so that each datatype constructor is given its full type annotation. The datatype definition for vectors (or size-indexed lists) is a prime example:

data $Vec\ p\ i$ **where**

$N_V :: Vec\ p\ Z$

$C_V :: p \rightarrow Vec\ p\ i \rightarrow Vec\ p\ (S\ i)$

Note the indices[11] vary in the *result types* of the data constructors: Z in the type of N_V and $(S\ i)$ in the type of C_V.

Nested datatypes, which we discussed earlier, are a special case of indexed datatypes that happened to be expressible within the recursive type equation syntax of Haskell, because the indices only vary in the recursive arguments of the data constructors, but not in the result type. For a clearer comparison, we express the bush datatype in GADT syntax as follows:[12]

data $Bush\ i$ **where**

$NB :: Bush\ i$

$CB :: i \rightarrow Bush\ (Bush\ i) \rightarrow Bush\ i$

Note, the type argument varies in the second argument of CB, which is $Bush\ (Bush\ i)$, but both the result types of NP and CP are $Bush\ i$.

In Figure 3.10, we define the vector datatype Vec as $\mu_{*\rightarrow *}\ (V\ p)\ i$, in Mendler style. That is, we apply $\mu_{*\rightarrow *}$ to the partial application of the base V to the

[11]The Z and S used in Vec are type-level representations of natural numbers, which are empty types that are not inhabited by any value. They are only intended to be used as indices.

[12]We can translate any recursive type equation into a definition using the GADT syntax since GADTs are indeed *generalized* algebraic datatypes.

parameter p, and then apply the resulting fixpoint to the index i. The base datatype $V\ p\ r\ i$ is a GADT with a parameter p and an index i. Recall that by convention we place the parameter p before the type argument r for recursion points, followed by the index i. We can express the *copy* function that traverses a given vector and reconstructs that vector with the same elements, in Mendler style, using the Mendler-style iteration combinator $\mathbf{mit}_{*\to*}$ at kind $* \to *$. We can express the *switch2* function that switches every two elements of the given vector, in Mendler style, using the course-of-values iteration combinator $\mathbf{mcvit}_{*\to*}$ at kind $* \to *$. The definitions of $\mathbf{mit}_{*\to*}$ and $\mathbf{mcvit}_{*\to*}$ are exactly the same as the definitions of $\mathbf{mit}_*$ and $\mathbf{mcvit}_*$, except that $\mathbf{mit}_{*\to*}$ and $\mathbf{mcvit}_{*\to*}$ have richer type signatures (see Figures 3.2 and 3.3). Thus, defining functions using $\mathbf{mit}_{*\to*}$ and $\mathbf{mcvit}_{*\to*}$ is no more complicated than defining functions for regular datatypes using $\mathbf{mit}_*$ and $\mathbf{mcvit}_*$. The one proviso to this statement is that we need to give explicit type signatures for φ because GHC does not support type inference for higher-rank types (i.e., types with inner $\forall$s that are not top-level). Again, in a language where Mendler-style combinators were language constructs rather than functions, we believe this annoying burden could be lifted.

3.7.3 Mutually recursive datatypes

We can express mutual recursion over mutually recursive datatypes in Mendler style using an indexed base datatype. The context extension function *extend* and the expression evaluation function *eval* in Figure 3.11 are mutually recursive functions over the mutually recursive datatypes of declaration *Dec* and expression *Exp*. The general recursive version on the left-hand side of Figure 3.11 is a self-explanatory standard evaluator implementation for the expression.

To express this in Mender style (right), we first define the common base *DecExpF*, which is indexed by D and E. Note the data constructors of *DecExpF* include the data constructors of declarations (*Def*) and expressions (*Var*, *Val*, *Add*, and *Let*).

```
data Z
data S i
data Vec p i where
  N_V :: Vec p Z
  C_V :: p → Vec p i → Vec p (S i)

copy :: Vec p i → Vec p i
copy N_V        = N_V
copy (C_V x xs) = C_V x (copy xs)

switch2 :: Vec p i → Vec p i
switch2 N_V        = N_V
switch2 (C_V x xs) =
  case xs of
    N_V      → C_V x N_V
    C_V y ys → C_V y (C_V x (switch2 ys))
```

```
data V p r i where
  N_V :: V p r Z
  C_V :: p → r i → V p r (S i)
type Vec p i = μ_{*→*} (V p) i
nil_V        = In_{*→*} N_V
cons_V x xs = In_{*→*} (C_V x xs)

copy :: Vec p i → Vec p i
copy = mit_{*→*} φ where
  φ :: (∀i.r i → Vec p i) → V p r i → Vec p i
  φ cp N_V        = nil_V
  φ cp (C_V x xs) = cons_V x (cp xs)

switch2 :: Vec p i → Vec p i
switch2 = mcvit_{*→*} φ where
  φ  :: (∀i.r i → V p r i) →
        (∀i.r i → Vec p i) → V p r i → Vec p i
  φ out sw2 N_V        = nil_V
  φ out sw2 (C_V x xs) =
    case out xs of
      N_V      → cons_V x nil_V
      C_V y ys → cons_V y (cons_V x (sw2 ys))
```

Figure 3.10: Recursion (*copy*) and course-of-values recursion (*switch2*) over size-indexed lists (*Vec*) expressed in terms of $\mathbf{mit}_{*\to*}$ and $\mathbf{mcvit}_{*\to*}$.

```
type Name = String
type Env = [(Name, Int)]

data Dec = Def Name Exp
data Exp = Var Name
         | Val Int
         | Add Exp Exp
         | Let Dec Exp
```

```
extend :: Dec → Env → Env
extend (Def x e) =
       λσ → (x, eval e σ) : σ

eval :: Exp → Env → Int
eval   (Var x)     =
       λσ → fromJust (lookup x σ)
eval   (Val v)     =
       λσ → v
eval   (Add e1 e2) =
       λσ → eval e1 σ + eval e2 σ
```

```
data D
data E

data DecExpF (r :: * → *) (i :: *) where
  Def :: Name → r E → r D → DecExpF r D
  Var :: Name             → DecExpF r E
  Val :: Int              → DecExpF r E
  Add :: r E → r E        → DecExpF r E
  Let :: r D → r E        → DecExpF r E

type Dec = μ(*→*) DecExpF D
type Exp = μ(*→*) DecExpF E

data family Ret i :: *
newtype instance Ret D = Ret_D (Env → Env)
newtype instance Ret E = Ret_E (Env → Int)

π_D  f = λx → case f x of Ret_D f_D → f_D
π_E  f = λx → case f x of Ret_E f_E → f_E

extev :: μ(*→*) DecExpF i → Ret i
extev = mit(*→*) φ where
  φ :: (∀i.r i → Ret i) → DecExpF r i → Ret i
  φ f (Def x e)  =
      Ret_D $ λσ → (x, ev e σ) : σ
                 where ev = π_E f
  φ f (Var x)    =
      Ret_E $ λσ → fromJust (lookup x σ)
  φ f (Val v)    =
      Ret_E $ λσ → v
  φ f (Add e1 e2) =
      Ret_E $ λσ → ev e1 σ + ev e2 σ
                 where ev = π_E f

extend :: Dec → Env → Env
extend = π_D extev

eval   :: Exp → Env → Int
eval   = π_E extev
```

Figure 3.11: Mutual recursion (*extend* and *eval* over *Dec* and *Exp*) expressed in terms of $\mathbf{mit}_{*\to*}$ over an indexed datatype *DecExpF*.

The data constructor for declarations is indexed by D and the other data constructors for expressions are indexed by E in their result types. Then, we can define *Dec* as $\mu_{*\to *}\ DecExpF\ D$ and *Exp* as $\mu_{*\to *}\ DecExpF\ E$. We wrap up the return types of the *eval* and *extend* functions with the data family *Ret*, for reasons similar to the return types of the summation functions in Section 3.7.1. We also define the projection functions $\pi_D :: Ret\ D \to (Env \to Env)$ and $\pi_E :: Ret\ E \to (Env \to Int)$ to open up the return type. Then, we can express the mutually recursive functions, both *eval* and *extend*, combined in one function definition *extev* using $\mathbf{mit}_{*\to *}$. You can observe that the definition of φ is very close to the definitions of the general recursive versions of *extend* and *eval* on the left. The difference is that we project out *ev* from f, which is the handle for the combined mutually recursive function, when we need to call the evaluation function for the recursion. Once we have defined the combined function *extev*, we can project out *extend* and *eval* using π_D and π_E.

3.8 MENDLER-STYLE PRIMITIVE RECURSION (mpr)

In Figure 3.12, we list type declarations and defining equations of several families of the Mendler-style recursion combinators. We give two versions for each family, one at kind $*$ and one at kind $* \to *$. The families of combinators increase in complexity from iteration (**mit**), through primitive recursion (**mpr**) and course-of-values iteration (**mcvit**), to course-of-values recursion (**mcvpr**). We saw **mit** and **mcvit** in the previous sections.

The Mendler-style primitive recursion family (**mpr**), when compared to the **mit** family, has an additional abstract operation, which we call *cast*. The *cast* operation explicitly converts a value of the abstract recursive type (r) into a value of the concrete recursive type ($\mu_*\ t$).

Similarly, the Mendler-style course-of-values recursion family (**mcvpr**), when compared to the **mcvit** family, also has an additional *cast* operation.

$$
\begin{array}{lll}
 & & \text{out} \qquad\qquad\qquad\qquad \text{cast} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \overset{\text{abstract}}{\underset{\text{recursive call}}{}} \\
\mathbf{mit}_{*} & :: (\forall r\ .\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\quad r\quad \to a\ ) \to (f\ r\quad \to a\ )) \to (\mu_{*}\ f\quad\ \to a\ ) \\
\mathbf{mit}_{*\to *} & :: (\forall r\ i. & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\forall i.r\ i \to a\ i) \to (f\ r\ i \to a\ i)) \to (\mu_{*\to *}\ f\ i \to a\ i) \\
\mathbf{mpr}_{*} & :: (\forall r\ .\ & \qquad\qquad\qquad\qquad (\quad r\quad \to \mu_{*}\ f\quad\ ) \to (\quad r\quad \to a\ ) \to (f\ r\quad \to a\ )) \to (\mu_{*}\ f\quad\ \to a\ ) \\
\mathbf{mpr}_{*\to *} & :: (\forall r\ i. & \qquad\qquad\qquad\qquad (\forall i.r\ i \to \mu_{*\to *}\ f\ i) \to (\forall i.r\ i \to a\ i) \to (f\ r\ i \to a\ i)) \to (\mu_{*\to *}\ f\ i \to a\ i) \\
\mathbf{mcvit}_{*} & :: (\forall r\ .(\quad r\quad \to f\ r\ ) \to & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\quad r\quad \to a\ ) \to (f\ r\quad \to a\ )) \to (\mu_{*}\ f\quad\ \to a\ ) \\
\mathbf{mcvit}_{*\to *} & :: (\forall r\ i.(\forall i.r\ i \to f\ r\ i) \to & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\forall i.r\ i \to a\ i) \to (f\ r\ i \to a\ i)) \to (\mu_{*\to *}\ f\ i \to a\ i) \\
\mathbf{mcvpr}_{*} & :: (\forall r\ .(\quad r\quad \to f\ r\ ) \to & \qquad\qquad\qquad\qquad (\quad r\quad \to \mu_{*}\ f\quad\ ) \to (\quad r\quad \to a\ ) \to (f\ r\quad \to a\ )) \to (\mu_{*}\ f\quad\ \to a\ ) \\
\mathbf{mcvpr}_{*\to *} & :: (\forall r\ i.(\forall i.r\ i \to f\ r\ i) \to & \qquad\qquad\qquad\qquad (\forall i.r\ i \to \mu_{*\to *}\ f\ i) \to (\forall i.r\ i \to a\ i) \to (f\ r\ i \to a\ i)) \to (\mu_{*\to *}\ f\ i \to a\ i)
\end{array}
$$

$$
\begin{array}{lllll}
\mathbf{mit}_{*} & \varphi\ (\mathsf{In}_{*}\quad x) = \varphi & & (\mathbf{mit}_{*}\quad\ \varphi)\ x \\
\mathbf{mit}_{*\to *} & \varphi\ (\mathsf{In}_{*\to *}\ x) = \varphi & & (\mathbf{mit}_{*\to *}\quad \varphi)\ x \\
\mathbf{mpr}_{*} & \varphi\ (\mathsf{In}_{*}\quad x) = \varphi & id\ (\mathbf{mpr}_{*}\quad\ \varphi)\ x \\
\mathbf{mpr}_{*\to *} & \varphi\ (\mathsf{In}_{*\to *}\ x) = \varphi & id\ (\mathbf{mpr}_{*\to *}\quad \varphi)\ x \\
\mathbf{mcvit}_{*} & \varphi\ (\mathsf{In}_{*}\quad x) = \varphi\ \mathsf{out}_{*} & (\mathbf{mcvit}_{*}\quad\ \varphi)\ x \\
\mathbf{mcvit}_{*\to *} & \varphi\ (\mathsf{In}_{*\to *}\ x) = \varphi\ \mathsf{out}_{*\to *} & (\mathbf{mcvit}_{*\to *}\quad \varphi)\ x \\
\mathbf{mcvpr}_{*} & \varphi\ (\mathsf{In}_{*}\quad x) = \varphi\ \mathsf{out}_{*} & id\ (\mathbf{mcvpr}_{*}\quad\ \varphi)\ x \\
\mathbf{mcvpr}_{*\to *} & \varphi\ (\mathsf{In}_{*\to *}\ x) = \varphi\ \mathsf{out}_{*\to *} & id\ (\mathbf{mcvpr}_{*\to *}\quad \varphi)\ x
\end{array}
$$

Figure 3.12: The Mendler-style primitive recursion (**mpr**) and the Mendler-style course-of-values recursion (**mcvpr**) at kinds $*$ and $* \to *$, in comparison with **mit** and **mcvit**.

```
data Nat                          data N r = Z | S r
   = Zero                         type Nat = μ* N
   | Succ Nat                     zero    = In* Z
                                  succ n = In* (S n)

                                  factorial = mpr* φ where
fac Zero      = Succ Zero            φ cast fac Z      = succ zero
fac (Succ n) = times (Succ n) (fac n)   φ cast fac (S n) = times (succ (cast n)) (fac n)
```

Figure 3.13: $\mathbf{mpr}_*$ example: factorial function.

Since **mpr** has an additional abstract operation, when compared to **mit**, it can express all the functions expressible with **mit**. In some programs, the additional *cast* operation can increase the efficiency of the program by supporting constant time access to the concrete value of the recursive component.

A typical example of primitive recursion is the factorial function. Figure 3.13 illustrates the general recursive version (right) and the Mendler-style version (left) of the factorial function, where $times :: Nat \to Nat \to Nat$ is the usual multiplication operation on natural numbers. Note the definition of φ in Mendler style is similar to the definition of *fac* in the general recursive version, except that it uses the explicit *cast* to convert from an abstract value $(n : r)$ to a concrete value $(cast\ n : Nat)$.

The primitive recursion family also enables programmers to define non-recursive functions, such as a constant time predecessor for natural numbers (Figure 3.14) and a constant time tail function for lists (Figure 3.15). Although it is possible to implement *factorial*, *pred*, and *tail* in terms of **mit**, those implementations will be less efficient. The time complexity of *factorial* in terms of iteration will be quadratic in the size of the input rather than being linear. The time complexity of *pred* and *tail* in terms of iteration will be linear in the size of the input rather

than being constant.

The course-of-values recursion family can be defined by adding the *out* operation to the **mpr** family, as is shown in Figure 3.12, just as the **mcvit** family can be defined by adding the *out* operation to **mit**. The **mcvpr** family is only guaranteed to terminate for positive datatypes, for the same reason that the **mcvit** family is only guaranteed to terminate for positive datatypes (recall Figure 3.7).

A simple variation of the Fibonacci function, shown in Figure 3.16, is an example of a course-of-values recursion. The Fibonacci function *fib* and the Lucas function *luc* satisfy the following recurrence relations:[13]

$$\begin{aligned} \mathit{fib}\ (n+2) &= \mathit{fib}\ (n+1) + \mathit{fib}\ n \\ \mathit{luc}\ (n+2) &= \mathit{luc}\ (n+1) + \mathit{luc}\ n + n \end{aligned}$$

Note the trailing "$\cdots + n$" in the recurrence relation for *luc*. We need the ability of the course-of-values recursion because n is a deep recursive component of $n+2$ (i.e., n is the predecessor of the predecessor of $n+2$). We need primitive recursion, since we not only perform a recursive call over n ($\cdots + \mathit{luc}\ n + \cdot s$), but also add the value of n itself ($\cdots + n$). The **mcvpr** family provides both *out* and *cast* operations for accessing deep recursive components and casting from an abstract value to a concrete recursive value.

It is strongly believed that the primitive recursion family cannot be embedded in F_ω in a reduction preserving manner, since it is known that induction is not derivable from second-order dependent calculi [39]. As we mentioned in Section 3.1.2, the termination properties of Mendler-style primitive recursion can be shown by embedding **mpr** into Fix_ω [3] (also described in Section 5.2). Additionally, we discovered how to embed **mcvpr** within Fix_ω. However, our embedding of

[13] The *luc* function in Figure 3.16 is slightly different from the original version of Lucas numbers. What *luc n* implements is the function $\mathit{Lucas}(n+1)-(n+1)$, where *Lucas* is the original definition of the Lucas number. Mathematically, Lucas numbers are just a Fibonacci sequence with different base values. They can be understood as a Fibonacci number offset by linear term. For instance, *luc* can be turned into a Fibonacci function via change of variable by $\mathit{fib}\ n = \mathit{luc}\ n + n + 1$.

```
pred Zero     = Zero
pred (Succ n) = n
```

```
pred = mpr* φ where
  φ cast pr Z     = zero
  φ cast pr (S n) = cast n
```

Figure 3.14: $\mathbf{mpr}_*$ example (non-recursive): a constant time predecessor.

```
data List a
  = Nil
  | Cons a (List a)

tail Nil         = Nil
tail (Cons x xs) = xs
```

```
data L a r = N | C a r
type List a = μ* (L a)
nil       = In* N
cons x xs = In* (C x xs)

tail = mpr* φ where
  φ cast tl N       = nil
  φ cast tl (C x xs) = cast xs
```

Figure 3.15: $\mathbf{mpr}_*$ example (non-recursive): a constant time tail function for lists.

```
luc Zero     = Zero
luc (Succ n) =
  case n of
    Zero    → Succ Zero
    Succ n' → plus (plus (luc n) (luc n'))
                   n'
```

```
lucas = mcvpr* φ where
  φ out cast luc Z     = zero
  φ out cast luc (S n) =
    case out n of
      Z    → succ zero
      S n' → plus (plus (luc n) (luc n'))
                  (cast n')
```

Figure 3.16: Lucas number (`http://oeis.org/A066982`) example illustrating the use of the $\mathbf{mcvpr}_*$ family.

mcvpr into Fix_ω (or Fix_i) is not reduction preserving. We will explain the details of the embedding of **mcvpr** into Fix_ω in Section 5.3.

3.9 MENDLER-STYLE ITERATION WITH SYNTACTIC INVERSES

While it is known that iteration and primitive recursion terminate for all types [3, 5], they are not particularly expressive over negative recursive types. Identifying additional Mendler-style operators that work naturally, and are more expressive than iteration, is one of the important results of this dissertation.

Interesting examples of Mendler-style operators over negative recursive types have been neglected in the literature. One of the reasons, we think, is because it is often possible to encode negative recursive types into positive recursive ones (e.g., [19]). Because conventional iteration and primitive recursion normalize for positive recursive types, one can use standard techniques on these encodings, which are translations of negative recursive types into positive recursive types. What we gain by using such encodings must be traded off against the loss in transparency that such encodings force upon the implementation. The natural structures, which were evident in the negative datatype, become obscured by such encodings.

A series of papers [30, 32, 63, 74, 96] has studied techniques that define recursion schemes directly over negative recursive types in the conventional setting. In our recent paper [6], we discovered that iteration over negative recursive types can be naturally captured as a kind-indexed family of Mendler style combinator. The **msfit** combinator (a.k.a. *msfcata*) at kind $*$ corresponds to the conventional recursion combinator discovered by Fegaras and Sheard [32] and later refined by Washburn and Weirich [96]. With this new **msfit** family, we were able to write many interesting programs, involving negative recursive types that may be impossible, or very unnatural, to write with the ordinary Mendler-style iteration family (**mit**, a.k.a. *mcata*).

3.9.1 Formatting HOAS

To lead up to the Mendler-style solution to formatting HOAS, we first review some earlier work on turning HOAS expressions into strings. This solution was suggested by Fegaras and Sheard [32]. They were studying yet another abstract recursion scheme described by Paterson [74] and Meijer and Hutton [63] that could only be used if the combining function had a true inverse. This seemed a bit limiting, so Fegaras and Sheard introduced the idea of a syntactic inverse. The syntactic inverse was realized by augmenting the μ_* type with a second constructor. This augmented μ_* had the same structure as $\breve{\mu}_*$ in Figure 3.1, but with a different type.

The algorithm works, but the augmentation introduces junk. Washburn and Weirich [96] eliminated the junk by exploiting parametricity. It is a coincidence that Mendler-style recursion combinators 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.

A general recursive implementation for open HOAS

The recursive datatype Exp_g in Figure 3.17 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 $(Exp_g \to Exp_g)$. 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:

$$
\begin{aligned}
k_g &= Lam_g\ (\lambda x \to Lam_g\ (\lambda y \to x)) \\
s_g &= Lam_g\ (\lambda x \to Lam_g\ (\lambda y \to Lam_g\ (\lambda z \to App_g\ (App_g\ x\ z)\ (App_g\ y\ z)))) \\
w_g &= Lam_g\ (\lambda x \to App_g\ x\ x) \\
skk_g &= App_g\ (App_g\ s_g\ k_g)\ k_g
\end{aligned}
$$

```
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) = λvs       → "(" ++ show' x vs ++ "  " ++ show' y vs ++ ")"
        show' (Lam_g z)   = λ(v : vs) → "(\\" ++ v ++ "->" ++ show' (z (Var_g v)) vs ++ ")"
        show' (Var_g v)   = λvs       → v

data ExpF r = Lam (r → r) | App r r
type Exp' a = ŭ_* ExpF a
type Exp = ∀a.Exp' a
  -- lam :: (∀a.Exp' a → Exp' a) → Exp
lam e     = lĭn_* (Lam e)
  -- app :: Exp → Exp → Exp
app f e   = lĭn_* (App f e)
showExp :: Exp → String
showExp e = msfit_* φ e vars
  where φ :: (([String] → String) → r) → (r → ([String] → String)) → ExpF r → ([String] → String)
        φ inv show' (App x y) = λvs       → "(" ++ show' x vs ++ "  " ++ show' y vs ++ ")"
        φ inv show' (Lam z)   = λ(v : vs) → "(\\" ++ v ++ "->" ++ show' (z (inv (const v))) vs ++ ")"

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

Figure 3.17: **msfit**$_*$ example: String formatting function for HOAS.

Since we can build any untyped lambda expression 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 3.17.

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 ($z :: Exp_g \to Exp_g$) 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 $show'$ (and hence also $showExp_g$) are total as long as the function values embedded in the Lam_g constructors are total.

We can use $showExp_g$ to print out the terms as follows:

$>$ *putStrLn* ($showExp_g\ k_g$)

```
(\a->(\b->a))
```

$>$ *putStrLn* ($showExp_g\ s_g$)

```
(\a->(\b->(\c->((a c) (b c)))))
```

$>$ *putStrLn* ($showExp_g\ w_g$)

```
(\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, which is $(Var_g\ v)$, was well behaved. If we had applied z to the expression $(Lam_g\ (\lambda x \to x))$ 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 $(Lam_g\ (\lambda x \to x))$, then what happens?

$$show'\ (Lam_g\ z)\ (v : vs) = \texttt{"(\\"} +\!\!+\ v +\!\!+ \texttt{"->"} +\!\!+ \\ show'\ (z\ (Lam_g\ (\lambda x \to x))\ vs +\!\!+ \texttt{")"}$$

The function is no longer total. To format $(z\ (Lam_g\ (\lambda x \to x)))$ 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 $Lam_g\ (\lambda x \to x)$ expression and keeps on recursing.

A Mendler-style solution for closed HOAS

Our exploration of the code in Figure 3.17 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 $Var_g :: String \to Exp_g$. In general, we may not be so lucky.

In Figure 3.17, we define Exp_g in anticipation of our need to write a function $showExp_g :: Exp_g \to String$, by including a constructor $Var_g :: String \to Exp_g$. Had we anticipated another function $f :: Exp_g \to Int$, we would have needed another

constructor $C :: Int \to Exp_g$. Clearly we need a better solution. The solution is to generalize the kind of the datatype from $Exp_g :: *$ to $Exp :: * \to *$, and add a universal inverse.

$$\begin{aligned}
&\textbf{data } Exp\ a = App\ (Exp\ a)\ (Exp\ a) \\
&\qquad\qquad \mid Lam\ (Exp\ a \to Exp\ a) \\
&\qquad\qquad \mid Inv\ a
\end{aligned}$$

$$\begin{aligned}
&countLam :: Exp\ Int \to Int \\
&countLam\ (Inv\ n) = n \\
&countLam\ (App\ x\ y) = countLam\ x + countLam\ y \\
&countLam\ (Lam\ f) = countLam\ (f\ (Inv\ 1))
\end{aligned}$$

Generalizing from *countLam*, we can define a function from *Exp* to any type. How do we lift this kind of solution to Mendler style? Fegaras and Sheard [32] proposed moving the general inverse from the base type to the datatype fixpoint. Later, this approach was refined by Washburn and Weirich [96] to remove the junk introduced by that augmentation (i.e., things such as $App\ (Inv\ 1)\ (Inv\ 1)$).

We use the same inverse-augmented datatype fixpoint appearing in Washburn and Weirich [96]. Here, we call it $\breve{\mu}_*$ (see Figure 3.1). The inverse-augmented datatype fixpoint $\breve{\mu}_*$ is similar to the standard datatype fixpoint μ_*. The difference is that $\breve{\mu}_*$ has an additional type index a and an additional data constructor $\mathsf{Inverse}_* :: a \to \breve{\mu}_*\ f\ a$, corresponding to the universal inverse. The data constructor $\mathsf{I\breve{n}}_*$ and the projection function $\breve{\mathsf{out}}_*$ correspond to In_* and out_* of the normal fixpoint μ_*. As usual, we restrict the use of $\breve{\mathsf{out}}_*$, or pattern matching against $\mathsf{I\breve{n}}_*$.

We illustrate this in the second part of Figure 3.17. 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:

$$\begin{aligned}
k &= lam\ (\lambda x \to lam\ (\lambda y \to x)) \\
s &= lam\ (\lambda x \to lam\ (\lambda y \to lam\ (\lambda z \to app\ (app\ x\ z)\ (app\ y\ z)))) \\
w &= lam\ (\lambda x \to app\ x\ x) \\
skk &= app\ (app\ s\ k)\ k
\end{aligned}$$

However, there is another way to construct Exp' values that is problematic. Using the constructor $\mathsf{Inverse}_*$, we can turn values of arbitrary type t into values of $Exp'\ t$ (e.g., $\mathsf{Inverse}_*\ True :: Exp'\ Bool$). This value is junk, since it does not correspond to any lambda term. By design, we wish to hide $\mathsf{Inverse}_*$ behind an abstraction boundary. We should never allow the user to construct expressions such as $\mathsf{Inverse}_*\ True$, except for using them as callers 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, and w are of type $(Exp'\ a)$. The expressions that contain $\mathsf{Inverse}_*$ have more specific types (e.g., $(\mathsf{Inverse}_*\ True) :: (Exp'\ Bool)$). Therefore, we define the type of Exp to be $\forall a.Exp'\ a$. Then, expressions of type Exp are guaranteed to be be inverse-free. Using parametricity to sort out the junk introduced by the inverse is the key idea of Washburn and Weirich [96], and the inverse-augmented fixpoint μ_* is the key idea of Fegaras and Sheard [32]. The contribution we make in this work is putting together these ideas in a Mender-style setting. By doing so, we are able to define recursion combinators over types with negative occurrences, which have well-understood termination properties enforced by parametricity. We define four such combinators: $\mathbf{msfit}_*$, $\mathbf{msfcvit}_*$, $\mathbf{msfit}_{*\to*}$, and $\mathbf{msfcvit}_{*\to*}$. The combinator $\mathbf{msfit}_*$ is the simplest. To define it, we generalize over $\mathbf{mit}_*$ by using the same device we used earlier. We abstract the combining function over an additional argument, this time, an abstract inverse.

- The combining function φ becomes a function of 3 arguments: an abstract inverse, an recursive caller, and a base structure.

$$\begin{aligned}\mathbf{msfit}\ \varphi\ (\mathsf{In}_*\ x) &= \varphi\ \mathsf{Inverse}_*\ (\mathbf{msfit}\ \varphi)\ x\\ \mathbf{msfit}\ \varphi\ (\mathsf{Inverse}_*\ z) &= z\end{aligned}$$

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

$$\begin{aligned}\mathbf{msfit}_* :: (\forall r.(a \to r\ a) \to \\ (r\ a \to a) \to \\ (f\ (r\ a) \quad \to a)) \to (\forall a.\breve{\mu}_*\ f\ a) \to a\end{aligned}$$

- Note the abstract recursive type r is parameterized by the answer type a because the inverse-augmented fixpoint $\breve{\mu}_*$ is parameterized by the answer type a.

  Also, note the second argument of $\mathbf{msfit}_*$, the object being operated on, has the higher-rank type $(\forall a.\breve{\mu}_*\ f\ a)$, insisting the input value to be inverse-free by enforcing a to be abstract.

In Figure 3.17, using $\mathbf{msfit}_*$, it is easy to define *showExp*, the string formatting function for *Exp*, as in Figure 3.17. The *App* case is similar to the general recursive implementation. The body of φ 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 role as $(Var_g\ v)$ in $show'$.

Does $\mathbf{msfit}_*$ really guarantee termination? To prove this, we need to address the first two of the three potential problems described on page 135. We assume that the first problem (embedded functions may be partial) won't happen. The second problem (cyclic use of constructors as arguments to embedded functions) is addressed by the same argument we used in Section 3.6. The abstract type of the inverse doesn't allow it to be applied to constructors, as they're not abstract enough. Just as we couldn't define p_m (in Section 3.6), we can't apply z to things

$$\textbf{type}\ \breve{\mu}_*\ f\ r\ a = (r\ a) + (((r\ a \to a) \to f\ (r\ a) \to a) \to a)$$

$$\textbf{newtype}\ Id\ x = Id\ \{unId :: x\}$$

$$\begin{aligned}\textbf{msfit}_*\ &::\ (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a)\\ &\quad \to (\forall a.\breve{\mu}_*\ f\ Id\ a) \to a\end{aligned}$$

$$\textbf{msfit}_*\ \varphi\ x = case_+\ x\ unId\ (\lambda f \to f\ (\varphi\ Id))$$

$$lift :: ((Id\ a \to a) \to f\ (Id\ a) \to a) \to \breve{\mu}_*\ f\ Id\ a \to Id\ a$$

$$lift\ h\ x = case_+\ x\ id\ (\lambda x \to Id\ (x\ h))$$

$$\textbf{type}\ a + b = \forall c.(a \to c) \to (b \to c) \to c$$

$$in_L :: a \to (a + b)$$

$$in_L\ a = \lambda f\ g \to f\ a$$

$$in_R :: b \to (a + b)$$

$$in_R\ b = \lambda f\ g \to g\ b$$

$$case_+ :: (a + b) \to (a \to c) \to (b \to c) \to c$$

$$case_+\ x\ f\ g = x\ f\ g$$

Figure 3.18: F_ω encoding of $\breve{\mu}_*$, $\textbf{msfit}_*$, and the sum type $(+)$.

such as $(Lam\ (\lambda x \to x))$. In Section 3.9.2, we provide an embedding of **msfit**, along with several examples (including the HOAS formatting example) into the strongly normalizing language F_ω. This constitutes a proof that **msfit** terminates for all inductive datatypes, even those with negative occurrences.

3.9.2 F_ω encoding of $\breve{\mu}_*$ and $\textbf{msfit}_*$

Figure 3.18 is the F_ω encoding[14] of the inverse-augmented datatype $\breve{\mu}_*$ and its iteration $\textbf{msfit}_*$. We use the sum type to encode $\breve{\mu}_*$ since it consists of two constructors, one for the inverse and the other for the recursion. The newtype Id

[14] Using a fragment of Haskell, which we believe to be a subset of F_ω.

```
data ExpF x = App x x | Lam (x → x)
type Exp' a = ŭ_* ExpF Id a
type Exp = ∀a.Exp' a
app :: Exp' a → Exp' a → Exp' a
app x y = in_R (λh → h unId (App (lift h x) (lift h y)))
lam :: (Exp' a → Exp' a) → Exp' a
lam f = in_R (λh → h unId (Lam (λx → lift h (f (in_L x)))))
showExp :: Exp → String
showExp e = msfit_* φ e vars where
  φ inv show' (App x y) = λvs        →
    "(" ++ show' x vs ++ "  " ++ show' y vs ++ ")"
  φ inv show' (Lam z)   = λ(v : vs) →
    "(\\" ++ v ++ "->" ++ show' (z (inv (const v))) vs ++ ")"
```

Figure 3.19: HOAS string formatting example in F_ω.

wraps answer values inside the inverse. The iteration combinator $\mathbf{msfit}_*$ unwraps the result (*unIn*) when x is an inverse. Otherwise, $\mathbf{msfit}_*$ runs the combining function φ over the recursive structure ($\lambda f \to f\ (\varphi\ Id)$). The utility function *lift* abstracts a common pattern, useful when we define the shorthand constructors (*lam* and *app*).

Figure 3.18 also contains the F_ω encoding of the sum type (+) and its constructors (or injection functions) in_L and in_R. The case expression $case_+$ for the sum type is just a binary function application. In the F_ω encoding, this could be omitted (i.e., $case_+\ x\ f\ g$ simplifies to $x\ f\ g$). But, we choose to write in terms of $case_+$ to make the definitions easier to read.

In Figure 3.19, we define both a recursive datatype for HOAS (*Exp*) and the string formatting function (*showExp*), with these F_ω encodings, just as we did in Section 3.9.1. We can define simple expressions using the shorthand constructors and print out those expressions using *showExp*. For example,

$> putStrLn\ (showExp\ (lam\ (\lambda x \to lam\ (\lambda y \to x))))$

```
(\a->(\b->a))
```

It is important to note that we embedded $\breve{\mu}_*$ and $\mathbf{msfit}_*$ into F_ω in Figure 3.19, but we have not embedded $\mathsf{In}\breve{}_*$ into F_ω. Instead, we embedded the two constructors of *Exp*, *app* and *lam*, into F_ω. Note that *app* and *lam* are defined in terms of in_R, *unId*, and *lift*, which are definable in F_ω as in Figure 3.18.

The situation is different from embedding of the Mendler-style iteration into F_ω, where μ_*, $\mathbf{mit}_*$, and also In_* are embedded into F_ω (see Figure 3.23 in Section 3.10). Then, the embeddings for data constructors of recursive types are simply given in terms of In_* (see the embedding of natural numbers on page 150, Section 3.10).

Unfortunately, for the Mendler-style iteration with syntactic inverses, we have not found a way to factor out $\breve{\mu}_*$ as an F_ω-term to reuse it for embedding data

constructors of inverse-augmented recursive types. We can only embed data constructors (*app* and *lam*) of a specific recursive type (*Exp*). This is analogous to the situation where we can embed any given regular recursive type in System F, but can not factor out μ or In as we can do in System F_ω.

We strongly believe that there exist systematic algorithms of embedding any given regular recursive types $\breve{\mu}_*$ and $\mathbf{msfit}_*$. We can already see the pattern: *lift* is applied to recursive arguments in positive positions and in_L is used in recursive arguments in negative positions. More precise and general description of the algorithm for embedding and a proof that the algorithm leads to desired the embeddings would be interesting future work.

3.9.3 Evaluating Simply Typed HOAS

We can write an evaluator for a simply-typed HOAS in a surprisingly simple manner as in Figure 3.20, using the Mendler-style iteration with syntactic inverses.

We first define the simply-typed HOAS as a recursive indexed datatype $Exp :: * \to *$. We take the fixpoint using $\breve{\mu}_{*\to*}$ (the fixpoint operation that supports a syntactic inverse). This fixpoint is taken over a non-recursive base structure ($ExpF :: (* \to *) \to (* \to *)$). Note that *ExpF* is an indexed type. So expressions will be indexed by their type. Using $\breve{\mu}_{*\to*}$, the fixpoint of any structure is also parameterized by the type of the answer.

The use of **msfit** requires that *Exp* should be parametric in this answer type (by defining **type** $Exp\ t = \forall a.Exp'\ a$) just as we did in the untyped HOAS formatting example in Figure 3.17.

Using general recursion, one would have defined the datatype $Exp_g :: * \to *$ that corresponds to *Exp* as follows, using Haskell's native recursive datatype definition.

data $Exp_g\ t$ **where**

$Lam_g :: (Exp_g\ a \to Exp_g\ b) \to Exp_g\ (a \to b)$

$App_g :: Exp_g\ (a \to b) \to Exp_g\ a \to Exp_g\ b$

```
data ExpF r t where
  Lam :: (r a → r b) → ExpF r (a → b)
  App :: r (a → b) → r a → ExpF r b
type Exp' a t = ŭ_{*→*} ExpF a t
type Exp t = ∀a.Exp' a t
  -- lam :: Exp' a t1 → Exp' a t2 → Exp' a (t1 → t2)
lam e   = lĭ_{*→*} (Lam e)
  -- app :: Exp (t1 → t2) → Exp t1 → Exp t2
app f e = lĭ_{*→*} (App f e)

newtype Id a = MkId {unId :: a}
type Phi f a = ∀r.(∀i.a i → r a i) → (∀i.r a i → a i) → (∀i.f (r a) i → a i)

evalHOAS :: Exp t → Id t
evalHOAS e = msfit_{*→*} φ e where
  φ :: Phi ExpF Id
  φ inv ev (Lam f) = MkId (λv → unId (ev (f (inv (MkId v)))))
  φ inv ev (App f x) = MkId (unId (ev f) (unId (ev x)))
```

Figure 3.20: **msfit**$_{*\to*}$ example: an evaluator for the simply-typed HOAS.

The definition of *evalHOAS* specifies how to evaluate an HOAS expression to a host-language value (i.e., Haskell) wrapped by the identity type (*Id*). In the description below, we ignore the wrapping (*MkId*) and unwrapping (*unId*) of *Id* by completely dropping them from the description. See Figure 3.20 (where they are not omitted) if you care about these details. We discuss the evaluation for each of the constructors of *Exp*:

- Evaluating an HOAS abstraction (*Lam f*) lifts an object-language function (f) over *Exp* into a host-language function over values: ($\lambda v \rightarrow ev\ (f\ (inv\ v))$). In the body of this host-language lambda abstraction, the inverse of the (host-language) argument value v is passed to the object-language function f. The resulting HOAS expression ($f\ (inv\ v)$) is evaluated by the recursive caller (ev) to obtain a host-language value.
- Evaluating an HOAS application (*App f x*) lifts the function f and argument x to host-language values ($ev\ f$) and ($ev\ x$), and uses the host-language application to compute the resulting value. Note that the host-language application (($ev\ f$) ($ev\ x$)) is type-correct since $ev\ f :: a \rightarrow b$ and $ev\ x :: a$; thus the resulting value has type b.

We can be confident that *evalHOAS* indeed terminates since $\breve{\mu}_{*\rightarrow *}$ and $\mathbf{msfit}_{*\rightarrow *}$ can be embedded into F_ω in a manner similar to the embedding of $\breve{\mu}_*$ and $\mathbf{msfit}_*$ into F_ω in Figure 3.18.

Figure 3.20 highlights two advantages of Mendler style over conventional style in one example. This example shows that the Mendler-style Sheard–Fegaras iteration is useful for both *negative* and *indexed* datatypes. *Exp* in Figure 3.20 has both negative recursive occurrences and type indices.

The *showHOAS* example in Figure 3.17, which we discussed in the previous subsection, has appeared in other work [32], written in conventional style. So, the *showHOAS* example only shows that Mendler style is as expressive as conventional

style (although it is perhaps syntactically more pleasant than conventional style). However, it is not obvious how one could extend the conventional-style Sheard–Fegaras iteration over indexed datatypes.

In contrast, the Mendler-style Sheard–Fegaras iteration is naturally defined over indexed datatypes of arbitrary kinds. In fact, both $\mathbf{msfit}_{*\to *}$ used in *evalHOAS* and $\mathbf{msfit}_{*}$ used in *showHOAS* have exactly the same syntactic definition. They differ only in their type signatures. This is illustrated in Figures 3.2 and 3.3 on pages 100-101.

3.9.4 A graph datatype with cycles and sharing

Another example of a negative datatype is the graph with cycles and sharing in Figure 3.21. For further details, see the paper by Fegaras and Sheard [32].

3.9.5 Additional Mendler-style combinators

The combinator $\mathbf{msfcvit}_{*}$ generalizes $\mathbf{mcvit}_{*}$ by the addition of an abstract inverse to a combinator that already has an abstract unroller. The combining function φ becomes a function of 4 arguments: an abstract inverse, an abstract unroller, a recursive caller, and a base structure.

The combinators $\mathbf{msfit}_{*\to *}$ and $\mathbf{msfcvit}_{*\to *}$ (at kind $* \to *$) generalize the combinators $\mathbf{msfit}_{*}$ and $\mathbf{msfcvit}_{*}$ (at kind $*$) to combinators on types with a type index. The pattern of generalization is quite evident in Figures 3.2 (p.100), and 3.3 (p.101) and the reader is encouraged to study those figures for a complete understanding of the results of this chapter.

We believe $\mathbf{msfit}_{*\to *}$ might be useful for writing functions over negative datatypes with type indices. The combinator $\mathbf{msfcvit}_{*\to *}$, like its kind $*$ counterpart $\mathbf{msfcvit}_{*}$, may not terminate given ill-behaved φ functions. Such functions use the unroller to *reach down inside a tree* to extract an embedded function and then apply that function to an ancestor that contains that function. Yet, they may be nevertheless

$$\textbf{data}\ G\ p\ r = N\ p\ [r] \mid R\ (r \to r) \mid S\ (r \to r)\ r$$

$$\textbf{type}\ \mathit{Graph}\ p = \forall a.\breve{\mu}_*\ (G\ p)\ a$$

$$\mathit{node}\ v\ \mathit{gs} = \breve{\mathsf{in}}_*\ (N\ v\ \mathit{gs})$$

$$\mathit{rec}\ f \quad = \breve{\mathsf{in}}_*\ (R\ f)$$

$$\mathit{share}\ f\ g\ = \breve{\mathsf{in}}_*\ (S\ f\ g)$$

$$\mathit{flatG} :: \mathit{Graph}\ a \to [a]$$

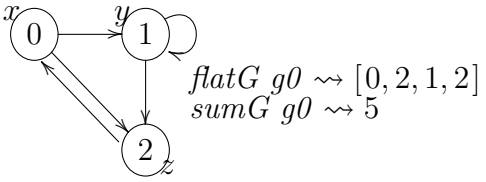
$$\mathit{flatG} = \textbf{msfit}_*\ \varphi\ \textbf{where}$$

$$\begin{aligned}
&\varphi\ \mathit{inv}\ \mathit{flat}\ (N\ v\ \mathit{gs}) &&= v : \mathit{concatMap}\ \mathit{flat}\ \mathit{gs}\\
&\varphi\ \mathit{inv}\ \mathit{flat}\ (R\ f) &&= \mathit{flat}\ (f\ (\mathit{inv}\ [\,]))\\
&\varphi\ \mathit{inv}\ \mathit{flat}\ (S\ f\ g) &&= \mathit{flat}\ (f\ g)
\end{aligned}$$

$$\mathit{sumG} :: \mathit{Graph}\ \mathit{Int} \to \mathit{Int}$$

$$\mathit{sumG} = \textbf{msfit}_*\ \varphi\ \textbf{where}$$

$$\begin{aligned}
&\varphi\ \mathit{inv}\ \mathit{sumg}\ (N\ n\ \mathit{gs}) &&= n + \mathit{sum}\ (\mathit{map}\ \mathit{sumg}\ \mathit{gs})\\
&\varphi\ \mathit{inv}\ \mathit{sumg}\ (R\ f) &&= \mathit{sumg}\ (f\ (\mathit{inv}\ 0))\\
&\varphi\ \mathit{inv}\ \mathit{sumg}\ (S\ f\ g) &&= \mathit{sumg}\ (f\ g)
\end{aligned}$$

$\mathit{g0} :: \mathit{Graph}\ \mathit{Int}$ -- 

$\mathit{flatG}\ \mathit{g0} \rightsquigarrow [0,2,1,2]$
$\mathit{sumG}\ \mathit{g0} \rightsquigarrow 5$

$$\begin{aligned}
&\mathit{g0} = \mathit{rec}\ (\lambda x \to\\
&\qquad \mathit{share}\ \ (\lambda z \to \mathit{node}\ 0\ [z, \mathit{rec}\ (\lambda y \to \mathit{node}\ 1\ [y, z])])\\
&\qquad\qquad (\mathit{node}\ 2\ [x]))
\end{aligned}$$

Figure 3.21: A graph datatype with cycles and sharing [32]

$$
\begin{aligned}
&\mathbf{mopenit}_* :: (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a) \\
&\quad \to (\forall a.\breve{\mu}_*\ f\ a \to \breve{\mu}_*\ f\ a) \to (a \to a) \\
&\mathbf{mopenit}_*\ \varphi\ x\ v = \mathbf{msfit}\ \varphi\ (x\ (\mathsf{Inverse}_*\ v)) \\
&\quad \mathbf{where} \\
&\quad \mathbf{msfit} :: (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a) \to \breve{\mu}_*\ f\ a \to a \\
&\quad \mathbf{msfit}\ \varphi\ (\breve{\mathsf{In}}_*\ x) \qquad = \varphi\ \mathsf{Inverse}_*\ (\mathbf{msfit}\ \varphi)\ x \\
&\quad \mathbf{msfit}\ \varphi\ (\mathsf{Inverse}_*\ z) \quad = z
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{data}\ E\ r = A\ r\ r \mid L\ (r \to r) \quad \text{-- base structure for HOAS} \\
&\mathbf{type}\ \mathit{Exp}\ a = \breve{\mu}_*\ E\ a \\
&\mathit{lam}\ g \quad\ = \breve{\mathsf{In}}_*\ (L\ g) \\
&\mathit{app}\ e_1\ e_2 = \breve{\mathsf{In}}_*\ (A\ e_1\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\quad \text{-- } \mathit{False} \text{ for } (\lambda x \to \mathit{lam}\ (\lambda y \to y)),\ \mathit{True} \text{ for } (\lambda x \to \mathit{lam}\ (\lambda y \to x)) \\
&\mathit{freevarused} :: (\forall a.\mathit{Exp}\ a \to \mathit{Exp}\ a) \to \mathit{Bool} \\
&\mathit{freevarused}\ e = \mathbf{mopenit}_*\ \varphi\ e\ \mathit{True} \\
&\quad \mathbf{where} \\
&\quad \varphi :: \forall r.(\mathit{Bool} \to r\ \mathit{Bool}) \to (r\ \mathit{Bool} \to \mathit{Bool}) \to E\ (r\ \mathit{Bool}) \to \mathit{Bool} \\
&\quad \varphi\ \mathit{inv}\ \mathit{fvused}\ (L\ g) \qquad = \mathit{fvused}\ (g\ (\mathit{inv}\ \mathit{False})) \\
&\quad \varphi\ \mathit{inv}\ \mathit{fvused}\ (A\ e_1\ e_2) = \mathit{fvused}\ e_1 \vee \mathit{fvused}\ e_2
\end{aligned}
$$

Figure 3.22: The Mendler-style open-iteration $\mathbf{mopenit}_*$, which allows one free variable, and the *freevarused* function defined using $\mathbf{mopenit}_*$.

$$\textbf{type}\ \mu_*\ f = \forall a.(\forall r.(r \to a) \to f\ r \to a) \to a$$

$$\textbf{mit}_* :: (\forall r.(r \to a) \to f\ r \to a) \to \mu_*\ f \to a$$

$$\textbf{mit}_*\ \varphi\ r = r\ \varphi$$

$$in_0 :: f\ (\mu_*\ f) \to \mu_*\ f$$

$$in_0\ r\ \varphi = \varphi\ (\textbf{mit}_*\ \varphi)\ r$$

Figure 3.23: F_ω encoding of μ_* and $\textbf{mit}_*$ in Haskell.

useful for well-behaved φ functions.

In HOAS, a meta-level function (from expressions to expressions) represents an expression with a single free variable. For example, $\lambda x \to app\ (lam\ (\lambda f \to app\ f\ x))$ represents $\lambda f \to f\ x$, where x is free. The Mendler-style open-iterator ($\textbf{mopenit}_*$) supports computation over terms with one free variable represented in this fashion. We write $\textbf{mopenit}_*\ (\lambda x \to e)\ v$ for the open-iteration over an expression e with a free variable x. The iteration should compute v when the computation reaches x. For instance, the function *freevarused* defined using $\textbf{mopenit}_*$ in Figure 3.22 checks whether x appears in e, or is simply never mentioned. There is an open-iteration combinator at each kind (e.g., $\textbf{mopenit}_{*\to*}$ at kind $* \to *$), just like other combinators. Washburn and Weirich [96] studied open-iterations that support more than one free variable, although not in Mendler style.

3.10 PROPERTIES OF RECURSION COMBINATORS

We close this chapter by summarizing the termination properties of Mendler-style recursion combinators (Table 3.1) and the relationships between those combinators (Figure 3.24) (i.e., which combinators can be defined in terms of others).

We give a termination proof for the Mendler-style iteration (at kind $*$) in Figure 3.23. The proof takes the form of an embedding into F_ω, which is known to be strongly normalizing. The proof in Figure 3.23 is adapted from work by Abel

	positive	negative	example
cata	proof [43]	undefined	*len* Section 3.3
$\mathbf{mit}_*$	proof Figure 3.23	proof Figure 3.23	*len* Figure 3.5
$\mathbf{mcvit}_*$	proof [94]	no	*fib* Figure 3.6
$\mathbf{msfit}_*$	proof Section 3.9.2	proof Section 3.9.2	*showExp* Figure 3.17
$\mathbf{msfcvit}_*$	argument Section 3.5	no	*loopFoo* Figure 3.7
$\mathbf{mpr}_*$	proof [3]	proof [3]	*factorial* Figure 3.13
$\mathbf{mcvpr}_*$	conjecture Section 5.3	no	*lucas* Figure 3.16
$\mathbf{mit}_{*\to*}$	proof [5]	proof [5]	*bsum* Figure 3.9
			extev Figure 3.11
$\mathbf{mcvit}_{*\to*}$	similar to $\mathbf{mcvpr}_{*\to*}$	no	*switch2* Figure 3.10
$\mathbf{msfit}_{*\to*}$	similar to $\mathbf{msfit}_*$	similar to $\mathbf{msfit}_*$	
$\mathbf{msfcvit}_{*\to*}$	similar to $\mathbf{msfcvit}_*$	no	

Table 3.1: Termination properties of Mendler-style recursion combinators.

et al. [5]. They prove termination of the Mendler-style iteration at arbitrary kinds. A proof similar to Figure 3.23 is also given by Vene [94].

The definitions given in Figure 3.23 are F_ω terms, but are also legal Haskell terms that execute in GHC. Try the following code with the definitions of μ_* and $\mathbf{mit}_*$ from Figure 3.23. They run and return the expected results!

$$
\begin{array}{l|l}
\textbf{data}\ N\ c = Z \mid S\ c & n2i :: Nat \to Int \\
\textbf{type}\ Nat = \mu_*\ N & n2i = \mathbf{mit}_*\ \varphi\ \textbf{where} \\
zer = in_0\ Z & \quad \varphi\ n2i'\ Z = 0 \\
suc = in_0 \circ S & \quad \varphi\ n2i'\ (S\ n) = 1 + n2i'\ n
\end{array}
$$

Abel and Matthes [3] proved termination of Mendler-style primitive recursion (**mpr**) by a reduction preserving embedding of **mpr** into Fix_ω. We discuss the details of this embedding in Section 5.2. We know that the Mendler-style course-of-values recursion (**mcvpr**) does not terminate for negative datatypes since **mcvit** does not terminate for negative datatypes. Any computation that can be defined by **mit** can also be defined by **mcvpr** (where it may be more efficient). We show a partial proof that $\mathbf{mcvpr}_*$ terminates for regular positive datatypes in Section 5.3, and we conjecture that **mcvpr** terminates for positive datatypes at higher-kinds as well.

Vene [94] stated that we can deduce the termination of the Mendler-style course-of-values iteration for positive datatypes from its relation to the conventional course-of-values iteration, but he did not clearly discuss whether the termination property holds for negative datatypes. In our work, we demonstrated that $\mathbf{mcvit}_*$ may not terminate for negative datatypes by exhibiting the counterexample (Figure 3.7) in Section 3.6.

Figure 3.24 illustrates a well-known fact that a standard iteration (**mit**) is a special case of a course-of-values iteration (**mcvit**). Note that **mit** is defined in terms of **mcvit** by ignoring the inverse operation (*out*). Similarly, we can

define **mit** in terms of **mpr** and **mcvit** in terms of **mcvpr** by ignoring the casting operation of the primitive recursion families.

$$
\begin{aligned}
&\mathbf{mit}_{*} && \varphi\ r = \mathbf{mcvit}_{*}\ (\mathit{const}\ \varphi)\ r \\
&\mathbf{mit}_{*\to *} && \varphi\ r = \mathbf{mcvit}_{*\to *}\ (\mathit{const}\ \varphi)\ r \\
&\mathbf{msfit}_{*} && \varphi\ r = \mathbf{msfcvit}_{*}\ (\lambda \mathit{inv}\ \_ \to \varphi\ \mathit{inv})\ r \\
&\mathbf{msfit}_{*\to *} && \varphi\ r = \mathbf{msfcvit}_{*\to *}\ (\lambda \mathit{inv}\ \_ \to \varphi\ \mathit{inv})\ r
\end{aligned}
$$

Figure 3.24: Alternative definition of iteration via the course-of-values iteration.

Part III

Term-Indexed Lambda Calculi

Chapter 4

SYSTEM F_i

It is well known that datatypes can be embedded into polymorphic lambda calculi by means of functional encodings such as the Church and Boehm-Berarducci encodings [14].

In System F, one can embed *regular datatypes* like homogeneous lists:

Haskell: `data List a = Cons a (List a) | Nil`

System F: $\texttt{List}\ A \triangleq \forall X.(A \to X \to X) \to X \to X$

$\texttt{Cons} \triangleq \lambda w.\lambda x.\lambda y.\lambda z.\, y\ w\ (x\ y\ z),\ \texttt{Nil} \triangleq \lambda y.\lambda z.z$

In such regular datatypes, constructors have an algebraic structure that directly translates into polymorphic operations on abstract types, as encapsulated by universal quantification over types (of kind $*$).

In the more expressive System F_ω, where one can abstract over type constructors of any kind, one can encode more general *type-indexed datatypes* that go beyond the regular datatypes. For example, one can embed powerlists with heterogeneous elements in which an element of type `a` is followed by an element of the product type `(a,a)`:

Haskell:
```
data Powl a = PCons a (Powl(a,a)) | PNil
-- PCons 1 (PCons (2,3) (PCons ((3,4),(1,2)) PNil)) ::  Powl Int
```

System F_ω: $\texttt{Powl} \triangleq \lambda A^*.\forall X^{*\to *}.(A \to X(A \times A) \to XA) \to XA \to XA$

Note the non-regular occurrence (`Powl(a,a)`) in the definition of (`Powl a`) and the use of universal quantification over higher-order kinds ($\forall X^{*\to *}$). The term

encodings for `PCons` and `PNil` are exactly the same as the term encodings for `Cons` and `Nil`, but with different types.

What about term-indexed datatypes? What extensions to System F_ω are needed to embed term-indices as well as type-indices? Our solution is System F_i.

In a functional language that supports term-indexed datatypes, we envisage that the classic example of homogeneous length-indexed lists would be defined along the following lines (in Nax-like syntax):

```
data Nat = S Nat | Z
data Vec : * -> Nat -> * where
  VCons : a -> Vec a {i} -> Vec a {S i}
  VNil  : Vec a {Z}
```

Here, the type constructor `Vec` is defined to admit parameterisation by both a type parameter and a term index.[1] For instance, the type (`Vec (List Nat) {S(S Z)}`) is a vector whose elements are lists of natural numbers. By design, our syntax directly reflects the difference between type and term arguments by enclosing the latter in curly braces. We also make this distinction in System F_i, where it is useful within the type system to guarantee the static nature of term-indexing.

The encoding of the vector datatype in System F_i is as follows:

$$\texttt{Vec} \triangleq \ \lambda A^{*}.\lambda i^{\texttt{Nat}}.\forall X^{\texttt{Nat}\to *}.(\forall j^{\texttt{Nat}}.A \to X\{j\} \to X\{\texttt{S}\ j\}) \to X\{\texttt{Z}\} \to X\{i\}$$

where `Nat`, `Z`, and `S` encode the natural number type and its two constructors, zero, and successor, respectively. Again, the term encodings for `VCons` and `VNil` are exactly the same as the encodings for `Cons` and `Nil`, but with different types.

Without going into the details of the formalism given in the next section, one sees that such a calculus that incorporates term-indexing structure needs four additional constructs (see Figure 4.1 for the highlighted extended syntax).

[1] Recall in Chapter 3, we classify the arguments of type constructors either as parameters that appear uniformly in the datatype definition (e.g., `a` in `Vec`, or as indices that vary (e.g., `i`, `S i`, or `Z`). Type arguments are sometimes used as parameters and sometimes used as as indices. Term arguments, on the other hand, are almost always used as indices, except for some degenerate cases (e.g., term-indexing by a unit value).

1. Type-indexed kind $A \to \kappa$ (as `Nat→*` in the example above), where the compile-time nature of term-indexing is reflected in the typing rules, forcing A to be a closed type (rule (Ri) in Figure 4.2).
2. Term-index abstraction $\lambda i^A.F$ (as $\lambda i^{\mathtt{Nat}}.\cdots$ in the example above) for constructing (or introducing) type-indexed kinds (rule (λi) in Figure 4.2).
3. Term-index application $F\{s\}$ (as $X\{\mathtt{Z}\}$, $X\{j\}$, and $X\{\mathtt{S}\ j\}$ in the example above) for destructing (or eliminating) type-indexed kinds, where the compile-time nature of indexing is reflected in the typing rules, forcing the index to be statically typed (rule $(@i)$ in Figure 4.2).
4. Term-index polymorphism $\forall i^A.B$ (as $\forall j^{\mathtt{Nat}}.\cdots$ in the example above), where the compile-time nature of polymorphic term-indexing is reflected in the typing rules, forcing the variable i to be static of closed type A (rule $(\forall Ii)$ in Figure 4.2).

As described above, System F_i maintains a clear-cut separation between type-indexing and term-indexing. This adds a level of abstraction to System F_ω and yields types that, in addition to parametric polymorphism, also keep track of inductive invariants using term-indices. All term-index information can be erased, as it is only used at compile-time. It is possible to project any well-typed System F_i term into a well-typed System F_ω term. For instance, the erasure of the F_i-type `Vec` is the F_ω-type `List`. This is established in Section 4.3 and used to deduce the strong normalization of System F_i.

A conference paper [7] presented the contents of this chapter at TLCA in 2013 has been published.

4.1 SYSTEM F_i

System F_i is a higher-order polymorphic lambda calculus designed to extend System F_ω by the inclusion of term indices. The syntax and rules of System F_i are described in Figures 4.1, 4.2, and 4.3. The extensions new to System F_i that are not originally part of System F_ω are highlighted by grey boxes. Eliding all the grey boxes from Figures 4.1, 4.2 and 4.3, one obtains a version of System F_ω with Curry-style terms and the typing context separated into two parts (type-level context Δ and term-level context Γ).

In this section, we first discuss the rational for our design choices (Section 4.1.1) and then introduce the new constructs of System F_i (Section 4.1.2).

4.1.1 Design of System F_i

Terms in F_i are Curry style. That is, term-level abstractions are unannotated ($\lambda x.t$), and type generalizations ($\forall I$) and type instantiations ($\forall E$) are implicit at the term-level. A Curry-style calculus generally has an advantage over its Church-style counterpart when reasoning about properties of reduction. For instance, the Church-Rosser property naturally holds for β-, η-, and $\beta\eta$-reduction in the Curry style, but may not hold in the Church style. This is caused by the presence of annotations in the abstractions [68].[2]

Type constructors, on the other hand, remain Church style in F_i. That is, type-level abstractions are annotated by kinds ($\lambda X^\kappa.F$). Choosing type constructors to be Church style makes the kind of a type constructor visually explicit. The choice of style for type constructors is not as crucial as the choice of style for terms because the syntax and kinding rules at the type-level are essentially a simply-typed lambda calculus. Annotating the type-level abstractions with kinds makes

[2]The Church-Rosser property, in its strictest sense (i.e., α-equivalence over terms), generally does not hold in Church-style calculi , but may hold under certain approximations, such as modulo ignoring the annotations in abstractions.

the kinds explicit in the type syntax. Because F_i is essentially an extension of F_ω with a new kind formation rule, making kinds explicit is a pedagogical tool to emphasize the consequences of this new formation rule. As a notational convention, we write A and B instead of F and G, where A and B to are expected to be types (i.e., nullary type constructors) of kind $*$.

In a language with term indices, terms appear in types (e.g., the length index $(n+m)$ in the type *Vec Nat* $\{n+m\}$). Such terms contain variables and the binding sites of these variables matter. In F_i, we expect such variables to be statically bound. Dynamically bound index variables would require a dependently-typed calculus such as the calculus of constructions. To reflect this design choice, typing contexts are separated into type-level contexts (Δ) and term-level contexts (Γ). Type level (static) variables (X, i) are bound in Δ and term (dynamic) variables (x) are bound in Γ. Type level variables are either type constructor variables (X) or term variables to be used as indices (i). As a notational convention, we write i instead of x when term variables are to be used as indices (i.e., introduced by either index abstraction or index polymorphism).

In contrast to our design choice, System F_ω is most often formalized using a single context, which binds both type variables (X) and term variables (x). In such a formalization, the free type variables in the typing of the term variable must be bound earlier in the context. For example, if X and Y appear free in the type of f, they must appear earlier in the single context (Γ) as below:

$$\Gamma = \ldots, X^*, \ldots, Y^*, \ldots, (f : \forall Z^*.X \to Y \to Z), \ldots$$

In such a formalization, the side condition ($X \notin \Gamma$) in the ($\forall I$) rule of Figure 4.1 is not necessary, since such a condition is already a part of the well-formedness condition for the context (i.e., Γ, X^κ is well-formed when $X \notin \mathrm{FV}(\Gamma)$). Thus, for F_ω, it is only a matter of taste whether to formalize the system using a single context or two contexts as they are equivalent formalizations with comparable

Syntax:

Sort	$\Box$	
Term Variables	x, i	
Type Constructor Variables	X	
Kinds	κ	$::= \ * \mid \kappa \to \kappa \mid A \to \kappa$
Type Constructors	A, B, F, G	$::= \ X \mid A \to B$
		$\mid \lambda X^{\kappa}.F \mid F\,G \mid \forall X^{\kappa}.B$
		$\mid \lambda i^{A}.F \mid F\,\{s\} \mid \forall i^{A}.B$
Terms	r, s, t	$::= \ x \mid \lambda x.t \mid r\ s$
Typing Contexts	Δ	$::= \ \cdot \mid \Delta, X^{\kappa} \mid \Delta, i^{A}$
	Γ	$::= \ \cdot \mid \Gamma, x : A$

Reduction: $\boxed{t \rightsquigarrow t'}$

$$\frac{}{(\lambda x.t)\ s \rightsquigarrow t[s/x]} \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \qquad \frac{r \rightsquigarrow r'}{r\ s \rightsquigarrow r'\ s} \qquad \frac{s \rightsquigarrow s'}{r\ s \rightsquigarrow r\ s'}$$

Figure 4.1: Syntax and Reduction rules of F_i.

complexity.

However, in F_i, we separate the context into two parts to distinguish the term variables used in types (called index variables, or indices, and bound as Δ, i^A) from the ordinary use of term variables (bound as $\Gamma, x : A$). The expectation is that indices should have no effect on reduction at the term-level. Although it is imaginable to formalize F_i with a single typing context and distinguish index variables from ordinary term variables using more general concepts (e.g., capability, or modality), we believe that splitting the typing context into two parts is the simplest solution for our purposes.

Well-formed typing contexts:

$$\boxed{\vdash \Delta} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Delta \quad \vdash \kappa : \Box}{\vdash \Delta, X^{\kappa}} (X \notin \mathrm{dom}(\Delta))$$

$$\frac{\vdash \Delta \quad \cdot \vdash A : *}{\vdash \Delta, i^{A}} (i \notin \mathrm{dom}(\Delta))$$

$$\boxed{\Delta \vdash \Gamma} \qquad \frac{\vdash \Delta}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash A : *}{\Delta \vdash \Gamma, x : A} (x \notin \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta))$$

Sorting: $\boxed{\vdash \kappa : \Box}$

$$(A)\frac{}{\vdash * : \Box} \qquad (R)\frac{\vdash \kappa : \Box \quad \vdash \kappa' : \Box}{\vdash \kappa \to \kappa' : \Box} \qquad (Ri)\frac{\cdot \vdash A : * \quad \vdash \kappa : \Box}{\vdash A \to \kappa : \Box}$$

Kinding: $\boxed{\Delta \vdash F : \kappa}$ $\qquad (Var)\frac{X^{\kappa} \in \Delta \quad \vdash \Delta}{\Delta \vdash X : \kappa} \qquad (\to)\frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$

$$(\lambda)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa} \vdash F : \kappa'}{\Delta \vdash \lambda X^{\kappa}.F : \kappa \to \kappa'} \qquad (\lambda i)\frac{\cdot \vdash A : * \quad \Delta, i^{A} \vdash F : \kappa}{\Delta \vdash \lambda i^{A}.F : A \to \kappa}$$

$$(@)\frac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'} \qquad (@i)\frac{\Delta \vdash F : A \to \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash F\,\{s\} : \kappa}$$

$$(\forall)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa} \vdash B : *}{\Delta \vdash \forall X^{\kappa}.B : *} \qquad (\forall i)\frac{\cdot \vdash A : * \quad \Delta, i^{A} \vdash B : *}{\Delta \vdash \forall i^{A}.B : *}$$

$$(Conv)\frac{\Delta \vdash A : \kappa \quad \Delta \vdash \kappa = \kappa' : \Box}{\Delta \vdash A : \kappa'}$$

Typing: $\boxed{\Delta; \Gamma \vdash t : A}$ $\qquad (:)\frac{(x : A) \in \Gamma \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : A} \qquad (:i)\frac{i^{A} \in \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i : A}$

$$(\to I)\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B} \qquad (\to E)\frac{\Delta; \Gamma \vdash r : A \to B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash r\ s : B}$$

$$(\forall I)\frac{\vdash \kappa : \Box \quad \Delta, X^{\kappa}; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B}(X \notin \mathrm{FV}(\Gamma)) \qquad (\forall E)\frac{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t : B[G/X]}$$

$$(\forall Ii)\frac{\cdot \vdash A : * \quad \Delta, i^{A}; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall i^{A}.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \qquad (\forall Ei)\frac{\Delta; \Gamma \vdash t : \forall i^{A}.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t : B[s/i]}$$

$$(=)\frac{\Delta; \Gamma \vdash t : A \quad \Delta \vdash A = B : *}{\Delta; \Gamma \vdash t : B}$$

Figure 4.2: Sorting, Kinding, and Typing rules of F_i.

Kind equality: $\boxed{\vdash \kappa = \kappa' : \Box}$

$$\frac{}{\vdash * = * : \Box} \qquad \frac{\vdash \kappa_1 = \kappa_1' : \Box \quad \vdash \kappa_2 = \kappa_2' : \Box}{\vdash \kappa_1 \to \kappa_2 = \kappa_1' \to \kappa_2' : \Box} \qquad \frac{\cdot \vdash A = A' : * \quad \vdash \kappa = \kappa' : \Box}{\vdash A \to \kappa = A' \to \kappa' : \Box}$$

$$\frac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa' = \kappa : \Box} \qquad \frac{\vdash \kappa = \kappa' : \Box \quad \vdash \kappa' = \kappa'' : \Box}{\vdash \kappa = \kappa'' : \Box}$$

Type constructor equality: $\boxed{\Delta \vdash F = F' : \kappa}$

$$\frac{\Delta, X^\kappa \vdash F : \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\, G = F[G/X] : \kappa'} \qquad \frac{\Delta, i^A \vdash F : \kappa \quad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

$$\frac{\Delta \vdash X : \kappa}{\Delta \vdash X = X : \kappa} \qquad \frac{\Delta \vdash A = A' : * \quad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X^\kappa.F = \lambda X^\kappa.F' : \kappa \to \kappa'} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F = F' : \kappa}{\Delta \vdash \lambda i^A.F = \lambda i^A.F' : A \to \kappa}$$

$$\frac{\Delta \vdash F = F' : \kappa \to \kappa' \quad \Delta \vdash G = G' : \kappa}{\Delta \vdash F\, G = F'\, G' : \kappa'} \qquad \frac{\Delta \vdash F = F' : A \to \kappa \quad \Delta; \cdot \vdash s = s' : A}{k\Delta \vdash F\, \{s\} = F'\, \{s'\} : \kappa}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa \vdash B = B' : *}{\Delta \vdash \forall X^\kappa.B = \forall X^\kappa.B' : *} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B = B' : *}{\Delta \vdash \forall i^A.B = \forall i^A.B' : *}$$

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \frac{\Delta \vdash F = F' : \kappa \quad \Delta \vdash F' = F'' : \kappa}{\Delta \vdash F = F'' : \kappa}$$

Term equality: $\boxed{\Delta; \Gamma \vdash t = t' : A}$

$$\frac{\Delta; \Gamma, x : A \vdash t : B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash (\lambda x.t)\, s = t[s/x] : B} \qquad \frac{\Delta; \Gamma \vdash x : A}{\Delta; \Gamma \vdash x = x : A}$$

$$\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t = t' : B}{\Delta; \Gamma \vdash \lambda x.t = \lambda x.t' : B} \qquad \frac{\Delta; \Gamma \vdash r = r' : A \to B \quad \Delta; \Gamma \vdash s = s' : A}{\Delta; \Gamma \vdash r\ s = r'\ s' : B}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, X^\kappa; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B}\,(X \notin \mathrm{FV}(\Gamma)) \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall X^\kappa.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t = t' : B[G/X]}$$

$$\frac{\cdot \vdash A : * \quad \Delta, i^A; \Gamma \vdash t = t' : B}{\Delta; \Gamma \vdash t = t' : \forall i^A.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(t'), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} \qquad \frac{\Delta; \Gamma \vdash t = t' : \forall i^A.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t = t' : B[s/i]}$$

$$\frac{\Delta; \Gamma \vdash t = t' : A}{\Delta; \Gamma \vdash t' = t : A} \qquad \frac{\Delta; \Gamma \vdash t = t' : A \quad \Delta; \Gamma \vdash t' = t'' : A}{\Delta; \Gamma \vdash t = t'' : A}$$

Figure 4.3: Equality rules of F_i.

4.1.2 System F_i compared to System F_ω

We assume readers are familiar with System F_ω and focus on describing the new constructs of F_i. These appear in grey boxes.

Kinds: The key extension to F_ω is the addition of type-indexed arrow kinds of the form $A \to \kappa$. This allows type constructors to have terms as indices. The development of F_i follows naturally from this single extension.

Sorting: The formation of indexed arrow kinds is governed by the sorting rule (Ri). This rule specifies that an indexed arrow kind $A \to \kappa$ is well-sorted when A has kind $*$ under the empty type-level context $(\cdot)$ and κ is well-sorted.

Requiring the use of the empty context avoids dependent kinds (i.e., kinds depending on the type-level or the value-level bindings). The type A appearing in the index arrow kind $A \to \kappa$ must be well-kinded under the empty type-level context $(\cdot)$. That is, A should to be a closed type of kind $*$ that does not contain any free type variables or index variables. For example, $(\mathit{List}\, X \to *)$ is not a well-sorted kind, while $((\forall X^{*}.\, \mathit{List}\, X) \to *)$ is a well-sorted kind.

Typing contexts: Typing contexts are split into two parts: the type-level contexts (Δ) for type-level (static) bindings and the term-level contexts (Γ) for term-level (dynamic) bindings. A new form of index variable binding (i^A) can appear in type-level contexts in addition to the traditional type variable bindings (X^κ). There is only one form of term-level binding ($x : A$) that appears in term-level contexts.

Well-formed typing contexts: A type-level context Δ is well-formed if it is either (1) empty, (2) extended by a type variable binding X^κ whose kind κ is well-sorted, or (3) extended by an index binding i^A whose type A is well-kinded under the empty type-level context at kind $*$. This restriction is similar to the one

that occurs in the sorting rule (Ri) for type-indexed arrow kinds (see paragraph ***Sorting***). The consequence of this is that, in typing contexts and sorts, A must be a closed type (i.e., nullary type constructor) without free variables.

A term-level context Γ is well-formed under a type-level context Δ when it is either empty or extended by a term variable binding $x : A$ whose type A is well-kinded under Δ.

Type constructors and their kinding rules: We extend the type constructor syntax by three constructs and extend the kinding rules accordingly.

Abstraction $\lambda i^A.F$ is the type-level abstraction over an index (or, index abstraction). Index abstractions introduce indexed arrow kinds by the kinding rule (λi). Note the use of the new form of context extension i^A in the kinding rule (λi).

Application $F\,\{s\}$ is the type-level index application. In contrast to the ordinary type-level application $(F\,G)$ where the argument (G) is a type constructor, the argument of an index application $(F\,\{s\})$ is a term (s). We use the curly brace notation around an index argument in a type to emphasize the transition from ordinary type to term and emphasize that s is an index term that is erasable. Index applications eliminate indexed arrow kinds by the kinding rule $(@i)$. We type check index term (s) under the current type-level context paired with the empty term-level context $(\Delta; \cdot)$ because we do not want it to depend on any term-level bindings. Allowing such a dependency would admit true dependent types.

Forall type $\forall i^A.B$ that quantifies over a term-index variable is called an index polymorphic type. The formation of indexed polymorphic types is governed by the kinding rule $\forall i$, which is very similar to the formation rule $(\forall)$ for ordinary polymorphic types.

In addition to rules (λi), $(@i)$, and $(\forall i)$, we need a conversion rule $(Conv)$ at the kind-level. This is because the new extension to the kind syntax $A \to \kappa$

involves types. Since kind syntax involves types, we need more than a simple structural equality over kinds. The new equality over kinds is the usual structural equality extended by the type constructor equality when comparing indexed arrow kinds (see Figure 4.3).

Terms and their typing rules. The term syntax is exactly the same as other Curry-style calculi. We write x for ordinary term variables introduced by term-level abstractions ($\lambda x.t$). We write i for index variables introduced by index abstractions ($\lambda i^A.F$) and index polymorphic types ($\forall i^A.B$). As discussed earlier, the distinction between x and i is for the convenience of readability.

Since F_i has index polymorphic types ($\forall i^A.B$), we need typing rules for index polymorphism: $(\forall Ii)$ for index generalization and $(\forall Ei)$ for index instantiation.

The index generalization rule $(\forall Ii)$ is similar to the type generalization rule $(\forall I)$, but generalizes over index variables (i) rather than type constructor variables (X). Rule $(\forall Ii)$ has two side conditions while rule $(\forall I)$ has only one. The additional side condition $i \notin \mathrm{FV}(t)$ in the $(\forall Ii)$ rule prevents terms from accessing the type-level index variables introduced by index polymorphism. Without this side condition, $\forall$-binder would no longer behave polymorphically, but instead would behave as a dependent function, which is usually denoted by the Π-binder in dependent type theories. The rule $(\forall I)$ for ordinary type generalization does not need such an additional side condition because type variables cannot appear in the syntax of terms. The side conditions on generalization rules for polymorphism are fairly standard in dependently-typed languages supporting distinctions between polymorphism (i.e., erasable arguments) and dependent functions (e.g., IPTS[69], ICC[68]).

The index instantiation rule $(\forall Ei)$ is similar to the type instantiation rule $(\forall Ei)$, except that we type check that the index term s is instantiated for i in the current type-level context paired with the empty term-level context $(\Delta; \cdot)$ rather

$$(\lambda i)\dfrac{\cdot \vdash A : *\qquad (@i)\dfrac{\Delta, i^A \vdash F : A \to \kappa \qquad (:i)\dfrac{i^A \in \Delta, i^A \qquad \Delta \vdash \cdot}{\Delta, i^A; \cdot \vdash i : A}}{\Delta, i^A \vdash F\{i\} : \kappa}}{\Delta \vdash \lambda i^A.F\{i\} : A \to \kappa}$$

Figure 4.4: Kinding derivation for an index abstraction.

than the current term-level context. Because index terms are at type-level, they should not depend on term-level bindings.

In addition to the rules $(\forall Ii)$ and $(\forall Ei)$ for index polymorphism, we need an additional variable rule $(:i)$ to be able to access the index variables already in scope. Terms (s) used at type-level in index applications $(F\{s\})$ should be able to access index variables already in scope. For example, $\lambda i^A.F\{i\}$ should be well-kinded under a context where F is well-kinded; this is justified by the derivation in Figure 4.4.

4.2 EMBEDDING DATATYPES AND MENDLER-STYLE ITERATORS

System F_i can express a rich collection of datatypes. First, we illustrate embeddings for both non-recursive and recursive datatypes using Church encodings [20] to define data constructors (Section 4.2.1). Second, we illustrate a more involved embedding for recursive datatypes based on two-level types (Section 4.2.2). Lastly, we discuss an encoding of equality over term indices (Section 4.2.3).

4.2.1 Embedding datatypes using Church-encoded terms

Church [20] invented an embedding of the natural numbers into the untyped λ-calculus, which he used to argue that the λ-calculus was expressive enough

$$
\begin{array}{lll}
\texttt{Bool} & = & \forall X.X \to X \to X \\
\texttt{true} & : \texttt{Bool} & = \lambda x_1.\lambda x_2.x_1 \\
\texttt{false} & : \texttt{Bool} & = \lambda x_1.\lambda x_2.x_2 \\
\texttt{elim}_{\texttt{Bool}} & : \texttt{Bool} \to \forall X.X \to X \to X & \\
 & = \lambda x.\lambda x_1.\lambda x_2.x\ x_1\ x_2 & (\textbf{if } x \textbf{ then } x_1 \textbf{ else } x_2)
\end{array}
$$

$$
\begin{array}{lll}
A_1 \times A_2 & = \forall X.(A_1 \to A_2 \to X) \to X & \\
\texttt{pair} & : \forall A_1^*.\forall A_2^*.A_1 \times A_2 & = \lambda x_1.\lambda x_2.\lambda x'.x'\ x_1\ x_2 \\
\texttt{elim}_{(\times)} & : \forall A_1^*.\forall A_2^*.A_1 \times A_2 \to \forall X.(A_1 \to A_2 \to X) \to X & \\
 & = \lambda x.\lambda x'.x\ x' & \\
 & \text{(by passing appropriate values to } x', \text{ we get} & \\
 & \ \texttt{fst} = \lambda x.x(\lambda x_1.\lambda x_2.x_1),\ \texttt{snd} = \lambda x.x(\lambda x_1.\lambda x_2.x_2)\ ) &
\end{array}
$$

$$
\begin{array}{lll}
A_1 + A_2 & = \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X & \\
\texttt{inl} & : \forall A_1^*.\forall A_2^*.A_1 \to A_1 + A_2 & = \lambda x.\lambda x_1.\lambda x_2.x_1\ x \\
\texttt{inr} & : \forall A_1^*.\forall A_2^*.A_2 \to A_1 + A_2 & = \lambda x.\lambda x_1.\lambda x_2.x_2\ x \\
\texttt{elim}_{(+)} & : \forall A_1^*.\forall A_2^*.(A_1 + A_2) \to & \\
 & \quad \forall X^*.(A_1 \to X) \to (A_2 \to X) \to X & \\
 & = \lambda x.\lambda x_1.\lambda x_2.x\ x_1\ x_2 & \\
 & \quad (\textbf{case } x \textbf{ of } \{\texttt{inl}\ x' \to x_1\ x'; \texttt{inr}\ x' \to x_2\ x'\}) &
\end{array}
$$

Figure 4.5: Embedding non-recursive datatypes.

$$
\begin{array}{lll}
\texttt{List} & = & \lambda A^*.\forall X^*.(A \to X \to X) \to X \to X \\
\texttt{cons} & : & \forall A^*.A \to \texttt{List}\,A \to \texttt{List}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\ x_c\,x_n) \\
\texttt{nil} & : & \forall A^*.\texttt{List}\,A \ = \ \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{List}} & : & \forall A^*.\texttt{List}\,A \to \forall X^*.(A \to X \to X) \to X \to X \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\ x_c\,x_n \qquad (\mathit{foldr}\ x_z\ x_c\ x \ \text{ in Haskell})
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Powl} & = & \lambda A^*. \\
 & & \forall X^{*\to *}.(A \to X(A \times A) \to XA) \to XA \to XA \\
\texttt{pcons} & : & \forall A^*.A \to \texttt{Powl}(A \times A) \to \texttt{Powl}\,A \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\ x_c\,x_n) \\
\texttt{pnil} & : & \forall A^*.\texttt{Powl}\,A \ = \ \lambda x_c.\lambda x_n.\lambda x_n \\
\texttt{elim}_{\texttt{Powl}} & : & \forall A^*.\texttt{Powl}\,A \to \\
 & & \forall X^{*\to *}.(A \to X(A \times A) \to XA) \to XA \to XA \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\ x_c\,x_n
\end{array}
$$

$$
\begin{array}{lll}
\texttt{Vec} & = & \lambda A^*.\lambda i^{\texttt{Nat}}. \\
 & & \forall X^{\texttt{Nat}\to *}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{S}\,i\}) \to \\
 & & \qquad X\{\texttt{Z}\} \to X\{i\} \\
\texttt{vcons} & : & \forall A^*.\forall i^{\texttt{Nat}}.A \to \texttt{Vec}\,A\,\{i\} \to \texttt{Vec}\,A\,\{\texttt{S}\,i\} \\
 & & \qquad = \lambda x_a.\lambda x.\lambda x_c.\lambda x_n.x_c\,x_a\,(x\ x_c\,x_n) \\
\texttt{vnil} & : & \forall A^*.\texttt{Vec}\,A\,\{\texttt{Z}\} \ = \ \lambda x_c.\lambda x_n.x_n \\
\texttt{elim}_{\texttt{Vec}} & : & \forall A^*.\forall i^{\texttt{Nat}}.\texttt{Vec}\,A\,\{i\} \to \\
 & & \forall X^{\texttt{Nat}\to *}.(\forall i^{\texttt{Nat}}.A \to X\{i\} \to X\{\texttt{S}\,i\}) \to \\
 & & \qquad X\{\texttt{Z}\} \to X\{i\} \\
 & = & \lambda x.\lambda x_c.\lambda x_n.x\ x_c\,x_n
\end{array}
$$

Figure 4.6: Embedding recursive datatypes.

for the foundation of logic and arithmetic. Church encoded the data constructors of natural numbers, successor and zero, as higher-order functions, $\texttt{succ} = \lambda x.\lambda x_s.\lambda x_z.x_s(x\, x_s x_z)$ and $\texttt{zero} = \lambda x_s.\lambda x_z.x_z$. The key concept of the Church encoding is that a value is encoded as an elimination function. The bound variables x_s and x_z (in `succ` and `zero`) denote the operations needed to eliminate the successor and zero cases respectively. The Church encodings of successor states that to eliminate $\texttt{succ}\, x$, one must "apply x_s to the elimination of the predecessor $(x\, x_s x_z)$", and, to eliminate `zero`, one may simply "return x_z". Since values *are* elimination functions, the eliminator can be defined as an application of the value itself to the needed operations, one for each of the data constructors. For instance, we can define an eliminator for the natural numbers as $\texttt{elim}_{\texttt{Nat}} = \lambda x.\lambda x_s.\lambda x_z.x\, x_s x_z$. This is simply an η-expansion of the identity function $\lambda x.x$. The Church encoded natural numbers are typable in polymorphic λ-calculi, such as System F_ω, as follows:

$$
\begin{array}{lcl}
\texttt{Nat} & = & \forall X^*.(X \to X) \to X \to X \\
\texttt{S} & : \texttt{Nat} \to \texttt{Nat} \;=\; & \lambda x.\lambda x_s.\lambda x_z.x_s(x\, x_s x_z) \\
\texttt{Z} & : \texttt{Nat} \;=\; & \lambda x_s.\lambda x_z.x_z \\
\texttt{elim}_{\texttt{Nat}} & : \texttt{Nat} \to \forall X^*.(X \to X) \to X \to X & \\
 & = & \lambda x.\lambda x_s.\lambda x_z.x\, x_s x_z
\end{array}
$$

Other datatypes are also embeddable into polymorphic λ-calculi in a similar fashion. Embeddings of some well-known non-recursive datatypes are illustrated in Figure 4.5, and embeddings of the list-like recursive datatypes, which we discussed as motivating examples in the beginning of this chapter, are illustrated in Figure 4.6. Note that the term encodings for the constructors and eliminators of the list-like datatypes in Figure 4.6 are exactly the same. For instance, the term encodings for `nil`, `pnil`, and `vnil` are all the same term: $\lambda x_s.\lambda x_z.x_z$. The term encodings for nil and cons capture the linear nature of lists, hence they are the same for all list-like structures. However, their types differ, capturing different

invariants, for example, element shape (`Powl`) and list length (`Vec`).

4.2.2 Embedding recursive datatypes as two-level types

We can divide a recursive datatype definition into two parts – a recursive type operator and a base structure. The operator "weaves" recursion into the datatype definition and the base structure describes its shape (i.e., the number of data constructors and their types). One can program with two-level types in any functional language that supports higher-order polymorphism[3] such as Haskell. In Figure 4.7, we illustrate this with an example of a two-level definition for ordinary lists (all the other types in this paper have similar definitions).

The use of two-level types has been recognized as a useful functional programming pearl [85] because two-level types separate the two concerns of (1) recursion on recursive subcomponents and (2) handling different cases by pattern matching over the shape of the (non-recursive) base structure. An advantage of such an approach is that a single eliminator can be defined once for all datatypes of the same kind. For example, the function $\texttt{mit}_\kappa$ describes Mendler-style iteration (a.k.a., fold, or catamorphism) for the recursive types defined by μ_κ. Although it is possible to write programs using two level datatypes in a general purpose functional language, one could not expect logical consistency in such systems.

Interestingly, there exist embeddings of the recursive type operator μ_κ, its data constructor $\texttt{In}_\kappa$, and the Mendler-style iterator $\texttt{mit}_\kappa$ for each kind κ into the higher-order polymorphic λ-calculus F_i, as illustrated in Figure 4.8. In addition to illustrating the general form of embedding μ_κ, we also fully expand the embeddings for some instances (μ_*, $\mu_{*\to*}$, $\mu_{\texttt{Nat}\to*}$), which are used in Figure 4.7. These embeddings support the embedding of arbitrary type- and term-indexed recursive datatypes into System F_i. Thus we can reason about these datatypes in a logically

[3] This is also known as higher-kinded polymorphism, or type-constructor polymorphism

```
newtype μ* (f :: * -> *) = In* (f (μ* f))

data ListF (a::*) (r::*) = Cons a r | Nil
type List a = μ* (ListF a)
cons x xs = In* (Cons x xs)
nil       = In* Nil

mit* :: (∀ r.(r->x) -> f r -> x) -> Mu0 f -> x
mit* phi (In* z) = phi (mit* phi) z

newtype μ(*→*) (f :: (*->*) -> (*->*)) (a::*)
  = In(*→*) (f (Mu(*→*) f)) a

data PowlF (r::*->*) (a::*) = PCons a (r(a,a)) | PNil
type Powl a = μ(*→*) PowlF a
pcons x xs = In(*→*) (PCons x xs)
pnil       = In(*→*) PNil

mit(*→*) :: (∀ r a.(∀a.r a->x a) -> f r a -> x a)
         -> μ(*→*) f a -> x a
mit(*→*) phi (In(*→*) z) = phi (mit(*→*) phi) z

-- above is Haskell (with some GHC extensions)
-- below is Haskell-ish pseudocode

newtype μ(Nat→*) (f::(Nat->*)->(Nat->*)) {n::Nat}
  = In(Nat→*) (f (μ(Nat→*) f)) {n}

data VecF (a::*) (r::Nat->*) {n::Nat} where
  VCons :: a -> r n -> VecF a r {S n}
  VNil  :: VecF a r {Z}
type Vec a {n::Nat} = μ(Nat→*) (VecF a) {n}
vcons x xs = In(Nat→*) (VCons x xs)
vnil       = In(Nat→*) VNil

mit(Nat→*)::(∀ r n.(∀n.r{n}->x{n})->f r {n}->x{n})
         -> μ(Nat→*) f {n} -> x{n}
mit(Nat→*) phi (In(Nat→*) z) = phi (mit(Nat→*) phi) z
```

Figure 4.7: Two-level types and their Mendler-style iterators in Haskell.

consistent calculus.

However, it is important to note that there does not exist an embedding that arbitrarily destructs (i.e., pattern matches away) the $\mathtt{In}_\kappa$ constructor. It is known that combining arbitrary recursive datatypes with the ability to destruct (or unroll) their values is powerful enough to define non-terminating computations in a type-safe way, leading to logical inconsistency. Some systems maintain consistency by restricting which recursive datatypes can be defined, but allow arbitrary unrolling. In Mendler style, one can define any datatype, but unrolling recursive values is restricted to Mendler-style recursion combinators. Such datatypes and Mendler-style recursion combinators are embeddable in F_i (and some in Fix_i). The family of Mendler-style recursion schemes are quite expressive, capturing at least iteration, primitive recursion, and course-of-values recursion.

Example 4.2.1. Datatype of λ-terms in context

```
data Lam ( C: Nat -> * ) { i: Nat } where
  LVar : C{i} -> Lam{i}
  LApp : Lam{i} -> Lam{i} -> Lam{i}
  LAbs : Lam{S i} -> Lam{i}
```

is encoded as:

$$\begin{aligned}\mathtt{Lam} \triangleq \lambda C^{\mathtt{Nat}\to *}\lambda i^{\mathtt{Nat}}.\forall X^{\mathtt{Nat}\to *}.\\ (\forall j^{\mathtt{Nat}}.C\{j\}\to X\{j\})\\ \to(\forall j^{\mathtt{Nat}}.X\{j\}\to X\{j\}\to X\{j\})\\ \to(\forall j^{\mathtt{Nat}}.X\{\mathtt{S}\,j\}\to X\{j\})\\ \to X\{i\}\end{aligned}$$

For a concrete representation one can consider `Lam Fin` where

```
data Fin { i: Nat } where
  FZ : Fin{S i}
  FS : Fin{i} -> Fin{S i}
```

This is encoded as

$$\mathtt{Fin} \triangleq \lambda i^{\mathtt{Nat}}.\forall X^{\mathtt{Nat}\to *}.(\forall j^{\mathtt{Nat}}.X\{\mathtt{S}\,j\})\to(\forall j^{\mathtt{Nat}}.X\{j\}\to X\{\mathtt{S}\,j\})\to X\{i\}$$

$$\text{notation:}\qquad \boldsymbol{\lambda}\mathbb{I}^{\kappa}.F = \lambda I_1^{K_1}.\cdots.\lambda I_n^{K_n}.F \qquad\qquad F\mathbb{I} = FI_1\cdots I_n$$
$$\forall\mathbb{I}^{\kappa}.B = \forall I_1^{K_1}.\cdots.\forall I_n^{K_n}.B \qquad\qquad F \overset{\kappa}{\rightarrow} G = \forall\mathbb{I}^{\kappa}.F\mathbb{I} \to G\mathbb{I}$$

where

$\kappa = K_1 \to \cdots \to K_n \to *$ and I_i is an index variable (i_i) when K_i is a type,

$\mathbb{I} = I_1, \ \ldots\ \ldots\ , I_n$ a type constructor variable (X_i) otherwise (i.e., $K_i = \kappa_i$).

$$\begin{array}{lcll}
\mu_{\kappa} & : & (\kappa \to \kappa) \to \kappa & = \lambda F^{\kappa\to\kappa}.\boldsymbol{\lambda}\mathbb{I}^{\kappa}.\forall X^{\kappa}.(\forall X_r^{\kappa}.(X_r \overset{\kappa}{\rightarrow} X) \to (FX_r \overset{\kappa}{\rightarrow} X)) \to X\mathbb{I} \\
\mu_{*} & : & (* \to *) \to * & = \lambda F^{*\to *}. \qquad \forall X^{*}.(\forall X_r^{*}.(X_r \to X) \to (F\,X_r \to X)) \to X \\
\mu_{*\to *} & : & ((* \to *) \to (* \to *)) \to (* \to *) & \\
 & = & \lambda F^{(*\to *)\to(*\to *)}.\lambda X_1^{*}.\forall X^{*\to *}.(\forall X_r^{*\to *}.(\forall X_1^{*}.X_r X_1 \to X X_1) \to (\forall X_1^{*}.F\,X_r X_1 \to X X_1)) \to X X_1 & \\
\mu_{\mathtt{Nat}\to *} & : & ((\mathtt{Nat} \to *) \to (\mathtt{Nat} \to *)) \to (\mathtt{Nat} \to *) & \\
 & = & \lambda F^{(\mathtt{Nat}\to *)\to(\mathtt{Nat}\to *)}.\lambda i_1^{\mathtt{Nat}}.\forall X^{\mathtt{Nat}\to *}.(\forall X_r^{\mathtt{Nat}\to *}.(\forall i_1^{\mathtt{Nat}}.X_r i_1 \to X i_1) \to (\forall i_1^{\mathtt{Nat}}.F\,X_r i_1 \to X i_1)) \to X i_1 &
\end{array}$$

$$\begin{array}{lcll}
\mathtt{In}_{\kappa} & : & \forall F^{\kappa\to\kappa}.F(\mu_{\kappa}F) \overset{\kappa}{\rightarrow} \mu_{\kappa}F & = \lambda x_r.\lambda x_{\varphi}.x_{\varphi}\,(\mathtt{mit}_{\kappa}\ x_{\varphi})\,x_r \\
\mathtt{mit}_{\kappa} & : & \forall F^{\kappa\to\kappa}.\forall X^{\kappa}.(\forall X_r^{\kappa}.(X_r \overset{\kappa}{\rightarrow} X) \to (FX_r \overset{\kappa}{\rightarrow} X)) \to (\mu_{\kappa}F \overset{\kappa}{\rightarrow} X) & = \lambda x_{\varphi}.\lambda x_r.x_r\,x_{\varphi}
\end{array}$$

Figure 4.8: Embedding of the recursive operators (μ_{κ}), their data constructors ($\mathtt{In}_{\kappa}$), and the Mendler-style iterators ($\mathtt{mit}_{\kappa}$) in F_i.

4.2.3 Leibniz index equality

The quantification over type-indexed arrow kind available in System F_i allows the definition of *Leibniz-equality type* constructor $\mathtt{LEq}_A : A \to A \to *$ on closed types A, defined as follows:

$$\mathtt{LEq}_A \triangleq \lambda i^A.\, \lambda j^A.\, \forall X^{A\to *}.\, X\{i\} \to X\{j\}$$

Observe that the following types are inhabited:

(Reflexive) $\forall i^A.\, \mathtt{LEq}_A\{i\}\{i\}$

(Transitive) $\forall i^A.\, \forall j^A.\, \forall k^A.\, \mathtt{LEq}_A\{i\}\{j\} \to \mathtt{LEq}_A\{j\}\{k\} \to \mathtt{LEq}_A\{i\}\{k\}$

(Logical) $\forall i^A.\, \forall j^A.\, \mathtt{LEq}_A\{i\}\{j\} \to \forall f^{A\to B}.\, \mathtt{LEq}_B\{f\, i\}\{f\, j\}$

$\forall f^{A\to B}.\, \forall g^{A\to B}.\, \mathtt{LEq}_{A\to B}\{f\}\{g\} \to \forall i^A.\, \mathtt{LEq}_B\{f\, i\}\{g\, i\}$

In addition to the above, one also has the inhabitation of the following type:[4]

(Symmetric) $\forall i^A.\, \forall j^A.\, \mathtt{LEq}_A\{i\}\{j\} \to \mathtt{LEq}_A\{j\}\{i\}$

Hence Leibniz equality is a congruence.

In applications, the types $\mathtt{LEq}_A$ are useful in constraining the term-indexing of datatypes. A general construction is given by the type constructors $\mathtt{Ran}_{A,B} : (A \to B) \to (A \to *) \to B \to *$. These are defined as

$$\mathtt{Ran}_{A,B} \triangleq \lambda f^{A\to B}.\, \lambda X^{A\to *}.\, \lambda j^B.\forall i^A.\, \mathtt{LEq}_B\{j\}\{f\, i\} \to X\{i\}$$

and are, in spirit, right Kan extensions, a notion that is extensively used in programming, e.g. [5, 52].

One of their usefulness comes from the fact that the following type is inhabited by a section:

$$\forall Y^{B\to *}.\, \forall X^{A\to *}.\, \forall f^{A\to B}.\, \Big(\forall i^A.\, Y\{f\, i\} \to X\{i\}\Big) \to \Big(\forall j^B.\, Y\{j\} \to (\mathtt{Ran}_{A,B}\{f\}X)\{j\}\Big)$$

[4] Intuitively, this is obvious, since we can swap the order of consecutive universal quantification over indices. That is, from $(\forall i^A.\, \forall j^A.\, \cdots)$ to $(\forall j^A.\, \forall i^A.\, \cdots)$.

This allows one to represent functions from input datatypes with constrained indices as plain indexed functions, and vice versa. For instance, by means of the iterators of the previous section, one can define a vector tail function of type

$$\forall X^{*}.\, \forall j^{\mathtt{Nat}}.\, \mathtt{Vec}\, X\, \{j\} \to \Big(\mathtt{Ran}_{\mathtt{Nat},\mathtt{Nat}}\, \{\mathtt{S}\}(\mathtt{Vec}\, X)\Big)\{j\}$$

and retract it to one of type

$$\forall X^{*}.\, \forall i^{\mathtt{Nat}}.\, \mathtt{Vec}\, X\, \{\mathtt{S}\, i\} \to \mathtt{Vec}\, X\, \{i\}\ .$$

Analogously, one can use an iterator to define a single-variable capture-avoiding substitution function of type

$$\forall i^{\mathtt{Nat}}.\, (\mathtt{Lam}\, \mathtt{Fin})\{i\} \to \Big(\mathtt{Ran}_{\mathtt{Nat},\mathtt{Nat}}\{\mathtt{S}\}(\lambda j^{\mathtt{Nat}}.\, \mathtt{Lam}\, \mathtt{Fin}\{j\} \to \mathtt{Lam}\, \mathtt{Fin}\{j\})\Big)\{i\}$$

and then retract it to one of type

$$\forall i^{\mathtt{Nat}}.\, (\mathtt{Lam}\, \mathtt{Fin})\{\mathtt{S}\, i\} \to (\mathtt{Lam}\, \mathtt{Fin})\{i\} \to (\mathtt{Lam}\, \mathtt{Fin})\{i\}\ .$$

Type constructors $\mathtt{Lan}_{A,B} : (A \to B) \to (A \to *) \to B \to *$, which are in spirit left Kan extensions, permit the encoding of functions of type $(\forall i^{A}.\, F\{i\} \to G\{t\, i\})$ for $F : A \to *$, $G : B \to *$, and $t : A \to B$, as functions of type

$$\forall j^{B}.\, (\mathtt{Lan}_{A,B}\{t\}F)\{j\} \to G\{j\}\ .$$

Left Kan extensions are dual to right Kan extensions, but to define them as such, one needs existential and product types. In formalisms without them, these have to be encoded. This can be done as follows:

$$\mathtt{Lan}_{A,B} \triangleq \lambda f^{A\to B}.\, \lambda X^{A\to *}.\, \lambda j^{B}.\, \forall Z^{*}.\, (\forall i^{A}.\, \mathtt{LEq}_{B}\{f\, i\}\{j\} \to X\{i\} \to Z) \to Z$$

The type

$$\forall X^{A\to *}.\, \forall Y^{B\to *}.\, \forall f^{A\to B}.\, (\forall i^{A}.\, X\{i\} \to Y\{f\, i\}) \to (\forall j^{B}.\, (\mathtt{Lan}_{A,B}\{f\}X)\{j\} \to Y\{j\})$$

is thus inhabited by a section, providing a retractable coercion between the two functional representations.

Left Kan extensions come with a canonical section of type

$$\forall f^{A\to B}.\,\forall X^{A\to *}.\,\forall i^{A}.\,X\{i\} \to (\mathtt{Lan}_{A,B}\{f\}X)\{f\,i\}$$

that, according to a reindexing function $t : A \to B$, embeds an A-indexed type F (at index s) into the B-indexed type $\mathtt{Lan}_{A,B}\{t\}F$ (at index $t\,s$). For instance, the type constructor $\mathtt{Lan}_{A,A\times A}\{\lambda x.\,\mathtt{pair}\,x\,x\}$ embeds arrays of types into matrices along the diagonal; while the type constructors $\mathtt{Lan}_{A\times A,A}\{\mathtt{fst}\}$ and $\mathtt{Lan}_{A\times A,A}\{\mathtt{snd}\}$ respectively encapsulate matrices of types as arrays by columns and rows.

4.3 METATHEORY

The expectation is that System F_i has all the nice properties of System F_ω, yet is more expressive because of the addition of term-indexed types.

We show some basic well-formedness properties for the judgments of F_i in Section 4.3.1. We prove erasure properties of F_i, which captures the idea that indices are erasable because they are irrelevant for reduction in Section 4.3.2. We show strong normalization, logical consistence, and subject reduction for F_i by reasoning about well-known calculi related to F_i in Section 4.3.3.

4.3.1 Well-formedness properties and substitution lemmas

We need to show that kinding and typing derivations give well-formed results under well-formed contexts. That is, kinding derivations ($\Delta \vdash F : \kappa$) result in well-sorted kinds ($\vdash \kappa$) under well-formed type-level contexts ($\vdash \Delta$) (Proposition 4.3.1), and typing derivations ($\Delta;\Gamma \vdash t : A$) result in well-kinded types ($\Delta;\Gamma \vdash A : *$) under well-formed type and term-level contexts (Proposition 4.3.2).

Proposition 4.3.1. $\dfrac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\vdash \kappa : \Box}$

Proposition 4.3.2. $\dfrac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$

We can prove these well-formedness properties by induction over the judgment and using the well-formness lemmas on equalities (Lemmas 4.3.1, 4.3.2, and 4.3.3) and substitution lemma (Lemma 4.3.4). The proofs for Propositions 4.3.1 and 4.3.2 are mutually inductive. Hence, we prove these two propositions at the same time, using a combined judgment J, which is either a kinding judgment or a typing judgment (i.e., $J ::= \Delta \vdash F : \kappa \mid \Delta; \Gamma \vdash t : A$). See Appendix B for the detailed proofs.

Lemma 4.3.1 (kind equality is well-sorted)**.** $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa : \Box \quad \vdash \kappa' : \Box}$

Proof. By induction on the derivation of kind equality and by the sorting rules. □

Lemma 4.3.2 (type constructor equality is well-kinded)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F : \kappa \quad \Delta \vdash F' : \kappa}$$

Proof. By induction on the derivation of type constructor equality and by the kinding rules. Also use the type substitution lemma (Lemma 4.3.4(1)) and the index substitution lemma (Lemma 4.3.4(2)). □

Lemma 4.3.3 (term equality is well-typed)**.**

$$\frac{\Delta, \Gamma \vdash t = t' : A}{\Delta, \Gamma \vdash t : A \quad \Delta, \Gamma \vdash t' : A}$$

Proof. By induction on the derivation of term equality and by the typing rules. Also use the term substitution lemma (Lemma 4.3.4(3)). □

The proofs for the three lemmas above are straightforward once we have dealt with the interesting cases for the equality rules involving substitution. We can prove those interesting cases by applying the substitution lemmas. The other cases fall into two categories: first, the equality rules that follow the same structure as

the sorting, kinding, and typing rules; and second, the reflexive rules and the transitive rules. The proof for the first category can be proved by induction and applying the corresponding sorting, kinding, and typing rules. The proof for the second category can be proved simply by induction.

Lemma 4.3.4 (substitution)**.**

1. (type substitution) $\dfrac{\Delta, X^{\kappa} \vdash F : \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash F[G/X] : \kappa'}$
2. (index substitution) $\dfrac{\Delta, i^{A} \vdash F : \kappa \qquad \Delta; \cdot \vdash s : A}{\Delta \vdash F[s/i] : \kappa}$
3. (term substitution) $\dfrac{\Delta; \Gamma, x : A \vdash t : B \qquad \Delta; \Gamma \vdash s : A}{\Delta, \Gamma \vdash t[s/x] : B}$

The substitution lemma is fairly standard and comparable to substitution lemmas in other well-known systems such as F_{ω} or ICC.

4.3.2 Erasure properties

We define a meta-operation of index erasure that projects F_i-types to F_{ω}-types.

Definition 4.3.1 (index erasure)**.**

$\boxed{\kappa^{\circ}}$ $\quad *^{\circ} = * \qquad (\kappa_1 \to \kappa_2)^{\circ} = {\kappa_1}^{\circ} \to {\kappa_2}^{\circ} \qquad (A \to \kappa)^{\circ} = \kappa^{\circ}$

$\boxed{F^{\circ}}$ $\quad X^{\circ} = X \qquad (A \to B)^{\circ} = A^{\circ} \to B^{\circ}$

$(\lambda X^{\kappa}.F)^{\circ} = \lambda X^{{\kappa}^{\circ}}.F^{\circ} \qquad (\lambda i^{A}.F)^{\circ} = F^{\circ}$

$(F\ G)^{\circ} = F^{\circ}\ G^{\circ} \qquad (F\ \{s\})^{\circ} = F^{\circ}$

$(\forall X^{\kappa}.B)^{\circ} = \forall X^{{\kappa}^{\circ}}.B^{\circ} \qquad (\forall i^{A}.B)^{\circ} = B^{\circ}$

$\boxed{\Delta^{\circ}}$ $\quad \cdot^{\circ} = \cdot \qquad (\Delta, X^{\kappa})^{\circ} = \Delta^{\circ}, X^{{\kappa}^{\circ}} \qquad (\Delta, i^{A})^{\circ} = \Delta^{\circ}$

$\boxed{\Gamma^{\circ}}$ $\quad \cdot^{\circ} = \cdot \qquad (\Gamma, x : A)^{\circ} = \Gamma^{\circ}, x : A^{\circ}$

Example 4.3.1. The meta-operation of index erasure simply discards all indexing information. The effect of this on most datatypes is to project the indexing invariants while retaining the type structure. This is clearly seen for the vector type constructor `Vec` whose index erasure is the list type constructor `List`, as in Figure 4.6. One can however build pathological examples. For instance, the type $\mathtt{P}_A \triangleq \forall i^A.\, \forall j^A.\, \mathtt{LEq}_A\{i\}\{j\}$ has index erasure $\mathtt{Unit} \triangleq \forall X^*.\, X \to X$.

Theorem 4.3.1 (index erasure on well-sorted kinds)**.** $\dfrac{\vdash \kappa : \Box}{\vdash \kappa^\circ : \Box}$

Proof. By induction on the sorting derivation. □

Remark 4.3.1. For any well-sorted kind κ in F_i, κ° is a kind in F_ω.

Theorem 4.3.2 (index erasure on well-formed type-level contexts)**.**

$$\frac{\vdash \Delta}{\vdash \Delta^\circ}$$

Proof. By induction on the derivation for well-formed type-level context and by Theorem 4.3.1. □

Remark 4.3.2. For any well-formed type-level context Δ in F_i, Δ° is a well-formed type-level context in F_ω.

Theorem 4.3.3 (index erasure on kind equality)**.** $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa^\circ = \kappa'^\circ : \Box}$

Proof. By induction on the kind equality judgement. □

Remark 4.3.3. For any well-sorted kind equality $\vdash \kappa = \kappa' : \Box$ in F_i, $\vdash \kappa^\circ = \kappa'^\circ : \Box$ is a well-sorted kind equality in F_ω.

The three theorems above on kinds are rather simple to prove as there is no need to consider mutual recursion in the definition of kinds because of the erasure operation on kinds. Recall that this operation discards the type (A) appearing in the index arrow type ($A \to \kappa$). So, there is no need to consider the types appearing in kinds and the index terms appearing in those types after the erasure.

Theorem 4.3.4 (index erasure on well-kinded type constructors)**.**

$$\frac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\Delta^\circ \vdash F^\circ : \kappa^\circ}$$

Proof. By induction on the kinding derivation.

case (Var) Use Theorem 4.3.2.

case $(Conv)$ By induction and using Theorem 4.3.3.

case (λ) By induction and using Theorem 4.3.1.

case $(@)$ By induction.

case (λi) We need to show that $\Delta^\circ \vdash (\lambda i^A.F)^\circ : (A \to \kappa)^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 4.3.1.

By induction, we know that $(\Delta, i^A)^\circ \vdash F^\circ : \kappa^\circ$, which simplifies $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 4.3.1.

case $(@i)$ We need to show that $\Delta^\circ \vdash (F\ \{s\})^\circ : \kappa^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 4.3.1.

By induction we know that $\Delta^\circ \vdash F^\circ : (A \to \kappa)^\circ$, which simplifies to $\Delta^\circ \vdash F^\circ : \kappa^\circ$ by Definition 4.3.1.

case $(\to)$ By induction.

case $(\forall)$ We need to show that $\Delta^\circ \vdash (\forall X^\kappa.B)^\circ : *^\circ$, which simplifies to $\Delta^\circ \vdash \forall X^{\kappa^\circ}.B^\circ : *$ by Definition 4.3.1.

From Theorem 4.3.1, we know that $\vdash \kappa^\circ : \Box$.

By induction we know that $(\Delta, X^\kappa)^\circ \vdash B^\circ : *^\circ$, which simplifies to $\Delta^\circ, X^{\kappa^\circ} \vdash B^\circ : *$ by Definition 4.3.1.

From the kinding rule $(\forall)$, we get exactly what we need to show: $\Delta^\circ \vdash \forall X^{\kappa^\circ}.B^\circ : *$.

case $(\forall i)$ We need to show that $\Delta^\circ \vdash (\forall i^A.B)^\circ : *^\circ$, which simplifies to $\Delta^\circ \vdash B^\circ : *$ by Definition 4.3.1.

By induction we know that $(\Delta, i^A)^\circ \vdash B^\circ : *^\circ$, which simplifies $\Delta^\circ \vdash B^\circ : *$ by Definition 4.3.1.

□

Theorem 4.3.5 (index erasure on type constructor equality)**.**

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ}$$

Proof. By induction on the derivation of type constructor equality.

Most cases are proven by applying the induction hypothesis and sometimes using Proposition 4.3.1.

The only interesting cases that are worth elaborating are the equality rules involving substitution. There are two such rules.

$$\frac{\Delta, X^\kappa \vdash F : \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^\kappa.F)\, G = F[G/X] : \kappa'}$$

We need to show $\Delta^\circ \vdash ((\lambda X^\kappa.F)\, G)^\circ = (F[G/X])^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ \vdash (\lambda X^{\kappa^\circ}.F^\circ)\, G^\circ = (F[G/X])^\circ : \kappa'^\circ$ by Definition 4.3.1.

By induction, we know that $(\Delta, X^\kappa)^\circ \vdash F^\circ : \kappa'^\circ$, which simplifies to $\Delta^\circ, X^{\kappa^\circ} \vdash F^\circ : \kappa'^\circ$ by Definition 4.3.1.

Using the kinding rule (λ), we get $\Delta^\circ \vdash \lambda X^{\kappa^\circ}.F^\circ : \kappa^\circ \to \kappa'^\circ$.

Using the kinding rule (@), we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ}.F^\circ)G^\circ : \kappa'^\circ$.

Using the very same equality rule of this case,
we get $\Delta^\circ \vdash (\lambda X^{\kappa^\circ} F^\circ)G^\circ = F^\circ[G^\circ/X] : \kappa'^\circ$.

We only need to check is $(F[G/X])^\circ = F^\circ[G^\circ/X]$, which is easy to see.

$$\frac{\Delta, i^A \vdash F : \kappa \qquad \Delta; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

By induction we know that $\Delta^\circ \vdash F^\circ : \kappa^\circ$.

The erasure of the left hand side of the equality is
$((\lambda i^A.F)\,\{s\})^\circ = (\lambda i^A.F)^\circ = F^\circ$.

We only need to show is $(F[s/i])^\circ = F^\circ$, which is obvious since index variables can only occur in index terms that are always erased. Recall the index erasure over type constructors in Definition 4.3.1, in particular, $(\lambda i^A.F)^\circ = F^\circ$, $(F\{s\})^\circ = F^\circ$, and $(\forall i^A.B)^\circ = B^\circ$. □

Remark 4.3.4. For any well-kinded type constructor equality $\Delta \vdash F = F' : \kappa$ in F_i, $\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ$ is a well-kinded type constructor equality in F_ω.

The proofs for the two theorems on type constructors above need not consider mutual recursion in the type constructor definition because of the erasure operation. Recall that the erasure operation on type constructors discards the index term (s) appearing in the index application ($F\ \{s\}$). So, there is no need to consider the index terms appearing in the types after the erasure.

Theorem 4.3.6 (index erasure on well-formed term-level contexts)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash \Gamma^\circ}$$

Proof. By induction on Γ.

case ($\Gamma = \cdot$) It trivially holds.

case ($\Gamma = \Gamma', x : A$) We know that $\Delta \vdash \Gamma'$ and $\Delta \vdash A : *$ by the well-formedness rules and that $\Delta^\circ \vdash \Gamma'^\circ$ by induction.

From $\Delta \vdash A : *$, we know that $\Delta^\circ \vdash A^\circ : *$ by Theorem 4.3.4.

We know that $\Delta^\circ \vdash \Gamma'^\circ, x : A^\circ$ from $\Delta^\circ \vdash \Gamma'^\circ$ and $\Delta^\circ \vdash A^\circ : *$ by the well-formedness rules.

Because $\Gamma'^\circ, x : A^\circ = (\Gamma', x : A)^\circ = \Gamma^\circ$ by definition, we know that $\Delta^\circ \vdash \Gamma^\circ$.

□

Theorem 4.3.7 (index erasure on index-free well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; \Gamma^\circ \vdash t : A^\circ} \ (\mathrm{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset)$$

Proof. By induction on the typing derivation. Interesting cases are the index related rules $(: i)$, $(\forall I i)$, and $(\forall E i)$. Proofs for the other cases are straightforward by induction and application of other erasure theorems corresponding to the judgment forms.

case $(:)$ By Theorem 4.3.6, we know that $\Delta^\circ \vdash \Gamma^\circ$ when $\Delta \vdash \Gamma$. By the definition of erasure in term-level context, we know that $(x : A^\circ) \in \Gamma^\circ$ when $(x : A) \in \Gamma$.

case $(: i)$ Vacuously true because t does not contain any index variables (i.e., $\mathrm{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset$).

case $(\to I)$ By Theorem 4.3.4, we know that $\cdot \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ, x : A^\circ \vdash t^\circ : B^\circ$. Applying the $(\to I)$ rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash \lambda x.t^\circ : A^\circ \to B^\circ$.

case $(\to E)$ Straightforward by induction.

case $(\forall I)$ By Theorem 4.3.1, we know that $\vdash \kappa^\circ : \Box$. By induction, we know that $\Delta^\circ, X^{\kappa^\circ}; \Gamma^\circ \vdash t : B^\circ$. Applying the $(\forall I)$ rule to what we know, we have $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$.

case $(\forall E)$ By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : \forall X^{\kappa^\circ}.B^\circ$. By Theorem 4.3.4, we know that $\Delta^\circ \vdash G^\circ : \kappa^\circ$. Applying the $(\forall E)$ rule, we have $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ[G^\circ/X]$.

case $(\forall I i)$ By Theorem 4.3.4, we know that $\cdot \vdash A^\circ : *$. By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what is required since $(\forall i^A.B)^\circ = B^\circ$.

case $(\forall E i)$ By induction, we know that $\Delta^\circ; \Gamma^\circ \vdash t : B^\circ$, which is what is required since $(B[s/i])^\circ = B^\circ$.

case $(=)$ By Theorem 4.3.5 and induction.

□

Example 4.3.2. The theorem yields that the pathological type $\mathtt{P}_A$ of Example 4.3.1 is not inhabited, as it is impossible to have both $t : \mathtt{P}_A$ and $t : (\mathtt{P}_A)^\circ = \mathtt{Unit}$. It follows as a corollary that the implication of Theorem 4.3.7 does not admit a converse.

In this context, for $A = \mathtt{Void}$, note that even though one has $i^{\mathtt{Void}}; \cdot \vdash \lambda x.\, i : \forall j^{\mathtt{Void}}.\, \forall X^{\mathtt{Void} \to *}.\, X\{i\} \to X\{j\}$, this open term cannot be closed by rule $(\forall Ii)$ because of its side condition. This is in stark contrast to what is possible in calculi with -ull type dependency. In System F_i, the index variables in a type-level context Δ cannot appear dynamically at the term-level. Conversely, term variables in the term-level context Γ cannot be used for instantiation of index polymorphic types (rule $(\forall Ei)$).

We introduce an index variable selection meta-operation that selects all the index variable bindings from the type-level context.

Definition 4.3.2 (index variable selection)**.**

$$\cdot^\bullet = \cdot \qquad (\Delta, X^\kappa)^\bullet = \Delta^\bullet \qquad (\Delta, i^A)^\bullet = \Delta^\bullet, i : A$$

Theorem 4.3.8 (index erasure on well-formed term-level contexts prepended by index variable selection)**.**

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash (\Delta^\bullet, \Gamma)^\circ}$$

Proof. Straightforward, by Theorem 4.3.6 and the typing rule $(: i)$. □

The following result is the appropriate version of Theorem 4.3.7 without the side condition therein.

Theorem 4.3.9 (index erasure on well-typed terms)**.**

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; (\Delta^\bullet, \Gamma)^\circ \vdash t : A^\circ}$$

Proof. The proof is almost the same as that of Theorem 4.3.7, except for the $(: i)$ case. The proof for the rule $(: i)$ case is easy because $(i : A) \in \Delta^\bullet$ when $i^A \in \Delta$ by definition of the index variable selection operation. Prepending indices to Γ that come from Δ do not affect the proof for the other cases. □

4.3.3 Strong normalization and logical consistency

Strong normalization is a corollary of the erasure property since we know that System F_ω is strongly normalizing.

Logical consistency is immediate because System F_i is a strict subset of the *restricted implicit calculus* [67], which is in turn a restriction of ICC [68]. Subject reduction is also immediate for the same reason.

We can also give a more direct proof of logical consistency by showing that the void type $\forall X^*.X$ is uninhabited in F_i. By type erasure, no terms inhabit F_i-types other than the corresponding F_ω-types. Since we already know that the void type $\forall X^*.X$ is uninhabited in F_ω, it must be the case that the void type is uninhabited in F_i.

Chapter 5

SYSTEM Fix_i

In this chapter, we investigate how System F_ω needs to be extended in order to prove termination of the Mendler-style primitive recursion (**msfit**) by a reduction preserving embedding. Recall that **msfit** supplies access to immediate subterms as well as the value of recursive calls over those subterms. The factorial function is the classic example of a computation described by primitive recursion that cannot be simulated efficiently by iteration.

It is fairly well known that there cannot be a reduction preserving embedding of primitive recursion[1] in System F. A proof of this is outlined in the paper *Induction is not derivable in second-order dependent type theory* [39]. For similar reasons, researchers strongly believe that there is no reduction-preserving embedding of primitive recursion in System F_ω. Fortunately, all hope is not lost for finding a reduction preserving embedding in a relatively simple calculus. Abel and Matthes [3] have designed Fix_ω, an extension of F_ω, that embeds primitive recursion with the desired reduction behavior. Their embedding relies on a novel use of the two extensions to F_ω – polarized kinds and an equi-recursive fixpoint type operator – in order to define primitive recursion within Fix_ω.

As a natural extension of these ideas, we present Fix_i, an extension of Fix_ω with erasable term-indices, that embeds primitive recursion over term-indexed datatypes as well as type-indexed datatypes and regular datatypes.

[1] Although we can define primitive recursion for positive datatypes in terms of iteration, which is embeddable in System F, such an embedding would not be reduction preserving. That is, it would require more reduction steps than the usual definition of primitive recursion.

The organization of this chapter is analogous to Chapter 4, where we added term-indices to F_ω to obtain F_i. Here, we add term indices to Fix_ω to obtain Fix_i. We describe Fix_i focusing on its differences from F_i. Readers may refer back to Chapter 4 for those details that remain unchanged from System F_i.

We describe syntax and typing rules (Section 5.1), illustrate embeddings of primitive recursion (Section 5.2), and discuss embeddings of course-of-values recursion (Section 5.3). Finally, we discuss the metatheory of Fix_i (Section 5.4).

5.1 SYSTEM Fix_i

The syntax and rules of System F_i are described in Figures 5.1, 5.2, 5.3, and 5.4. The extensions new to System Fix_i that are not original parts of either System F_ω or System Fix_ω are highlighted by either dashed boxes or grey boxes, respectively.

The extensions that not originally part of System Fix_ω are highlighted by grey boxes. Those extensions support term indexing. Eliding all the grey boxes from Figures 5.1, 5.2, 5.3, and 5.4, one obtains a version of System Fix_ω with typing contexts separated into two parts.[2]

The extensions that are not originally part of System F_ω but present in System Fix_ω are highlighted by dashed boxes. Those extensions support equi-recursive types. Eliding all the dashed boxes, as well as all the grey boxes, from Figures 5.1, 5.2, 5.3 and 5.4, one obtains the Curry-style System F_ω with typing contexts separated into two parts.

The grey-boxed extensions for term-indexing are essentially the same as those grey-boxed extensions in System F_i (Section 4.1). Hence, we will only focus our description on the dashed-box extensions regarding polarities (Section 5.1.1) and equi-recursive types (Section 5.1.2).

[2] The original description of Fix_ω[3] has one combined typing context.

Syntax:

Sort	$\Box$	
Term Variables	x, i	
Type Constructor Variables	X	
Polarities	$\bar{p}$	$::= + \mid - \mid 0$
Kinds	κ	$::= * \mid \bar{p}\kappa \to \kappa \mid A \to \kappa$
Type Constructors	A, B, F, G	$::= X \mid A \to B \mid \mathsf{fix} F$
		$\mid \lambda X^{p\kappa}.F \mid F\,G \mid \forall X^{\kappa}.B$
		$\mid \lambda i^{A}.F \mid F\,\{s\} \mid \forall i^{A}.B$
Terms	r, s, t	$::= x \mid \lambda x.t \mid r\ s$
Typing Contexts	Δ	$::= \cdot \mid \Delta, X^{p\kappa} \mid \Delta, i^{A}$
	Γ	$::= \cdot \mid \Gamma, x : A$

Reduction: $\boxed{t \rightsquigarrow t'}$

$$\frac{}{(\lambda x.t)\ s \rightsquigarrow t[s/x]} \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'} \qquad \frac{r \rightsquigarrow r'}{r\ s \rightsquigarrow r'\ s} \qquad \frac{s \rightsquigarrow s'}{r\ s \rightsquigarrow r\ s'}$$

Figure 5.1: Syntax and Reduction rules of Fix_i.

Well-formed typing contexts:

$$\boxed{\vdash \Delta} \quad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Delta \quad \vdash \kappa : \square}{\vdash \Delta, \boxed{X^{p\kappa}}}(X \notin \mathrm{dom}(\Delta)) \qquad \frac{\vdash \Delta \quad \cdot \vdash A : *}{\vdash \Delta, i^A}(i \notin \mathrm{dom}(\Delta))$$

$$\boxed{\Delta \vdash \Gamma} \quad \frac{\vdash \Delta}{\boxed{0\Delta} \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash A : *}{\Delta \vdash \Gamma, x : A}(x \notin \mathrm{dom}(\Gamma))$$

Sorting: $\boxed{\vdash \kappa : \square}$ $(A)\dfrac{}{\vdash * : \square}$ $(R)\dfrac{\vdash \kappa : \square \quad \vdash \kappa' : \square}{\vdash \boxed{p\kappa} \to \kappa' : \square}$ $(Ri)\dfrac{\cdot \vdash A : * \quad \vdash \kappa : \square}{\vdash A \to \kappa : \square}$

Kinding: $\boxed{\Delta \vdash F : \kappa}$ $(Var)\dfrac{\boxed{X^{p\kappa}} \in \Delta \quad \vdash \Delta}{\Delta \vdash X : \kappa}\boxed{(p \in \{+, 0\})}$

$$(\to)\frac{\boxed{-\Delta} \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *} \qquad \boxed{(\mathsf{fix})\frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix} F : \kappa}}$$

$$(\lambda)\frac{\vdash \kappa : \square \quad \Delta, \boxed{X^{p\kappa}} \vdash F : \kappa'}{\Delta \vdash \lambda \boxed{X^{p\kappa}}.F : \boxed{p\kappa} \to \kappa'} \qquad (\lambda i)\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F : \kappa}{\Delta \vdash \lambda i^A.F : A \to \kappa}$$

$$(@)\frac{\Delta \vdash F : \boxed{p\kappa} \to \kappa' \quad \boxed{p\Delta} \vdash G : \kappa}{\Delta \vdash F\, G : \kappa'} \qquad (@i)\frac{\Delta \vdash F : A \to \kappa \quad \boxed{0\Delta}; \cdot \vdash s : A}{\Delta \vdash F\, \{s\} : \kappa}$$

$$(\forall)\frac{\vdash \kappa : \square \quad \Delta, \boxed{X^{0\kappa}} \vdash B : *}{\Delta \vdash \forall X^{\kappa}.B : *} \qquad (\forall i)\frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B : *}{\Delta \vdash \forall i^A.B : *}$$

$$(Conv)\frac{\Delta \vdash A : \kappa \quad \Delta \vdash \kappa = \kappa' : \square}{\Delta \vdash A : \kappa'}$$

Typing: $\boxed{\Delta; \Gamma \vdash t : A}$ $(:)\dfrac{(x : A) \in \Gamma \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : A}$ $(: i)\dfrac{i^A \in \Delta \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i : A}$

$$(\to I)\frac{\Delta \vdash A : * \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda x.t : A \to B} \qquad (\to E)\frac{\Delta; \Gamma \vdash r : A \to B \quad \Delta; \Gamma \vdash s : A}{\Delta; \Gamma \vdash r\ s : B}$$

$$(\forall I)\frac{\vdash \kappa : \square \quad \Delta, \boxed{X^{0\kappa}}; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B}(X \notin \mathrm{FV}(\Gamma)) \quad (\forall E)\frac{\Delta; \Gamma \vdash t : \forall X^{\kappa}.B \quad \Delta \vdash G : \kappa}{\Delta; \Gamma \vdash t : B[G/X]}$$

$$(\forall Ii)\frac{\cdot \vdash A : * \quad \Delta, i^A; \Gamma \vdash t : B}{\Delta; \Gamma \vdash t : \forall i^A.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix} (\forall Ei)\frac{\Delta; \Gamma \vdash t : \forall i^A.B \quad \Delta; \cdot \vdash s : A}{\Delta; \Gamma \vdash t : B[s/i]}$$

$$(=)\frac{\Delta; \Gamma \vdash t : A \quad \Delta \vdash A = B : *}{\Delta; \Gamma \vdash t : B}$$

Figure 5.2: Sorting, Kinding, and Typing rules of Fix_i.

Kind equality: $\boxed{\vdash \kappa = \kappa' : \Box}$

$$\frac{}{\vdash * = * : \Box}$$

$$\frac{\vdash \kappa_1 = \kappa_1' : \Box \quad \vdash \kappa_2 = \kappa_2' : \Box}{\vdash \boxed[dashed]{p}\kappa_1 \to \kappa_2 = \boxed[dashed]{p}\kappa_1' \to \kappa_2' : \Box} \qquad \frac{\cdot \vdash A = A' : * \quad \vdash \kappa = \kappa' : \Box}{\vdash A \to \kappa = A' \to \kappa' : \Box}$$

$$\frac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa' = \kappa : \Box} \qquad \frac{\vdash \kappa = \kappa' : \Box \quad \vdash \kappa' = \kappa'' : \Box}{\vdash \kappa = \kappa'' : \Box}$$

Type constructor equality: $\boxed{\Delta \vdash F = F' : \kappa}$

$$\frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}F = F(\mathsf{fix}F) : \kappa}$$

$$\frac{\Delta, \boxed[dashed]{X^{p\kappa}} \vdash F : \kappa' \quad \boxed[dashed]{p\Delta} \vdash G : \kappa}{\Delta \vdash (\lambda X^{p\kappa}.F)\, G = F[G/X] : \kappa'} \qquad \frac{\Delta, i^A \vdash F : \kappa \quad \boxed[dashed]{0\Delta}; \cdot \vdash s : A}{\Delta \vdash (\lambda i^A.F)\, \{s\} = F[s/i] : \kappa}$$

$$\frac{\Delta \vdash X : \kappa}{\Delta \vdash X = X : \kappa} \qquad \frac{-\Delta \vdash A = A' : * \quad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *} \qquad \frac{\Delta \vdash F = F' : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}F = \mathsf{fix}F' : \kappa}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, \boxed[dashed]{X^{p\kappa}} \vdash F = F' : \kappa'}{\Delta \vdash \lambda \boxed[dashed]{X^{p\kappa}}.F = \lambda \boxed[dashed]{X^{p\kappa}}.F' : \boxed[dashed]{\kappa} \to \kappa'} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash F = F' : \kappa}{\Delta \vdash \lambda i^A.F = \lambda i^A.F' : A \to \kappa}$$

$$\frac{\Delta \vdash F = F' : \boxed[dashed]{p\kappa} \to \kappa' \quad \boxed[dashed]{p\Delta} \vdash G = G' : \kappa}{\Delta \vdash F\, G = F'\, G' : \kappa'}$$

$$\frac{\Delta \vdash F = F' : A \to \kappa \quad \boxed[dashed]{0\Delta}; \cdot \vdash s = s' : A}{\Delta \vdash F\, \{s\} = F'\, \{s'\} : \kappa}$$

$$\frac{\vdash \kappa : \Box \quad \Delta, \boxed[dashed]{X^{0\kappa}} \vdash B = B' : *}{\Delta \vdash \forall X^{\kappa}.B = \forall X^{\kappa}.B' : *} \qquad \frac{\cdot \vdash A : * \quad \Delta, i^A \vdash B = B' : *}{\Delta \vdash \forall i^A.B = \forall i^A.B' : *}$$

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \frac{\Delta \vdash F = F' : \kappa \quad \Delta \vdash F' = F'' : \kappa}{\Delta \vdash F = F'' : \kappa}$$

Figure 5.3: Kind and type-constructor equality rules of Fix_i.

Term equality: $\boxed{\Delta;\Gamma \vdash t = t' : A}$

$$\frac{\Delta;\Gamma, x : A \vdash t : B \qquad \Delta;\Gamma \vdash s : A}{\Delta;\Gamma \vdash (\lambda x.t)\, s = t[s/x] : B} \qquad \frac{\Delta;\Gamma \vdash x : A}{\Delta;\Gamma \vdash x = x : A}$$

$$\frac{\Delta \vdash A : * \qquad \Delta;\Gamma, x : A \vdash t = t' : B}{\Delta;\Gamma \vdash \lambda x.t = \lambda x.t' : B}$$

$$\frac{\Delta;\Gamma \vdash r = r' : A \to B \qquad \Delta;\Gamma \vdash s = s' : A}{\Delta;\Gamma \vdash r\ s = r'\ s' : B}$$

$$\frac{\vdash \kappa : \Box \qquad \Delta, \boxed{X^{0\kappa}};\Gamma \vdash t = t' : B}{\Delta;\Gamma \vdash t = t' : \forall X^{\kappa}.B}(X \notin \mathrm{FV}(\Gamma))$$

$$\frac{\Delta;\Gamma \vdash t = t' : \forall X^{\kappa}.B \qquad \Delta \vdash G : \kappa}{\Delta;\Gamma \vdash t = t' : B[G/X]}$$

$$\frac{\cdot \vdash A : * \qquad \Delta, i^{A};\Gamma \vdash t = t' : B}{\Delta;\Gamma \vdash t = t' : \forall i^{A}.B}\begin{pmatrix} i \notin \mathrm{FV}(t), \\ i \notin \mathrm{FV}(t'), \\ i \notin \mathrm{FV}(\Gamma) \end{pmatrix}$$

$$\frac{\Delta;\Gamma \vdash t = t' : \forall i^{A}.B \qquad \Delta;\cdot \vdash s : A}{\Delta;\Gamma \vdash t = t' : B[s/i]}$$

$$\frac{\Delta;\Gamma \vdash t = t' : A}{\Delta;\Gamma \vdash t' = t : A} \qquad \frac{\Delta;\Gamma \vdash t = t' : A \qquad \Delta;\Gamma \vdash t' = t'' : A}{\Delta;\Gamma \vdash t = t'' : A}$$

Figure 5.4: Term equality rules of Fix_i.

5.1.1 Polarities

Polarities track how type constructor variables are used. A polarity (p) is either covariant ($+$), contravariant ($-$), or avariant (0). When a type variable is bound, its polarity is made explicit both at its binding site, and in the context. The avariant polarity (0) means that a variable can be used both covariantly and contravariantly[3] We prefix a kind by a polarity (i.e., $p\kappa$) to specify the variable's kind and polarity. For example,

$$\begin{array}{rcl} X_1^{-*}, X_2^{+*} \vdash X_1 \to X_2 : * & \text{justifies} & \lambda X_1^{-*}.\lambda X_2^{+*}.X_1 \to X_2 \\ X_1^{0*}, X_2^{0*} \vdash X_1 \to X_2 : * & \text{also justifies} & \lambda X_1^{0*}.\lambda X_2^{0*}.X_1 \to X_2 \\ X^{0*} \vdash X \to X : * & \text{justifies} & \lambda X^{0*}.X \to X \end{array}$$

Note that we can replace $+$ and $-$ in the first example with 0 as in the second example, since the variables of avariant polarity can be used in any position that is both in a covariant and contravariant position. In the third example, the polarity of X can be neither $+$ nor $-$, but must must be 0, since X appears in both covariant and contravariant positions.

Syntax using polarized kinds: The kind syntax is polarized. That is, the domain kind (κ) of an arrow kind ($p\kappa \to \kappa'$) must be prefixed by its polarity (p). Type abstractions ($\lambda X^{p\kappa}.F$) in the type syntax are annotated by polarity-prefixed kinds ($p\kappa$). Type constructor variables (X) bound in the type-level contexts (Δ) are likewise annotated by polarity-prefixed kinds ($p\kappa$). Note the syntax for extending the type-level context $\Delta, X^{p\kappa}$ in Figure 5.1. The kinding rule (λ) exploits all these three uses of polarized kinds – in type abstractions, in kind arrows, and in type-level contexts.

[3] The word "invariant" is sometimes used (see [3]), but we think this notation is quite misleading, The polarity 0 means that the system *does not care* about that variable's polarity, rather than indicating some unchanging set of properties about the variable's polarity.

Polarity operation on type-level context ($p\Delta$): The kinding judgment $\Delta \vdash F : \kappa$ assumes that F is in covariant positions. This is why the (Var) rule requires the polarity of X to be either $+$ or 0 but not $-$. To judge well-kindedness of type constructors in contravariant positions (e.g., A in $A \to B$), we should invert the polarities of all the type constructor variables in the context. This idea of inverting polarities in the context is captured by the $-\Delta$ operation in the kinding rule $(\to)$. More generally, the well-kindedness F expected to be used as p-polarity can be determined by the judgement $p\Delta \vdash F : \kappa$, where $p\Delta$ operation is defined as:

- when p is either $+$ or $-$

$$\begin{array}{lcll} p\cdot & = & \cdot & \\ p(\Delta, X^{p'\kappa}) & = & p\Delta, X^{(pp')\kappa} & \left(pp' \text{ is the usual sign product } \begin{array}{l} {}^{+p' = p'}_{-+ = -} \\ {}^{-- = +}_{-0 = 0} \end{array}\right) \\ p(\Delta, i^A) & = & p\Delta, i^A & \end{array}$$

- when $p = 0$

$$\begin{array}{lcll} 0\cdot & = & \cdot & \\ 0(\Delta, X^{0\kappa}) & = & 0\Delta, X^{0\kappa} & \\ 0(\Delta, X^{p\kappa}) & = & 0\Delta & (p \neq 0) \\ 0(\Delta, i^A) & = & 0\Delta, i^A & \end{array}$$

Note the use of $p\Delta$ operation in the kinding rule (@) in order to determine the well-kindedness of G expected to be used as p-polarity by the type constructor $F : p\kappa \to \kappa'$ being applied to G.

Where polarities are irrelevant (i.e., avariant): Polarities are irrelevant (i.e., avariant) for universally quantified variables and indices as well as in the typing rules. This is because the sole purpose of tracking polarities in Fix_i is to ensure that we only take the equi-recursive fixpoint over covariant type constructors, as in the kinding rule (μ). Note that we can only take fixpoints over type constructors of covariant arrow kinds whose domain and codomain coincide ($+\kappa \to \kappa$). We

can never take fixpoints over forall types (or universal quantification) and type constructor that expect an index because they are not of arrow kinds. Forall types are always of kind $*$ and type constructors that expect an index are of arrow kinds ($A \to \kappa$). So, we give universally quantified variables avariant polarity ($X^{0\kappa}$ in the ($\forall$) rule) and nullify polarities when type checking indices (0Δ in the ($@i$) rule). For similar reasons, we assume that type-level contexts are nullified in the typing rules; note 0Δ in the well-formedness condition for $\Delta \vdash \Gamma$ in Figure 5.2. That is, we always type check under nullified type-level context for all terms in general as well as for indices appearing in type applications. As a result, the typing rules of Fix_i have no dashed-box extensions except for $X^{0\kappa}$ in the generalization rule ($\forall I$) where we introduce a universally quantified type constructor variable.

5.1.2 Equi-recursive type operator fix

Fix_i provides the equi-recursive type operator fix. The kinding rule (fix) in Figure 5.2 is similar to the (μ) rule of System F_i (see Figure 4.2 in Section 4.1), but requires base structure F to be covariant (or positive), that is, $F : +\kappa \to \kappa$. This restriction on the polarity of F is caused by the equi-recursive nature of fix, that is, $\mathsf{fix}F = F(\mathsf{fix}F)$, described by the first type constructor equality rule inside a dashed box in Figure 5.3. Restricting the polarity of the base structure, to which fix can be applied, is necessary to maintain strong normalization. Adding equi-recursive types without restricting the polarity breaks the strong normalization because it amounts to having both formation and elimination of arbitrary iso-recursive types.

Note that there is no explicit term syntax that guides the conversion between $\mathsf{fix}F$ and $F(\mathsf{fix}F)$, unlike in iso-recursive[4] systems where In and unIn are term syntaxes that explicitly guide rolling (from μF to $F(\mu F)$) and unrolling (from $F(\mu F)$ to μF). Because $\mathsf{fix}F = F(\mathsf{fix}F)$ is given definitionally (i.e., by the equality

[4] In this dissertation, all the other fixpoint type operators except fix are iso-recursive.

$$
\begin{array}{rcll}
\bot & \triangleq & \forall X^*.X & : * \\
\mathsf{Unit} & \triangleq & \forall X.\lambda X^{0*}.X & : * \\
\times & \triangleq & \lambda X_1^{+*}.\lambda X_2^{+*}.\forall X^*.(X_1 \to X_2 \to X) \to X & : +* \to +* \to * \\
+ & \triangleq & \lambda X_1^{+*}.\lambda X_2^{+*}.\forall X^*.(X_1 \to X) \to (X_2 \to X) \to X & : +* \to +* \to * \\
\exists_\kappa & \triangleq & \lambda X_F^{0\kappa \to *}.\forall X^*.(\forall X_1^\kappa.X_F\, X_1 \to X) \to X & : +(0\kappa \to *) \to * \\
\exists_A & \triangleq & \lambda X_F^{A \to *}.\forall X^*.(\forall i^A.X_F\{i\} \to X) \to X & : +(A \to *) \to *
\end{array}
$$

Figure 5.5: Embeddings of some well-known non-recursive datatypes in Fix_i.

rule definition), the type constructor conversion rule ($Conv$) can silently roll (from $F(\mathsf{fix}F)$ to $\mathsf{fix}F$) and unroll (from $F(\mathsf{fix}F)$ to $\mathsf{fix}F$) the recursive types, just as it can silently β-convert type constructors.

In the following section, we review how iso-recursive type operator μ_κ and its In_κ constructor, which is well-behaved (i.e., strongly normalizes) for base structures of arbitrary polarity, can be embedded into Fix_i defined in terms of the equi-recursive type operator fix that is only well-behaved for covariant base structures.

5.2 EMBEDDING DATATYPES AND PRIMITIVE RECURSION

Embedding for primitive recursion over datatypes of arbitrary polarities into Fix_i was discovered by Abel and Matthes [3]. We review these embeddings in the context of Fix_i.

The embeddings of non-recursive datatypes in Figure 5.5 are exactly the same as in F_i (see Section 4.2), other than tracking polarities of the type constructor variables. That is, we use the usual impredicative encodings for non-recursive datatypes such as void, unit, pairs, sums, and existential types. The examples in Figure 5.5 are mostly from Abel and Matthes [3], except for the last example of $\exists_A$, an existential type over term-indices of type A.

Embedding recursive datatypes and their Mendler-style primitive recursion

notation: $\boldsymbol{\lambda}\mathbb{I}^{\kappa}.F = \lambda I_1^{K_1}.\cdots.\lambda I_n^{K_n}.F$ $\quad F\mathbb{I} = F\,I_1\cdots I_n$

$\forall\mathbb{I}^{\kappa}.B = \forall I_1^{K_1}.\cdots.\forall I_n^{K_n}.B$ $\quad F \overset{\kappa}{\rightarrow} G = \forall\mathbb{I}^{\kappa}.F\mathbb{I} \rightarrow G\mathbb{I}$

where

$\kappa = K_1 \rightarrow \cdots \rightarrow K_n \rightarrow *$ and I_i is an index variable (i_i) when K_i is a type,

$\mathbb{I} = I_1,\ \ldots\ \ldots\ ,\ I_n$ a type constructor variable (X_i) otherwise (i.e., $K_n = p_i\kappa_i$).

$$\mu_{\kappa} \ :\ 0(0\kappa \rightarrow \kappa) \rightarrow \kappa$$
$$\mu_{\kappa} \triangleq \lambda X_F^{0(0\kappa\rightarrow\kappa)}.\mathsf{fix}(\Phi_{\kappa}\,X_F)$$
$$\Phi_{\kappa} \ :\ 0(0\kappa \rightarrow \kappa) \rightarrow +\kappa \rightarrow \kappa$$
$$\Phi_{\kappa} \triangleq \lambda X_F^{0(0\kappa\rightarrow\kappa)}.\lambda X_c^{+\kappa}.\boldsymbol{\lambda}\mathbb{I}^{\kappa}.\forall X^{\kappa}.(\forall X_r^{\kappa}.(X_r \overset{\kappa}{\rightarrow} X_c) \rightarrow (X_r \overset{\kappa}{\rightarrow} X) \rightarrow (X_F\,X_r \overset{\kappa}{\rightarrow} X)) \rightarrow X\,\mathbb{I}$$

$$\mathbf{mpr}_{\kappa} \ :\ \forall X_F^{0(0\kappa\rightarrow\kappa)}.\forall X^{\kappa}.(\forall X_r^{\kappa}.(X_r \overset{\kappa}{\rightarrow} \mu_{\kappa}X_F) \rightarrow (X_r \overset{\kappa}{\rightarrow} X) \rightarrow (X_F\,X_r \overset{\kappa}{\rightarrow} X)) \rightarrow (\mu_{\kappa}X_F \overset{\kappa}{\rightarrow} X)$$
$$\mathbf{mpr}_{\kappa} \triangleq \lambda s.\lambda r.r\ s$$

$$\mathsf{In}_{\kappa} \ :\ \forall X_F^{0(0\kappa\rightarrow\kappa)}.X_F(\mu_{\kappa}X_F) \overset{\kappa}{\rightarrow} \mu_{\kappa}X_F$$
$$\mathsf{In}_{\kappa} \triangleq \lambda t.\lambda s.s\ \mathit{id}\,(\mathbf{mpr}_{\kappa}\ s)\ t$$
$$\mathit{id} \triangleq \lambda x.x$$

Figure 5.6: Embedding of the recursive type operators (μ_{κ}), their data constructors (In_{κ}), and the Mendler-style primitive recursors ($\mathbf{mpr}_{\kappa}$) in Fix_i.

The type of r can be expanded by the defintion of fix and the equi-recursive equality rule on fix as follows:

$$\mu_\kappa X_F\,\mathbb{I} = \mathsf{fix}(\Phi_\kappa X_F)\mathbb{I} = \Phi_\kappa X_F(\mathsf{fix}(\Phi_\kappa X_F))\mathbb{I} = \Phi_\kappa X_F(\mu_\kappa X_F)\mathbb{I}$$
$$= \forall X^\kappa.\underbrace{(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X))}_{\text{exactly matches with the type of } s} \to X\,\mathbb{I}$$

$$\frac{\dfrac{X_F^{0(0\kappa\to\kappa)}, X^{0\kappa}, \mathbb{I}^\kappa;\ s : (\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X)), r : \mu_\kappa X_F\,\mathbb{I} \vdash r\ s : X\,\mathbb{I}}{X_F^{0(0\kappa\to\kappa)}, X^{0\kappa};\ s : (\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X)) \vdash \lambda s.r\ s : \mu_\kappa X_F \xrightarrow{\kappa} X}}{\cdot;\cdot \vdash \lambda s.\lambda r.r\ s : \forall X_F^{0\kappa\to\kappa}.\forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X)) \to (\mu_\kappa X_F \xrightarrow{\kappa} X)}$$

Figure 5.7: Well-typedness of the **mpr** embedding in Fix_i.

Let $\Delta = X_F^{0(0\kappa\to\kappa)}, \mathbb{I}^\kappa, X^{0\kappa}$ and $\Gamma = t : X_F(\mu_\kappa X_F)\mathbb{I}, s : (\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X))$.

$$\begin{array}{llllll}
\Delta;\Gamma \vdash s & : (\mu_\kappa X_F \xrightarrow{\kappa} \mu_\kappa X_F) & \to & (\mu_\kappa X_F \xrightarrow{\kappa} X) & \to & (X_F(\mu_\kappa X_F) \xrightarrow{\kappa} X) \qquad \text{(by instantiating } X_r \text{ with } \mu_\kappa X_F) \\
\Delta;\Gamma \vdash id & : (\mu_\kappa X_F \xrightarrow{\kappa} \mu_\kappa X_F) & & & & \\
\Delta;\Gamma \vdash (\mathbf{mpr}_\kappa\ s) & : & & (\mu_\kappa X_F \xrightarrow{\kappa} X) & & \\
\Delta;\Gamma \vdash t & : & & & & X_F(\mu_\kappa X_F)\mathbb{I}
\end{array}$$

$$\frac{\dfrac{\dfrac{X_F^{0(0\kappa\to\kappa)}, \mathbb{I}^\kappa, X^{0\kappa};\ t : X_F(\mu_\kappa X_F)\mathbb{I}, s : (\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X)) \vdash s\ id\ (\mathbf{mpr}_\kappa\ s)\ t : X\,\mathbb{I}}{X_F^{0(0\kappa\to\kappa)}, \mathbb{I}^\kappa;\ t : X_F(\mu_\kappa X_F)\mathbb{I} \vdash \lambda s.s\ id\ (\mathbf{mpr}_\kappa\ s)\ t : \forall X^\kappa.(\forall X_r^\kappa.(X_r \xrightarrow{\kappa} \mu_\kappa X_F) \to (X_r \xrightarrow{\kappa} X) \to (X_F\,X_r \xrightarrow{\kappa} X)) \to X\,\mathbb{I}}}{X_F^{0(0\kappa\to\kappa)}, \mathbb{I}^\kappa;\ t : X_F(\mu_\kappa X_F)\mathbb{I} \vdash \lambda s.s\ id\ (\mathbf{mpr}_\kappa\ s)\ t : \mu_\kappa X_F\,\mathbb{I}} \qquad \text{(We can expand the type into above as in Figure 5.7)}}{\cdot;\cdot \vdash \lambda t.\lambda s.s\ id\ (\mathbf{mpr}_\kappa\ s)\ t : \forall X_F^{\kappa\to\kappa}.X_F(\mu_\kappa X_F) \xrightarrow{\kappa} \mu_\kappa X_F}$$

Figure 5.8: Well-typedness of the In embedding in Fix_i.

amounts to embedding μ_κ, In_κ, and $\mathbf{mpr}_\kappa$ described in Section 3.8. Figure 5.6 illustrates the embeddings discovered by Abel and Matthes [3], reformatted using our conventions (see Figure 4.8 in Section 4.2.2) and taking term-indices into consideration. To confirm the correctness of these embeddings, we need to check that (1) the embeddings are well-kinded and well-typed and (2) the primitive recursion behaves well (i.e., $\mathbf{mpr}_\kappa\ s\ (\mathsf{In}_\kappa) \longrightarrow^+ s\ id\ (\mathbf{mpr}_\kappa\ s)\ t$). From the term encodings of $\mathbf{mpr}_\kappa$ and In_κ, it is obvious that the reduction of primitive recursion behaves well. Thus, we only need to check that μ_κ is well-kinded and that $\mathbf{mpr}_\kappa$ and In_κ are well-typed.

Note that the polarities appearing in the embedding of μ_κ are all 0. The embedding from a non-polarize kind κ into a polarized kind $\ulcorner\kappa\urcorner$ can be defined as:

$$\ulcorner *\urcorner = * \qquad \ulcorner \kappa_1 \to \kappa_2\urcorner = 0\ulcorner\kappa_1\urcorner \to \ulcorner\kappa_2\urcorner \qquad \ulcorner A \to \kappa\urcorner = A \to \ulcorner\kappa\urcorner.$$

It is easy to see that the embedding of the non-polarized recursive type operator $\mu_\kappa : 0(0\kappa \to \kappa) \to \kappa$ is well-kinded, provided that $\Phi_\kappa : 0(0\kappa \to \kappa) \to +\kappa \to \kappa$ is well-kinded. Note that Φ_κ turns an avariant type constructor $(0\kappa \to \kappa)$ into a positive type constructor $(+\kappa \to \kappa)$. From the definition of Φ_κ, we only need to check that $(X_r \overset{\kappa}{\to} X_c)$, $(X_r \overset{\kappa}{\to} X)$, $X_F\, X_r \overset{\kappa}{\to} X$, and $X\ \mathbb{I}$ are of kind $*$ under the context $X_F^{0(0\kappa\to\kappa)}, X_c^{+\kappa}, \mathbb{I}^\kappa, X^{0\kappa}$, which is not difficult to see.

Well-typedness of $\mathbf{mpr}_\kappa$ and In_κ are justified in Figures 5.7 and 5.8

5.3 EMBEDDING COURSE-OF-VALUES RECURSION

To add a new Mendler-style recursion scheme and show its termination, we need to address several issues:

- First, we need to add an appropriate type-level fixpoint operator (e.g., μ_κ for primitive recursion) that is used to build recursive types. This type-level operation needs to capture not only the tying of the recursive knot, but also the compatible structure needed to encode the new Mendler-style recursor.

$$\mu^+_\kappa \;:\; 0(+\kappa \to \kappa) \to \kappa$$
$$\mu^+_\kappa \triangleq \lambda X_F^{0(+\kappa\to\kappa)}.\mathsf{fix}(\Phi^+_\kappa\; X_F)$$
$$\Phi^+_\kappa \;:\; 0(+\kappa \to \kappa) \to +\kappa \to \kappa$$
$$\Phi^+_\kappa \triangleq \lambda X_F^{0(+\kappa\to\kappa)}.\lambda X_c^{+\kappa}.\boldsymbol{\lambda}\mathbb{I}^\kappa.\forall X^\kappa.(\forall X_r^\kappa.(X_r \overset{\kappa}{\to} X_F\, X_r) \to (X_r \overset{\kappa}{\to} X_c) \to (X_r \overset{\kappa}{\to} X) \to (X_F\, X_r \overset{\kappa}{\to} X)) \to X\,\mathbb{I}$$

$$\mathbf{mcvpr}_\kappa \;:\; \forall X_F^{+\kappa\to\kappa}.\forall X^\kappa.(\forall X_r^\kappa.(X_r \overset{\kappa}{\to} X_F\, X_r) \to (X_r \overset{\kappa}{\to} \mu^+_\kappa X_F) \to (X_r \overset{\kappa}{\to} X) \to (X_F\, X_r \overset{\kappa}{\to} X)) \to (\mu^+_\kappa X_F \overset{\kappa}{\to} X)$$
$$\mathbf{mcvpr}_\kappa \triangleq \lambda s.\lambda r.r\; s$$

$$\mathsf{In}_F \;:\; F(\mu^+_\kappa F) \overset{\kappa}{\to} \mu^+_\kappa F$$
$$\mathsf{In}_F \triangleq \lambda t.\lambda s.s\; \mathsf{unIn}_F\; id\; (\mathbf{mcvpr}_\kappa\; s)\; t$$

Provided that there exists $\mathsf{unIn}_F : \mu^+_\kappa F \overset{\kappa}{\to} F(\mu^+_\kappa F)$ for the base structure $F : +\kappa \to \kappa$, such that $\mathsf{unIn}_F(\mathsf{In}_F\; t) \longrightarrow^+ t$ where the reduction is constant regardless of t (although steps may vary between each base structure F).

See Figure 5.10 for embeddings of unrollers (unIn_F) for some well-known positive base structures (F).

Figure 5.9: Embedding of the recursive type operators (μ^+_κ), the Mendler-style course-of-values recursors ($\mathbf{mcvpr}_\kappa$), and the roller (In_F) in Fix_i, provided that the embedding of unIn_F exists.

Regular datatypes

$N \triangleq \lambda X^{+*}.X + \mathsf{Unit}$ $\quad \mathsf{unIn}_N \triangleq \mathbf{mcvpr}_*(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (\mathtt{InL} \circ cast)\ \mathtt{InR})$

$L \triangleq \lambda X_a^{+*}.\lambda X^{+*}.(X_a \times X) + \mathsf{Unit}$ $\quad \mathsf{unIn}_{(LA)} \triangleq \mathbf{mcvpr}_*(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (\mathtt{InL} \circ (id \times cast))\ \mathtt{InR})$

$R \triangleq \lambda X_a^{+*}.\lambda X^{+*}.(X_a \times \mathtt{List}\, X) \to X$ $\quad \mathsf{unIn}_{(RA)} \triangleq \mathbf{mcvpr}_*(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (id \times fmap_{\mathtt{List}}\ cast))$

Type-indexed datatypes

$P \triangleq \lambda X^{+*\to *}.\lambda X_a^{+*}.X_a \times X(X_a \times X_a) + \mathsf{Unit}$ $\quad \mathsf{unIn}_P \triangleq \mathbf{mcvpr}_{+*\to *}(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (\mathtt{InL} \circ (id \times cast))\ \mathtt{InR})$

$B \triangleq \lambda X^{+*\to *}.\lambda X_a^{+*}.X_a \times X(X\, X_a) + \mathsf{Unit}$ $\quad \mathsf{unIn}_B \triangleq \mathbf{mcvpr}_{+*\to *}(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (\mathtt{InL} \circ (id \times (cast \circ fmap\ cast)))\ \mathtt{InR})$

Term-indexed datatypes

$$V \triangleq \lambda X_a^*.\lambda X^{\mathtt{Nat}\to *}.\lambda i^{\mathtt{Nat}}.(\exists j^{\mathtt{Nat}}.((i = \mathtt{succ}\, j) \times X_a \times X\{j\})) + (i = \mathtt{zero})$$

$$\mathtt{VCons} \triangleq \lambda x_a.\lambda x.\mathtt{InL}(\mathtt{Ex}_{\mathtt{Nat}}(\mathtt{Eq}_{\mathtt{Nat}}, x_a, x)) \ : \ \forall X_a^*.\forall X^{\mathtt{Nat}\to *}.\forall i^{\mathtt{Nat}}.X_a \to X\,\{i\} \to V\,X_a\,X\,\{\mathtt{succ}\, i\}$$

$$\mathtt{VNil} \triangleq \mathtt{InR}\ \mathtt{Eq}_{\mathtt{Nat}} \ : \ \forall X_a^*.\forall X^{\mathtt{Nat}\to *}.V\,X_a\,X\,\{\mathtt{zero}\}$$

$$\mathsf{unIn}_{(V\,A)} \triangleq \mathbf{mcvpr}_{\mathtt{Nat}\to *}(\lambda\_.\lambda cast.\lambda\_.\lambda x.x\ (\mathtt{InL} \circ (id \times id \times cast))\ \mathtt{InR})$$

The notation $\exists j^A.B$ is shorthand for $\exists_A(\lambda j^A.B)$ where $\exists_A$ is defined in Figure 5.5.

$\mathtt{Ex}_{\mathtt{A}} : \forall F^{A\to *}.\exists_A F$ and $\mathtt{Eq}_{\mathtt{A}} : \forall i^A.\forall j^A.(i = j)$ are the data constructors of the existential and equality types, respectively.

See Figure 5.9 for the embedding of the Mendler-style course-of-values recursor ($\mathbf{mcvpr}_\kappa$).

Figure 5.10: Embeddings of unroller (unIn_F) for some well-known positive base structures (F).

- Second, we need to specify the behavior of the new Mendler-style recursor by discovering the characteristic equations it should obey (e.g., Haskell definition of Mendler-style recursor in Chapter 3).
- Third, we need to find an embedding that preserves the characteristic equations in the host calculus (e.g., embedding of In_κ and $\mathbf{mpr}_\kappa$).
- In practice, the second and third issues are intimately entwined as the equations and embedding are carefully designed (using a Church-style encoding) to achieve the desired result (e.g., In_κ is defined in terms of the Mendler-style recursor $\mathbf{mpr}_\kappa$).

To embed Mendler-style course-of-values recursion, we can follow the steps above just as we did for Mendler-style primitive recursion. In addition, we need to embed an unroller unIn_F, the key operation for Mendler-style course-of-values recursion. Recall the use of *out* in the Fibonacci number example (Figure 3.6) and the Lucas number example (Figure 3.16) in Section 3.5.

5.3.1 General form for the embedding of course-of-values recursion

Figure 5.9 illustrates the embedding of a new iso-recursive type operator (μ^+_κ), the Mendler-style course-of-values recursor ($\mathbf{mcvpr}_\kappa$), and the roller In_F in Fix_i. Since the embedding of In_F uses unIn_F, we also need to embed the unroller unIn_F in order to complete the embedding of the roller In_F. Embedding unIn_F is possible for a fairly large class of positive base structure F. Figure 5.10 illustrates some of these unrollers. But, it may not be possible to give an embedding of the unroller for some base structures. Recall that not all base structures can have well-defined course-of-values recursion that guarantees termination (see Figure 3.7 in Section 3.5).

The embeddings of μ^+_κ, $\mathbf{mcvpr}_\kappa$, and In_F for course-of-values recursion (see Figure 5.9) are very similar to the embeddings of μ_κ, $\mathbf{mpr}_\kappa$, and In_κ for primitive recursion (see Figure 5.6) discussed earlier in the previous section. The definition

of $\mathbf{mcvpr}_\kappa \triangleq \lambda s.\lambda r.r\ s$ is exactly the same as the definition of $\mathbf{mpr}_\kappa$, only differing in its type signature. The definition of In_F is similar to In_κ but uses the additional unIn_F that implements the abstract unroller. So, the last piece of the puzzle for embedding Mendler-style course-of-values recursion is the embedding of unIn_F.

In the following subsections, we elaborate on how to embed unrollers (unIn_F) through examples (Section 5.3.2), derive uniform embeddings of the unrollers generalizing from those examples, and discuss whether the embeddings of unrollers satisfy their desired properties.

5.3.2 Embedding unrollers

Embeddings of unrollers for some well-known positive datatypes are illustrated in Figure 5.10. The general idea is to use $\mathbf{mcvpr}_\kappa$ to define unIn_F for the base structure $F : +\kappa \to \kappa$ without using the abstract recursive call operation, only using the abstract cast operation to define constant time unrollers. To define an unroller, we map non-recursive components (X_a) as they use *id* and map abstract recursive components (X_r) to concrete recursive components ($\mu^+_\kappa F$) using the abstract *cast* operation provided by $\mathbf{mcvpr}_\kappa$. We can embed unrollers for regular datatypes such as natural numbers (base N) and lists (base L), type-indexed datatypes such as powerlists (base P), and term-indexed datatypes such as vectors (base V) in this way. The notation for combining functions for tuples are defined as $(f \times g) \triangleq \lambda x.(f\,x, g\,x)$, and $(f \times g \times h)$ are defined similarly for triples.

Embedding unrollers for regular datatypes: The embeddings of unIn_N and $\mathsf{unIn}_{(LA)}$ are self explanatory. The embedding of $\mathsf{unIn}_{(RA)}$ relies on the map function for lists, since the rose tree is indirect datatype where recursive subcomponents appear inside the list ($\texttt{List}\ X_r$). The $fmap_{\texttt{List}}$ function applies *cast* to each of the abstract recursive subcomponents of type X_r inside $F\ X_r$ values into a concrete recursive type $\mu^+_* F$ in order to obtain $F(\mu^+_* F)$ values.

Embedding unrollers for nested datatypes are no more complicated than embedding unrollers for regular datatypes. For instance, the embedding of unIn_P for powerlists is almost identical to the embedding of unIn_L for regular lists except for the use of $\mathbf{mcvpr}_{*\to*}$ instead of $\mathbf{mcvpr}_*$. This is because the cast operation provided by $\mathbf{mcvpr}_{*\to*}$ is polymorphic over the type index: $cast : \forall X_i.X_r X_i \to \mu^+_{*\to*} X_F X_i$. Since unrollers preserve indices, there is no extra work to be done other than toa apply the *cast*. In the embedding of unIn_P, the `cast` function performs an index-preserving cast from an abstract recursive type $X_r(X_a \times X_a)$ to the concrete powerlist type $\mu^+ P(X_a \times X_a)$.

Embedding unrollers for *truly nested datatypes* [5] such as bushes are similar to embedding unrollers for indirect regular recursive types. Truly nested datatypes are recursive datatypes whose indices may involve themselves. Truly nested datatypes are similar to indirect recursive types in the sense that a bunch of recursive components are contained in certain data structures – in case of truly nested datatypes those data structures are exactly the nested datatypes themselves. Assuming that the nested datatype has a notion of monotone map, we can use *fmap* to push down the *cast* to the inner structure and then *cast* the outer structure. Note the use of (*cast*$\circ$*fmap cast*) in the embedding of the unroller unIn_R for bushes.

Embedding unrollers for indexed datatypes are no more complicated than embedding unrollers for regular datatypes. To embed unrollers for term-indexed datatypes, we would often need existential types (Figure 5.5) and equality types. We can encode equality types in Fix_i as a Leibniz equality over indices, i.e., $(i = j) \triangleq \forall F^{A\to*}.F\{i\} \to F\{j\}$, as discussed in Section 4.2.3. These extra encodings for maintaining term-indices do not terribly complicate the embeddings of unrollers, as unrollers are index-preserving. The embedding of $\mathsf{unIn}_{(VA)}$ for length indexed lists is almost identical to the embedding of $\mathsf{unIn}_{(LA)}$ for regular lists, except that

it has one more *id*. The first *id* that appears in $(id \times id \times cast)$ is so that the index equality remain unchanged.

Giving different definitions of unIn_F for each different F, as illustrated in Figure 5.10 appears too ad-hoc. Hence, we discuss how to generalize the embeddings of the unIn operations, assuming that F has a notion of a monotone map (e.g., *fmap* for $F : +* \rightarrow^*$) in the following subsection. Later, in Section 5.4.2, we reason about what conditions for F to have a notion of a monotone map.

5.3.3 Deriving uniform embeddings of the unrollers

To derive uniform embeddings of the unrollers, we transcribed the embeddings of the unrollers appearing in Figure 5.10 into Haskell and observed common patterns among them. These results are summarized in Figures 5.11, 5.12, 5.13, and 5.14. This Haskell transcription exercise not only helps us derive uniform embeddings of the unrollers, but also helps us recognize the conditions that the base structures should satisfy in order to have an embedding of its unroller in Fix_i.

Haskell transcriptions of the unroller embeddings for regular datatypes are given in Figure 5.12 (Haskell definitions of `Mu0` and `mcvpr0` are given in Figure 5.11). Note that the definitions of `unInN`, `unInL`, and `unInR` are uniform: `mvcvp0 (λ_ cast _ →fmap cast)`. This uniform definition relies on the existence of *fmap* over the base structures – note the **`deriving Functor`** in the data declarations. In Section 5.4.2, we show that *fmap* exists for any $F : +* \rightarrow *$ in Fix_i. Hence, we can derive a uniform embedding for the unroller unIn_* for any base $F : +* \rightarrow *$ as follows:

$$\mathsf{unIn}_* : \forall X_F^{+*\rightarrow *}.\mu_*^+ X_F \rightarrow X_F(\mu_*^+ X_F)$$
$$\mathsf{unIn}_* \triangleq \mathbf{mcvpr}_*(\lambda\_.\lambda cast.\lambda\_.fmap_{X_F}\ cast)$$

Haskell transcriptions of the unroller embeddings for nested datatypes are given

```
newtype Mu0 f = In0 { unIn0 :: f(Mu0 f) }

mcvpr0 :: Functor f ⇒ (∀ r. (r → f r) →
                            (r → Mu0 f) →
                            (r → a) →
                            (f r → a) )
       → Mu0 f → a
mcvpr0 phi = phi unIn0 id (mcvpr0 phi) . unIn0

newtype Mu1 f i = In1 { unIn1 :: f(Mu1 f)i }

mcvpr1 :: Functor1 f ⇒
          (∀ r i'. Functor r ⇒ (∀ i. r i → f r i) →
                               (∀ i. r i → Mu1 f i) →
                               (∀ i. r i → a i) →
                               (f r i' → a i') )
       → Mu1 f i → a i
mcvpr1 phi = phi unIn1 id (mcvpr1 phi) . unIn1

class Functor1 h where
  fmap1'  :: Functor f ⇒ (∀ i j. (i → j) → f i → g j)
                       → (a → b) → h f a → h g b
  -- fmap1' h = fmap1 (h id)

  fmap1 :: Functor f ⇒ (∀ i. f i → g i)
                       → (a → b) → h f a → h g b
  fmap1 h = fmap1' (λf → h . fmap f)

instance (Functor1 h, Functor f) ⇒ Functor (h f) where
  fmap f = fmap1 id
         -- fmap1' (λf → id . fmap f)

instance Functor (f (Mu1 f)) ⇒ Functor (Mu1 f) where
  fmap f = In1 . fmap f . unIn1
```

Figure 5.11: μ_*, $\mathbf{mcvpr}_*$, and $\mu_{*\to*}$, $\mathbf{mcvpr}_{*\to*}$ transcribed into Haskell.

```
data N r   = S r   | Z  deriving Functor
type Nat = Mu0 N

unInN :: Mu0 N → N(Mu0 N)
unInN = mcvpr0 (λ _ cast _ →  fmap cast)

data L a r = C a r | N  deriving Functor
type List a = Mu0 (L a)

unInL :: Mu0(L a) → (L a) (Mu0(L a))
unInL = mcvpr0 (λ _ cast _ →  fmap cast)

data R a r = F a [r]    deriving Functor -- relies on (Functor [])
type Rose a = Mu0 (R a)

unInR :: Mu0(R a) → (R a) (Mu0(R a))
unInR = mcvpr0 (λ _ cast _ →  fmap cast)
```

Figure 5.12: Embeddings of unIn_N, $\mathsf{unIn}_{(LA)}$, $\mathsf{unIn}_{(RA)}$ transcribed into Haskell.

```
data P r i = PC i (r (i,i)) | PN
type Powl i = Mu1 P i

instance Functor1 P where
  fmap1' h f (PC x a) = PC (f x) (h (λ(i,j) → (f i,f j)) a)
  fmap1' _ _ PN = PN

unInP :: Mu1 P i → P(Mu1 P) i
unInP = mcvpr1 (λ _ cast _ → fmap1 cast id)
  -- mcvpr1 phi where
  --   phi _ cast _ (PC x xs) = PC x (cast xs)
  --   phi _ cast _ PN = PN

data B r i = BC i (r (r i)) | BN
type Bush i = Mu1 B i

instance Functor1 B where
  fmap1' h f (BC x a) = BC (f x) (h (h f) a)
  fmap1' _ _ BN = BN

unInB :: Mu1 B i → B (Mu1 B) i
unInB = mcvpr1 (λ _ cast _ → fmap1 cast id)
  -- mcvpr1 phi where
  --   phi _ cast _ (BC x xs) = BC x (cast (fmap cast xs))
  --   phi _ cast _ BN = BN
```

Figure 5.13: Embedding of unIn_P and unIn_B transcribed into Haskell.

```
class FunctorI1 (h :: (* → *) → * → *) where
  fmapI1 :: (∀ i . f i → g i) → h f j → h g j

mcvprI1 :: FunctorI1 f ⇒
           (∀ r j. (∀ i. r i → f r i) →
                   (∀ i. r i → Mu1 f i) →
                   (∀ i. r i → a i) →
                   (f r j → a j) )
        → Mu1 f i' → a i'
mcvprI1 phi = phi unIn1 id (mcvprI1 phi) . unIn1


data Succ n
data Zero

data V a r i where
  VC :: a → r n → V a r (Succ n)
  VN :: V a r Zero

instance FunctorI1 (V a) where
  fmapI1 h (VC x a) = VC x (h a)
  fmapI1 _ VN = VN

unInV :: Mu1 (V a) i → (V a) (Mu1 (V a)) i
unInV = mcvprI1 (λ_ cast _ → fmapI1 cast)


instance FunctorI1 P where
  fmapI1 h (PC x a) = PC x (h a)
  fmapI1 _ PN = PN

unInP' :: Mu1 P a → P (Mu1 P) a
unInP' = mcvprI1 (λ_ cast _ → fmapI1 cast)
```

Figure 5.14: Embedding of $\mathsf{unIn}_{(VA)}$ and another embedding of unIn_P transcribed into Haskell.

in Figure 5.13. (Haskell definitions of `Mu1`, `mcvpr1`, and `fmap1'` are given in Figure 5.11). Note that the definitions of the unrollers `unInP` for powerlists and `unInB` for bushes are uniform: `mvcvp1 (λ_ cast _ →fmap1 cast id)`. The function `fmap1` is the rank 2 monotone map for type constructors of kind $+(+* \to *) \to (+* \to *)$, which is analogous to rank 1 monotone map `fmap` for type constructors of kind $+* \to *$. Alternatively, one can think of the `Functor1` class as a bifunctor over type constructors of kind $+(+* \to *) \to +* \to *$, whose first argument $(+\underline{(+* \to *)} \to +* \to *)$ is a type constructor and second argument $(+(+* \to *) \to +\underline{*} \to *)$ is a type.

A Haskell transcription of the unroller embedding for the length-indexed list datatype, which is a term-indexed datatype, is given in Figure 5.14. To give an embedding of $\mathsf{unIn}_{(VA)}$, we define another version of Mendler-style course-of-values recursion combinator `mcvprI1` similar to `mcvpr1` in Figure 5.11 but requires the base structure to be an instance of the `FunctorI1` class rather than an instance of the `Functor1` class. The `FunctorI1` class is simpler than the `Functor1` class because `FunctorI1` requires only the first argument to be monotone while `Functor1` requires both the first and second arguments to be monotone – this is evident when comparing the types of their member functions `fmap1` and `fmapI1` side-by-side:

```
fmap1  :: (Functor f, Functor1 h) ⇒
          (∀ i. f i → g i) → (a → b) → h f a → h g b
fmapI1 :: FunctorI1 h ⇒
          (∀ i. f i → g i)           → h f a → h g a
```

Here, `fmapI1` does not have the extra `Functor` requirement since it does not require the second argument type to be monotone. The function `fmapI1` only transforms the first type constructor argument, preserving the second argument (which may be a type index or a term index), while `fmap1` is able to transform both the first and second arguments. For term-indexed datatypes, `fmapI1` is enough to construct embeddings of the unrollers, since there is no need to transform indices –

recall that term-indices do not appear at value level. Hence, there are no values to transform in the first place. Furthermore, even if we had such values, we need not transform them anyway because unrollers are index-preserving by definition. Thus, in the type signature of `mcvprI1`, we need not require the abstract recursive structure `r` to be a `Functor`, unlike in the type signature of `mcvpr1` where we did require `Functor r`. The embedding of $\mathsf{unIn}_{(VA)}$ has the expected shape: `mcvprI1 (λ_ cast _ →fmapI1 cast)`.

Interestingly, we can give yet another Haskell transcription of unIn_P in terms of `mcvprI1` (see `unInP'` in Figure 5.14) rather than in terms of `mcvpr1`. This is because we do not really need the ability to transform type indices to embed unrollers for powerlists. However, for truly nested datatypes such as bushes, this alternative is not possible. Recall that we do need to transform indices from abstract recursive type to concrete recursive type to embed unIn_B because a bush is indexed by its own structure. In summary, unroller embeddings for indexed datatypes, regardless of term-indexed or type-indexed, do not require indices to be monotone unless the datatype is truly nested. For truly nested datatypes, we must require indices as well as the recursive type constructor itself to be monotone in order to embed their unrollers. We can have a good approximation of this idea using with polarized kinds in Fix_i. We conjecture that all base structures of kind $+(+* \to *) \to +* \to *$ are instances of `Functor1`, while all base structures of kind $+(0* \to *) \to 0* \to *$ are instances of `Functor1`. We discuss this more formally in Section 5.4.2.

5.3.4 Properties of unrollers

We expect two properties to hold for the unroller embeddings. First, unIn_F must be a left identity of In_κ. That is, $\mathsf{unIn}_F(\mathsf{In}_\kappa t) \longrightarrow^+ t$ for any term t. Second, unIn_F should be a constant time operation, regardless of its supplied argument. That is, $\mathsf{unIn}_F(\mathsf{In}_\kappa t) \longrightarrow^+ t$ takes constant steps independent of t (but may vary between

differrent Fs). One difficulty is that some embeddings of the unrollers illustrated in Figure 5.10 are not constant time. However, we can safely optimize them into constant time functions because of the metatheoretic property of $\mathbf{mcvpr}_\kappa$ and *fmap*:

- The *cast* operation of $\mathbf{mcvpr}_\kappa$ is implemened as the identity function *id*.
- $(fmap_F\ id)\ t \longrightarrow^+ t$ for any $t : FA$. This property generalizes to monotone maps of higher kinds. For instance, $(fmap1_H\ id\ id)\ t \longrightarrow^+ t$ for any $t : H\,F\,A$ (see Figure 5.11 for the definition of *fmap1*).

For instance, $\mathsf{unIn}_{(RA)}$ for the rose tree datatype, which is an *indirect recursive datatype*, are not constant time. The map function for lists $fmap_{List}$ appearing in the definition of $\mathsf{unIn}_{(RA)}$ is obviously not a constant time function. That is, we traverse the list inside a rose tree to *cast* each element of the list. Thus, $\mathsf{unIn}_{(RA)}$ is linear to the length of the list appearing in the rose tree. We can safely optimize $\mathsf{unIn}_{(RA)}$ into a constant time operation by optimizing $(fmap_{\texttt{List}}\ cast)$ into the identity function *id*. This optimization is safe because the property of *cast* is implemented by *id* and the property of (*fmap id*) is equivalent to *id*. However, this does not mean we have a constant time embedding of $\mathsf{unIn}_{(RA)}$ within Fix_i, since the optimized term is not type-correct. The identity function $id \triangleq \lambda x.x$ cannot be given the same type as $(fmap_{\texttt{List}}\ id) : \texttt{List}(X_r\,X_a) \to \texttt{List}(\mu^+_\kappa R\,X_a)$.

For similar reasons, unIn_B for the bush datatype, which is a *truly nested datatype*, is not constant time either due to the use of $fmap\ cast : X_r(X_rX_a) \to X_r(\mu^+_{*\to *}BX_a)$, which traverses the outer X_r structure (an abstract bush) to cast each element from $(X_r\,X_a)$ to $(\mu^+_{*\to *}B\,X_a)$. However, in this case, there is yet another subtlety that must be addressed before we can address the embedding of unIn_B not being constant time. Note that we boldly assumed that the abstract recursive type X_r has an *fmap* operation (specified by `Functor r` in the Haskell transcription). Previously, in the embedding of unIn_R for the rose tree, we relied on a property of a

specific instance of *fmap* for `List`, which is a type well known to have *fmap* and is indeed equipped with the desired property. In the case of unIn_B, we cannot assume anything but the kind of $X_r : +* \to *$ because it is abstract. Hence, we should rely on a more general property that *fmap* is well-defined for any type constructors of kind $+* \to *$ in Fix_i. We discuss this general property of *fmap* in Section 5.4.2.

Lastly, we can even further optimize the unrollers based on the observation made in the previous subsection that all unroller embeddings have uniform shape:

$$\mathbf{mcvpr}_\kappa(\lambda\_.\lambda cast.\lambda\_.\ \underline{fmap??\ cast\ id \cdots id})$$

where the underlined part can be optimized to the identity function *id*.

It is important to note that embeddings of the Mendler-style course-of-values primitive recursion rely on the existence of unrollers. However, we have not formally proved the existence of unrollers in general. We only conjecture and strongly believe that unrollers exist for recursive types whose kinds are decorated with sufficient positive polarity. Clarifying these issues will be an interesting direction for future work. Some preliminary ideas are elaborated in Section 5.4.2.

5.4 METATHEORY OF Fix_i

Recall that we extended Fix_ω to Fix_i to support primitive recursion and course-of-values recursion. In this section, we show strong normalization and logical consistency of Fix_i. We also give a partial proof of the syntactic conditions necessary for well-behaved course-of-values recursion.

5.4.1 Strong normalization and logical consistency

We can prove strong normalization of Fix_i by the following strategy.

- Define a notion of index erasure that projects Fix_i types to Fix_ω types.
- Show that every well-typed Fix_i-term is a well-typed Fix_ω-term by index erasure.

- Fix_i inherits strongly normalization from Fix_ω because Fix_ω is strongly normalizing [3].

The definition of the index erasure operation for System Fix_i and the proofs of the related theorems are virtually the same as their counterparts in System F_i (see Section 4.3). So, we simply illustrate the definition and the theorems, but omit their proofs.

We define a meta-operation of index erasure ($\circ$) that projects Fix_i types to Fix_ω types and another meta-operation ($\bullet$) that selects only the index variable bindings (i^A) from the type-level context.

Definition 5.4.1 (index erasure)**.**

$$\boxed{\kappa^\circ} \quad *^\circ = * \qquad (p\kappa_1 \to \kappa_2)^\circ = p{\kappa_1}^\circ \to {\kappa_2}^\circ \qquad (A \to \kappa)^\circ = \kappa^\circ$$

$$\boxed{F^\circ} \quad X^\circ = X \qquad (A \to B)^\circ = A^\circ \to B^\circ \qquad (\mathsf{fix}F)^\circ = \mathsf{fix}F^\circ$$

$$(\lambda X^{p\kappa}.F)^\circ = \lambda X^{p{\kappa}^\circ}.F^\circ \qquad (\lambda i^A.F)^\circ = F^\circ$$

$$(F\ G)^\circ = F^\circ\ G^\circ \qquad (F\ \{s\})^\circ = F^\circ$$

$$(\forall X^\kappa.B)^\circ = \forall X^{\kappa^\circ}.B^\circ \qquad (\forall i^A.B)^\circ = B^\circ$$

$$\boxed{\Delta^\circ} \quad \cdot^\circ = \cdot \qquad (\Delta, X^{p\kappa})^\circ = \Delta^\circ, X^{p{\kappa}^\circ} \qquad (\Delta, i^A)^\circ = \Delta^\circ$$

$$\boxed{\Gamma^\circ} \quad \cdot^\circ = \cdot \qquad (\Gamma, x : A)^\circ = \Gamma^\circ, x : A^\circ$$

Definition 5.4.2 (index variable selection)**.**

$$\cdot^\bullet = \cdot \qquad (\Delta, X^{p\kappa})^\bullet = \Delta^\bullet \qquad (\Delta, i^A)^\bullet = \Delta^\bullet, i : A$$

These two definitions are exactly the same as the definitions of $^\circ$ and $^\bullet$ in F_i (defined in Section 4.3.3), except for the new constructs of Fix_i: (1) polarities in kinds and (2) the equi-recursive type operator fix.

Once $^\circ$ and $^\bullet$ are defined, the proof of strong normalization of Fix_i, by index erasure, is virtually the same as the proof of strong normalization of F_i (Section 4.3.3).

Here, we list only the theorems since their proofs can be trivially reconstructed by consulting the proofs of the corresponding theorems in Section 4.3.3.

Theorem 5.4.1 (index erasure on well-sorted kinds). $\dfrac{\vdash \kappa : \Box}{\vdash \kappa^\circ : \Box}$

Theorem 5.4.2 (index erasure on well-formed type-level contexts).

$$\frac{\vdash \Delta}{\vdash \Delta^\circ}$$

Theorem 5.4.3 (index erasure on kind equality). $\dfrac{\vdash \kappa = \kappa' : \Box}{\vdash \kappa^\circ = \kappa'^\circ : \Box}$

Theorem 5.4.4 (index erasure on well-kinded type constructors).

$$\frac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\Delta^\circ \vdash F^\circ : \kappa^\circ}$$

Theorem 5.4.5 (index erasure on type constructor equality).

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ}$$

Theorem 5.4.6 (index erasure on well-formed term-level contexts).

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash \Gamma^\circ}$$

Theorem 5.4.7 (index erasure on index-free well-typed terms).

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; \Gamma^\circ \vdash t : A^\circ} \ (\mathrm{dom}(\Delta) \cap \mathrm{FV}(t) = \emptyset)$$

Theorem 5.4.8 (index erasure on well-formed term-level contexts prepended by index variable selection).

$$\frac{\Delta \vdash \Gamma}{\Delta^\circ \vdash (\Delta^\bullet, \Gamma)^\circ}$$

Theorem 5.4.9 (index erasure on well-typed terms).

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta^\circ; (\Delta^\bullet, \Gamma)^\circ \vdash t : A^\circ}$$

As stated in the introduction to this section, strong normalization is a direct consequence of the erasure theorems above.

Logical consistency: We show the logical consistency of Fix_i by showing that the void type $\forall X^*.X$ is uninhabited. By index erasure, we know that that the set of terms that inhabit a Fix_i-type is a subset of the set of terms that inhabit the corresponding (erased) Fix_ω-type. Thus, if $\forall X^*.X$ is uninhabited in Fix_ω then $\forall X^*.X$ is uninhabited in Fix_i. Abel and Matthes [3] gives a saturated set interpretation for the type constructors in Fix_ω that is similar to the interpretation given in Section 2.3.2 for F_ω type constructors. The void type $\forall X^*.X$ is obviously uninhabited (by any closed term) according to this interpretation. That is, no strongly-normalizing, closed term inhabits $\forall X^*.X$.

$$[\![\forall X^*.X]\!]_\xi \in \mathsf{SAT}_* \to \mathsf{SAT}_*$$

$$[\![\forall X^*.X]\!]_\xi = \bigcap_{\mathcal{A} \in [\![*]\!]} [\![X]\!]_{\xi[X \to \mathcal{A}]} = \bigcap_{\mathcal{A} \in [\![*]\!]} \mathcal{A} = \bot$$

The minimal saturated set $\bot$ of SAT, which is saturated from the empty set, does not have any closed terms. See Section 2.1.1 for the definition of saturated sets.

5.4.2 Syntactic conditions for well-behaved course-of-values recursion

In Section 5.3, we embedded Mendler-style course-of-values recursion (**mcvpr**) in Fix_i for the base structures (F) that have maps (fmap_F), also known as *monotonicity witnesses* [58, 60]. The theoretical development of the termination of **mcvpr** by assuming the existence of maps is elegant, since it does not require ad-hoc syntactic restrictions on the formation of types. However, in a language implementation, it is not desirable to require users to manually witness fmap_F every time they need to convince the type system that **mcvpr** is well-defined over F.

It would be very convenient if we could categorize type constructors of Fix_i that have maps by analyzing their polarized kinds. As a consequence, for any type constructor F whose kind meets certain criteria, users can immediately assume the existence of fmap_F and that **mcvpr** always terminates for F.

For instance, we conjecture that any $F : +* \to *$ should have a map, as in

Conjecture 5.4.1 below. In this thesis, we only show that this conjecture holds for simple cases (Proposition 5.4.1) while still encompassing a broad range of types. The complete proof is left for future work. Matthes [60] showed that positive inductive types, in the context of System F, are monotone (i.e., map exists), but it has not yet been studied whether we can rely on polarized kinds to derive monotonicity in the context of System F_ω.

Conjecture 5.4.1. *For any $F : +* \to *$, there exists*

$$\mathit{fmap}_F : \forall X^*.\forall Y^*.(X \to Y) \to F\ X \to F\ Y \quad \textit{such that}$$

$$\begin{aligned} \mathit{fmap}_F\ \mathit{id} &= \mathit{id} \\ \mathit{fmap}_F\ f \circ \mathit{fmap}_F\ g &= \mathit{fmap}_F\ (f \circ g) \end{aligned}$$

Assuming that the conjecture above is true, we can show that **mcvpr** is well-defined for any $F : +* \to *$ as follows.

Conjecture 5.4.2. *For any $F : +* \to *$, there exists $\mathsf{unIn}_F : \mu^+ * F \to F(\mu^+ * F)$ such that $\mathsf{unIn}_F(\mathsf{In}_F\ t) \longrightarrow^+ t$.*

Proof. Because we know that fmap_F exists by Conjecture 5.4.1, we can define

$$\mathsf{unIn}_F = \mathbf{mcvpr}_* \ (\lambda\_.\lambda \mathit{cast}.\lambda\_.\lambda x.\mathit{fmap}_F\ \mathit{cast}\ x)$$

Because we know that $\mathit{fmap}_F\ \mathit{id}\ x \longrightarrow^+ x$, we can show that the unroller unIn_F has the desired property $\mathsf{unIn}_F(\mathsf{In}_F\ t) \longrightarrow^+ \mathit{fmap}_F\ \mathit{id}\ t \longrightarrow^+ t$. □

We believe that the above conjectures are true, but we prove only a small fragment (outlined below), which is a special case of Conjecture 5.4.1.

Proposition 5.4.1. *There exists $\mathit{fmap}_F : \forall X^*.\forall Y^*.(X \to Y) \to F\ X \to F\ Y$ for any $F : +* \to *$ such that*

- *F is non-recursive, that is, does not have* fix*,*

- *F is a value, that is, F is in normal form (i.e., has no redex) and closed (i.e., F has no free variables),*
- *All the bound variables within F are introduced by universal quantification and those variables are of kind $*$.*

Proof. We can derive $fmap_F$ from the structure of F. Since F has kind $+* \to *$, it must be a lambda abstraction ($\lambda Z^{+*}.B$), some kind of application (a normal application, $F'\,G$, or an index application $F'\{s\}$), or a variable of kind $+* \to *$. No other way of forming F can have kind $+* \to *$.

We also assume that F is in normal form, so if F were an application then F' must be a variable with an arrow kind. However, we have assumed that all variables have kind $*$. Hence, F cannot be a variable since we have assumed that F is a value with no free variables, thus F must have the form $\lambda Z^{+*}.B$. We proceed by analyzing the structure of B.

case ($Z \notin \mathrm{FV}(B)$, i.e., F is a constant function.) $fmap_{(\lambda Z^{+*}.B)} = \lambda\_.\lambda x.x$

Since $F\ X = F\ Y = B$, we simply return the identity function on B.

case ($F \triangleq \lambda Z^{+*}.Z$. i.e., F is the identity.) $fmap_{(\lambda Z^{+*}.Z)} = \lambda z.z$

Since $F\ X = X$ and $F\ X = Y$, we return the function $z : X \to Y$ itself.

case ($F \triangleq \lambda Z^{+*}.\forall X_1^*.B_1$, B is a universal quantification.)

$$fmap_{(\lambda Z^{+*}.\forall X_1^*.B_1)} = fmap_{(\lambda Z^{+*}.B_1)}$$

Here, we need to find an *fmap* that works for any valuation of X_1. That is, we must find an *fmap* that works for $\lambda Z^{+*}.B_1[V/X_1]$ for an arbitrary value V. Since values are closed, v cannot contain free variables including Z. Since v is completely independent of Z, the value V cannot make any difference to the derived *fmap*. Hence, we simply ignore X_1.

case ($F \triangleq \lambda Z^{+*}.A \to B_1$, i.e., B is a function.)

When $Z \notin \mathrm{FV}(A)$, $fmap_{(\lambda Z^{+*}.A \to B_1)} = \lambda z.\lambda y.\lambda x.fmap_{(\lambda Z^{+*}.B_1)}\ z\ (y\ x)$

When $A \triangleq \forall X_1^*.A_1$, $fmap_{(\lambda Z^{+*}.(\forall X_1^*.A_1)\to B_1)} = fmap_{(\lambda Z^{+*}.A_1\to B_1)}$

When $B_1 \triangleq \forall X_1^*.B_2$, $fmap_{(\lambda Z^{+*}.A\to\forall X_1^*.B_2)} = fmap_{(\lambda Z^{+*}.A\to B_2)}$

When $A \triangleq A_1 \to \cdots \to A_n \to \forall X_2^*.B_2'$,

$$fmap_{(\lambda Z^{+*}.(A_1\to\cdots\to A_n\to\forall X_2^*.B_2')\to B_1)} = fmap_{(\lambda Z^{+*}.(A_1\to\cdots\to A_n\to B_2')\to B_1)}$$

When $A \triangleq A_1 \to \cdots \to A_n \to B_2$, where B_2 is not an arrow type

$$\begin{aligned} & fmap_{(\lambda Z^{+*}.(A_1\to\cdots\to A_n\to B_2)\to B_1)} \\ = \;& \lambda z.\lambda y.\lambda x.\; fmap_{(\lambda Z^{+*}.B_1)}\; z\; (y\; (\lambda x_1.\ldots.\lambda x_n.\; x\; (fmap_{(\lambda Z^{+*}.A_1)}\; z\; x_1) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (fmap_{(\lambda Z^{+*}.A_n)}\; z\; x_n))) \end{aligned}$$

□

To illustrate that the *fmap*s derived in the proof above are indeed type-correct, we provide some examples in Haskell accepted by GHC in Figure 5.15. In fact, all the `Functor` instances in Figure 5.15 are automatically derivable using the `DeriveFunctor` extension in GHC version 7.4. However, GHC does not derive functor instances when there are type constructor variables other than $*$, since the kind system in GHC does not keep track of polarity as in Fix_ω or Fix_i.

Recall that we are proving a simplified version of the desired conjecture with several simplifying assumptions: F is a non-recursive closed value and bound variables introduced in F have kind $*$. Let us now generalize the latter restriction by allowing type constructor variables of kind $p* \to *$ as well as type variables of kind $*$. There are three possibilities for $X : p* \to *$. Note that the variable X would be used as the function part of an application, like $X\,G$, within type B.

case $(X : +* \to *)$ By induction,[5] there exists a map for any valuation of X. So, we denote that map as $fmap_X$. The map for $\lambda Z^{+*}.X\,G$ is

$$fmap_{(\lambda Z^{+*}.X\,G)} = fmap_X \circ fmap_{(\lambda Z^{+*}.G)}$$

[5] We need to make sure that this is a well-founded induction. It may be a coinductive proof.

```
{-# LANGUAGE RankNTypes #-}

data F1 x = C1 (Int → Bool)

instance Functor F1 where
  fmap f (C1 z) = C1 z

data F2 x = C2 x

instance Functor F2 where
  fmap f (C2 z) = C2 (f z)

data F3 x = C3 (([x] → Bool) → Maybe x)

instance Functor F3 where
  fmap z (C3 y) = C3 (λx → fmap z (y (λx1→ x (fmap z x1))))

data F4 x = C4 ((∀ y . [x] → y) → Maybe x)

instance Functor F4 where
  fmap z (C4 y) = C4 (λx → fmap z (y (λx1→ x (fmap z x1))))

data F5 x = C5 (∀ y . ([x] → y) → Maybe x)

instance Functor F5 where
  fmap z (C5 y) = C5 (λx → fmap z (y (λx1→ x (fmap z x1))))

data F6 x = C6 (([x] → ([x] → Bool)) → Maybe x)

instance Functor F6 where
  fmap z (C6 y) =
    C6 (λx → fmap z (y (λx1 x2 → x (fmap z x1) (fmap z x2))))
```

Figure 5.15: Haskell code example to illustrate well-typedness of *fmap*s derived in the proof of Proposition 5.4.1.

Note that $fmap_X$ is not fixed until it is instantiated, however, we definitely know that it exists for any valuation.

case $(X : -* \to *)$ According to the (@) rule applied to $X\ G$, the argument G should be well-kinded under $-\Delta$. Note that $Z^{-*} \in -\Delta$ since $Z^{+*} \in \Delta$. So, Z cannot appear in positive positions in G. It can only appear in negative positions (e.g., $G \triangleq Z \to G'$). Any valuation of X will have the form of $\lambda X_1^{-*}.B_1$ where X_1 appears in negative positions. As a consequence, $(\lambda X_1^{-*}.B_1)G = B_1[G/X_1]$ by a single step reduction. Note that $B_1[G/X_1]$ must be in normal form. Since G has kind $*$ (it cannot be a lambda), substitution of X_1 with G cannot introduce a new redex. Since G is substituted only into negative positions, any Z occurring in a negative position in G become a positive position in the substituted type. Thus, $\lambda Z^{+*}.B_1[G/X_1] : +* \to *$. Hence, by induction,[6] there exists a map for any valuation of X since we can derive $fmap_{(\lambda Z^{+*}.B_1[G/X_1])}$.

case $(X : 0* \to *)$ Note that G cannot have Z in it because Z has $+$ polarity in the context. Recall that, 0Δ ignores the variables with either $+$ or $-$ polarity. According to the (@) rule in Figure 5.2, G should be well-kinded under 0Δ. Since $Z \notin \mathrm{FV}(X\,G)$, we simply return the identity function as the map for $\lambda Z^{+*}.X\,G$.

In addition, having $X : A \to *$ does not make a difference since $X\{s\}$ cannot have Z either. Once we know that we can derive maps in the presence of type constructor variables with single argument, it is easy to generalize this to arbitrary rank-1 kinded type constructor variables (e.g., $+* \to A_1 \to -* \to A_2 \to 0* \to *$).

To complete the proof of Conjecture 5.4.1, we need to consider the equi-recursive type operator (fix) and type constructor variables of kind higher than rank 1. Considering fix makes the proof harder since it becomes less obvious

[6] We need to make sure that this is a well-founded induction. It may be a coinductive proof.

what "normal form" of a type constructor means. Recall that $(\mathsf{fix}\, F_1)$ expands to $F_1(\mathsf{fix}\, F_1)$. When F_1 is a lambda abstraction, the expansion of fix introduces a new redex at the type level. We hope to complete this proof in the future.

Once we have completed the proof of Conjecture 5.4.1, the next step is prove a similar conjecture for higher kinds (recall `fmap1` in Section 5.3.2). For instance, we conjecture that maps exist for type constructors of kind $+(+* \to *) \to (+* \to *)$.

Part IV

Nax Language

Chapter 6

INTRODUCTION TO FEATURES OF THE NAX LANGUAGE

This chapter provides an informal introduction to the Nax programming language. We go through several distinct features of Nax, providing one or more examples for each feature. Basic understanding of these features will be necessary to continue further discussions on design principles (Chapter 7) and type inference (Chapter 8) in the following chapters.

All the examples in this chapter run on our prototype implementation of Nax. An example usually consists of several parts:

- Introducing data definitions to describe the data of interest. Recursive data is introduced in two stages. We must be careful to separate parameters from indices when using indices to describe static properties of data.
- Introduce type synonyms and constructor functions, either by explicit definition or by automatic fixpoint derivation, to limit the amount of explicit notation that must be supplied by the programmer.
- Write a series of definitions that describes how the data is to be manipulated. Deconstruction of recursive data can only be performed with Mendler-style recursion combinators to ensure strong normalization.

6.1 TWO-LEVEL TYPES

Non-recursive datatypes are introduced by the **data** declaration. The data declaration can include arguments. For example, the three non-recursive datatypes,

Bool, *Either*, and *Maybe*, familiar to many functional programmers, are introduced by declaring the kind of the type and the type of each of the constructors. This is similar to the way GADTs are introduced in Haskell.

```
data Bool : ⋆          | data Either : ⋆ → ⋆ → ⋆   | data Maybe : ⋆ → ⋆
  where                |   where                  |   where
  False : Bool         |   Left  : a → Either a b  |   Nothing : Maybe a
  True  : Bool         |   Right : b → Either a b  |   Just    : a → Maybe a
```

Note the kind information ($Bool : \star$) declares *Bool* to be a type, ($Either : \star \rightarrow \star \rightarrow \star$) declares *Either* to be a type constructor with two type arguments, and ($Maybe : \star \rightarrow \star$) declares *Maybe* to be a type constructor with one type argument.

To introduce a recursive type, we first introduce a non-recursive datatype that uses a parameter where the usual recursive components occur. By design, normal parameters of the introduced type are written first (a in L below) and the type argument that stands for the recursive component is written last (the r of N and the r of L below).

```
-- The fixpoint of N will          | -- The fixpoint of (L a) will
-- be the natural numbers.         | -- be the polymorphic lists
data N : ⋆ → ⋆ where               | data L : ⋆ → ⋆ → ⋆ where
  Zero : N r                       |   Nil  : L a r
  Succ : r → N r                   |   Cons : a → r → L a r
```

A recursive type can be defined as the fixpoint of a (perhaps partially applied) non-recursive type constructor. Thus, the traditional natural numbers are typed by $\mu_{[\star]}\ N$ and the traditional lists with components of type a are typed by $\mu_{[\star]}\ (L\ a)$. Note that the recursive type operator $\mu_{[\kappa]}$ is itself specialized with a kind argument inside square brackets ($[\kappa]$). The recursive type $(\mu_{[\kappa]}\ f)$ is well-kinded only if the operand f has kind $\kappa \rightarrow \kappa$, in which case the recursive type $(\mu_{[k]}\ f)$ has kind κ.

Since both N and $(L\ a)$ have kind $\star \to \star$, the recursive types $\mu_{[\star]}\ N$ and $\mu_{[\star]}\ (L\ a)$ have kind $\star$. That is, they are both types, not type constructors.

6.2 CREATING VALUES

Values of a particular datatype are created by the use of constructor functions. For example, *True* and *False* are nullary constructors (or constants) of type *Bool*. (*Left* 4) is a value of type (*Either Int a*). Values of recursive types (i.e., those values with types such as $(\mu_{[k]}\ f)$ are formed by using the special $\mathsf{In}_{[\kappa]}$ constructor expression. Thus, *Nil* has type $L\ a$ and $(\mathsf{In}_{[\star]}\ Nil)$ has type $(\mu_{[\star]}\ (L\ a))$. In general, applying the operator $\mathsf{In}_{[k]}$ injects a term of type $f\ (\mu_{[k]}\ f)$ to the recursive type $(\mu_{[k]}\ f)$. Thus, a list of *Bool* could be created using the term $(\mathsf{In}_{[\star]}\ (Cons\ True\ (\mathsf{In}_{[\star]}\ (Cons\ False\ (\mathsf{In}_{[\star]}\ Nil)))))$. A general rule of thumb is to apply $\mathsf{In}_{[k]}$ to terms of non-recursive types to get terms of recursive types. Writing programs using two-level types and recursive injections has definite benefits, but it certainly makes programs rather annoying to write. Thus, we have provided Nax with a simple but powerful synonym (or macro) facility.

6.3 SYNONYMS, CONSTRUCTOR FUNCTIONS, AND FIXPOINT DERIVATION

We may codify that some type is the fixpoint of another, once and for all, by introducing a type synonym.

synonym $Nat = \mu_{[\star]}\ N$
synonym $List\ a = \mu_{[\star]}\ (L\ a)$

In a similar manner, we can introduce constructor functions that create recursive values without explicit mention of $\mathsf{In}_{[\kappa]}$ at their call sites (potentially many), but only at their site of definition (exactly once).

$$
\begin{aligned}
zero &= \mathsf{In}_{[\star]}\ Zero \\
succ\ n &= \mathsf{In}_{[\star]}\ (Succ\ n) \\
nil &= \mathsf{In}_{[\star]}\ Nil \\
cons\ x\ xs &= \mathsf{In}_{[\star]}\ (Cons\ x\ xs)
\end{aligned}
$$

This is such a common occurrence that recursive synonyms and constructor functions can be automatically derived. Automatic synonym and constructor derivation in Nax is both concise and simple. The clause "**deriving** fixpoint *List*" (below right) automatically derives the **synonym** definition for *List*. It also defines the constructor functions *nil* and *cons*. By convention, the constructor functions are named by dropping the initial upper-case letter in the name of the non-recursive constructors to lower-case. To illustrate, we provide side-by-side comparisons of Haskell and two different uses of Nax.

Haskell	*Nax with synonyms*	*Nax with derivation*
data *List a* $=$ *Nil* $\mid$ *Cons a* (*List a*)	**data** $L : \star \to \star \to \star$ **where** *Nil* : *L a r* *Cons* : $a \to r \to L\ a\ r$ **synonym** *List a* $= \mu_{[\star]}\ (L\ a)$ $nil = \mathsf{In}_{[\star]}\ Nil$ $cons\ x\ xs = \mathsf{In}_{[\star]}\ (Cons\ x\ xs)$	**data** $L : \star \to \star \to \star$ **where** *Nil* : *L a r* *Cons* : $a \to r \to L\ a\ r$ **deriving** fixpoint *List*
$x = Cons\ 3\ (Cons\ 2\ Nil)$	$x = cons\ 3\ (cons\ 2\ nil)$	$x = cons\ 3\ (cons\ 2\ nil)$

6.4 MENDLER COMBINATORS FOR NON-INDEXED TYPES

There are no restrictions on what kinds of datatypes can be defined in Nax. There are also no restrictions on the creation of values for those datatypes. Values of datatypes are created using data constructors and the recursive injection ($\mathsf{In}_{[k]}$). To ensure strong normalization, analysis (or elimination, pattern matching) of the constructed values has some restrictions. Values of non-recursive types can be

freely analyzed using pattern matching. Values of recursive types must be analyzed using one of the Mendler-style combinators. By design, we limit pattern matching to values of non-recursive types, by *not* providing any mechanism to match against the recursive injection ($\mathsf{In}_{[k]}$).

To illustrate simple pattern matching over non-recursive types, we give a multi-clause definition for the $\neg$ function[1] over the (non-recursive) *Bool* type, and a function that strips off the *Just* constructor over the (non-recursive) *Maybe* type using a case expression.

$$\begin{array}{l|l} \neg\ \mathit{True} = \mathit{False} & \mathit{unJust0}\ x = \mathbf{case}_{\{\}}\ x\ \mathbf{of}\ \mathit{Just}\ x \quad \to x \\ \neg\ \mathit{False} = \mathit{True} & \qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to 0 \end{array}$$

Of course, the $\neg$ function can also be defined as a single clause definition using the **case** expression and the *unJust0* function can also be defined as a multi-clause definition.

Analysis of recursive data is performed by Mendler-style recursion combinators. In our implementation, we provide 5 families of Mendler-style combinators: **mit**. (fold or catamorphism or iteration), **mpr**. (primitive recursion), **mcvit**. (courses-of-values iteration), **mcvpr**. (courses-of-values recursion), and **msfit**. (fold or catamorphism or iteration for recursive types with negative occurrences).

A Mendler-style combinator is written in a manner similar to a case expression. A Mendler-style combinator expression contains patterns, and the variables bound in the patterns are scoped over a term. This term is executed if the pattern matches. A Mendler-style combinator expression differs from a case expression in that it also introduces additional names (or variables) into scope. These variables play a role similar in nature to the operations of an abstract datatype and provide additional functionality aside to what can be expressed using just case analysis.

[1] $\neg$ is just a pretty-printed notation of *not* using lhs2TeX.

For a visual example, compare the **case** expression to the **mit**. expression. In the **case** expression, each *clause* following the **of** indicates a possible match of the scrutinee x. In the **mit**. expression, each *equation* following the **with**, binds the variable f, and matches the pattern to a value related to the scrutinee x.

$$\begin{array}{l|l} \begin{array}{rl} \textbf{case}_{\{\}}\ x\ \textbf{of}\ \mathit{Nil} & \rightarrow e_1 \\ \mathit{Cons}\ x\ xs & \rightarrow e_2 \end{array} & \begin{array}{rll} \textbf{mit}_{\{\}}\ x\ \textbf{with}\ f & (\mathit{Cons}\ x\ xs) & = e_1 \\ f & \mathit{Nil} & = e_2 \end{array} \end{array}$$

The number and type of the additional variables depends upon which family of Mendler combinators is used to analyze the scrutinee. Each equation specifies (a potential) computation in an abstract datatype depending on whether the pattern matches. For the **mit**. combinator (above), the abstract datatype has the following form. The scrutinee x is a value of some recursive type $(\mu_{[\star]}\ T)$ for a non-recursive type constructor T. In each clause, the pattern has type $(T\ r)$, for some abstract type r. The additional variable introduced (f) is an operator over the abstract type r that can safely manipulate only abstract values of type r.

Different Mendler-style combinators are implemented by different abstract types. Each abstraction safely describes a class of provably terminating computations over a recursive type. The number (and type) of abstract operations differs from one family of Mendler combinators to another. Below, we give descriptions of three families of Mendler combinators, their abstractions, and the types of the operators within the abstraction. In each description, the type *ans* represents the result type when the Mendler combinator is fully applied.

$\mathbf{mit}_{\{\}}\ x\ \mathbf{with}$ $\quad f\ \ p_i = e_i$	$\mathbf{mpr}_{\{\}}\ x\ \mathbf{with}$ $\quad f\ \ \mathit{cast}\ \ p_i = e_i$	$\mathbf{mcvit}_{\{\}}\ x\ \mathbf{with}$ $\quad f\ \ \mathit{out}\ \ p_i = e_i$
$x\ : \mu_{[\star]}\ T$ $f\ : r \to \mathit{ans}$	$x\ : \mu_{[\star]}\ T$ $f\ : r \to \mathit{ans}$ $\mathit{cast} : r \to \mu_{[\star]}\ T$	$x\ \ : \mu_{[\star]}\ T$ $f\ \ : r \to \mathit{ans}$ $\mathit{out} : r \to T\ r$
$p_i : T\ r$ $e_i : \mathit{ans}$	$p_i : T\ r$ $e_i : \mathit{ans}$	$p_i\ \ : T\ r$ $e_i\ \ : \mathit{ans}$
$\mathbf{mit}_{\{\psi\}}\ \varphi\ (\mathsf{In}_{[\star]}\ x)$ $\quad = \varphi\ (\mathbf{mit}_{\{\psi\}}\ \varphi)\ x$	$\mathbf{mpr}_{\{\psi\}}\ \varphi\ (\mathsf{In}_{[\star]}\ x)$ $\quad = \varphi\ (\mathbf{mpr}_{\{\psi\}}\ \varphi)\ (\mathsf{In}_{[\star]})\ x$	$\mathbf{mcvit}_{\{\psi\}}\ \varphi\ (\mathsf{In}_{[\star]}\ x)$ $\quad = \varphi\ (\mathbf{mcvit}_{\{\psi\}}\ \varphi)\ \mathit{out}\ x$ $\quad \mathbf{where}\ \mathit{out}\ (\mathsf{In}_{[\star]}\ x) = x$

A Mendler-style combinator implements a (provably terminating) recursive function applied to the scrutinee. The abstract type and its operations ensure termination. Note that every operation above includes an abstract operator, $f : r \to \mathit{ans}$. This operation represents a recursive call in the function defined by the Mendler-style combinator. Other operations, such as *cast* and *out*, support additional functionality within the abstraction in which they are defined (**mpr**. and **mcvit**., respectively). The equations at the bottom of each column above provide an operational understanding of how each Mendler-style combinator works. These can be safely ignored until after we see some examples of how a Mendler-style combinator works in practice. In Figure 6.1, the *length* function uses the simplest kind of recursion where each recursive call is an application to a direct subcomponent of the input. Operationally, *length* works as follows. The scrutinee y has type $(\mu_{[\star]}\ (L\ a))$ and has the form $(\mathsf{In}_{[\star]}\ x)$. The type of y implies that x must have the form *Nil* or $(\mathit{Cons}\ x\ xs)$. The **mit**. strips off the $\mathsf{In}_{[\star]}$ and matches x against the *Nil* and $(\mathit{Cons}\ x\ xs)$ patterns. If the *Nil* pattern matches, then 0 is returned. If the $(\mathit{Cons}\ x\ xs)$ pattern matches, x and xs are bound. The abstract type mechanism

$$
\begin{array}{lll}
\mathit{length}\ y & = \mathbf{mit}_{\{\}}\ y & \mathbf{with}\ \mathit{len}\ \mathit{Nil} = \mathit{zero} \\
 & & \quad \mathit{len}\ (\mathit{Cons}\ x\ xs) = (\mathit{succ}\ \mathit{zero}) + \mathit{len}\ xs \\
\mathit{tail}\ x & = \mathbf{mpr}_{\{\}}\ x & \mathbf{with}\ \mathit{tl}\ \mathit{cast}\ \mathit{Nil} = \mathit{nil} \\
 & & \quad \mathit{tl}\ \mathit{cast}\ (\mathit{Cons}\ y\ ys) = \mathit{cast}\ ys \\
\mathit{factorial}\ x & = \mathbf{mpr}_{\{\}}\ x & \mathbf{with}\ \mathit{fact}\ \mathit{cast}\ \mathit{Zero} = \mathit{succ}\ \mathit{zero} \\
 & & \quad \mathit{fact}\ \mathit{cast}\ (\mathit{Succ}\ n) = \mathit{times}\ (\mathit{succ}\ (\mathit{cast}\ n))\ (\mathit{fact}\ n) \\
\mathit{fibonacci}\ x & = \mathbf{mcvit}_{\{\}}\ x & \mathbf{with}\ \mathit{fib}\ \mathit{out}\ \mathit{Zero} = \mathit{succ}\ \mathit{zero} \\
 & & \quad \mathit{fib}\ \mathit{out}\ (\mathit{Succ}\ n) = \mathbf{case}_{\{\}}\ (\mathit{out}\ n)\ \mathbf{of} \\
 & & \qquad \mathit{Zero} \rightarrow \mathit{succ}\ \mathit{zero} \\
 & & \qquad \mathit{Succ}\ m \rightarrow \mathit{fib}\ n + \mathit{fib}\ m
\end{array}
$$

Figure 6.1: Illustrating the use of the Mendler-style recursion combinators provided in Nax by simple examples: *length*, *tail*, *factorial*, and *fibonacci*.

gives the pattern $(\mathit{Cons}\ x\ xs)$ and the type $(L\ a\ r)$, so $(x : a)$ and $(xs : r)$ for some abstract type r. The abstract operation, $(\mathit{len} : r \rightarrow \mathit{Int})$, can safely be applied to xs, obtaining the length of the tail of the original list. This value is incremented and then returned. The abstract operation of **mit**. provides a safe way to allow the user to make recursive calls, *len*, but the abstract type, r, limits its use to direct subcomponents, so termination is guaranteed.

Some recursive functions need access to the concrete values of the direct subcomponents (of type $\mu_{[\star]}\ T$), in addition to applying abstract recursive calls on the abstract handles of the direct subcomponents (of type r). The Mendler-style combinator **mpr**. provides a safe, yet abstract mechanism to support this.

There are two abstract operations provided by **mpr**.: the recursive caller with type $(r \rightarrow \mathit{ans})$ and a casting function with type $(r \rightarrow \mu_{[\star]}\ T)$. The casting operation allows the user to recover the original type from the abstract recursive type r, but since the recursive caller only works on the abstract recursive type r, the user cannot make a recursive call on one of these cast values. The functions

factorial (over the natural numbers) and *tail* (over lists) are both defined using **mpr**_.

Note how in *factorial* the original input is recovered (in constant time) by taking the successor of the concrete predecessor value (*cast* n) obtained by casting the abstract predecessor n. In the *tail* function, the abstract tail *ys* is cast to get the answer, and the recursive caller is not even used.

Some recursive functions need access, not only to the direct subcomponents, but also to even deeper subcomponents. The Mendler-style combinator **mcvit**_ provides a safe,[2] yet abstract mechanism to support this. The function *fibonacci* is a classic example of this kind of recursion. The recursion combinator **mcvit**_ provides two abstract operations: the recursive caller with type $(r \to ans)$ and a projection function with type $(r \to T\ r)$. The projection allows the programmer to observe the hidden T structure inside a value of the abstract recursive type r. In the *fibonacci* function above, we name the projection *out*. It is used to observe if the abstract predecessor, n, of the input, x, is either zero, or the successor of the second predecessor m of x. Note how recursive calls are made on the direct predecessor n and the second predecessor m.

Each recursion combinator can be defined by the equation at the bottom of its figure. Each combinator can be given a naive type involving the concrete recursive type $(\mu_{[\star]}\ T)$, but if we instead give it a more abstract type, abstracting values of type $(\mu_{[\star]}\ T)$ into some unknown abstract type r, one can safely guarantee a certain pattern of use that ensures termination. Informally, if the combinator works for some unknown type r, it will certainly also work for the actual type $(\mu_{[\star]}\ T)$, but because we cannot assume that r has any particular structure; the user is forced to use the abstract operations in carefully proscribed ways.

[2] Only for positive datatypes, of course.

6.5 TYPES WITH STATIC INDICES

Recall that a type can have both parameters and indices, and that indices can be either types or terms. We define three types below, each with one or more indices. Each example defines a non-recursive type and then uses fixpoint derivation to define synonyms for its fixpoint and constructor functions. By convention, in each example, the argument that abstracts the recursive components is called r. By design, arguments appearing before r are understood to be parameters, and arguments appearing after r are understood to be indices. To define a recursive type with indices, it is necessary to give the argument r a higher kind. That is, r should take indices as well, since it abstracts over a recursive type which takes indices.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nest} : (\star \to \star) \to \star \to \star\ \textbf{where} \\
\quad \mathit{Tip}\ \ \ : a \to \mathit{Nest}\ r\ a \\
\quad \mathit{Fork}\ \ : r\ (a, a) \to \mathit{Nest}\ r\ a \\
\quad\quad \textbf{deriving}\ \text{fixpoint}\ \mathit{PowerTree} \\
\textbf{data}\ V : \star \to (\mathit{Nat} \to \star) \to \mathit{Nat} \to \star\ \textbf{where} \\
\quad \mathit{Vnil}\ \ \ : V\ a\ r\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{Vcons} : a \to r\ \{n\} \to V\ a\ r\ \{\text{`}\mathit{succ}\ n\} \\
\quad\quad \textbf{deriving}\ \text{fixpoint}\ \mathit{Vector} \\
\textbf{data}\ \mathit{Tag} = E \mid O \\
\textbf{data}\ P : (\mathit{Tag} \to \mathit{Nat} \to \star) \to \mathit{Tag} \to \mathit{Nat} \to \star\ \textbf{where} \\
\quad \mathit{Base}\ \ : P\ r\ \{E\}\ \{\text{`}\mathit{zero}\} \\
\quad \mathit{StepO} : r\ \{O\}\ \{i\} \to P\ r\ \{E\}\ \{\text{`}\mathit{succ}\ i\} \\
\quad \mathit{StepE} : r\ \{E\}\ \{i\} \to P\ r\ \{O\}\ \{\text{`}\mathit{succ}\ i\} \\
\quad\quad \textbf{deriving}\ \text{fixpoint}\ \mathit{Proof}
\end{array}
$$

Note, to distinguish type indices from term indices (and to make parsing unambiguous), we enclose term indices in braces ($\{...\}$). We also backquote (‘) variables in terms that we expect to be bound in the current environment. Un-backquoted variables are taken to be universally quantified. By backquoting *succ*, we indicate that we want terms that are applications of the successor function, but not some

universally quantified variable[3]. For non-recursive types without parameters, the kind of the fixpoint is the same as the kind of the recursive argument r. If the non-recursive type has parameters, the kind of the fixpoint will be composed of the parameters $\rightarrow$ the kind of the recursive argument r. For example, study the kinds of the fixpoints for the non-recursive types declared above in the table below.

non-recursive type	*Nest*	*V*	*P*
recursive type	*PowerTree*	*Vector*	*Proof*
kind of T	$\star \rightarrow \star$	$\star \rightarrow Nat \rightarrow \star$	$Tag \rightarrow Nat \rightarrow \star$
kind of r	$\star \rightarrow \star$	$Nat \rightarrow \star$	$Tag \rightarrow Nat \rightarrow \star$
number of parameters	0	1	0
number of indices	1 (type)	1 (term)	2 (term,term)

Recall, indices are used to track static properties about values with those types. A well-formed (*PowerTree* x) contains a balanced set of parenthesized binary tuples of elements. The index x describes the type of values nested in the parentheses. The invariant is that the number of items nested is always an exact power of 2. A (*Vector* a $\{n\}$) is a list of elements of type a, with length exactly equal to n, and a (*Proof* $\{E\}$ $\{n\}$) witnesses that the natural number n is even, while a (*Proof* $\{O\}$ $\{m\}$) witnesses that the natural number m is odd. Some example values with these types are given below.

$$tree1 : PowerTree\ Int = tip\ 3$$
$$tree2 : PowerTree\ Int = fork\ (tip\ (2, 5))$$
$$tree3 : PowerTree\ Int = fork\ (fork\ (tip\ ((4, 7), (0, 2))))$$
$$v_2 : Vector\ Int\ \{succ\ (succ\ zero)\} = (vcons\ 3\ (vcons\ 5\ vnil))$$
$$p1 : P\ \{O\}\ \{succ\ zero\} = stepE\ base$$
$$p2 : P\ \{E\}\ \{succ\ (succ\ zero)\} = stepO\ (stepE\ base)$$

[3] In the design of Nax, we had a choice. Either explicitly declare each universally quantified variable or explicitly mark those variables not universally quantified. Since quantification is much more common than referring to variables already in scope, the choice was easy.

Note that in the types of the terms above, the indices in braces ($\{...\}$) are ordinary terms (not types). In these examples, we use natural numbers (e.g., *succ* (*succ zero*)) and elements (E and O) of the two-valued type *Tag*. It is interesting to note that sometimes the terms are of recursive types (e.g., *Nat* which is a synonym for $\mu_{[\star]}\ N$), while some are non-recursive types (e.g., *Tag*).

6.6 MENDLER-STYLE COMBINATORS FOR INDEXED TYPES

Mendler-style combinators generalize naturally to indexed types. The key observation that makes this generalization possible is that the types of the operations within abstraction have to be generalized to deal with the type indices in a consistent manner. How this is done is best first explained by example, and then later abstracted to its full general form.

Recall, a value of type (*PowerTree Int*) is a set of integers. This set is constructed as a balanced binary tree with pairs at the leaves (see *tree2* and *tree3* above). The number of integers in the set is an exact power of 2. Consider a function that adds up all those integers. One wants a function of type (*PowerTree Int* $\rightarrow$ *Int*). One strategy for writing this function is to write a more general function of type (*PowerTree* $a \rightarrow (a \rightarrow Int) \rightarrow Int$). In Nax, we can do this as follows:

$$
\begin{aligned}
&genericSum\ t = \mathbf{mit}_{\{a.\,(a \rightarrow Int) \rightarrow Int\}}\ t\ \mathbf{with} \\
&\qquad\qquad sum\ (Tip\ x)\ \ = \lambda f \rightarrow f\ x \\
&\qquad\qquad sum\ (Fork\ x) = \lambda f \rightarrow sum\ x\ (\lambda(a, b) \rightarrow f\ a + f\ b) \\
&sumTree\ t = genericSum\ t\ (\lambda x \rightarrow x)
\end{aligned}
$$

In general, the type of the result of a function over an indexed type can depend upon what the index is. Thus, a Mendler-style combinator over a value with an indexed type must be type-specialized to that value's index. Different values of the same general type will have different indices. After all, the role of an index is to witness an invariant about the value, and different values might have different

invariants. Capturing this variation is the role of the clause $\{\, a \,.\, (a \to \mathit{Int}) \to \mathit{Int}\}$ following the keyword **mit**.. We call such a clause an *index transformer*. In the same way that the type of the result depends upon the index, the type of the different components of the abstract datatype implementing the Mendler-style combinator also depend upon the index. In fact, everything depends upon the index in a uniform way. The index transformer captures this uniformity. One cannot abstract over the index transformer in Nax. Each Mendler-style combinator, over an indexed type, must be supplied with a concrete clause (inside the braces) that describe how the results depend upon the index. To see how the transformer is used, study the types of the terms in the following paragraph. Can you see the relation between the types and the transformer?

The scrutinee t has type $(\mathit{PowerTree}\ a)$, which is a synonym for $((\mu_{[\star\to\star]}\ \mathit{Nest})\ a)$. The recursive caller *sum* has type $(\forall a\,.\ r\ a \to (a \to \mathit{Int}) \to \mathit{Int})$, for some abstract type constructor r. Recall that r has an index. So r must be a type constructor, not a type. The patterns $(\mathit{Tip}\ x)$ and $(\mathit{Fork}\ x)$ have type $(\mathit{Nest}\ r\ a)$ and the right-hand sides of the equations $(\lambda f \to f\ x)$ and $(\lambda f \to \mathit{sum}\ x\ (\lambda(a,b) \to f\ a + f\ b))$ have type $((a \to \mathit{Int}) \to \mathit{Int})$. Note that the dependency of $((a \to \mathit{Int}) \to \mathit{Int})$ on the index a appears in both the result type and the type of the recursive caller. If we think of an index transformer such as $\{\, a\ .\ (a \to \mathit{Int}) \to \mathit{Int}\}$ as a function $\psi\ \ a = (a \to \mathit{Int}) \to \mathit{Int}$, we can succinctly describe the types of the abstract operations of **mit**.. In the table below, we put the general case for the general form $(\mathbf{mit}_{\{\psi\}}\ x\ \ \mathbf{with}\ \overline{f\ p_i = e_i})$ on the left, and terms from the *genericSum* example that illustrate the general case on the right.

$\psi : \kappa \to \star$	$\{\, a \,.\, (a \to Int) \to Int \,\}$	:	$\star \to \star$
$T : (\kappa \to \star) \to \kappa \to \star$	$Nest$	:	$(\star \to \star) \to \star \to \star$
$x : (\mu_{[\kappa \to \star]}\ T)\ a$	t	:	$(\mu_{[\star \to \star]}\ Nest)\ a$
$f : \forall a : \kappa \,.\, r\ a \to \psi\ a$	sum	:	$\forall a : \star \,.\, r\ a \to (a \to Int) \to Int$
$p_i : T\ r\ a$	$Fork\ x$	:	$Nest\ r\ a$
$e_i : \psi\ a$	$\lambda f \to f\ x$	:	$(a \to Int) \to Int$

The same scheme for **mit**. generalizes to type constructors with term indices and with multiple indices. To illustrate this, we give the generic schemes for type constructors with 2 or 3 indices. In the table, the variables κ_1, κ_2, and κ_3 stand for arbitrary kinds (either kinds for types such as $\star$ or kinds for terms such as Nat or Tag).

$T : (\kappa_1 \to \kappa_2 \to \star) \to (\kappa_1 \to \kappa_2 \to \star)$	$T : (\kappa_1 \to \kappa_2 \to \kappa_3 \to \star) \to (\kappa_1 \to \kappa_2 \to \kappa_3 \to \star)$
$\psi : \kappa_1 \to \kappa_2 \to \star$	$\psi : \kappa_1 \to \kappa_2 \to \kappa_3 \to \star$
$x : (\mu_{[\kappa_1 \to \kappa_2 \to \star]}\ T)\ a\ b$	$x : (\mu_{[\kappa_1 \to \kappa_2 \to \kappa_3 \to \star]}\ T)\ a\ b\ c$
$f : \forall a : \kappa_1\ (b : \kappa_2) \,.\, r\ a\ b \to \psi\ a\ b$	$f : \forall a : \kappa_1\ (b : \kappa_2)\ (c : \kappa_3) \,.\, r\ a\ b\ c \to \psi\ a\ b\ c$
$p_i : T\ r\ a\ b$	$p_i : T\ r\ a\ b\ c$
$e_i : \psi\ a\ b$	$e_i : \psi\ a\ b\ c$

The simplest form of index transformation is where the transformation is a constant function. This is the case of the function that computes the integer length of a length-indexed list (what we call a $Vector$). Independent of the length, the result is an integer. Such a function has type $Vector\ a\ \{\, n \,\} \to Int$. We can write this as follows:

$$\begin{aligned} vlen\ x = \mathbf{mit}_{\{\{i\}.\, Int\}}\ x\ \mathbf{with}\ & len\ Vnil & = 0 \\ & len\ (Vcons\ x\ xs) & = 1 + len\ xs \end{aligned}$$

Let's study an example with a more interesting index transformation. A term with type $(Proof\ \{\, E \,\}\ \{\, n \,\})$, which is synonymous with $(\mu_{[\,Tag \to Nat \to \star]}\ P\ \{\, E \,\}\ \{\, n \,\})$, witnesses that the term n is even. Can we transform such a term into a proof that

$n + 1$ is odd? We can generalize this by writing a function which has both of the types below:

$Proof\ \{E\}\ \{n\} \to Proof\ \{O\}\ \{`succ\ n\}$, and

$Proof\ \{O\}\ \{n\} \to Proof\ \{E\}\ \{`succ\ n\}$.

We can capture this dependency by defining the term-level function *flip* and using a **mit**. with the index transformer: $\{\{t\}\ \{i\}\ .\ Proof\ \{`flip\ t\}\ \{`succ\ i\}\}$.

$$
\begin{aligned}
&flip\ E = O\\
&flip\ O = E\\
&flop\ x = \mathbf{mit}_{\{\{t\}\ \{i\}.\ Proof\ \{`flip\ t\}\ \{`succ\ i\}\}}\ x\ \mathbf{with}\\
&\qquad f\ Base\ \ \ \ \ \ = stepE\ base\\
&\qquad f\ (StepO\ p) = stepE\ (f\ p)\\
&\qquad f\ (StepE\ p) = stepO\ (f\ p)
\end{aligned}
$$

For our last term-indexed example, every length-indexed list has a length, which is either even or odd. We can witness this fact by writing a function with type: $Vector\ a\ \{n\} \to Either\ (Even\ \{n\})\ (Odd\ \{n\})$. Here, *Even* and *Odd* are synonyms for particular kinds of *Proof*. To write this function, we need the index transformation: $\{\{n\}\ .\ Either\ (Even\ \{n\})\ (Odd\ \{n\})\}$.

$$
\begin{aligned}
&\mathbf{synonym}\ Even\ \{x\} = Proof\ \{E\}\ \{x\}\\
&\mathbf{synonym}\ Odd\ \ \{x\} = Proof\ \{O\}\ \{x\}\\
&proveEvenOrOdd\ x = \mathbf{mit}_{\{\{n\}.\ Either\ (Even\ \{n\})\ (Odd\ \{n\})\}}\ x\ \mathbf{with}\\
&\qquad\qquad prEOO\ Vnil\ \ \ \ \ \ \ \ \ \ = Left\ base\\
&\qquad\qquad prEOO\ (Vcons\ x\ xs) = \mathbf{case}_{\{\}}\ prEOO\ xs\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad Left\ p\ \ \ \to Right\ (stepE\ p)\\
&\qquad\qquad\qquad\qquad\qquad\qquad Right\ p \to Left\ (stepO\ p)
\end{aligned}
$$

6.7 RECURSIVE TYPES OF UNRESTRICTED POLARITY BUT RESTRICTED ELIMINATION

In Nax, programmers can define recursive data structures with both positive and negative polarity. The classic example is a datatype encoding the syntax of λ-calculus, which uses higher-order abstract syntax (HOAS). Terms in the λ-calculus are variables, applications, or abstractions. In a HOAS representation, one uses Nax functions to encode abstractions. We give a two-level description for recursive λ-calculus *Term*s, by taking the fixpoint of the non-recursive *Lam* datatype.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Lam} : \star \to \star\ \textbf{where} \\
\quad \mathit{App} :: r \to r \to \mathit{Lam}\ r \\
\quad \mathit{Abs} :: (r \to r) \to \mathit{Lam}\ r \\
\qquad \textbf{deriving}\ \text{fixpoint}\ \mathit{Term} \\
\mathit{apply} = \mathit{abs}\ (\lambda f \to \mathit{abs}\ (\lambda x \to \mathit{app}\ f\ x))
\end{array}
$$

Note that we don't need to include a constructor for variables, as variables are represented by Nax variables, bound by Nax functions. For example, the lambda term $(\lambda f.\lambda x.f\ x)$ is encoded by the Nax term *apply* above.

Note also, the constructor function $\mathit{abs} : (\mathit{Term} \to \mathit{Term}) \to \mathit{Term}$ introduced by the **deriving** fixpoint clause, has a negative occurrence of the type *Term*. In a language with unrestricted analysis, such a type could lead to non-terminating computations. The Mendler **mit**. and **mpr**. combinators limit the analysis of such types in a manner that precludes non-terminating computations. The Mendler-style combinator **mcvit**. is too expressive to exclude non-terminating computations and must be restricted to recursive datatypes with no negative occurrences.

Even though **mit**. and **mpr**. allow us to safely operate on values of type *Term*, they are not expressive enough to write many interesting functions. Fortunately, there is a more expressive Mendler-style combinator that is safe over recursive types with negative occurrences. We call this combinator **msfit**.. This combinator is based upon an interesting programming trick, first described by Sheard and

Fegaras [32], hence the "**sf**" in the name **msfit**.. The abstraction supported by **msfit**. is as follows:[4]

$\mathbf{msfit}_{\{\}}\ x\ \mathbf{with}$ $\quad f\ inv\ p_i = e_i$	$x \quad : \mu_{[\star]}\ T$ $f \quad : r \to ans$ $inv : ans \to r$ $p_i \quad : T\ r$ $e_i \quad : ans$

To use **msfit**. the inverse allows one to cast an answer into an abstract value. To see how this works, study the function that turns a *Term* into a string. The strategy is to write an auxiliary function *showHelp* that takes an extra integer argument. Every time we encounter a lambda abstraction, we create a new variable `x`n (see the function *new*), where n is the current value of the integer variable. When we make a recursive call, we increment the integer. In the comments (the rest of a line after --), we give the types of a few terms, including the abstract operations *sh* and *inv*.

```
                                -- cat : List String → String
                                -- new : Int → String
new n = cat ["x", show n]
                                -- showHelp : Term → (Int → String)
                                -- sh : r → (Int → String)
                                -- inv : (Int → String) → r
                                -- (λn → new m) : Int → String
showHelp x =
  msfit{} x with
    sh inv (App x y) = λm → cat ["(", sh x m, " ", sh y m, ")"]
    sh inv (Abs f)   = λm → cat ["(fn ", new m, " => ",
                                 sh (f (inv (λn → new m))) (m + 1), ")"]
```

[4] More precisely, we need to use $\breve{\mu}$, which is different from μ, for **msfit**. (see Section 10.2). We have not correctly implemented this in our current implementation, which we are using to run the examples in this dissertation. So, our example here just uses μ instead of $\breve{\mu}$. But, we are working it the right way in the new implementation.

showTerm x = *showHelp* x 0
showTerm apply : *List Char* = `"(fn x0 => (fn x1 => (x0 x1)))"`

The final line of the example above illustrates applying *showTerm* to *apply*. Recall that *apply* = *abs* $(\lambda f \rightarrow$ *abs* $(\lambda x \rightarrow$ *app* $f\ x))$, which is the HOAS representation of the λ-calculus term $(\lambda f.\lambda x.f\ x)$.

There are more details behind the **msfit**. and fixpoint derivations for the datatypes on which **msfit**. operates. Recall, in Chapter 3, we described in Haskell that **msfit** operates on recursive values of a fixpoint type ($\breve{\mu}$) augmented by a syntactic inverse, while other recursion schemes operate on recursive values of a standard fixpoint type (μ). For further discussions, see Section 10.2.

6.8 LESSONS FROM NAX

Nax is our attempt to build a strongly normalizing, sound and consistent logic based upon Mendler-style recursion combinators. We would like to emphasize the lessons we learned along the way.

- Writing types as the fixed point of a non-recursive type constructor (two-level types) is quite expressive. It supports a wide variety of types including regular types (*Nat* and *List*), nested types (*PowerTree*), GADTs (*Vector*), and mutually recursive types (*Even* and *Odd*).
- Two-level types, while expressive, are difficult to program with (all those $\mu_{[\kappa]}$ and $\mathsf{In}_{[\kappa]}$ annotations), so a strong synonym facility is necessary. With syntactic support of synonyms and automatic derivation of synonyms for recursive types, one hardly notices extra verbosity due to the use of two-level types.
- The use of term-indexed types allows programmers to write types that act as logical relations and form the basis for reasoning about programs. In Chapters 4 and 5, we formalized lambda calculi, which support term indices.

- Using Mendler-style combinators is expressive and, with syntactic support (the **with** equations of the Mendler combinators), easy to use. In fact, Nax programs are often no more complicated than their Haskell counterparts, except the use of Mendler-style recursion combinators instead of general recursion.
- Type inference is an important feature of a programming language. We hope you noticed, apart from index transformers and datatype declarations, no type information is supplied in any of the Nax examples. Our Nax implementation can reconstruct all other type information.
- Index transformers are the minimal information needed to extend Hindley–Milner type inference over GADTs. One can always predict where they are needed, and the Nax implementation can enforce that the programmer supplies them. They are never needed for non-indexed types. Nax faithfully extends Hindley–Milner type inference.

Chapter 7

DESIGN PRINCIPLES OF NAX'S TYPE SYSTEM

7.1 INTRODUCTION

During the past decade, the functional programming community has achieved partial success in their goal of maintaining fine-grained properties by only moderately extending functional language type systems [17, 18, 98]. This approach is often called *"lightweight"*[1] in contrast to the approach taken by fully dependent type systems (e.g., Coq, Agda). The Generalized Algebraic Data Type (GADT) extension, implemented in the Glasgow Haskell Compiler (GHC) and in OCaml [34, 56], has made the lightweight approach widely applicable to everyday functional programming tasks.

Unfortunately, most practical lightweight implementations lack **logical consistency** and **type inference**. In addition, they often lack term indexing, so **term indices are faked** (or simulated) by using an additional type structure to replicate the requisite term structure. A recent extension in GHC, datatype promotion [99], addresses the issue of term indices, but the issues of logical consistency and type inference remain.

Nax is a programming language designed to support both type- and term-indexed datatypes, logical consistency, and type inference.

(1) **Nax is strongly normalizing and logically consistent.**

Types in Nax can be given logical interpretations as propositions and the

[1] e.g., `http://okmij.org/ftp/Computation/lightweight-dependent-typing.html`

programs of those types as proofs of those propositions. Theories behind strong normalization and logical consistency include Mendler-style recursion [6] discussed in Chapter 3 and the lambda calculi, System F_i, and System Fix_i, discussed in Chapters 4 and 5.

(2) **Nax supports Hindley–Milner-style type inference.**

Nax needs few type annotations. In particular, annotations for top-level functions, which are usually required for bidirectional type checking in dependently-typed languages, are unnecessary. Type annotations are only required when introducing GADTs and as index transformers attached to pattern matching constructs (**case** and Mendler-style combinators such as **mit**.) for GADTs. We will discuss further details on type inference in Chapter 8.

(3) **Nax programs are expressive and concise.**

Nax programs are similar in size to their Haskell and Agda equivalents (Section 7.2), yet they still retain logical consistency and type inference. Despite several features unique to Nax, explained in Table 7.1, these features do not necessarily add verbosity.

(4) **Nax supports term indices within a relatively simple type system.**

The type system of Nax (Section 7.3.1) is based on a two-level universe structure, just like Haskell, yet it allows nested term indices (Section 7.3.2) as in languages based on a universe structure of countably many levels (e.g., Coq, Agda).

The detailed mechanisms behind (1) and (2) are discussed in other chapters. In this chapter, we demonstrate (3) and (4), through a series of examples: – a type-preserving evaluator (Section 7.2.1), a generic path datatype (Section 7.2.2), and a stack-safe compiler (Section 7.2.3). These examples demonstrate that programming in Nax can be as succinct as as programming in Haskell or Agda. Then, we discuss the key design principles behind indexed datatypes in Nax (Section 7.3.1)

and its strengths and limitations (Section 7.3.2).

7.2 THE TRILINGUAL ROSETTA STONE

In this section, we introduce three examples (Figures 7.1 and 7.2, Figures 7.3 and 7.4, and Figures 7.5 and 7.6) that use term-indexed datatypes to enforce program invariants. Each example is written in three different languages – like the Rosetta Stone – Haskell, Nax, and Agda. We have crafted these programs to look as similar to one another as possible by choosing the same identifiers and syntax structure whenever possible. So, anyone already familiar with Haskell-like languages or Agda-like languages could easily understand our Nax programs just by comparing them with the programs on the left or on the right. The features unique to Nax, which are used in this chapter, are summarized in Table 7.1 (review Chapter 6 for further details).

The three examples we introduce are the following:

- A type-preserving evaluator for a simple expression language (Section 7.2.1),
- A generic *Path* datatype that can be specialized to various list-like structures with indices (Section 7.2.2), and
- A stack-safe compiler for the same simple expression language, which uses the *Path* datatype (Section 7.2.3).

We adopt the examples from Conor McBride's keynote talk [62] at ICFP 2012 (originally written in Agda). All the example code was tested in GHC 7.4.1 (should also work in later versions such as GHC 7.6.x), our prototype Nax implementation, and Agda 2.3.0.1.

7.2.1 Type-preserving evaluator for an expression language

In a language that supports term indices, one writes a type-preserving evaluator as follows: (1) define a datatype TypeUniverse which encodes types of the object

The "**deriving** fixpoint T" clause after **data** $F : \overline{k} \to \kappa \to \kappa$ **where** $\cdots$ automatically derives a recursive type synonym $T\,\overline{a} = \mu_{[\kappa]}\ (F\,\overline{a}) : \kappa$ and its constructor functions. For instance, the deriving clause below left automatically derives the definitions below right:

data $L : \star \to \star \to \star$ **where**	**synonym** $List\ a = \mu_{[\star]}\ (L\ a)$
$Nil \quad : L\ a\ r$	$nil = \mathsf{In}_{[\star]}\ Nil$
$Cons : a \to r \to L\ a\ r$	$cons\ x\ xs = \mathsf{In}_{[\star]}\ (Cons\ x\ xs)$
deriving fixpoint $List$	

The **synonym** keyword defines a type synonym, just like Haskell's **type** keyword.

In Nax, **data** declarations cannot be recursive. Instead, to define recursive types, one uses a fixpoint type operator $\mu_{[\kappa]} : (\kappa \to \kappa) \to \kappa$ over non-recursive base structures of kind $\kappa \to \kappa$ (e.g., $(L\ a) : \star \to \star$). Nax provides the usual data constructor $\mathsf{In}_{[\kappa]}$ to construct recursive values of the type $\mu_{[\kappa]}$. $\mathsf{In}_{[\kappa]}$ is used to define the normal constructor functions of recursive types (e.g., *nil* and *cons*).

However, one cannot pattern match against $\mathsf{In}_{[\kappa]}\ e$ in Nax. Instead, Nax provides several well-behaved (i.e., always terminating) Mendler-style recursion combinators such as **mit**. that work naturally over μ types, even with indices.

To support type inference, Nax requires programmers to annotate Mendler-style combinators with index transformers. For instance, Nax can infer that the term $(\lambda x \to \mathbf{mit}_{\{\{i\}\,\{j\}.\ T_2\,\{j\}\,\{i\}\}}\ x\ \mathbf{with}\ \cdots)$ has type $T_1\ \{i\}\ \{j\} \to T_2\ \{j\}\ \{i\}$ using the information in the index transformer $_{\{\{i\}\,\{j\}\ .\ T_2\,\{j\}\,\{i\}\}}$.

Table 7.1: Nax features: **deriving** fixpoint, **synonym**, μ, `In`, and **mcata**.

Haskell + GADTs, DataKinds, KindSignatures

```
data Ty = I | B

data Val :: Ty → ⋆ where
  IV :: Int → Val I
  BV :: Bool → Val B

plusV :: Val I → Val I → Val I
plusV (IV n) (IV m) = IV (n + m)

ifV :: Val B → Val t → Val t → Val t
ifV (BV b) v1 v2 = if b then v1 else v2

data Expr :: Ty → ⋆ where
  VAL  :: Val t → Expr t
  PLUS :: Expr I → Expr I → Expr I
  IF   :: Expr B →
          Expr t → Expr t → Expr t

eval :: Expr t → Val t
eval (VAL v)       = v
eval (PLUS e1 e2) =
  plusV (eval e1) (eval e2)
eval (IF e0 e1 e2) =
  ifV (eval e0) (eval e1) (eval e2)
```

Nax

```
data Ty = I | B

data Val : Ty → ⋆ where
  IV : Int → Val {I}
  BV : Bool → Val {B}

-- plusV : Val {I} → Val {I} → Val {I}
plusV (IV n) (IV m) = IV (n + m)

-- ifV : Val {B} → Val {t} → Val {t} → Val {t}
ifV (BV b) v1 v2 = if b then v1 else v2

data E : (Ty → ⋆) → (Ty → ⋆) where
  VAL  : Val {t} → E r {t}
  PLUS : r {I} → r {I} → E r {I}
  IF   : r {B} → r {t} → r {t} → E r {t}
    deriving fixpoint Expr

-- eval : Expr {t} → Val {t}
eval e = mit_{{{t}. Val {t}}} e with
  ev (VAL v)       = v
  ev (PLUS e1 e2) =
    plusV (ev e1) (ev e2)
  ev (IF e0 e1 e2) =
    ifV (ev e0) (ev e1) (ev e2)
```

Figure 7.1: A type-preserving evaluator (*eval*) that evaluates an expression (*Expr*) to a value (*Val*), in Haskell and in Nax.

NAX

```
data Ty = I | B

data Val : Ty → ⋆ where
  IV : Int → Val {I}
  BV : Bool → Val {B}

-- plusV : Val {I} → Val {I} → Val {I}
plusV (IV n) (IV m) = IV (n + m)

-- ifV : Val {B} → Val {t} → Val {t} → Val {t}
ifV (BV b) v1 v2 = if b then v1 else v2

data E : (Ty → ⋆) → (Ty → ⋆) where
  VAL  : Val {t} → E r {t}
  PLUS : r {I} → r {I} → E r {I}
  IF   : r {B} → r {t} → r {t} → E r {t}
    deriving fixpoint Expr

-- eval : Expr {t} → Val {t}
eval e = mit_{{{t}. Val {t}}} e with
           ev (VAL v)      = v
           ev (PLUS e1 e2) =
             plusV (ev e1) (ev e2)
           ev (IF e0 e1 e2) =
             ifV (ev e0) (ev e1) (ev e2)
```

AGDA

```
data Ty : ⋆ where I : Ty
                 B : Ty

data Val : Ty → ⋆ where
  IV : ℕ → Val I
  BV : Bool → Val B

plusV : Val I → Val I → Val I
plusV (IV n) (IV m) = IV (n + m)

ifV : Val B → {t : Ty} →
      Val t → Val t → Val t
ifV (BV b) v1 v2 = if b then v1 else v2

data Expr : Ty → ⋆ where
  VAL  : {t : Ty} → Val t → Expr t
  PLUS : Expr I → Expr I → Expr I
  IF   : Expr B → {t : Ty} →
         Expr t → Expr t → Expr t

eval : {t : Ty} → Expr t → Val t
eval (VAL v)      = v
eval (PLUS e1 e2) =
  plusV (eval e1) (eval e2)
eval (IF e0 e1 e2) =
  ifV (eval e0) (eval e1) (eval e2)
```

Figure 7.2: A type-preserving evaluator (*eval*) that evaluates an expression (*Expr*) to a value (*Val*), in Nax and in Agda.

language; (2) define a datatype Value (the range of object language evaluation) indexed by terms of the type TypeUniverse; (3) define a datatype ObjectLanguage indexed by the same type TypeUniverse; and (4) write the evaluator (from expressions to values) that preserves the term indices representing the type of the object language. Once the evaluator is type checked, we are confident that the evaluator is type-preserving, relying on type preservation of the host-language type system. In Figures 7.1 and 7.2, we provide a concrete example of such a type-preserving evaluator for a very simple expression language (*Expr*).

Our TypeUniverse (*Ty*) for the expression language consists of numbers and booleans, represented by the constants `I` and `B`. We want to evaluate an expression to get a value, which may be either numeric (`IV` n) or boolean (`BV` b). Note that the both the *Expr* and the *Val* datatypes are indexed by constant terms (`I` and `B`) of TypeUniverse (*Ty*). The terms of TypeUniverse are also known as *type representations*.

An expression (*Expr*) is either a value (`VAL` v), a numeric addition (`PLUS` e_1 e_2), or a conditional (`IF` e_0 e_1 e_2). Note that the term indices of *Expr* ensure that expressions are type-correct by construction. For instance, a conditional expression `IF` e_0 e_1 e_2 can only be constructed when e_0 is a boolean expression (i.e., indexed by `B`) and e_1 and e_2 are expressions of the same type (i.e., both indexed by t). Then, we can write an evaluator (*eval*) (from expressions to values) which preserves the index that represents the object language type. The definition of *eval* is fairly straightforward, since our expression language is a very simple one. Note that the functions in Nax do not need type annotations (they appear as comments in gray). In fact, Nax currently does not support any syntax for type annotations on function declarations.

Curly braces in the Nax code above indicate the use of term indices in types. For instance, t appearing in $\{\, t \,\}$ is a term-index variable rather than a type variable.

7.2.2 Generic *Path*s parametrized by a binary relation

In this section, we introduce a generic *Path* datatype.[2] We will instantiate *Path* into three different types of lists: plain lists, length-indexed lists (*List′* and *Vec* in Figures 7.3 and 7.4) and a *Code* type, in order to write a stack-safe compiler (Figures 7.5 and 7.6).

The type constructor *Path* expects three arguments, that is, $Path\ x\ \{i\}\ \{j\} : \star$. The argument $x : \{\iota\} \to \{\iota\} \to \star$ is a binary relation describing legal transitions (i.e., $x\ \{i\}\ \{j\}$ is inhabited if one can legally step from i to j). The arguments $i : \iota$ and $j : \iota$ represent the initial and final vertices of *Path*. A term of type $Path\ x\ \{i\}\ \{j\}$ witnesses a (possibly many step) path from i to j following the legal transition steps given by the relation $x : \{\iota\} \to \{\iota\} \to \star$.

The *Path* datatype provides two ways of constructing witnesses of paths. First, $pNil : Path\ x\ \{i\}\ \{i\}$ witnesses an empty path (or ϵ-transition) from a vertex to itself, which always exists regardless of the choice of x. Second, $pCons{:}x\ \{i\}\ \{j\} \to Path\ x\ \{j\}\ \{k\} \to Path\ x\ \{i\}\ \{k\}$ witnesses a path from i to k, provided that there is a single step transition from i to j and that there exists a path from j to k.

The function $append : Path\ x\ \{i\}\ \{j\} \to Path\ x\ \{j\}\ \{k\} \to Path\ x\ \{i\}\ \{k\}$ witnesses that there exists a path from i to k provided that there exist two paths from i to j and from j to k. Note that the implementation of *append* is exactly the same as the usual append function for plain lists. We instantiate *Path* by providing a specific relation to instantiate the parameter x.

Plain lists ($List'\ a$) are path oblivious. That is, one can always add an element (a) to a list ($List'\ a$) to get a new list ($List'\ a$). We instantiate x to the degenerate relation $(Elem\ a) : Unit \to Unit \to \star$, which is tagged by a value of type a and which witnesses a step with no interesting information. Then, we can define $List'\ a$

[2] There is a Haskell library package for this: `http://hackage.haskell.org/package/thrist`

Haskell + GADTs, DataKinds, PolyKinds

```
data Path x i j where
  PNil  :: Path x i i
  PCons :: x i j → Path x j k
                 → Path x i k

append :: Path x i j → Path x j k
                     → Path x i k
append PNil           ys = ys
append (PCons x xs) ys =
       PCons x (append xs ys)

-- instantiating to a plain regular list
data Elem a i j where
  MkElem :: a → Elem a () ()
type List' a = Path (Elem a) () ()
nil' = PNil :: List' a
cons' :: a → List' a → List' a
cons' = PCons . MkElem
-- instantiating to a length-indexed list
data Nat = Z | S Nat
data Elem_V a i j where
  MkElem_V :: a → Elem_V a (S n) n
type Vec a n = Path (Elem_V a) n Z
vNil = PNil :: Vec a Z
vCons :: a → Vec a n → Vec a (S n)
vCons = PCons . MkElem_V
```

Nax

```
data P : ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆) where
  PNil  : P x r {i} {i}
  PCons : x {i} {j} → r {j} {k} → P x r {i} {k}
    deriving fixpoint Path
-- append : Path {i} {j} → Path {j} {k}
append l =      --      → Path {i} {k}
  mit{{i} {j}. Path x {j} {k}→Path x {i} {k}} l
    with
      app PNil          ys = ys
      app (PCons x xs) ys =
          pCons x (app xs ys)

-- instantiating to a plain regular list
data Unit = U
data Elem : ⋆ → Unit → Unit → ⋆ where
  MkElem : a → Elem a {U} {U}
synonym List' a = Path (Elem a) {U} {U}
nil' = pNil  -- :List' a
-- cons' : a → List' a → List' a
cons' x = pCons (MkElem x)
-- instantiating to a length-indexed list
data Elem_V : ⋆ → Nat → Nat → ⋆ where
  MkElem_V : a → Elem_V a {'succ n} {n}
synonym Vec a {n}
            = Path (Elem_V a) {n} {'zero}
vNil = pNil  -- :Vec a {'zero}
-- vCons : a → Vec a {n} → Vec a {'succ n}
vCons x = pCons (MkElem_V x)
```

Figure 7.3: A generic indexed list (*Path*) parameterized by a binary relation (x) over indices (i, j, k) and its instantiations (*List'*, *Vec*), in Haskell and in Nax.

Nax

```
data P : ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆) →
         ({ι} → {ι} → ⋆)  where
  PNil  : P x r {i} {i}
  PCons : x {i} {j} → r {j} {k} → P x r {i} {k}
    deriving fixpoint Path
-- append : Path {i} {j} → Path {j} {k}
append l =        --        → Path {i} {k}
  mit{{i} {j}. Path x {j} {k}→Path x {i} {k}} l
    with
      app PNil          ys = ys
      app (PCons x xs) ys =
          pCons x (app xs ys)

-- instantiating to a plain regular list
data Unit = U
data Elem : ⋆ → Unit → Unit → ⋆ where
  MkElem : a → Elem a {U} {U}
synonym List' a = Path (Elem a) {U} {U}
nil' = pNil  -- :List' a
-- cons' : a → List' a → List' a
cons' x = pCons (MkElem x)

-- instantiating to a length-indexed list
data Elem_V : ⋆ → Nat → Nat → ⋆ where
  MkElem_V : a → Elem_V a {'succ n} {n}
synonym Vec a {n}
            = Path (Elem_V a) {n} {'zero}
vNil = pNil  -- :Vec a {'zero}
-- vCons : a → Vec a {n} → Vec a {'succ n}
vCons x = pCons (MkElem_V x)
```

Agda

```
data Path {I : ⋆} (X : I → I → ⋆)
                  : I → I → ⋆
  where
    PNil  : {i : I} → Path X i i
    PCons : {i j k : I} → X i j →
            Path X j k → Path X i k
append : {I : ⋆} →
  {X : I → I → ⋆} →
  {i j k : I} →
  Path X i j → Path X j k
             → Path X i k
append PNil          ys = ys
append (PCons x xs) ys =
       PCons x (append xs ys)

-- instantiating to a plain regular list
record Unit : ⋆ where constructor ⟨⟩
List' : ⋆ → ⋆
List' a = Path (λ i j → a) ⟨⟩ ⟨⟩
nil' : {a : ⋆} → List' a
nil' = PNil
cons' : {a : ⋆} → a → List' a → List' a
cons' = PCons

-- instantiating to a length-indexed list
Vec : ⋆ → ℕ → ⋆
Vec a n = Path (λ i j → a) n zero
vNil : {a : ⋆} → Vec a zero
vNil = PNil
vCons : {a : ⋆} {n : ℕ} →
        a → Vec a n → Vec a (suc n)
vCons = PCons
```

Figure 7.4: A generic indexed list (*Path*) parameterized by a binary relation (x, X) over indices (i, j, k) and its instantiations (*List'*, *Vec*), in Nax and in Agda.

as a synonym of $Path\ (Elem\ a)\ \{\,U\,\}\ \{\,U\,\}$, and its constructors nil' and $cons'$.

Length-indexed lists ($Vec\ a\ \{\,n\,\}$) need a natural number index to represent the length of the list. So, we instantiate x to a relation over natural numbers $(Elem_V\ a) : Nat \to Nat \to \star$ tagged by a value of type a witnessing steps of size one. The relation $(Elem_V\ a)$ counts down exactly one step, from $succ\ n$ to n, as described in the type signature of $MkElem_V : a \to Elem\ a\ \{\text{`}succ\ n\}\ \{\,n\,\}$. Then, we define $Vec\ a\ \{\,n\,\}$ as a synonym $Path\ (Elem_V\ a)\ \{\,n\,\}\ \{\text{`}zero\}$, counting down from n to $zero$. In Nax, in a declaration, backquoted identifiers appearing inside index terms enclosed by braces refer to functions or constants in the current scope (e.g., `$zero$ appearing in $Path\ (Elem_V\ a)\ \{\,n\,\}\ \{\text{`}zero\}$ refers to the predefined $zero : Nat$). Names without backquotes (e.g., n and a) are implicitly universally quantified.

For plain lists and vectors, the relations $(Elem\ a)$ and $(Elem_V\ a)$ are parameterized by the type a. That is, the transition step for adding one value to the path is always the same, independent of the value. Note that both $Elem$ and $Elem_V$ have only one data constructor $MkElem$ and $MkElem_V$, respectively, since all "small" steps are the same. In the next subsection, we will instantiate $Path$ with a relation witnessing stack configurations, with multiple constructors, each witnessing different transition steps for different machine instructions.

The Haskell code is similar to the Nax code, except that it uses general recursion and kinds are not explicitly annotated on datatypes.[3] In Agda, there is no need to define wrapper datatypes such as $Elem$ and $Elem_V$ since type-level functions are no different from term-level functions.

HASKELL + GADTs, DataKinds, PolyKinds

```
data List a = Nil | a :. List a ;  infixr :.
data Inst :: List Ty → List Ty → ⋆ where
  PUSH  :: Val t → Inst ts (t :. ts)
  ADD   :: Inst (I :. I :. ts) (I :. ts)
  IFPOP :: Path Inst ts ts' →
           Path Inst ts ts' →
           Inst (B :. ts) ts'

type Code sc sc' = Path Inst sc sc'

compile :: Expr t → Code ts (t :. ts)
compile (VAL v) =
  PCons (PUSH v) PNil
compile (PLUS e1 e2) =
  append (append (compile e1) (compile e2))
         (PCons ADD PNil)
compile (IF e0 e1 e2) =
  append (compile e0)
         (PCons (IFPOP (compile e1)
                       (compile e2))
                PNil)
```

NAX

```
data Instr : (List Ty → List Ty → ⋆) →
             (List Ty → List Ty → ⋆) where
  PUSH  : Val {t} → Instr r {ts} {`cons t ts}
  ADD   : Instr r {`cons I (`cons I ts)} {`cons I ts}
  IFPOP : Path r {ts} {ts'} →
          Path r {ts} {ts'} →
          Instr r {`cons B ts} {ts'}
    deriving fixpoint Inst

synonym Code {sc} {sc'} = Path Inst {sc} {sc'}
  --  Path (μ[List Ty→List Ty→⋆] Instr) {sc} {sc'}

compile e =
  mit{{t}. Code {ts} {`cons t ts}} e with
    cmpl (VAL v)       =
      pCons (pUSH v) pNil
    cmpl (PLUS e1 e2) =
      append (append (cmpl e1) (cmpl e2))
             (pCons aDD pNil)
    cmpl (IF e0 e1 e2) =
      append (cmpl e0)
             (pCons (iFPOP (cmpl e1)
                           (cmpl e2))
                    pNil)
```

Figure 7.5: A stack-safe compiler, in Haskell and in Nax.

NAX

```
data Instr : (List Ty → List Ty → ⋆) →
             (List Ty → List Ty → ⋆) where
  PUSH  : Val {t} → Instr r {ts} {`cons t ts}
  ADD   : Instr r {`cons I (`cons I ts)} {`cons I ts}
  IFPOP : Path r {ts} {ts'} →
          Path r {ts} {ts'} →
          Instr r {`cons B ts} {ts'}
    deriving fixpoint Inst

synonym Code {sc} {sc'} = Path Inst {sc} {sc'}
  -- Path (μ[List Ty→List Ty→⋆] Instr) {sc} {sc'}

compile e =
  mit{{t}. Code {ts} {`cons t ts}} e with
    cmpl (VAL v)       =
      pCons (pUSH v) pNil
    cmpl (PLUS e1 e2) =
      append (append (cmpl e1) (cmpl e2))
             (pCons aDD pNil)
    cmpl (IF e0 e1 e2) =
      append (cmpl e0)
             (pCons (iFPOP (cmpl e1)
                           (cmpl e2))
                    pNil)
```

AGDA

```
data Inst : List Ty → List Ty → ⋆ where
  PUSH  : {t : Ty} {ts : List Ty} →
          Val t → Inst ts (t :: ts)
  ADD   : {ts : List Ty} →
          Inst (I :: I :: ts) (I :: ts)
  IFPOP : {ts ts' : List Ty} →
          Path Inst ts ts' →
          Path Inst ts ts' →
          Inst (B :: ts) ts'

Code : List Ty → List Ty → ⋆
Code sc sc' = Path Inst sc sc'

compile  : {t : Ty} → {ts : List Ty} →
           Expr t → Code ts (t :: ts)
compile (VAL v)       =
  PCons (PUSH v) PNil
compile (PLUS e1 e2) =
  append (append (compile e1) (compile e2))
         (PCons ADD PNil)
compile (IF e0 e1 e2) =
  append (compile e0)
         (PCons (IFPOP (compile e1)
                       (compile e2))
                PNil)
```

Figure 7.6: A stack-safe compiler, in Nax and in Agda

7.2.3 Stack-safe compiler for the expression language

In Figures 7.5 and 7.6, we implement a stack-safe compiler for the same expression language (*Expr* in Figures 7.1 and 7.2) discussed in Section 7.2.1. In Figures 7.1 and 7.2 of that section, we implemented an index-preserving evaluator $eval : Expr\ \{t\} \to Val\ \{t\}$. Here, the stack-safe compiler $compile : Expr\ \{t\} \to Code\ \{ts\}\ \{`cons\ t\ ts\}$ uses the index to enforce stack safety – an expression of type t compiles to some code, which when run on a stack machine with an initial stack configuration ts terminates with the final stack configuration $cons\ t\ ts$.

A stack configuration is an abstraction of the stack that tracks only the types of the values stored there. We represent a stack configuration as a list of type representations (*List Ty*).[4] For instance, the configuration for the stack containing three values (from top to bottom) [3, True, 4] is *cons* `I` (*cons* `B` (*cons* `I` *Nil*)).

To enforce stack safety, each instruction ($Inst : List\ Ty \to List\ Ty \to \star$) is indexed with its initial and final stack configuration. For example, $aDD : Inst\ \{`cons\ \texttt{I}\ (`cons\ \texttt{I}\ ts)\}\ \{`cons\ \texttt{I}\ ts\}$ instruction expects two numeric values on top of the stack. Running the *aDD* instruction will consume those two values, replacing them with a new numeric value (the result of the addition) on top of the stack leaving the rest of the stack unchanged.

We define *Code* as a *Path* of stack-consistent instructions (i.e., $Code\ \{ts\}\ \{ts'\}$ is a synonym for $Path\ Inst\ \{ts\}\ \{ts'\}$ from Section 7.2.2). For example, the compiled code consisting of the three instructions $inst_1 : Inst\ \{ts_0\}\ \{ts_1\}$, $inst_2 :$

[3] In Haskell, kinds are inferred by default. The `KindSignatures` extension in GHC allows kind annotations.

[4] The astute reader may wonder why we use *List* instead of the already defined *List′* in Figures 7.3 and 7.4, which is exactly the plain list we want. In Nax and Agda, it is possible to have term indices of *List′ Ty* instead of *List Ty*. (In Nax and Agda, the *List* datatype is defined in their standard libraries.) Unfortunately, this is not the case in Haskell. Haskell's datatype promotion does not allow promoting datatypes indexed by the already promoted datatypes. Recall that *List′ Ty* is a synonym of *Path* (*Elem Ty*) () (), which cannot be promoted to an index since it is indexed by the already promoted unit term (). In the following section, we will discuss further on how the two approaches of Nax and Haskell differ in their treatment of term-indexed types.

$Inst\ \{ts_1\}\ \{ts_2\}$, and $inst_3 : Inst\ \{ts_2\}\ \{ts_3\}$ has the type $Code\ \{ts_0\}\ \{ts_3\}$.

7.3 DISCUSSION

Indexed types (e.g., *Val* in Figure 7.1) are classified by kinds (e.g., $Ty \to \star$). What do valid kinds look like? Sorting rules define kind validity (or well-sortedness). Different programming languages that support term indices have made different design choices. In this section, we compare the sorting rules of Nax with the sorting rules of other languages (Section 7.3.1). Then, we compare the class of indexed datatypes supported by Nax with those supported in other languages (Section 7.3.2).

7.3.1 Universes, kinds, and well-sortedness

The concrete syntax for kinds appears similar among Haskell, Nax, and Agda. For instance, in Figure 7.1, the kind $Ty \to \star$ has exactly the same textual representation in all of the three languages. However, each language has its own universe structure, kind syntax, and sorting rules, as summarized in Figure 7.7.

Figure 7.8 illustrates differences and similarities between the mechanism for checking well-sortedness, by comparing the justification for the well-sortedness of the kind $List\ Ty \to \star$. The important lessons of Figure 7.8 are that the Nax approach is closely related to *universe subtyping* in Agda and the datatype promotion in Haskell is closely related to *universe polymorphism* in Agda.

In Nax, we may form a kind arrow $\{A\} \to \kappa$ whenever A is a type (i.e., $\vdash_{\mathsf{ty}} A : \star$). Note that types may only appear in the domain (the left-hand side of the arrow) but not in the codomain (the right-hand side of the arrow). Modulo right associativity of arrows (i.e., $\kappa_1 \to \kappa_2 \to \kappa_3$ means $\kappa_1 \to (\kappa_2 \to \kappa_3)$), kinds in Nax always terminate in $\star$. For example,[5] $\star \to \star \to \star$, $\{Nat\} \to \{Nat\} \to \star$, and

[5] The Nax implementation allows programmers to omit curly braces in kinds when the domain

Haskell + DataKinds	Nax	Agda
$\star : \Box$	$\star : \Box$	$\star_0 : \star_1 : \star_2 : \star_3 : \cdots$ $\Vert \quad \Vert$ $\star \quad \Box$
$\kappa ::= \star \mid \kappa \to \kappa \mid T\,\overline{\kappa}$	$\kappa ::= \star \mid \kappa \to \kappa \mid \{A\} \to \kappa$	term/type/kind/sort merged into one pseudo-term syntax
$(\to)\dfrac{\vdash_{\mathsf{k}} \kappa_1 : \Box \qquad \vdash_{\mathsf{k}} \kappa_2 : \Box}{\vdash_{\mathsf{k}} \kappa_1 \to \kappa_2 : \Box}$	$(\to)\dfrac{\vdash_{\mathsf{k}} \kappa_1 : \Box \qquad \vdash_{\mathsf{k}} \kappa_2 : \Box}{\vdash_{\mathsf{k}} \kappa_1 \to \kappa_2 : \Box}$	$(\to)\dfrac{\vdash \kappa_1 : \star_i \qquad \vdash \kappa_2 : \star_i}{\vdash \kappa_1 \to \kappa_2 : \star_i}$
$(\uparrow^{\Box}_{\star})\dfrac{\vdash_{\mathsf{ty}} T : \star^n \to \star \quad \vdash_{\mathsf{k}} \kappa : \Box \text{ for each } \kappa \in \overline{\kappa}}{\vdash_{\mathsf{k}} T\,\overline{\kappa} : \Box}$	$(\{\}\to)\dfrac{\vdash_{\mathsf{ty}} A : \star \qquad \vdash_{\mathsf{k}} \kappa : \Box}{\vdash_{\mathsf{k}} \{A\} \to \kappa : \Box}$	$(\leqslant)\dfrac{\vdash \kappa : s \qquad s \leqslant s'}{\vdash \kappa : s'}$

Figure 7.7: Universes, kind syntax, and selected sorting rules of Haskell, Nax, and Agda. Haskell's and Nax's kind syntax are simplified to exclude kind polymorphism. Agda's $(\to)$ rule is simplified to only allow non-dependent kind arrows.

$(\{Nat\} \to \star) \to \{Nat\} \to \star$ are valid kinds in Nax. The sorting rule $(\{\}\to)$ could be understood as a specific use of universe subtyping $(\star \leqslant \Box)$ hard-wired within the arrow formation rule. Agda needs a more general notion of universe subtyping, since it is a dependently-typed language with stratified universes, which we will shortly explain.

Agda has countably many stratified type universes for several good reasons. When we form a kind arrow $\kappa_1 \to \kappa_2$ in Agda, the domain κ_1 and the codomain κ_2 must be the same universe (or sort), as specified by the $(\to)$ rule in Figure 7.7, and the arrow kind also lies in the same universe. However, requiring κ_1, κ_2, and $\kappa_1 \to \kappa_2$ to be in exactly the same universe can cause a lot of code duplication. For example, $List\ Ty \to \star_0$ cannot be justified by the $(\to)$ rule since $\vdash List\ Ty : \star_0$ while $\vdash \star_0 : \star_1$. To work around the universe difference, one could define the datatypes $List'$ and Ty', which are isomorphic to $List$ and Ty, only at one higher

of an arrow kind obviously looks like a type. For instance, $Nat \to \star$ is considered as $\{Nat\} \to \star$ since Nat is obviously a type because it starts with an uppercase. In Section 7.2, we omitted curly braces to help readers compare Nax with other languages. From now on, we will consistently put curly braces in kinds.

$$\textsc{Nax}\quad \dfrac{\dfrac{\vdash_{\mathsf{ty}} \mathit{List} : \star \to \star \quad \vdash_{\mathsf{ty}} \mathit{Ty} : \star}{\vdash_{\mathsf{ty}} \mathit{List}\ \mathit{Ty} : \star}\,(\{\}\to) \qquad \vdash_{\mathsf{k}} \star : \Box}{\vdash_{\mathsf{k}} \{\mathit{List}\ \mathit{Ty}\} \to \star : \Box}$$

$$\textsc{Agda}\quad \dfrac{\dfrac{\dfrac{\vdash \mathit{List} : \star \to \star \quad \vdash \mathit{Ty} : \star}{\vdash \mathit{List}\ \mathit{Ty} : \star}\,(\to) \qquad \star \leqslant \Box}{\vdash \mathit{List}\ \mathit{Ty} : \Box}\,(\leqslant) \qquad \vdash \star : \Box}{\vdash \mathit{List}\ \mathit{Ty} \to \star : \Box}\,(\to)$$

$$\textsc{Haskell}\quad \dfrac{\dfrac{\vdash_{\mathsf{ty}} \mathit{List} : \star \to \star \qquad \dfrac{\vdash_{\mathsf{ty}} \mathit{Ty} : \star}{\vdash_{\mathsf{k}} \mathit{Ty} : \Box}\,(\uparrow^{\Box}_{\star})}{\vdash_{\mathsf{k}} \mathit{List}\ \mathit{Ty} : \Box}\,(\to) \qquad \vdash_{\mathsf{k}} \star : \Box}{\vdash_{\mathsf{k}} \mathit{List}\ \mathit{Ty} \to \star : \Box}\,(\uparrow^{\Box}_{\star})$$

$$\textsc{Agda}\ +\ \text{universe polymorphism}\quad \dfrac{\dfrac{\dfrac{\vdash \mathit{List} : \forall\{i\} \to \star_i \to \star_i}{\vdash \mathit{List} : \Box \to \Box} \qquad \dfrac{\vdash \mathit{Ty} : \forall\{i\} \to \star_i}{\vdash \mathit{Ty} : \Box}}{\vdash \mathit{List}\ \mathit{Ty} : \Box}\,(\to) \qquad \vdash \star : \Box}{\vdash \mathit{List}\ \mathit{Ty} \to \star : \Box}\,(\to)$$

Figure 7.8: Justifications for well-sortedness of the kind $\mathit{List}\ \mathit{Ty} \to \star$ in Nax, Haskell, Agda.

level, such that $\vdash \mathit{List}'\ \mathit{Ty}' : \star_1$. Only then, can one construct $\mathit{List}'\ \mathit{Ty}' \to \star_0$. Furthermore, if one needs to form $\mathit{List}\ \mathit{Ty} \to \star_1$, we would need yet another set of duplicate datatypes List'' and Ty'' at yet another higher level. Universe subtyping provides a remedy to such a code duplication problem by allowing objects in a lower universe to be considered as objects in a higher universe. This gives us a notion of subtyping such that $\star_i \leqslant \star_j$ where $i \leqslant j$.[6] With universe subtyping, we can form arrows from Ty to any level of universe (e.g., $\mathit{List}\ \mathit{Ty} \to \star_0$, $\mathit{List}\ \mathit{Ty} \to \star_1$, ...). Relating Agda's universes to sorts in Haskell and Nax, $\star_0$ and $\star_1$ correspond to $\star$ and $\Box$. So, we write $\star$ and $\Box$ instead of $\star_0$ and $\star_1$ in the justification of well-formedness of $\mathit{List}\ \mathit{Ty} \to \star$ in Agda, to make the comparisons align in Figure 7.8.

[6] See Ulf Norell's thesis [71] (Section 1.4) for the full description on universe subtyping.

In addition to universe subtyping, Agda also supports universe polymorphism,[7] which is closely related to datatype promotion. In fact, it is more intuitive to understand the datatype promotion in Haskell as a special case of universe polymorphism. Since there are only two universes $\star$ and $\Box$ in Haskell, we can think of datatypes such as *List* and *Ty* being defined polymorphically at both $\star$ and $\Box$. That is, $List : \Box \to \Box$ as well as $List : \star \to \star$, and similarly, $Ty : \Box$ as well as $Ty : \star$. So, $List : \Box \to \Box$ can be applied to $Ty : \Box$ at the kind-level, just as $List : \star \to \star$ can be applied at the type-level.

In summary, Nax provides a new way of forming kind arrows by allowing types that are already fully applied at the type-level as the domain of an arrow. On the contrary, Haskell first promotes type constructors (e.g., *List*) and their argument types (e.g., *Ty*) to the kind-level, and everything else (application of *List* to *Ty* and kind arrow formation) happens at the kind-level.

7.3.2 Nested Term Indices and Datatypes Containing Types

Nax supports nested term indices, while Haskell's datatype promotion cannot. The examples in Section 7.2 only used rather simple indexed datatypes, whose term indices are of non-indexed types (e.g., *Nat*, *List Ty*). One can imagine more complex indexed datatypes, where some term indices are themselves of term-indexed datatypes. Such nested term indices are often useful in dependently-typed programming. For instance, Brady and Hammond [16] used an environment datatype with nested term indices in their EDSL implementation for verified resource usage protocols. Figure 7.9 illustrates transcriptions of their environment datatype (*Env*), originally written in Idris [15], into Nax and Agda. The datatype *Env* is indexed by a length indexed list (*Vec*), which is again indexed by a natural number (n). Note that the nested term index n appears inside the curly braces nested

[7] See `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.UniversePolymorphism`.

Nax

```
-- Environments of stateful resources index by length-indexed lists
data V : ⋆ → (Nat → ⋆) → Nat → ⋆ where
  VNil : V a r {`zero}
  VCons : a → r {n} → V a r {`succ n}
    deriving fixpoint Vec

data Envr : (({st} → ⋆) → {Vec st {n}} → ⋆)
          → (({st} → ⋆) → {Vec st {n}} → ⋆) where
  Empty : Envr r res {`vNil}
  Extend : res {x} → r res {xs} → Envr r res {`vCons x xs}
    deriving fixpoint Env

-- Usage example: resource (Res) indexed by its state (St)
data St = Read | Write

data Res : St → ⋆ where File1 : Res {Read}
                        File2 : Res {Write}

-- myenv : Env Res {`vCons Read (`vCons Write `vNil)}
myenv = extend File1 (extend File2 empty)

-- Environments additionaly indexed by singleton natural numbers
data SN : (Nat → ⋆) → (Nat → ⋆) where Szer : SN r {`zero}
                                      Ssuc : r {n} → SN r {`succ n}
                                        deriving fixpoint SNat

data Envr' : (({st} → ⋆) → {SNat {n}} → {Vec st {n}} → ⋆)
           → (({st} → ⋆) → {SNat {n}} → {Vec st {n}} → ⋆) where
  Empty' : Envr' r res {`szer} {`vNil}
  Extend' : res {x} → r res {n} {xs} → Envr' r res {`ssuc n} {`vCons x xs}
    deriving fixpoint Env'

-- myenv' : Env' Res {`ssuc (`ssuc`szer)} {`vCons Read (`vCons Write `vNil)}
myenv' = extend' File1 (extend' File2 empty')
```

Agda

```
data Vec (a : ⋆) : ℕ → ⋆ where VNil  : {n : ℕ} → Vec a n
                               VCons : {n : ℕ} → a → Vec a n → Vec a (suc n)

data Env {st} (res : st → ⋆) : {n : ℕ} → Vec st n → ⋆ where
  Empty : Env res {0} VNil
  Extend : {n : ℕ} {x : st} {xs : Vec st n} →
           res x → Env res xs → Env res {suc n} (VCons x xs)
```

Figure 7.9: Environments of stateful resources indexed by the length-indexed list of states.

HASKELL + GADTs, DataKinds, PolyKinds

```
data List a = Nil | a :. List a ;  infixr :.
data HList :: List * -> * where
  HNil :: HList Nil
  HCons :: t -> HList ts -> HList (t :. ts)
hlist :: HList (Int :. Bool :. List Int :. Nil)
hlist = HCons 3 (HCons True (HCons (1 :. 2 :. Nil) HNil))
```

Figure 7.10: Heterogeneous lists (*HList*) indexed by the list of element types (*List* $\star$).

twice ($\{$ *Vec st* $\{n\}\}$). There is no Haskell transcription for *Env* because datatype promotion is limited to datatypes without term indices.

On the contrary, Haskell supports promoted datatypes that hold types as elements, although limited to types without term indices, while Nax does not. The heterogeneous list datatype (*HList*) in Figure 7.10 is a well-known example[8] that uses datatypes containing types. Note that *HList* is indexed by *List* $\star$, which is a promoted list whose elements are of kind $\star$, that is, the element are types. For instance, *hlist* in Figure 7.10 contains three elements $3 : Int$, $True : Bool$, and $(1 :. 2 :. Nil) : List\ Int$, and its type is $HList\ (Int :. Bool :. List\ Int :. Nil)$.

7.4 RELATED WORK

Singleton types, first coined by Hayashi [45], have been used in lightweight verification to simulate dependent types [53, 97]. Sheard, Hook, and Linger [86] demonstrated that singleton types can be defined just like any other datatype in Omega [83], a language equipped with GADTs and a rich kind structure. Nax's universe and kind structure is much simpler than Omega's (e.g., no user-defined

[8]The *HList* library in Haskell by Kiselyov, Lämmel, and Schupke [54] was originally introduced using type class constraints, rather than using GADTs and other relatively new extensions.

kinds in Nax), yet singleton types are definable with fewer worries about code duplication across different universes. Singleton types are typically indexed by the values of their non-singleton counterparts. For example, in Figure 7.9, singleton natural numbers (*SNat*) are indexed by natural numbers (*Nat*). Note that we can index datatypes by singleton types in Nax, while datatype promotion cannot (recall Section 7.3.2). For instance, *Env′* indexed by *SNat* in Figure 7.9 can better simulate the dependently-typed version than *Env*, since *Env′* has a direct handle on size of the environment at the type-level, just by referring to the *SNat* index, without extra type-level computation on the *Vec* index.

Eisenberg and Weirich [31], in the setting of Haskell's datatype promotion, automatically derived a singleton type (e.g., singleton natural numbers) and its associated functions (e.g., addition over singleton natural numbers) from their non-singleton counterparts (e.g., natural numbers and their addition). We think it would be possible to apply similar strategies to Nax, and even better, singleton types for already indexed datatypes would be derivable.

The kind arrow $(\{A\} \to \kappa)$, from a type to a kind, predates Nax. Our kind syntax in Figure 7.7, although developed independently, happens to coincide with the kind syntax of Deputy [24], a dependently-typed system for low-level imperative languages with variable mutation and a heap allocated structure.

Curly braces in Nax are different from those in Agda or SHE.

In Nax, curly braces mean that the things inside them are *erasable* (i.e., must still type-correct without all the curly braces). Agda's curly braces mean that the things in them would often be *inferable* so that programmers may omit them.

The concrete syntax for kinds in SHE[9] appears almost identical to Nax's concrete kind syntax, even using curly braces around types. However, SHE's (abstract) kind syntax is virtually identical to the (abstract) kind syntax of datatype promotion, thus quite different from Nax, since $\{A\} :: \Box$ in SHE.

Kind polymorphism in Nax may be polymorphic over term-index variables ($i : A$) and type variables ($\alpha : \star$), as well as over kind variables ($\mathcal{X} : \Box$). That is, polymorphic kinds (or kind schemes) in Nax may be kind polymorphic ($\forall \mathcal{X} \,.\, \kappa$), type polymorphic ($\forall \alpha . \kappa$), term-index polymorphic ($\forall i . \kappa$), or combinations of them ($\forall \mathcal{X}\ \alpha\ i \,.\, \kappa$). For example, the kinds of P and *Path* in Figure 7.3 are polymorphic over the type variable $\iota : \star$. In contrast, datatype promotion in Haskell only needs to consider polymorphic kinds ($\forall \mathcal{X} \,.\, \kappa$) quantified over kind variables ($\mathcal{X} : \Box$) since everything is already promoted to the kind-level.

In Nax, kind polymorphism is limited to rank-1 since it is well known that higher-rank kind polymorphism leads to a paradox [51]. In fact, type polymorphism in Nax is limited to rank-1 as well since type inference is based on Hindley-Milner [66].

Concoqtion [33] is an extension of MetaOCaml with indexed types. Concoqtion shares some similar design principles — Hindley–Milner-style type inference and *gradual typing by erasure* over (term) indices. Both in Nax and in Concoqtion, a program using indexed types must still type check within the non-indexed sub-language (OCaml for Concoqtion) when all indices are erased from the program. However, indices in Concoqtion differ from the term indices discussed in this chapter (Nax, datatype promotion, and dependently-typed languages like Agda). Concoqtion indices are Coq terms rather than OCaml terms. Although this obviously leads to code duplication between the index world (Coq) and the program

[9] `http://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`

world (OCaml), Concoqtion enjoys practical benefits of having access to the Coq libraries for reasoning about indices. Comparison of Concoqtion and other related systems can be found in the technical report by Pasalic, Siek, and Taha [73].

7.5 SUMMARY AND FUTURE WORK

In Nax, programmers can enforce program invariants using indexed types, without excessive annotations (like functional programming languages) while enjoying logical consistency (like dependently-typed proof assistants).

There are two approaches that allow term indices without code duplication at every universe. *Universe subtyping* is independent of the number of universes. Even scaled down to two universes ($\star, \Box$), it adds no additional restrictions – term indices can appear at arbitrary depth. *Universe polymorphism* is sensitive to the number of universes. Unless there are countably infinite universes, nested term indices are restricted to depth $n - 1$ where n is the number of universes.

On the other hand, universe polymorphism can reuse datatypes at the term-level (*List a* where $a : \star$) at the type-level to contain type elements (e.g., *List* $\star$), which is beyond universe subtyping. We envision that Nax extended with first-class datatype descriptions [27] would be able express the same concept reflected at the term level, so that we would have no need for type-level datatypes.

Chapter 8

TYPE INFERENCE IN NAX

Type inference for a language that supports indexed datatypes is known to be difficult. In this chapter, we illustrate the key idea that enables a conservative extension of Hindley-Milner type inference (HM). We will not be as formal and detailed on proofs as we did for HM in Section 2.4. We extrapolate from the properties of HM that the same property (soundness of type inference) should hold for a subset of Nax, which is structurally similar to HM. Then, we will argue that some key new features in Nax, which are not present in HM preserve those properties.

Index transformers, which are type annotations on pattern matching constructs, play a key role in inferring types for Nax programs involving indexed datatypes. We introduce a subset of Nax, SmallNax, only considering non-recursive datatypes defined by equational declarations, but omitting other details of Nax (Section 8.1). Next, we extend SmallNax with recursive types and Mendler-style iteration, describe their kinding and typing rules, and discuss the role of index transformers for type inference (Section 8.2). Then, we discuss how we treat other Nax features such as GADT-style definitions and term indices in our implementation (Section 8.3).

8.1 SMALLNAX

The syntax of SmallNax is illustrated in Definition 8.1.1, its kinding and typing rules are illustrated in Figure 8.1.

Definition 8.1.1 (Syntax of SmallNax)**.**

Term	t, s	$::=$	$x \mid \lambda x.t \mid t\ s \mid \mathbf{let}\ x = s\ \mathbf{in}\ t \mid C \mid \varphi^{\psi}$
Type constructor	F, G, A, B	$::=$	$X \mid A \to B \mid T \mid F\ G$
Type scheme	σ	$::=$	$\forall X^{\kappa}.\sigma \mid A$

Definition 8.1.2 (Type scheme ordering (or, generic instantiation))**.** $\boxed{\sigma \sqsubseteq_{\Delta} \sigma'}$

$$\textsc{GInst}\ \frac{X'_1, \ldots, X'_m \notin \mathrm{FV}(\forall X_1^{\kappa_1} \ldots X_n^{\kappa_n}.\sigma) \qquad \Delta \vdash \forall X_1^{\kappa_1} \ldots X_n^{\kappa_n}.\sigma : * \qquad \Delta \vdash \forall X_1^{\prime\kappa'_1} \ldots X_m^{\prime\kappa'_m}.\, A[F_1/X_1] \cdots [F_n/X_n] : *}{\forall X_1^{\kappa_1} \ldots X_n^{\kappa_n}.\sigma \ \sqsubseteq_{\Delta}\ \forall X_1^{\prime\kappa'_1} \ldots X_m^{\prime\kappa'_m}.\, A[F_1/X_1] \cdots [F_n/X_n]}$$

The syntax of SmallNax is similar to the syntax of HM in Section 2.4. SmallNax has data constructors (C) and case functions (φ^{ψ}) in addition to the terms of HM. A case function φ^{ψ} is a list of alternatives ($\varphi ::= \overline{C\overline{x} \to t}$) annotated with an index transformer ψ.[1] The case expression $\mathbf{case}_{\psi}\ s\ \mathbf{of}\ \varphi$ in Nax corresponds to $\varphi^{\psi}\ s$, an application of the case function (φ^{ψ}) to the scrutinee (s). Considering case expressions as applications simplifies the typing rules because we do not need a separate typing rule for case expressions. In addition to the types of HM, the type constructor syntax in SmallNax includes type constructor names (T) and type constructor applications ($F\ G$). The type schemes in SmallNax ($\forall X^{\kappa}.\sigma$) is similar to the type schemes ($\forall X.\sigma$) in HM, but the universally quantified type variable (X) is annotated with its kind (κ).

We assume that type constructor names and their associated data constructors are introduced into the context by preprocessing non-recursive equational datatype definitions. For example, **data** `Maybe` a = `Just` a | `Nil` introduces a type constructor name `Maybe` and its associated data constructors `Just` and `Nil` into the context (Δ and Γ in Figure 8.1). That is, $\texttt{Maybe}^{*\to *} \in \Delta$ and

[1] Our Nax implementation supports nested patterns (e.g., $(C_1\, x_1\, (C_2\, x_2)\, x_3)$, but SmallNax only allows simple patterns (i.e., data constructor followed by variables) in alternatives.

$\texttt{Just} : \forall X_a^*.X_a \to \texttt{Maybe}\, X_a,\ \texttt{Nil} : \forall X_a^*.\texttt{Maybe}\, X_a \in \Gamma$. Data constructors introduced from an equational datatype definition have uniform return types ($T\ \overline{X}$) and no existential variables in their types. For instance, return types of both `Just` and `Nil` have the form of $\texttt{Maybe}\, X_a$. For such non-recursive equational datatype definitions, index transformer annotations are not needed. So, we either omit the annotation on the case function (φ) or write a dot ($\varphi^{\cdot}$). We will need index transformers to infer types involving recursive datatypes (Section 8.2) and GADTs (Section 8.3).

Declarative typing rules and syntax-directed typing rules. The typing rules of SmallNax (Figure 8.1), excluding the rules for datatypes (Con, Case, Alt in the declarative rules and their corresponding syntax-directed rules), are structurally similar to the typing rules of HM (Figure 2.9). Each of those typing rules in SmallNax has its corresponding typing rule with the same name in HM. The differences from HM are the existence of kinding rules to ensure well-kindedness of type constructors (which can have kinds other than $*$) and the additional context Δ in the typing rules to keep track of whether type constructor variables are in scope with correctly assigned kinds. The generic instantiation rule (GInst) also takes Δ into consideration so that both sides of $\sqsubseteq$ are well-kinded. We can view HM as a restriction of SmallNax (excluding the features for datatypes) where kinds are always $*$. So, we know that the syntax-directed typing rules (excluding Con_s, Case_s, Alt_s) are sound (Theorem 8.1.1) and complete (Theorem 8.1.2) with respect to the declarative typing rules (excluding Con, Case, Alt) in SmallNax.

Theorem 8.1.1 ($\vdash_{\mathsf{s}}$ is sound with respect to $\vdash$)**.** $\dfrac{\Delta; \Gamma \vdash_{\mathsf{s}} t : A}{\Delta; \Gamma \vdash t : A}$

Theorem 8.1.2 ($\vdash_{\mathsf{s}}$ is complete with respect to $\vdash$)**.**

$$\frac{\Delta; \Gamma \vdash t : \sigma}{\exists A.\ \Delta; \Gamma \vdash_{\mathsf{s}} t : A\ \wedge\ \overline{\Delta; \Gamma}(A) \sqsubseteq_{\Delta} \sigma}$$

Kinding rules $\boxed{\Delta \vdash F : \kappa}$

$$\textsc{TVar}\ \frac{X^\kappa \in \Delta}{\Delta \vdash X : \kappa} \qquad \textsc{TArr}\ \frac{\Delta \vdash A : * \quad \Delta \vdash B : *}{\Delta \vdash A \to B : *}$$

$$\textsc{TCon}\ \frac{T^\kappa \in \Delta}{\Delta \vdash T : \kappa} \qquad \textsc{TApp}\ \frac{\Delta \vdash F : \kappa \to \kappa' \quad \Delta \vdash G : \kappa}{\Delta \vdash F\ G : \kappa'}$$

Declarative typing rules $\boxed{\Delta;\Gamma \vdash t : \sigma}$

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Delta;\Gamma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Delta \vdash A : * \quad \Delta;\Gamma, x : A \vdash t : B}{\Delta;\Gamma \vdash \lambda x.t : A \to B}$$

$$\textsc{App}\ \frac{\Delta;\Gamma \vdash t : A \to B \quad \Delta;\Gamma \vdash s : A}{\Delta;\Gamma \vdash t\ s : B} \qquad \textsc{Let}\ \frac{\Delta;\Gamma \vdash s : \sigma \quad \Delta;\Gamma, x : \sigma \vdash t : B}{\Delta;\Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : B}$$

$$\textsc{Inst}\ \frac{\Delta;\Gamma \vdash t : \sigma \quad \sigma \sqsubseteq_\Delta \sigma'}{\Delta;\Gamma \vdash t : \sigma'} \qquad \textsc{Gen}\ \frac{\Delta, X^\kappa;\Gamma \vdash t : \sigma}{\Delta;\Gamma \vdash t : \forall X^\kappa.\sigma}\ (X \notin \mathrm{FV}(\Gamma))$$

$$\textsc{Con}\ \frac{C : \sigma \in \Gamma}{\Delta;\Gamma \vdash C : \sigma} \qquad \textsc{Case}\ \frac{\overline{\Delta;\Gamma \vdash^\psi C\overline{x} \to t : \sigma}}{\Delta;\Gamma \vdash (\overline{C\overline{x} \to t})^\psi : \sigma}$$

$\boxed{\Delta;\Gamma \vdash^\psi C\overline{x} \to t : \sigma}$

$$\textsc{Alt}\ \frac{\Delta;\Gamma \vdash C : \overline{A} \to T\overline{B}\,\overline{A'} \quad \Delta;\Gamma, \overline{x : A} \vdash t : \psi(\overline{A'})}{\Delta;\Gamma \vdash^\psi C\overline{x} \to t : \forall\overline{X^\kappa}.T\overline{B}\,\overline{X} \to \psi(\overline{X})}$$

Syntax-directed typing rules $\boxed{\Delta;\Gamma \vdash^{\mathsf{s}} t : A}$

$$\textsc{Var}_s\ \frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq_\Delta A}{\Delta;\Gamma \vdash^{\mathsf{s}} x : A} \qquad \textsc{Abs}_s\ \frac{\Delta \vdash A : * \quad \Delta;\Gamma, x : A \vdash^{\mathsf{s}} t : B}{\Delta;\Gamma \vdash^{\mathsf{s}} \lambda x.t : A \to B}$$

$$\textsc{App}_s\ \frac{\Gamma \vdash^{\mathsf{s}} t : A \to B \quad \Gamma \vdash^{\mathsf{s}} s : A}{\Gamma \vdash^{\mathsf{s}} t\ s : B} \qquad \textsc{Let}_s\ \frac{\Delta;\Gamma \vdash^{\mathsf{s}} s : A \quad \Delta;\Gamma, x : \overline{\Delta;\Gamma}(A) \vdash^{\mathsf{s}} t : B}{\Gamma \vdash^{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : B}$$

$\overline{\Delta;\Gamma}(A) = \forall\vec{X}.A$ where $\vec{X} = \mathrm{FV}(A) \backslash \mathrm{dom}(\Delta) \backslash \mathrm{FV}(\Gamma)$

$$\textsc{Con}_s\ \frac{C : \sigma \in \Gamma \quad \sigma \sqsubseteq_\Delta A}{\Delta;\Gamma \vdash^{\mathsf{s}} C : A} \qquad \textsc{Case}_s\ \frac{\overline{\Delta;\Gamma \vdash^{\mathsf{s}\psi} C\overline{x} \to t : \sigma} \quad \sigma \sqsubseteq_\Delta A}{\Delta;\Gamma \vdash (\overline{C\overline{x} \to t})^\psi : A}$$

$\boxed{\Delta;\Gamma \vdash^{\mathsf{s}\psi} C\overline{x} \to t : \sigma}$

$$\textsc{Alt}_s\ \frac{\Delta;\Gamma \vdash^{\mathsf{s}} C : \overline{A} \to T\overline{B}\,\overline{A'} \quad \Delta;\Gamma, \overline{x : A} \vdash^{\mathsf{s}} t : \psi(\overline{A'})}{\Delta;\Gamma \vdash^{\mathsf{s}\psi} C\overline{x} \to t : \forall\overline{X^\kappa}.T\overline{B}\,\overline{X} \to \psi(\overline{X})}$$

Figure 8.1: Kinding and typing rules of SmallNax

We have argued that Theorem 8.1.1 and Theorem 8.1.2 hold for all the typing rules in SmallNax excluding the rules for datatypes. So, we only need to check whether these two theorems hold for the rules for datatypes, that is, for the declarative rules Con, Case, and Alt, and their syntax-directed counterparts $\textsc{Con}_s$, $\textsc{Case}_s$, and $\textsc{Alt}_s$. They obviously hold for the rules Con and $\textsc{Con}_s$ because these rules have exactly the same structure as Var and $\textsc{Var}_s$. Once we know that Alt is sound and complete with respect to $\textsc{Alt}_s$, it is quite straightforward to show that Case is sound and complete with respect to $\textsc{Case}_s$ because the typing one can get from $\textsc{Case}_s$ is a generic instantiation of the typing one can get from Case. It is indeed the case that Alt is sound and complete with respect to $\textsc{Alt}_s$ because they have exactly the same structure. Unlike other syntax-directed typing rules of the form $\Delta; \Gamma \vdash^{\mathsf{s}} t : A$, which assign a type to a monomorphic type (A), the rule $\textsc{Alt}_s$ of the form $\Delta; \Gamma \vdash^{\mathsf{s}\psi} C\overline{x} \to t : \sigma$ assigns a polymorphic type scheme (σ). Since Alt and $\textsc{Alt}_s$ have exactly the same structure, one calling on $\vdash$ and the other calling on $\vdash^{\mathsf{s}}$ in their premises, they must be sound and complete with respect to each other.

SmallNax is strongly normalizing and logically consistent. The type system of SmallNax is sound with respect to System F_ω. That is, when $\Delta; \Gamma \vdash t : \sigma$ in SmallNax, then $\Delta; \Gamma \vdash t : \sigma$ in System F_ω. Considering the let-term ($\mathbf{let}\ x = s\ \mathbf{in}\ t$) as a syntactic sugar of a lambda term applied to the scrutinee ($(\lambda x.t)\, s$), the terms of SmallNax, except data constructors and case expressions, are exactly the same as the term of System F_ω, which we discussed in Section 2.3. For SmallNax terms involving data constructors and case expressions, we use the Church encoding to translate them into System F_ω terms. We show the soundness of typing with respect to System F_ω by reasoning about the declarative typing rules. Recall that we discussed the soundness of typing for HM with respect to System F by reasoning about the declarative typing rules of HM in Section 2.4.

The kinding and typing rules, except those rules for datatypes, are admissible in System F_ω. The kinding rules except TCon are exactly the same as the kinding rules with the same name in System F_ω (see Figure 2.6 on page 61). The declarative typing rules except Con, Case, and Alt are admissible in System F_ω. The rules Var, Abs, App, and Gen[2] are exactly the same as the typing rules of System F_ω. We can show that the rules Let and Inst in SmallNax are admissible in System F_ω by following virtually the same argument that we used to show that the rules Let and Inst in HM are admissible in System F (see p.82 in Section 2.4). The Let rule in SmallNax corresponds to a consecutive use of App and Abs in System F_ω. A single derivation step of Inst in SmallNax corresponds to multiple uses of the rules TyAbs and TyApp in System F_ω.

The kinding and typing rules involving datatypes (TCon, Con, Case, and Alt) can be understood as being admissible in System F_ω via the Church encodings of datatypes. In Section 2.2.1 and Section 2.3.1, we discussed how non-recursive datatypes (e.g., unit, void, boolean, sums, products) can be encoded as functions. The rules TCon, Con, Case, and Alt are compatible with those encodings. Thus, the type system of SmallNax, described in Figure 8.1, is sound with respect to System F_ω. Therefore, SmallNax is strongly normalizing and logically consistent.

From System F_ω to SmallNax. We will discuss informal and high-level design concepts of what restrictions from System F_ω make SmallNax feasible for type inference, rather than formally discussing concrete type inference algorithms. There are two restrictions from System F_ω, rank-1 polymorphism and type constructor names, which make type inference decidable in SmallNax. In addition, we discuss the role of index transformers in type inference. Although index transformers are not essential for pattern matching of datatypes defined by non-recursive equations,

[2] The Gen rule in SmallNax corresponds to the TyAbs rule in System F_ω(Figure 2.6). The other rules, Var, Abs, App corresponds to the rules with the same name in System F_ω.

they do play essential roles in inferring types of recursive datatype definitions and GADT-style datatype definitions.

Let us review what restriction from System F makes HM (without recursion) feasible for type inference. Type inference is undecidable in System F due to its arbitrary rank polymorphism (i.e., polymorphic types can appear arbitrary deep inside type constructor arguments, in particular, inside the left-hand side of $\rightarrow$). Type inference becomes decidable in HM by restricting the polymorphism to be rank-1 (i.e., universal quantification can only appear at the top level). Similarly, we restrict the polymorphism to be rank-1 in SmallNax (see Definition 8.1.1).

In addition to arbitrary rank polymorphism, type abstractions ($\lambda X^{\kappa}.F$) in System F_{ω} are another feature that makes type inference undecidable. Type inference algorithms involving type abstractions would require higher-order unification (i.e., unification involving reconstruction of function implementation), which is known to be undecidable [42]. In SmallNax, we can avoid higher-order unification because there are no type abstractions. Datatypes in SmallNax are introduced into the context as primitives, that is, type constructor names into Δ and their associated data constructors into Γ. So, we only need first-order unification for inferring types in SmallNax.

8.2 SMALLNAX WITH MENDLER-STYLE RECURSION

In this section, we extend SmallNax with Mendler-style recursion combinators. We first review type inference and recursion in Section 8.2.1. Then, we introduce the typing rules for Mendler-style iteration in SmallNax and discuss the role of index transformers in type inference in Section 8.2.2.

8.2.1 A review of monomorphic recursion and polymorphic recursion

The Hindley–Milner type system (HM) [29] supports monomorphic (general) recursion by assigning a monomorphic type (A) to the recursive variable (x), as described in the rule Fix-m below right.[3] The Milner–Mycroft type system (MM) [70] supports polymorphic recursion by assigning a polymorphic type (σ) to the recursive variable (x), as described in the rule Fix-p below left.

$$\textsc{Fix-p}\frac{\Gamma, x : \sigma \vdash t : \sigma}{\Gamma \vdash \mathbf{fix}\ x.t : \sigma} \qquad \textsc{Fix-m}\frac{\Gamma, x : A \vdash t : A}{\Gamma \vdash \mathbf{fix}\ x.t : A}$$

Polymorphic recursion is necessary for writing recursive programs involving nested datatypes. However, type inference for MM is known to be undecidable [46]. That is, we cannot generally decide whether a suitable σ exists for $\mathbf{fix}\ x.t$ in the rule Fix-p.

What makes HM (including Fix-m) particularly suitable for type inference while type inference for MM is undecidable? Henglein [46] summarized the peculiarity of HM is that occurrences of a recursive definition "*inside* the body of its definition can only be used *monomorphically*", whereas occurrences "*outside* its body can be used *polymorphically*".

8.2.2 Typing rules for Mendler-style recursion combinators

We design the typing rules for recursion combinators in SmallNax based on a similar idea to what makes HM suitable for type inference. We first discuss a simplified version (with less polymorphism) of the typing rule for **mit** in Figure 8.2. Then, we illustrate a more polymorphic version, which is actually used in the Nax implementation, in Figure 8.3.

What makes SmallNax, including the mit′ rule in Figure 8.2, suitable for type

[3] In Section 2.4, we excluded the general recursion in our formalization of HM, although its original presentation has general recursion, because our language does not support general recursion.

Syntax

Term	t, s	$::=$	$\ldots \mid \mathbf{mit}_\kappa\ x\ \varphi^\psi$
Type constructor	F, G, A, B	$::=$	$\ldots \mid \mu_\kappa(T\,\overline{G})$

Kinding rules $\quad \textsc{mu}\dfrac{\Delta \vdash T\,\overline{G} : \kappa \to \kappa}{\Delta \vdash \mu_\kappa(T\,\overline{G}) : \kappa}$

Typing rules

$$\textsc{mit}'\frac{\Delta, X_r^\kappa; \Gamma, x : \forall\overline{X^{\kappa'}}.X_r\,\overline{X} \to \psi(\overline{X}) \vdash \varphi^\psi : \forall\overline{X^{\kappa'}}.F\,X_r\,\overline{X} \to \psi(\overline{X})}{\Delta; \Gamma \vdash \mathbf{mit}_\kappa\ x\ \varphi^\psi : \forall\overline{X^{\kappa'}}.\mu_\kappa F\,\overline{X} \to \psi(\overline{X})}$$

Figure 8.2: SmallNax extended with μ_κ and $\mathbf{mit}_\kappa$, using a simplified version of the inference rule for $\mathbf{mit}_\kappa$

inference is that the *type parameters* of the recursive function argument are *monomorphic*, whereas the *type indices* are *polymorphic* inside the body of the recursive function definition. Outside the body, the recursive function can be used polymorphically over both type parameters and type indices.

Figure 8.2 highlights the extended parts of the syntax, kinding rules, and typing rules of SmallNax from Figure 8.1. The term syntax is extended with the Mendler-style iteration combinator. An application of the Mendler-style iteration combinator to a term $(\mathbf{mit}_\kappa\ x\ \overline{C\overline{x} \to t}^\psi)\ s$ in SmallNax corresponds to $\mathbf{mit}_\psi\ s\ \overline{x\ (C\overline{x}) \to t}$ in Nax, where x is the name for the abstract recursive function call used in each case branch t. We relaxed the syntax of the Mendler-style iteration combinator to be used as a first-class function without its recursive argument s in SmallNax: for the same reason we relaxed the syntax of case expressions $(\mathbf{case}_\psi\ s\ \mathbf{of}\ \varphi)$ in Nax into case functions (φ^ψ) in SmallNax so that it could be used without the scrutinee (s) – we do not need a separate rule for the application of the Mendler-style iteration combinator. The kinding rule MU and the typing rule MIT′ are admissible in System F_ω by the embedding of μ_κ and $_\kappa$ into System F_ω as discussed in

Section 4.2.2.

The rule MIT′ is suitable for type inference. Unlike the rule FIX-P in MM, which cannot always determine the type scheme (σ) of the general recursion ($\mathbf{fix}\ x.t$), the rule MIT′ unambiguously determines the type scheme ($\forall\overline{X^{\kappa'}}.\mu_\kappa F\,\overline{X} \to \psi(\overline{X})$) of the Mendler-style iteration ($\mathbf{mit}_\kappa\ x\ \varphi^\psi$) from the index transformer (ψ) – the number and kinds of universally quantified variables ($\forall\overline{X^{\kappa'}}.\cdots$) in the type scheme must match the number and kinds of the arguments to the index transformer. The rule MIT′ uniquely determines the type scheme of the Mendler-style iteration, except F. The type constructor F in the rule MIT′ is determined by the rules CON, CASE, and ALT in Figure 8.1.

We can formulate the syntax-directed counterpart of the rule MIT′ as follows:

$$\text{MIT}'_s\ \frac{\Delta, X_r^\kappa; \Gamma, x : \forall\overline{X^{\kappa'}}.X_r\,\overline{X} \to \psi(\overline{X}) \vdash^{\mathsf{s}} \varphi^\psi : \forall\overline{X^{\kappa'}}.F\,X_r\,\overline{X} \to \psi(\overline{X})}{\Delta; \Gamma \vdash^{\mathsf{s}} \mathbf{mit}_\kappa\ x\ \varphi^\psi : \mu_\kappa F\,\overline{G} \to \psi(\overline{G})}$$

Note that the type $\mu_\kappa F\,\overline{G} \to \psi(\overline{G})$ in MIT′$_s$ is a generic instance of the type scheme $\forall\overline{X^{\kappa'}}.\mu_\kappa F\,\overline{X} \to \psi(\overline{X})$ in MIT′. So, the relation between MIT′ and MIT′$_s$ are similar to the relation between CASE and CASE$_s$. Therefore, the declarative typing rules including MIT′$_s$ are sound and complete with respect to the syntax-directed rules including MIT′$_s$, for similar reasons as to why CASE$_s$ is sound and complete with respect to CASE.

We need additional polymorphism in the typing rule for **mit** when we have free variables in the index transformer ($\psi(\overline{X})$), other than the indices ($\overline{X}$) of the argument type. For example, consider the index transformer $\{\{t\}\,.\,$`Code`$\,\{ts\}\,\{$`‘cons`$\ t\ ts\}\}$ appearing in the definition of the stack-safe compiler (*compile*) in Section 7.2.3. The free variable (*ts*) should be generalized as well as the index (t) to infer the type of the *compile* function. The type scheme for the recursive caller (x in the typing rule for **mit**) should be fully generalized, except for the part of the input type up to its parameters. That is, we should generalize both the indices and the free variables of the index transformer. The idea described in this paragraph is

$$\textsc{mit}\ \frac{\Delta, X_r^{\kappa}; \Gamma, x : \forall \overline{X'^{\kappa'}}.X_r\ \overline{X} \to \psi(\overline{X}) \vdash \varphi^{\psi} : \forall \overline{X'^{\kappa'}}.F\ X_r\ \overline{X} \to \psi(\overline{X})}{\Delta; \Gamma \vdash \mathbf{mit}_{\kappa}\ x\ \varphi^{\psi} : \forall \overline{X'^{\kappa'}}.\mu_{\kappa} F\ \overline{X} \to \psi(\overline{X})}$$

$$\begin{array}{rl} \text{where} & \overline{X'} = \overline{X} \cup \mathrm{FV}(\psi(\overline{X})) \setminus \mathrm{dom}(\Delta) \setminus \mathrm{FV}(\Gamma) \\ \text{and} & \text{each } \kappa' \text{ is an appropriate kind for each } X' \end{array}$$

Figure 8.3: A more polymorphic version of the inference rule for $\mathbf{mit}_{\kappa}$

summarized as the rule MIT in Figure 8.3.

Among many recursion schemes, we discuss only about Mendler-style iteration in this chatper. Nevertheless, our discussions througout this chapter are applicable to the formulation of typing rules for other Mendler-style recursion schemes as well.

8.3 SMALLNAX WITH GADTS

So far in this chapter, we have only considered those datatypes defined by equations.[4] Such equational datatypes are either regular (e.g., homogeneous lists) or nested (e.g., powerlists, bushes). As discussed in Section 8.1, data constructors introduced from an equational definition (i.e., $\mathbf{data}\ T\ \overline{X} = \overline{C\overline{G}}$) have uniform return types ($T\ \overline{X}$) and no existential variables in their types. GADT definitions can introduce a wider range of datatypes, including data constructors with non-uniform return types and data constructors with existential type variables in their types.

8.3.1 Existential type variables

GADT definitions can introduce *existential type variables* in the types of data constructors. Existential type variables are type variables that do not appear in the result type. In fact, we have already seen an example of a GADT definition

[4] A recursive datatype defined using μ_{κ} over non-recursive equational datatypes can also be described by a recursive equation.

that contains existential type variables in earlier chapters. Consider the simply-typed HOAS datatype, which we discussed in Section 3.9.3, defined as a GADT in Haskell:[5]

```
data Exp t where
  Lam :: (Exp a -> Exp b) -> Exp (a -> b)
  App :: Exp a -> Exp (a -> b) -> Exp b
```

Note that the return types of the two data constructors are not uniform. The return type of `Lam` is `Exp (a -> b)` and the return type of `App` is `Exp b`. Also note that the type variable `a` in the type of `App` does not appear in the result type `Exp b`. So, `a` is an existential type variable by definition. When we have an application expression `(App e1 e2) :: Exp b`, we know that there exists some `a` such that `e1 :: a` and `e2 :: a -> b`, but there is no way to statically know what `a` is even when we have more information about `b`. Existential type variables must remain abstract in pattern matching. That is, they can never be instantiated inside alternatives.

To handle existential variables, we adjust the rule Alt in Figure 8.1 as follows:

$$\textsc{Alt}\frac{\begin{array}{c}\Delta, \exists_\Gamma(C); \exists_C(\Gamma) \vdash C : \overline{A} \to T\overline{B}\,\overline{A'} \\ \Delta, \exists_\Gamma(C); \Gamma, \overline{x : A} \vdash t : \psi(\overline{A'})\end{array}}{\Delta; \Gamma \vdash^{\psi} C\overline{x} \to t : \forall\overline{X^{\kappa}}.T\overline{B}\,\overline{X} \to \psi(\overline{X})} \tag{8.3.1}$$

where $\exists_\Gamma(C)$ is the list of existential variable bindings of C and $\exists_C(\Gamma)$ drops the universal quantification of the existential variables in the type scheme of C (i.e., makes them into free variables) so that they become abstract. That is, when $C : \forall\overline{X^{\kappa}}.A \in \Gamma$ and $\overline{X'^{\kappa'}} = \overline{X^{\kappa}} \setminus \exists(C)$, then $C : \forall\overline{X'^{\kappa'}}.A \in \exists_C(\Gamma)$ and all other bindings in $\exists_C(\Gamma)$ remain the same as in Γ. We only need to make existential variables abstract when assigning types for pattern variables ($\overline{x : A}$). So, we use $\exists_C(\Gamma)$ only in the first premise. In the second premise, where we type check the

[5] In Nax, we should define `Exp` in two levels by taking the fixpoint $\mu_{*\to*}$ over a non-recursive GADT. For simplicity, we illustrate the type-indexed expression datatype using Haskell since Haskell GADTs allow recursive definitions.

body (t) of the alternative, we use the original Γ so that all quantified variables in the type scheme of C can be instantiated. For example, we should be able to apply `App :: Exp a -> Exp (a -> b) -> Exp b` in the example, which we discussed above, to an expression of any type (e.g., `Exp Int`, `Exp (Int -> Bool)`).

8.3.2 Generalized existential type variables and index transformers

Lin [55] generalized the notion of existential type variables, calling them *generalized existential type variables*, while developing a practical type inference algorithm for GADTs. Intuitively, generalized existential type variables are "type variables introduced by a pattern that receive no parametric instantiation" [55]. Generalized existential type variables are a conservative extension of existential type variables. That is, all existential type variables are generalized existential type variables. Here, we focus on the generalized type variables that are not existential type variables.

Consider the type representation (whose value describes the structure of a type. a.k.a. type universe) defined as a GADT in Haskell:

```
data Rep t where
  INT :: Rep Int
  PAIR :: Rep a -> Rep b -> Rep (a, b)
  FUN :: Rep a -> Rep b -> Rep (a -> b)
```

There are no existential type variables in the definition of `Rep`. The first type constructor `INT` does not have any type variables in its type. In the other two type constructors `PAIR` and `FUN`, both the type variables `a` and `b` appear in their result types, `Rep (a,b)` and `Rep (a -> b)`. These type variables could be *generalized existential variables*, which should not be instantiated to other types, just like existential type variables. For example, when we pattern match against a value of `Rep t`, `a` and `b` must remain polymorphic inside the alternatives. For instance, in the alternative for `PAIR`, we know that `t = (a,b)`, but we should instantiate neither `a` nor `b` because `t` is polymorphic.

Generalized existential type variables are extrinsic properties of data constructors, unlike existential type variables, which are intrinsic properties of data constructors [55]. Existential type variables always remain abstract (or polymorphic) inside alternatives regardless of the scrutinee type. On the other hand, generalized existential type variables depend on the scrutinee type. That is, we could have a different set of generalized existential type variables for the very same data constructor, depending on the type of the scrutinee we pattern match against. For example, when we pattern match against a value of `Exp (Int,Bool)`, there is no generalized existential type variable inside the alternative for `PAIR`,[6] because `a` and `b` are instantiated to `Int` and `Bool`.

We further adjust the rule Alt, adopting Lin's notion of generalized existential variables, as follows:

$$\textsc{Alt}\frac{\begin{array}{c}\Delta, \exists^{\psi}_{\Gamma}(C); \exists^{\psi}_{C}(\Gamma) \vdash C : \overline{A} \to T\overline{B}\,\overline{A'} \\ \Delta, \exists^{\psi}_{\Gamma}(C); \Gamma, \overline{x : A} \vdash t : \psi(\overline{A'})\end{array}}{\Delta; \Gamma \vdash^{\psi} C\overline{x} \to t : \forall \overline{X^{\kappa}}.T\overline{B}\,\overline{X} \to \psi(\overline{X})} \tag{8.3.2}$$

The change from the previous `Alt` rule (8.3.1) on page 274 is using $\exists^{\psi}_{\Gamma}$ and $\exists^{\psi}_{C}$ for generalized existential type variables instead of $\exists^{\psi}_{\Gamma}$ and $\exists^{\psi}_{C}$ for existential type variables. For example, $\exists^{\psi}_{\Gamma}(\texttt{PAIR})$ is the list of two variables consisting of `a` and `b`, while $\exists_{\Gamma}(\texttt{PAIR})$ is an empty list. Accordingly, the type scheme binding for `PAIR` in $\exists^{\psi}_{\texttt{PAIR}}(\Gamma)$ is a monomorphic type `Rep a -> Rep b -> Rep (a, b)` where `a` and `b` are free,[7] while the type scheme binding for `PAIR` in $\exists_{\texttt{PAIR}}(\Gamma)$ is a polymorphic type scheme $\forall \texttt{a}^*.\forall \texttt{b}^*.$ `Rep a -> Rep b -> Rep (a, b)`.

The Alt rule above (8.3.2) still lacks consideration for case functions that only expect scrutinee types with more specific indices than fully polymorphic variables

[6] Other alternatives, for `INT` and `FUN` are unreachable. So, for the scrutinee with type `Exp (Int,Bool)`, all cases are covered by a single alternative for `PAIR`. To make coverage checking for such scrutinee types aware of unreachable cases, we need an advanced coverage checking algorithm rather than just making sure that there exists an alternative for every data constructor. Coverage checking for our Nax implementation remains future work.

[7] They are bound in $\exists^{\psi}_{\Gamma}(\texttt{PAIR})$, of course.

(e.g., pattern matching only against values of type `Exp (Int,Bool)`, rather than `Exp t` for any type `t`). In our Nax implementation, we do support the syntax for such pattern matching by allowing index transformers of the form such as { `(Int,Bool)` . A }, whose argument list on the left-hand side of the dot could contain more specific forms than just type variables. Formulation of typing rules, which further adjust from the Alt rule (8.3.2), and implementation of coverage checking considering such index transformers with constrained input index are left for future work.

Part V

Postlude

Chapter 9

RELATED WORK

In this chapter, we discuss additional related work that was not discussed in previous chapters (Section 3.1 and Section 7.4). We discuss five categories of related work: Mendler-style co-recursion schemes over co-data (Section 9.1), Mendler-style recursion schemes over multiple values (Section 9.2), dependently-typed Mendler-style induction (Section 9.3), the use of sized-types to explain the termination of Mendler-style recursion schemes (Section 9.4), and the comparison of our Mendler-style approach to logical frameworks (Section 9.5).

9.1 MENDLER-STYLE CO-ITERATION AND CO-RECURSION

Data structures have natural duals, known as co-data. Data is characterized by how it is constructed and co-data is characterized by how it is observed (i.e., destructed).

Mendler-style recursion schemes generalize (or, dualize) naturally to co-data. We call these generalizations Mendler-style co-recursion schemes. These co-recursion schemes generate possibly infinite structures. For instance, an infinite sequence of natural numbers.

The Mendler-style co-iteration, **mcoit**, (a.k.a. anamorphism or unfold) is dual to the Mendler-style iteration, **mit**, (a.k.a. catamorpihsm or fold). Figure 9.1 (adapted from Uustalu and Vene [93]) illustrates a Haskell transcription of **mit** and its dual **mcoit**. We use the same style of Haskell code used in Chapter 3. The reversal of the function arrows is typical of a dual construction. Note that domain

```
  -- Mendler-style co-fixpoint ν* and co-iterator mcoit*
data ν* f = UnOut* {out* :: f (ν* f)}   -- use of UnOut* is restricted
mcoit* :: (∀r.(a → r) → (a → f r)) → (a → ν* f)
mcoit* φ v = UnOut* (φ (mcoit* φ) v)

  -- Mendler-style fixpoint μ* and iterator mit*
data μ* f = In* {unIn* :: f (μ* f)}   -- use of UnIn0 is restricted
mit*    :: (∀r.(r → a) → (f r → a)) → (μ* f → a)
mit* φ x = φ (mit* φ) (unIn* x)
```

Figure 9.1: A Haskell transcription of Mendler-style co-iteration (**mcoit**) in comparison to Mendler-style iteration (**mit**) at kind $*$.

and the codomain of the abstract operations are flipped: $(a \to r)$, $(a \to f\ r)$, $(a \to \nu_*\ f)$ verses $(r \to a)$, $(f\ r \to a)$, $(\mu_*\ f \to a)$. We illustrate Mendler-style co-iteration at kind $*$. Mendler-style co-iteration naturally generalizes to higher kinds just as Mendler-style iteration generalizes to higher kinds.

In order to understand co-recursive datatypes, we review recursive datatypes. In Mendler style, recursive datatypes are defined as fixpoints of (non-recursive) base structures. For example, we can define datatypes for natural numbers and lists in two steps: define the base structure (N and L) and take fixpoints of them (using μ_*):

```
data N r = Zero | Succ r          data L a r = Nil | Cons a r
type Nat = μ* N                   type List a = μ* (L a)
zero    = In* Zero                nil         = In* Nil
succ n = In* (Succ n)             cons x xs = In* (Cons x xs)
```

The constructor functions *zero*, *succ*, *nil*, and *cons* are ordinary definitions defined in terms of In_*. Using the conventions described in Chapter 3, the use of In_* is unrestricted, but its inverse $unIn_*$ (or pattern matching against In_*) is restricted. To eliminate a list or a natural number, one must use a Mendler-style recursion scheme, like $\mathbf{mit}_*$. In Mendler style, one can freely *construct* recursive values but

cannot freely destruct them. For example, one cannot define head or tail functions for *List* by simple pattern matching. Instead, one must define them via Mendler-style recursion schemes.

Conversely, when we define co-data, one can freely *tear down* their values, but one cannot freely construct them. To *construct* co-recursive values, one must rely on Mendler-style co-recursion schemes. Co-recursive datatypes are defined as the co-fixpoint ν_* of (non-recursive) base structures. For example, infinite streams are defined as a co-recursive datatype as follows:

$$\begin{array}{l}\textbf{data}\ \mathit{StreamF}\ a\ r = \mathit{SCons}\ a\ r\\ \textbf{type}\ \mathit{Stream}\ a = \nu_*\ (\mathit{StreamF}\ a)\\ \\ \mathit{head}\ s = \textbf{case}\ (\mathsf{out}_*\ s)\ \textbf{of}\ \mathit{SCons}\ h\ \_ \to h\\ \mathit{tail}\ \ s = \textbf{case}\ (\mathsf{out}_*\ s)\ \textbf{of}\ \mathit{SCons}\ \_\ t\ \to t\end{array}$$

Note that we can define destructor functions for streams, $\mathit{head} :: \mathit{Stream}\ a \to a$ and $\mathit{tail} :: \mathit{Stream}\ a \to \mathit{Stream}\ a$, simply by pattern matching, since we can freely use $\mathsf{out}_* :: \nu_*\ f \to f\ (\nu_*\ f)$.

However, without the help of a Mendler-style co-recursion scheme, one cannot define a constructor function, such as $\mathit{scons} :: a \to \mathit{Stream}\ a \to \mathit{Stream}\ a$, that builds up a new stream from an element and an existing stream. One must use Mendler-style co-recursion schemes to construct co-recursive values. This limitation follows from the restriction we place on the use of UnOut_*. We can pattern match against a value $(\mathit{UnOut}_*\ x)$ (or freely use the function out_*), but we cannot freely use UnOut_* to construct co-data. The last step of constructing co-data (applying UnOut_*) must be done using a Mendler-style co-recursion scheme, just as the first step of eliminating data (stripping off In_*) must be done using a Mendler-style recursion scheme.

As an example of constructing a *Stream*, we define a function $\mathit{upfrom} :: \mathit{Nat} \to \mathit{Stream}\ \mathit{Nat}$, which builds up a stream starting from a given natural number n where each element (n) is allways followed by its successor $(\mathit{succ}\ n)$, as follows:

$$upfrom\ n = \mathbf{mcoit}_*\ \varphi\ n\ \mathbf{where}$$
$$\varphi\ upfrm\ n = SCons\ n\ (upfrm\ (succ\ n))$$

For instance, (*upfrom zero*) is a stream of all the natural numbers, starting from zero, and counting upwards.

Note that the φ function is similar in structure to the general recursive implementation below, which exploits the laziness of Haskell:

$$\mathbf{data}\ Stream_g\ a = SCons_g\ a\ (Stream_g\ a)$$

$$upfrom_g\ n = SCons_g\ n\ (upfrom_g\ (succ\ n))$$

Although the streams built by *upfrom* conceptually stand for infinite lists, they do not diverge. The stream (*upfrom zero*) can be understood as a generator, ready to generate the next natural number (using *head*) or the next stream (using *tail*).

For example, we can write a function $take :: Nat \to Stream\ a \to List\ a$, where $take\ n\ s$ produces a list consisting of the prefix of length n of the stream s, as follows:

$$take\ n = \mathbf{mit}_*\ \varphi\ n\ \mathbf{where}$$
$$\varphi\ tk\ Zero = \lambda\_ \to nil$$
$$\varphi\ tk\ (Succ\ n) = \lambda s \to cons\ (head\ s)\ (tk\ n\ (tail\ s))$$

For instance, (*take three* (*upfrom zero*)) produces a list with three elements starting from zero (*cons* (*one* (*cons two* (*cons three nil*)))) where $one = succ\ zero$, $two = succ\ one$ and $three = succ\ two$.

Note that the φ function is similar in structure to how one would typically implement Haskell's standard prelude function $take :: Int \to [a] \to [a]$ over Haskell lists. Unlike the Haskell prelude function, which is partial (e.g., *take* 2 [ ] is undefined), our *take* funciton over streams are total because *Stream*s are always infinite by definition.

One could define a possibly finite stream by taking the co-fixpoint over L, sharing the same base structure with *List*, as follows:

$$\mathbf{type}\ Stream'\ a = \nu_*\ (L\ a)$$

$$
\begin{aligned}
head'\ s = \mathbf{case}\ \mathsf{out}_*\ s\ \mathbf{of}\ & \mathit{Nil} & \to \mathit{Nothing} \\
& \mathit{Cons}\ h\ \_ & \to \mathit{Just}\ h
\end{aligned}
$$

$$
\begin{aligned}
tail'\ \ s = \mathbf{case}\ \mathsf{out}_*\ s\ \mathbf{of}\ & \mathit{Nil} & \to \mathit{Nothing} \\
& \mathit{Cons}\ \_\ t & \to \mathit{Just}\ t
\end{aligned}
$$

Here, the destructors $head'$ and $tail'$ become slightly more complicated because $Stream'$ can be finite, terminating in Nil.

Because of laziness, datatypes in Haskell have characteristics of both recursive and co-recursive datatypes. However, when we use Haskell to explain Mendler-style concepts, we always distinguish recursive and co-recursive datatypes by adhering to the conventions we discussed: no general recursion except to define the (co-)fixpoint[1] operators themselves (μ_*, ν_*) and their (co-)recursion schemes ($\mathbf{mit}_*$, $\mathbf{mcoit}_*$). We also restrict the use of $unIn_*$ and $UnOut_*$ as described.

Matthes [59] extended System F with Mendler-style (co-)iteration and primitive (co-)recursion, and studied their properties. Abel et al. [5] embedded Mendler-style (co-)iteration into System F_ω. Abel and Matthes [3] discovered a reduction preserving embedding of Mendler-style primitive recursion into Fix_ω. They mention that an embedding of primitive co-recursion is similarly possible.

Uustalu and Vene [90, 91, 93] studied Mendler-style recursion schemes in a categorical setting, while the works mentioned in the paragraph above are set in the context of typed lambda calculi. Vene Vene [94] relates several Mendler-style recursion schemes with their non Mendler-style counterparts – (co-)iteration, primitive (co-)recursion, and course-of-values (co-)iteration.

[1] A word prefixed by '(co-)' refers to the words both with and without '(co-)'. That is, (co-)iteration refers to both iteration and co-iteration.

9.2 MENDLER-STYLE RECURSION SCHEMES OVER MULTIPLE VALUES

There are many Mendler-style recursion schemes in addition to those discussed in Chapter 3. Here, we introduce two Mendler-style recursion schemes that work over two (or more) structures simultaneously.

9.2.1 Simultaneous iteration

Uustalu and Vene [92] studied course-of-value iteration (a.k.a. histomorphism) and simultaneous iteration (a.k.a. multimorhpism). They formulate these two recursion schemes in both the conventional and Mendler style. They show that the formulations are equivalent provided that the base structures for recursive types are functors (i.e., positive). We have already discussed Mendler-style course-of-values iteration in previous chapters. Here, we introduce Mendler-style simultaneous iteration over multiple recursive values, using the examples adopted from Uustalu and Vene [92]. For simplicity, we only consider simultaneous iteration over two recursive values, which can be transcribed into Haskell as follows:

$$\mathbf{msimit}_{*,*} :: (\forall r_1\ r_2.(r_1 \to r_2 \to a) \to f_1\ r_1 \to f_2\ r_2 \to a) \to \mu_*\ f_1 \to \mu_*\ f_2 \to a$$

$$\mathbf{msimit}_{*,*}\ \varphi\ (\mathsf{In}_*\ x_1)\ (\mathsf{In}_*\ x_2) = \varphi\ (\mathbf{msimit}_{*,*}\ \varphi)\ x_1\ x_2$$

This recursion scheme simplifies function definitions that simultaneously iterate over two arguments. For example, we can define $\mathit{lessthan} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}$ and $\mathit{take} :: \mathit{Nat} \to \mathit{List}\ a \to \mathit{List}\ a$ as follows:

$$\mathit{lessthan} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Bool}$$

$$\mathit{lessthan} = \mathbf{msimit}_{*,*}\ \varphi\ \mathbf{where}$$

$$\begin{aligned}
&\varphi\ \mathit{lt}\ \mathit{Zero}\ \ \ \ \ \ \mathit{Zero}\ \ \ \ \ \ &&= \mathit{False}\\
&\varphi\ \mathit{lt}\ \mathit{Zero}\ \ \ \ \ \ (\mathit{Succ}\ _) &&= \mathit{True}\\
&\varphi\ \mathit{lt}\ (\mathit{Succ}\ _)\ \ \mathit{Zero}\ \ \ \ \ \ &&= \mathit{False}\\
&\varphi\ \mathit{lt}\ (\mathit{Succ}\ m)\ (\mathit{Succ}\ n) &&= \mathit{lt}\ m\ n
\end{aligned}$$

$$
\begin{array}{l}
take :: Nat \to List\ a \to List\ a \\
take = \mathbf{msimit}_{*,*}\ \varphi\ \mathbf{where} \\
\quad \varphi\ tk\ Zero\ \ \_ \qquad\qquad = nil \\
\quad \varphi\ tk\ (Succ\ \_)\ Nil \qquad\quad = nil \\
\quad \varphi\ tk\ (Succ\ n)\ (Cons\ x\ xs) = cons\ x\ (tk\ n\ xs)
\end{array}
$$

Note that the φ functions above are similar in structure to how one would typically define *lessthan* and *take* using general recursion in Haskell. Although it is possible to define these functions using multiple nested uses of $\mathbf{mit}_*$, it is certainly not as simple as the definitions above.

The termination behavior of simultaneous iteration (**msimit**) has not been studied when negative datatypes are involved. Nor do we know of any studies that have embeded **msimit** into a strongly normalizing typed lambda calculus. For both course-of-values iteration (**mcvit**) and recursion (**mcvpr**), we have found counterexamples that nontermination is possible for negative datatypes (see Figure 3.7 in Section 3.5 on p.113). We also showed that **mcvpr** can be embedded into Fix_i (or Fix_ω) assuming monotonicity (Section 5.3).

One can imagine simultaneous primitive recursion ($\mathbf{msimpr}_{*,*}$), which has additional casting operations, as follows:

$$
\begin{array}{l}
\mathbf{msimpr}_{*,*} :: (\forall r_1\ r_2\ .\ (r_1 \to r_2 \to a) \quad \text{-- recursive call} \\
\qquad\qquad\qquad\qquad \to (r_1 \to \mu_*\ f_1) \qquad \text{-- cast1} \\
\qquad\qquad\qquad\qquad \to (r_2 \to \mu_*\ f_2) \qquad \text{-- cast2} \\
\qquad\qquad\qquad\qquad \to f_1\ r_1 \to f_2\ r_2 \to a) \to \mu_*\ f_1 \to \mu_*\ f_2 \to a \\
\mathbf{msimpr}_{*,*}\ \varphi\ (\mathsf{In}_*\ x_1)\ (\mathsf{In}_*\ x_2) = \varphi\ (\mathbf{msimpr}_{*,*}\ \varphi)\ id\ id\ x_1\ x_2
\end{array}
$$

To extend primitive recursion ($\mathbf{mpr}_*$), which has has only one casting operation, into simultaneous primitive recursion, multiple casting operations are needed – one for each of recursive arguments. Here, we formulated $\mathbf{msimpr}_{*,*}$ with two recursive arguments. So, we have two casting operations, whose types are $(r_1 \to \mu_*\ f_1)$ and

$(r_2 \to \mu_*\ f_2)$.

9.2.2 Lexicographic recursion

Some recursive functions over multiple recursive values justify termination because their arguments decrease at every recursive call under a lexicographic ordering. Note that this is different from simultaneous iteration where each of the arguments decreases in every recursive call. In a lexicographic ordering, some arguments may stay the same (in more-significant positions) or increase (in less-significant positions) while another argument decreases. A typical example of lexicographic recursion is the Ackermann function, which we can define using general recursion in Haskell as follows:

$$
\begin{array}{l}
\textbf{data}\ Nat_g = Zero_g \mid Succ_g\ Nat_g \\
acker_g\ Zero_g\ \ \ \ n\ \ \ \ = Succ_g\ n \\
acker_g\ (Succ_g\ m)\ Zero_g\ \ \ \ = Succ_g\ (acker_g\ m\ Zero_g) \\
acker_g\ (Succ_g\ m)\ (Succ_g\ n) = acker_g\ m\ (acker_g\ (Succ_g\ m)\ n)
\end{array}
$$

Observe that the first argument is more significant than the second. In the third equation, the first argument m of the outer recursive call decreases (i.e., is smaller than $(Succ_g\ m)$) while the second argument $(acker_g\ (Succ_g\ m)\ n)$ may increase (i.e., may be larger than $(Succ_g\ n)$).

The following Mendler-style recursion scheme captures the idea of lexicographic recursion over two arguments.

$$
\begin{array}{l}
\textbf{mlexpr}_{*,*} :: (\forall r_1\ r_2\ .\ (r_1 \to \mu_*\ f_2 \to a) \quad \text{-- outer recursive call} \\
\qquad\qquad \to (r_2 \to a) \qquad\qquad \text{-- inner recursive call} \\
\qquad\qquad \to (r_1 \to \mu_*\ f_1) \qquad\qquad \text{-- cast1} \\
\qquad\qquad \to (r_2 \to \mu_*\ f_2) \qquad\qquad \text{-- cast2} \\
\qquad\qquad \to f_1\ r_1 \to f_2\ r_2 \to a) \to \mu_*\ f_1 \to \mu_*\ f_2 \to a \\
\textbf{mlexpr}_{*,*}\ \varphi\ (\mathsf{In}_*\ x_1)\ (\mathsf{In}_*\ x_2) = \varphi\ (\textbf{mlexpr}_{*,*}\ \varphi)\ (\textbf{mlexpr}_{*,*}\ \varphi\ (\mathsf{In}_*\ x_1))\ id\ id\ x_1\ x_2
\end{array}
$$

The Mendler-style lexicographic recursion $\mathbf{mlexpr}_{*,*}$[2] is similar to the Mendler-style simultaneous recursion $\mathbf{msimpr}_{*,*}$ introduced in the previous section, but has two abstract operations for inner and outer recursion. Note the types of these two recursive calls $(r_1 \to \mu_*\ f_2 \to a)$ and $(r_2 \to a)$. The outer recursive call expects its first argument to be a direct subcomponent by requiring its type to be r_1. The second argument has type $\mu_*\ f_2$, which means that it could be any value, because it is the less-significant factor of the lexicographic ordering. The inner recursive call only expects its second argument to be a direct subcomponent by requiring its type is required to be r_2. Since it is assumed that the first argument stays the same in the inner call, the first argument is omitted. Using $\mathbf{mlexpr}_{*,*}$, we can define the Ackermann function as follows:

$$
\begin{array}{l}
\mathit{acker} = \mathbf{mlexpr}_{*,*}\ \varphi\ \mathbf{where} \\
\quad \varphi\ \mathit{ack}\ \mathit{ack}'\ \mathit{cast1}\ \mathit{cast2}\ \mathit{Zero}\ \ \ \mathit{Zero}\ \ \ = \mathit{succ}\ \mathit{zero} \\
\quad \varphi\ \mathit{ack}\ \mathit{ack}'\ \mathit{cast1}\ \mathit{cast2}\ \mathit{Zero}\ \ \ (\mathit{Succ}\ n) = \mathit{succ}\ (\mathit{succ}\ (\mathit{cast2}\ n)) \\
\quad \varphi\ \mathit{ack}\ \mathit{ack}'\ \mathit{cast1}\ \mathit{cast2}\ (\mathit{Succ}\ m)\ \mathit{Zero}\ \ \ = \mathit{succ}\ (\mathit{ack}\ m\ \mathit{zero}) \\
\quad \varphi\ \mathit{ack}\ \mathit{ack}'\ \mathit{cast1}\ \mathit{cast2}\ (\mathit{Succ}\ m)\ (\mathit{Succ}\ n) = \mathit{ack}\ m\ (\mathit{ack}'\ n)
\end{array}
$$

We strongly believe that $\mathbf{mlexpr}_{*,*}$ terminates for all positive datatypes. The termination behavior for negative (or mixed-variant) datatypes needs further investigation.

9.3 MENDLER-STYLE INDUCTION

The dependently-typed version of primitive recursion is called induction. We formulate Mendler-style induction over regular datatypes as follows.

$$
\begin{array}{l}
\mathbf{mind}_* \ :\ \forall (F : * \to *)(A : \mu_* F \to *). \\
\qquad \Big(\forall (r : *).\ (\mathit{cast} : r \to \mu_* F) \\
\qquad\qquad \to (\mathit{call} : (x : r) \to A\ (\mathit{cast}\ x)) \\
\qquad\qquad \to (y : F\ r) \to A\ (\mathsf{In}_*(\mathit{fmap}_F\ \mathit{cast}\ y))\Big) \to (z : \mu_*\ f) \to A\ z
\end{array}
$$

[2] The idea for $\mathbf{mlexpr}_{*,*}$ originated in a conversation between Tarmo Uustalu and Tim Sheard at the TYPES 2013 workshop (not published anywhere else at the moment).

$$\mathbf{mind}_*\ \varphi\ (\mathsf{In}_*\ x)\ =\ \varphi\ \ id\ \ (\mathbf{mind}_*\ \varphi)\ \ x$$

The definition of Mendler-style induction **mind**[3] shows that induction is essentially the same as the Mendler-style primitive recursion **mpr**, except that the type signature involves dependent types. Note, the final answer type $(A\ z)$ is dependent on the recursive argument $z : \mu_* F$. Since $A : \mu_* F \to *$ expects a concrete recursive value, we use *cast* in the type signature of the φ function to cast $(x : r)$ and $(y : F\, r)$ into $\mu_* F$ values, so that they can be passed to A. In the type signature of **mind**, *cast* comes before *call* because the type signature of *call* depends on *cast*. When defining **mpr**, *cast* and *call* can come in any order since there is no dependency in the type signature of **mpr**.

One important aspect of $\mathbf{mind}_*$ is that it is well-defined only over positive F, because we relied on the existence of $fmap_F$ to write its type signature. It is an open question whether one can formulate a Mendler-style induction that works for negative datatypes.

In the future work section (Section 10.1), we introduce another Mendler-style recursion scheme that is useful for mixed-variant datatypes. The work of a Mendler stylist is never done!

9.4 TYPE-BASED TERMINATION AND SIZED TYPES[4]

Type-based termination (coined by Barthe, Frade, Giménez, Pinto, and Uustalu [9]) stands for approaches that integrate termination into type checking, as opposed to syntactic approaches that reason about termination over untyped term structures. The Mendler-style approach is, of course, type-based. In fact, the idea of type-based termination was inspired by Mendler [64, 65]. In Mendler style, we know

[3] The idea behind $\mathbf{mind}_*$ comes from discussion with Tarmo Uustalu. He described this on a whiteboard when I met with him at the University of Cambridge in Fall 2011.

[4] We plan to submit a modified version of this section as a part of the TYPES post-proceedings draft.

that well-typed functions defined using Mendler-style recursion schemes always terminate. This guarantee follows from the design of the recursion scheme, where the use of higher-rank polymorphic types in the abstract operations enforce the invariants necessary for termination.

Abel [1, 2] summarizes the advantages of type-based termination as follows: *communication* (programmers think using types), *certification* (types are machine-checkable certificates), *a simple theoretical justification* (no additional complication for termination other than type checking), *orthogonality* (only small parts of the language are affected, e.g., principled recursion schemes instead of general recursion), *robustness* (type system extensions are less likely to disrupt termination checking), *compositionality*[5] (one needs only types, not the code, for checking the termination), and *higher-order functions and higher-kinded datatypes* (works well even for higher-order functions and non-regular datatypes, as a consequence of compositionality). In his dissertation [1] (Section 4.4) on sized types, Abel views the Mendler-style approach as enforcing size restrictions using higher-rank polymorphism as follows:

- The abstract recursive type r in Mendler style corresponds to $\mu^{\alpha} F$ in his sized-type system (System $\mathsf{F}_{\omega}^{\hat{}}$), where the sized type for the value being passed in corresponds to $\mu^{\alpha+1} F$.
- The concrete recursive type μF in Mendler style corresponds to $\mu^{\infty} F$ since there is no size restriction.
- By subtyping, a type with a smaller size-index can be cast to the same type with a larger size-index.

This view is based on the same intuition we discussed in Chapter 3. Mendler-style recursion schemes terminate — for positive datatypes — because r-values are direct subcomponents of the value being eliminated. They are always smaller than the

[5]This is not listed in Abel's thesis, but comes from his invited talk in FICS 2012.

value being passed in. Types enforce that recursive calls are only well-typed, when applied to smaller subcomponents.

Abel's System $\mathsf{F}_{\omega}^{\hat{}}$ can express primitive recursion quite naturally using subtyping. The casting operation $(r \to \mu F)$ in Mendler-style primitive recursion corresponds to an implicit conversion by subtyping from $\mu^{\alpha} F$ to $\mu^{\infty} F$ because $\alpha \leqslant \infty$.

System $\mathsf{F}_{\omega}^{\hat{}}$ [1] is closely related to System Fix_{ω} [3]. Both of these systems are base on equi-recursive fixpoint types over positive base structures. Both of these systems are able to embed (or simulate) Mendler-style primitive recursion (which is based on iso-recursive types) via the encoding [37] of arbitrary base structures into positive base structures. In Section 5.2, we rely on the same encoding, denoted by Φ, when embedding **mpr** into System Fix_i.

Abel's sized-type approach evidences good intuition concerning the reasons that certain recursion schemes terminate over positive datatypes. But, it is not a useful intuition of whether or not those recursion schemes would terminate for negative datatypes, unless there is an encoding that can translate negative datatypes into positive datatypes. For primitive recursion, this is possible (as we mentioned above). However, for our recursion scheme **msfit**, which is especially useful over negative datatypes, we do not know of an encoding that can map the inverse augmented fixpoints into positive fixpoints. So, it is not clear whether Abel's the sized-type approach based on positive equi-recursive fixpoint types can provide a good explanation for the termination of **msfit**. In Section 10.1, we will discuss another Mendler-style recursion scheme (**mprsi**), which is also useful over negative datatypes and has a termination property (not yet proved) based on the size of the index in the datatype.

9.5 LOGICAL FRAMEWORKS BASED ON THE $\lambda\Pi$-CALCULUS

A "logical framework", in a broad sense, refers to any system that serves as "a meta-language for the formalization of deductive systems" [77]. In a more narrow sense, logical frameworks are systems closely related to to the Edinburgh Logical Framework (LF or ELF) [44], which uses the $\lambda\Pi$-calculus as its specification language. In this section, we discuss *logical frameworks* in this more narrow sense.

The $\lambda\Pi$-calculus (a.k.a. $\lambda\mathbf{P}$) is one of the corners in Barendregt's λ-cube [8] that is adjacent to the simply-typed lambda calculus (STLC, or, $\lambda\!\rightarrow$). The $\lambda\Pi$-calculus extends the STLC with dependent types, but without polymorphism or functions from types to types (type operators). The syntax of $\lambda\Pi$, extended with constants (c), is describe below:

Kinds	K	$::= \texttt{type} \mid \Pi x : A.K$
Type Families	A, B	$::= c \mid \Pi x : A.B \mid \lambda x : A.B \mid AM$
Objects	M, N	$::= c \mid x \mid \lambda x : A.M \mid MN$

In logical frameworks, one can introduce new constants by naming types and objects. These are intended to represent datatypes such as natural numbers, lists, and may even involve higher-order abstract syntax. However, these constants are merely syntactic descriptions, not necessary tied to any specific semantics or logical interpretations. That is, introducing constants does not automatically supply any recursion schemes or induction principles, as is done in functional languages or proof assistants that support new datatypes as a feature. Each logical framework supports its own meta-logic to give meanings to the logic (or, the language) specified by introducing such constants. The choice of meta-logic can be either relational (like a logic programming language), functional (like a functional programming language), or something else.

Logical frameworks are very flexible for describing many different logical systems (i.e., formalizing a language) by using a two-layered approach of a minimal

specification language ($\lambda\Pi$) and a meta-logic. However, this two-layered approach is not ideal as a programming system. One can model arbitrary programming languages, giving them semantics in the logical framework. But, the programming capability of the specification language and the meta-logic is limited. In the remainder of this section, we discuss Twelf, whose meta-logic is relational, and, Beluga and Delphin, whose meta-logics are functional.

Twelf[6] is the most widely used logical framework. In Twelf, you can define abstract syntax for datatypes by introducing constants for types and objects involving those types. For example, you can define natural numbers as follows.[7]

```
nat : type.       %%% define a type constant
z : nat.          %%% define a constant for zero
s : nat -> nat.   %%% define a constant for sucessor
```

At this point, the constants `z` and `s` are just typed syntax. Introduction of constants is not associated with any semantics for the constants, unlike the natively supported inductive datatypes in Coq or Agda. So, there are no restrictions on how these constants may be used, such as the positivity constraint on inductive datatypes in Coq or Agda. One can give meanings to the natural number constants by defining inductive relations over them. For example, we can define addition as a ternary relation over natural numbers, as follows:

```
plus : nat -> nat -> nat -> type.
plus/z : plus z Y Y.
plus/s : plus (s X) Y (s Z)
      <- plus X Y Z.
```

The right-hand sides (after the colon) of `plus/z` and `plus/s` look like a Prolog program defining addition. Twelf's meta-logic is, in fact, typed first-order relational

[6] `http://twelf.org/`

[7] Twelf examples are adopted from Boyland's Twelf Library on Github. `https://github.com/boyland/twelf-library`

logic. At the type-level, Twelf predicates are like pure Prolog programs with type-checking. All computational issues, such as termination checking, are present at the level of these relational definitions (as opposed to the introduction of new constants). Twelf has a termination checker for inductive relations (external to the type checker) based on lexicographic subterm ordering over untyped terms. In addition to the type signatures of the relations, one can optionally specify input/output modes for each of their arguments, if necessary, in order to guide the termination checker to consider only the input arguments for termination.[8]

One cannot write higher-order relations natively in Twelf because Twelf's meta-logic is first-order, not higher-order. To write a program using higher-order functions in Twelf, one has to model one's own object language that is able to support higher-order functions, then program within that object language rather than programming in Twelf's meta-logic. We summarize the steps necessary to program using higher-order functions in Twelf:

(1) Define an object language syntax (as the syntax `z` and `s` for natural numbers) with bindings (this is done by HOAS), applications, and whatever needed to express higher-order functions.

(2) Define the evaluation semantics of the object language using inductive relations (i.e., write an evaluation relation for the object language in a Prolog-like way).

(3) Write programs in the object language by putting together pieces of the syntax you defined in (1).

(4) Finally, evaluate the program by reasoning based on the evaluation relation defined in (2).

[8]There are various directives to guide checking input/output modes, coverage, and termination in Twelf. For further information, see the documentations from its homepage.

This process is clearly not ideal if the desire is simply to "program" with higher-order functions in a type-safe way, possibly with some termination guarantees. One does not always want to reason about the meta-theory of the object language in general.

Beluga [78] is similar to Twelf, but it is closer to a functional language since the inductive definitions are functional, rather than relational. Beluga supports higher-order functions, unlike Twelf. One can write a natural number addition function in Beluga as follows:[9]

```
rec add : [. nat ] -> [. nat ] -> [. nat ]
  = fn x => fn y =>
    case x of
    | [. z ]   => y
    | [. s N ] => let [. R ] = add [. N ] y in [. s R ]
    ;
```

Types like `nat` and `nat -> nat` are called representation-level types. So, objects like `z` and `s` are called representation-level objects. Types like `[. nat ]` and `[. nat ] -> [. nat ]` are called computation-level types. This `add` function definition is almost identical to typical recursive function definition of natural number addition in functional languages, except for the new representation-level variable binding `R` in the second case branch. In Beluga, one cannot write `[. s (add [. N ] y) ]` because `s` expects a representation-level object as its argument. In Twelf-style logical frameworks, representation-level types are inhabited only by η-long β-normal representation-level objects, which do not include application forms of computational-level objects.

More generally, computation-level types can have the form `[g . t]` where `g` is a context object and `t` is a representation-level type. One of the Beluga's unique features is supporting pattern matching over computational objects with contexts,

[9] Adopted from the Beluga tutorial. `http://complogic.cs.mcgill.ca/beluga/`

and also coverage checking of those patterns. Computational types with the empty context, of the form `[. t]`, are inhabited by closed values, which do not involve any free (representation-level) variables. Since Beluga has explicit access to context objects, we believe that it can express what **msfit** can express, and in addition, it can also express what **openit** (Section 3.9.5) can express.

One can also write higher-order functions (e.g., map)[10] in Beluga almost the same way one does in functional programming languages, except, perhaps, for the tedious representation-level bindings (e.g., `R` in the `add` function above). In regards to higher-order functions, Beluga is in a much better position than Twelf. Recall that, in Twelf, one needs to model a whole new functional language by describing its semantics with inductive relations in order to express higher-order functions.

Termination is not type based in Beluga either. Like Twelf, it needs an external termination (or totality) checker, but its prototype implementation currently lacks such a checker. We suspect one of the reasons why the Beluga implementation does not yet contain a termination checker is due to the difficulty of checking termination of higher-order functions. The syntactic approaches to termination, used by logical frameworks based on first-order meta-logic, may fail to check termination for many higher-order functions.

Delphin [79] has goals similar to Beluga, supporting functional programming rather than relational reasoning. For example, the addition function over natural numbers can be defined in Delphin as follows.

```
fun plus : <nat> -> <nat> -> <nat>
  = fn <z>   <M> => <M>
    |  <s N> <M> => let val <x> = plus <N> <M> in <s x> end
    ;
```

Although both Beluga and Delphin support similar features with similar syntax, their theoretical foundations differ [78] on how they treat contexts. Delphin cannot

[10] A map function over natural number lists is given in the Beluga tutorial.

distinguish open values from closed values as is done in Beluga, since Delpin does not explicitly manage contexts. Pientka [78] also points out that Delphin tries to reuse Twelf's infrastructure as much as possible. For instance, the termination checker of Delphin is based on lexicographic subterm ordering, which is also the case in Twelf.

Although Delphin and Beluga do support higher-order functions, they do not support polymorphism, but only dependent types by term indexing. That is, one can only write monomorphic functions. Recall that, in $\lambda\Pi$, one can only index type families by terms, not types. Indexing by types would support polymorphism. This is inconvenient for programming higher-order functions, because many higher-order functions are polymorphic in nature; users need to duplicate their definitions for each different type needed.

Chapter 10

FUTURE WORK

We summarize some ongoing and future work in this chapter: designing a new Mendler-style recursion scheme that is useful for negative datatypes (Section 10.1), different fixpoint types (Section 10.2), deriving monotonicity from polarized kinds (Section 10.4), and kind polymorphism and kind inference (Section 10.4).

10.1 ANOTHER MENDLER-STYLE RECURSION SCHEME FOR MIXED-VARIANT DATATYPES [1]

In Section 3.9, we discussed Mendler-style iteration with a syntactic inverse, **msfit**, which is particularly useful for defining functions over negative (or mixed-variant) datatypes. We demonstrated the usefulness of **msfit** by defining functions over HOAS:

- the string formatting function *showHOAS* for the untyped HOAS using $\mathbf{msfit}_{*}$ (Figure 3.17 on p.133) and
- the type-preserving evaluator *evalHOAS* for the simply-typed HOAS using $\mathbf{msfit}_{*\to *}$ (Figure 3.20 on p.143).

In this section, we speculate about another Mendler-style recursion scheme, **mprsi**, motivated by an example similar to the *evalHOAS* function. The name **mprsi** stands for Mendler-style primitive recursion with a sized index.

[1] This section is an extended and revised version of our extended abstract (without the introduction section) in the TYPES 2013 workshop.

```
data ExpF r t where Lam :: (r t1 → r t2) → ExpF r (t1 → t2)
                    App :: r (t1 → t2) → r t1 → ExpF r t2
type Exp' a t = ŭ*→* ExpF a t
type Exp t = ∀a.Exp' a t
lam :: (∀a.Exp' a t1 → Exp' a t2) → Exp (t1 → t2)
lam e = Iĭ*→* (Lam e)
app :: Exp (t1 → t2) → Exp t1 → Exp t2
app e1 e2 = Iĭ*→* (App e1 e2)

newtype Id a = MkId {unId :: a}
type Phi f a = ∀r.(∀i.a i → r a i) → (∀i.r a i → a i) → (∀i.f (r a) i → a i)

evalHOAS :: Exp t → Id t
evalHOAS e = msfit*→* φ e where
  φ :: Phi ExpF Id
  φ inv ev (Lam f) = MkId (λv → unId (ev (f (inv (MkId v)))))
  φ inv ev (App f x) = MkId (unId (ev f) (unId (ev x)))

  -- The code above is the same as the code in Figure 3.20 in Section 3.9.
  -- We repeat it here, in order to review the evalHOAS example.

data V r t where VFun :: (r t1 → r t2) → V r (t1 → t2)
type Val t = μ*→* V t
val f = In*→* (VFun f)

vevalHOAS :: Exp t → Val t
vevalHOAS e = msfit*→* φ e where
  φ :: Phi ExpF (μ*→* V)
  φ inv ev (Lam f) = val (λv → ev (f (inv v)))
  φ inv ev (App e1 e2) = unVal (ev e1) (ev e2)

  -- unVal does not follow the restrictions of the Mendler style.
  -- Its definition relies on pattern matching against In*→*.

unVal :: Val (t1 → t2) → (Val t1 → Val t2)
unVal (In*→* (VFun f)) = f
```

Figure 10.1: Two evaluators for the simply-typed λ-calculus in HOAS. One uses a native (Haskell) value domain (*evalHOAS*), the other uses a user-defined value domain (*vevalHOAS*).

We review the *evalHOAS* example and then compare it to our motivating example *vevalHOAS* for **mprsi**. Both *evalHOAS* and *vevalHOAS* are illustrated in Figure 10.1. Recall that this code is written in Haskell, following the Mendler-style conventions. The function $evalHOAS :: Exp\ t \to Id\ t$ is a type-preserving evaluator that evaluates an HOAS expression of type t to a (Haskell) value of type t. The *evalHOAS* function always terminates because $\mathbf{msfit}_{*\to*}$ always terminates. Recall that $\mathbf{msfit}_{*\to*}$ and $\breve{\mu}_{*\to*}$ can be embedded into System F_ω (or System F_i, if we need term indices).

The motivating example $vevalHOAS :: Exp\ t \to Val\ t$ is also a type-preserving evaluator. Unlike *evalHOAS*, it evaluates to a user-defined value domain *Val* of type t (rather than a Haskell value). The definition of *vevalHOAS* is similar to *evalHOAS*; both of them are defined using $\mathbf{msfit}_{*\to*}$. The first equation of φ for evaluating the *Lam*-expression is essentially the same as the corresponding equation in the definition of *evalHOAS*. The second equation of φ for evaluating the *App*-expression is also similar in structure to the corresponding equation in the definition of *evalHOAS*. However, the use of *unVal* is problematic. In particular, the definition of *unVal* relies on pattern matching against $\mathsf{In}_{*\to*}$. Recall that one cannot freely pattern match against a recursive value in Mendler style. Recursive values must be analyzed (or eliminated) by using Mendler-style recursion schemes. It is not a problem to use *unId* in the definition of *evalHOAS* because *Id* is non-recursive.

It is not likely that *unVal* can be defined using any of the existing Mendler-style recursion schemes discussed earlier. So, we designed a new Mendler-style recursion scheme that can express *unVal*. The new recursion scheme **mprsi** extends **mpr** with an additional uncast operation. Recall that **mpr** has two abstract operations, call and cast. So, **mprsi** has three abstract operations, call, cast, and uncast. In the following paragraphs, we explain the design of **mprsi** step-by-step.

Let us try to define *unVal* using $\mathbf{mpr}_{*\to*}$ and examine where it falls short. $\mathbf{mpr}_{*\to*}$ provides two abstract operations, *cast* and *call*, as it can be seen from the type signature below:

$$\begin{array}{lll}\mathbf{mpr}_{*\to*} :: (\forall r\ i.(\forall i.r\ i \to \mu_{*\to*}\ f\ i) & \text{-- cast} \\ \quad \to \quad (\forall i.r\ i \to a\ i) & \text{-- call} \\ \quad \to \quad (f\ r\ i \to a\ i) &) \to \mu\ f\ i \to a\ i\end{array}$$

We attempt to define *unVal* using $\mathbf{mpr}_{*\to*}$ as follows:

$$\begin{array}{l} unVal :: \mu_{*\to*}\ V\ (t_1 \to t_2) \to (\mu_{*\to*}\ V\ t_1 \to \mu_{*\to*}\ V\ t_2) \\ unVal = \mathbf{mpr}_{*\to*}\ \varphi\ \mathbf{where} \\ \quad \varphi\ cast\ call\ (VFun\ f) = ... \end{array}$$

Inside the φ function, we have a function $f :: (r\ t_1 \to r\ t_2)$ over abstract recursive values. We need to cast f into a function over concrete recursive values $(\mu\ V\ t_1 \to \mu\ V\ t_2)$. We should not need to use *call*, since we do not expect to use any recursion to define *unVal*. So, the only available operation is $cast :: (\forall i.r\ i \to \mu\ f\ i)$. Composing *cast* with f, we can get $(cast \circ f) :: (r\ t_1 \to \mu\ V\ t_2)$, whose codomain $(\mu\ V\ t_2)$ is exactly what we want. But, the domain is still abstract $(r\ t_1)$ rather than being concrete $(\mu\ V\ t_1)$. We are stuck.

What additional abstract operation would help us complete the definition of *unVal*? We need an abstract operation to cast from $(r\ t_1)$ to $(\mu\ V\ t_1)$ in a contravariant position. If we had an inverse of cast, $uncast :: (\forall i.\mu\ f\ i \to r\ i)$, we could complete the definition of *unVal* by composing *uncast*, f, and *cast*. That is, $(uncast \circ f \circ cast) :: (\mu_{*\to*}\ V\ t_1 \to \mu_{*\to*}\ V\ t_2)$. Thus, we can formulate $\mathbf{mprsi}_{*\to*}$ with a naive type signature as follows:

$$\begin{array}{lll}\mathbf{mprsi}_{*\to*} :: (\forall r\ i. & (\forall i.r\ i \to \mu_{*\to*}\ f\ i) & \text{-- cast} \\ & \to (\forall i.\mu_{*\to*}\ f\ i \to r\ i) & \text{-- uncast} \\ & \to (\forall i.r\ i \to a\ i) & \text{-- call} \\ & \to (f\ r\ i \to a\ i) &) \to \mu\ f\ i \to a\ i\end{array}$$

$$\mathbf{mprsi}_{*\to*}\ \varphi\ (\mathsf{In}_{*\to*}\ x) = \varphi\ id\ id\ (\mathbf{mprsi}_{*\to*}\ \varphi)\ x$$

Although the type signature above is type-correct, it is too powerful. The Mendler-style approach uses types to forbid, as ill-typed, non-terminating computations. Having both *cast* and *uncast* supports the same ability as freely pattern matching over recursive values, which can lead to non-termination. To recover the guarantee of termination, we need to restrict the use of either *cast* or *uncast*, or both.

Let us see how this non-termination might occur. If we allowed $\mathbf{mprsi}_{*\to *}$ with the naive type signature above, we would be able to write an evaluator (similar to *vevalHOAS* but for an untyped HOAS) that does not always terminate. This evaluator would diverge for terms with self application. Here, we walk through the process of defining an untyped HOAS with a dummy index. The base structures for the untyped HOAS and its value domain can be defined as follows:

$$\mathbf{data}\ \mathit{ExpF}_u\ r\ t = \mathit{Lam}_u\ (r\ t \to r\ t) \mid \mathit{App}_u\ (r\ t)\ (r\ t)$$

$$\mathbf{data}\ V_u\ r\ t = \mathit{VFun}_u\ (r\ t \to r\ t)$$

Fixpoints of the structures above represent the untyped HOAS and its value domain. Here, the index t is bogus; that is, it does not track the type of an HOAS expression but remains constant everywhere. Using the naive version of $\mathbf{mprsi}_{*\to *}$ above, we can write an evaluator similar to *vevalHOAS* for the untyped HOAS $(\mu_{*\to *}\ \mathit{ExpF}_u\ ())$ via the value domain $(\mu_{*\to *}\ V_u\ ())$, which would obviously not terminate for some input.

Why did we believe that *vevalHOAS* always terminates? Because it evaluates a well-typed HOAS, whose type is encoded as an index t in the recursive datatype $(\mathit{Exp}\ t)$. That is, the use of indices as types is the key to the termination property. Therefore, our idea is to restrict the use of the abstract operations in $\mathbf{mprsi}_{*\to *}$ by enforcing constraints over their indices; in that way, we would still be able write *vevalHOAS* for the typed HOAS, but would get a type error when we try to write an evaluator for the untyped HOAS.

We suggest that some of the abstract operations of $\mathbf{mprsi}_{*\to*}$ should only be applied to the abstract values whose indices are smaller in size compared to the size of the argument index. For the *vevalHOAS* example, we define being smaller as the structural ordering over types, that is, $t_1 < (t_1 \to t_2)$ and $t_2 < (t_2 \to t_1)$. We have two candidates for the type signature of $\mathbf{mprsi}_{*\to*}$:

- Candidate 1: restrict uses of both *cast* and *uncast*

$$\begin{array}{lll} \mathbf{mprsi}_{*\to*} :: (\forall r\ j. & (\forall i.(i<j) \Rightarrow r\ i \to \mu_{*\to*}\ f\ i) & \text{-- cast} \\ \to (\forall i.(i<j) \Rightarrow \mu_{*\to*}\ f\ i \to r\ i) & & \text{-- uncast} \\ \to (\forall i. & r\ i \to a\ i) & \text{-- call} \\ \to (f\ r\ j \to a\ j) & &) \to \mu\ f\ i \to a\ i \end{array}$$

- Candidate 2: restrict the use of *uncast* only

$$\begin{array}{lll} \mathbf{mprsi}_{*\to*} :: (\forall r\ j. & (\forall i. & r\ i \to \mu_{*\to*}\ f\ i) \quad \text{-- cast} \\ \to (\forall i.(i<j) \Rightarrow \mu_{*\to*}\ f\ i \to r\ i) & & \text{-- uncast} \\ \to (\forall i. & r\ i \to a\ i) & \text{-- call} \\ \to (f\ r\ j \to a\ j) & &) \to \mu\ f\ i \to a\ i \end{array}$$

We strongly believe that the first candidate always terminates, but it might be overly restrictive. Maybe the second candidate is enough to guarantee termination? Both candidates allow defining *vevalHOAS*, because one can define *unVal* using $\mathbf{mprsi}_{*\to*}$ with either one of the candidates, but both forbid the evaluator over the untyped HOAS, because neither supports extracting functions from the untyped value domain.

We need further studies to prove the termination properties of **mprsi**. The sized-type approach, discussed in Section 9.4, seems to be relevant for showing the termination of **mprsi**. However, existing theories on sized-types are not directly applicable to **mprsi** because they are focused on positive datatypes, but not negative datatypes.

10.2 CONVERSION BETWEEN DIFFERENT FIXPOINT TYPES

In Chapter 3, we introduced several Mendler-style recursion schemes by describing them in Haskell, following certain stylistic conventions. Most of the recursion schemes, including **mit** and **mpr**, share the same standard fixpoint representation in Haskell, denoted by μ, except those recursion schemes involving inverse operations, such as **msfit**. The recursion schemes involving inverse operations operate on the inverse augmented fixpoint, denoted by $\breve{\mu}$. Recall the Haskell definitions of the two different fixpoint type operators, μ and $\breve{\mu}$, at kind $*$, repeated below:

$$\textbf{newtype}\ \mu_*\ f \quad = \mathsf{In}_*\ (f\ (\mu_*\ f)) \qquad \text{-- } \textbf{mit}_*, \textbf{mpr}_*, \ldots$$

$$\textbf{data} \qquad \breve{\mu}_*\ f\ a = \mathsf{I\breve{n}}_*\ (f\ (\breve{\mu}_*\ f\ a)) \mid \mathsf{Inverse}_*\ a \quad \text{-- } \textbf{msfit}_*$$

We want to establish an isomorphism,[2] $\mu_*\ f \simeq (\forall a.\breve{\mu}_*\ f\ a)$, between these two fixpoint types, because we want the Nax language to have one fixpoint rather than two. Naively thinking, there is likely to be a one-to-one mapping between the μ_*-values and the $\breve{\mu}_*$-values that do not involve the constructor $\mathsf{Inverse}_*$. Since μ_* and $\breve{\mu}_*$ look structurally isomorphic to each other excuding $\mathsf{Inverse}_*$, one could expect that the quantification $\forall a$ in $(\forall a.\breve{\mu}_*\ f\ a)$ would prevent the constructor $\mathsf{Inverse}_*$ from appearing in values of type $(\forall a.\breve{\mu}_*\ f\ a)$.

To establish an isomorphism between μ_* and $\breve{\mu}_*$, we must construct two mapping (or coercion) functions of type $\mu_*\ f \to (\forall a.\breve{\mu}_*\ f\ a)$ and $(\forall a.\breve{\mu}_*\ f\ a) \to \mu_*\ f$ (that are each other's inverse). At first glance, we thought it would be easier to find a mapping of type $\mu_*\ f \to (\forall a.\breve{\mu}_*\ f\ a)$ by replacing all the In_*s with $\mathsf{I\breve{n}}_*$s. However, contrary to our expectation, the other mapping turns out to be more natural. We illustrate this by using the HOAS datatype as an example. At the end of this section, we will contemplate on why this is so.

Figure 10.2 illustrates a mapping from $(\forall a.\breve{\mu}_*\ E\ a)$ to $\mu_*\ E$ implemented using

[2]It is more than an isomorphism since we want to preserve the structure as well. But, for simplicity, we will just say isomorphism here.

$\mathbf{msfit}_*$, where E is a base structure for the untyped HOAS. Since we have two fixpoint type operators, $\breve{\mu}_*$ and μ_*, we can define two recursive datatypes from E: *Exp* defined as $(\forall a.\breve{\mu}_*\ E\ a)$ and *Expr* defined as $\mu_*\ E$. The function $exp2expr :: Exp \to Expr$ implements the mapping from $\breve{\mu}_*$-based HOAS expressions to μ_*-based HOAS expressions. Note, *exp2expr* is defined using $\mathbf{msfit}_*$. This indicates that the mapping from $(\forall a.\breve{\mu}_*\ f\ a)$ to $\mu_*\ f$, for any given f, is admissible within our theory, System F_i.

Figure 10.3 illustrates an incomplete attempt to define a mapping the other direction. Finding a mapping from $(\mu_*\ E)$ to $(\forall a.\breve{\mu}_*\ E\ a)$ turns out to be difficult (perhaps impossible). Instead, we found a possible candidate ($expr2exp'$)[3] for a mapping from *Expr* to $(Exp'\ Expr)$. The codomain $(Exp'\ Expr)$ is an instantiation of $(\forall a.Exp'\ a)$ where a is instantiated to *Expr*. To define $expr2exp'$, we need its inverse function $exp'2expr :: Exp'\ Expr \to Expr$, whose implementation is structurally identical to *exp2expr* in Figure 10.2, but its type signature instantiates a by *Expr*. Note that $exp'2expr$ is defined using $\mathbf{msfit}'$, whose definition is structurally identical to $\mathbf{msfit}_*$, but recurses over values of $\breve{\mu}_*\ f\ a$ rather than $(\forall a.\breve{\mu}_*\ f\ a)$. We can prove that $\mathbf{msfit}'$ always terminates by embedding it into System F_ω (see Figure 10.4). Thus, $exp'2expr$ is admissible within our theory.

Lastly, we define $expr2exp'$ similar in structure to its inverse $exp'2expr$. Instead of an abstract recursive call and an abstract inverse, we use general recursion and the actual inverse function $exp'2expr$. Here, we use general recursion and pattern matching against In_* because we do not know of a Mendler-style recursion scheme to define $expr2exp'$. We need further investigation on whether $expr2exp'$ would always terminate and whether it is possible to make it work for *Exp* rather than $Exp'\ Expr$.

Let us contemplate on why the coercion from $(\forall a.\breve{\mu}_*\ E\ a)$ to $\mu_*\ E$ exists,

[3] We ended up using general recursion while defining $expr2exp'$. So, we do not know whether $expr2exp'$ is total.

$$
\begin{array}{l}
\textbf{data}\ E\ r = Lam\ (r \to r) \mid App\ r\ r \\
\textbf{type}\ Expr = \mu_*\ E \\
\textbf{type}\ Exp'\ a = \breve{\mu}_*\ E\ a \\
\textbf{type}\ Exp = (\forall a.Exp'\ a) \quad \text{-- i.e., } (\forall a.\breve{\mu}_*\ E\ a) \\
exp2expr :: Exp \to Expr \quad \text{-- i.e., } (\forall a.\breve{\mu}_*\ E\ a) \to \mu_*\ E \\
exp2expr = \textbf{msfit}_*\ \varphi\ \textbf{where} \\
\quad \varphi\ inv\ p2r\ (Lam\ f) \quad\ = \mathsf{In}_*\ (Lam\ ((\lambda x \to p2r\ (f\ (inv\ x))))) \\
\quad \varphi\ inv\ p2r\ (App\ e_1\ e_2) = \mathsf{In}_*\ (App\ (p2r\ e_1)\ (p2r\ e_2))
\end{array}
$$

Figure 10.2: Conversion from $\breve{\mu}$-values to μ-values using **msfit**.

$$
\begin{array}{l}
\textbf{msfit}_* :: (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a) \to (\forall a.\breve{\mu}_*\ f\ a) \to a \\
\textbf{msfit}_*\ \varphi\ r = \textbf{msfit}'\ \varphi\ r \\
\textbf{msfit}' :: (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a) \to \quad \breve{\mu}_*\ f\ a \to a \\
\textbf{msfit}'\ \varphi\ (\breve{\mathsf{In}}_*\ x) \quad\ = \varphi\ \mathsf{Inverse}_*\ (\textbf{msfit}'\ \varphi)\ x \\
\textbf{msfit}'\ \varphi\ (\mathsf{Inverse}_*\ z) \quad = z \\
exp'2expr :: Exp'\ Expr\ Expr \to Expr \quad \text{-- i.e., } \breve{\mu}_*\ E\ (\mu_*\ E) \to \mu_*\ E \\
exp'2expr = \textbf{msfit}'\ \varphi\ \textbf{where} \\
\quad \varphi\ inv\ p2r\ (Lam\ f) \quad\ = \mathsf{In}_*\ (Lam\ ((\lambda x \to p2r\ (f\ (inv\ x))))) \\
\quad \varphi\ inv\ p2r\ (App\ e_1\ e_2) = \mathsf{In}_*\ (App\ (p2r\ e_1)\ (p2r\ e_2)) \\
expr2exp' :: Expr \to Exp'\ Expr\ Expr \quad \text{-- i.e., } \mu_*\ E \to \breve{\mu}_*\ E\ (\mu_*\ E) \\
expr2exp'\ (\mathsf{In}_*\ (Lam\ f)) \quad\ = \breve{\mathsf{In}}_*\ (Lam\ (\lambda x \to expr2exp'\ (f\ (exp'2expr\ x)))) \\
expr2exp'\ (\mathsf{In}_*\ (App\ e_1\ e_2)) = \breve{\mathsf{In}}_*\ (App\ (expr2exp'\ e_1)\ (expr2exp'\ e_2))
\end{array}
$$

Figure 10.3: An incomplete attempt to convert from μ-values to $\breve{\mu}$-values.

$$
\begin{array}{l}
\textbf{msfit} :: (\forall r.(a \to r\ a) \to (r\ a \to a) \to f\ (r\ a) \to a) \to \breve{\mu}_*\ f\ Id\ a \to a \\
\textbf{msfit}\ \varphi\ x = caseSum\ x\ unId\ (\lambda f \to f\ (\varphi\ Id))
\end{array}
$$

Figure 10.4: F_ω encoding of **msfit**′ in Haskell (see with Figure 3.18 on p.139).

but the coercion the other direction is difficult (perhaps impossible) to find. We believe that $\mathbf{msfit}_*$ can express more functions than $\mathbf{mit}_*$ (e.g., *showHOAS* in Figure 3.17). Then, it may be the case that values of $(\forall a.\breve{\mu}_*\ f\ a)$ are in fact more restrictive than the values of $(\mu_*\ f)$. The additional expressiveness of $\mathbf{msfit}_*$ may be a compensation for the restrictions on the value of $(\forall a.\breve{\mu}_*\ f\ a)$. In summary, we strongly believe that $(\forall a.\breve{\mu}_*\ f\ a)$ is a subset of $(\mu_*\ f)$. From this observation, we plan to design Nax with two fixpoints ($\breve{\mu}$ and μ) and built-in support for coercion from inverse-augmented fixpoint types to standard fixpoint types (but not the other direction).

In this section, we discussed what we should consider when using both **mit** and **msfit** together. For some recursion schemes, it is quite trivial to establish a theory for using them together. For instance, there is no problem using **mpr** together with **mit** since **mpr** subsumes **mit** — we can implement **mit** in terms of **mpr**. However, for some recursion schemes, such as **mpr** and **msfit**, it is not trivial to establish a theory for using them together. Developing theories for using such recursion schemes together is an important future work.

10.3 MONOTONICITY FROM POLARIZED KINDS

We first review the summary of discussions in Section 5.3 and then list future work on monotonicity and polarized kinds.

Summary of the discussions in Section 5.3

In Section 5.3, we embedded Mendler-style course-of-values recursion into System Fix_i assuming monotonicity. Recall that kinds are polarized in System Fix_i. For instance, $F : p* \to *$ is a type constructor that expects a type argument, whose polarity is p, and returns a type. We discussed that, for a regular recursive datatype $(\mu_* F)$, monotonicity amounts to its base structure $(F : p* \to *)$ being a functor.

When F is a functor, there exists $fmap_F : \forall X^* Y^*.(X \to Y) \to FX \to FY$, which satisfies the desired properties of a functor.

We can generalizes the concept of "being a functor" to type constructors of arbitrary kinds, and such type constructors are called *monotone*. A *monotonicity witness* is a generalization of $fmap_F$, which witnesses F being a functor, and its type is called *monotonicity*, denoted by $mon_\kappa F$. For example, monotonicity for F at kind $*$ is denoted by $mon_* F$, thus, $fmap_F : mon_* F$. More generally, when the type constructor F has more than one argument, there can be more than one notion of monotonicity for F. For example, consider $F : p_1\kappa_1 \to p_2\kappa_2 \to *$. We say that F is monotone on its first argument when $(X_1 \to X_2)$ implies $(FX_1Y \to FX_2Y)$ and that F is monotone on its second argument when $(Y_1 \to Y_2)$ implies $(FXY_1 \to FXY_2)$. One possible notion of monotonicity for F is to require only the first argument be monotone. Another possible notion is to require both of the arguments be monotone.

We discussed in Section 5.3 that there are more than one notion of monotonicity witness at higher kinds. For a non-regular recursive type $(\mu_{p*\to *} F)$, where $F : p_r(p* \to *) \to (p* \to *)$, there are two different notions of monotonicity.

$$
\begin{aligned}
mon_{p*\to *} F = \ & \forall G_1^{p*\to *}.\forall G_2^{p*\to *}.(\forall X.G_1 X \to G_2 X) \to (\forall X.F\, G_1 X \to F\, G_2 X) \\
mon'_{p*\to *} F = \ & \forall G_1^{p*\to *}.\forall G_2^{p*\to *}.mon_* G_1 \to \\
& \qquad (\forall X^*.G_1 X \to G_2 X) \to \\
& \qquad \forall X_1^*.\forall X_2^*.(X_1 \to X_2) \to F\, G_1 X_1 \to F\, G_2 X_2
\end{aligned}
$$

The former, $mon_{p*\to *} F$, requires F to be monotone on its first argument, which is the recursive argument. We discussed that $mon_{p*\to *} F$ is sufficient for the embedding of **mcvpr** over non-truly nested datatypes, such as powerlists.

The latter, $mon'_{p*\to *} F$, requires F to be monotone on both arguments (i.e., both the recursive argument and the index argument). We discussed that we need this stronger notion of monotonicity in order to embed **mcvpr** over truly

nested datatypes, such as bushes, whose index involves the recursive argument in its definition.

Future work on deriving monotonicity from polarized kinds.

According to the embedding of $\mathbf{mcvpr}_\kappa$ in Section 5.3, one needs to witness monotonicity of F to ensure that $\mathbf{mcvpr}_\kappa$ always terminates for $(\mu_\kappa F)$-values. That is, one must show the existence of $mon_\kappa F$ with the desired properties to use $\mathbf{mcvpr}_\kappa$ with a termination guarantee. However, it is not desirable to require programmers to manually derive $mon_\kappa F$ for each F. It more desirable for the language implementation to automatically derive a monotonicity witness for F. It would be even better if the language type system can guarantee the existence of a monotonicity witness by examining the polarized kind of F, rather than actually deriving monotonicity for F by examining its definition.

For System F, it is known that $mon_* F$ exists for any positive F (i.e., $F : +* \to *$ if given a polarized kind) [60]. However, it is still an open question whether any $F : +* \to *$ is monotone in higher-order polymorphic calculi, such as Fix_i. In Section 5.3, we proved that $mon_* F$ exists for a certain class of $F : +* \to *$, and proof that $mon_* F$ exists for any $F : +* \to *$ in Fix_i is left for future work.

We discussed that there are two notions of monotonicity at kind $p* \to *$, one ($mon_{p*\to*}F$) for non-truly nested datatypes and the other ($mon'_{p*\to*}F$) for truly nested datatypes. We conjecture that $mon_{p*\to*}F$ exists for any non-truly nested $F : +(p* \to *) \to (p* \to *)$, and, that $mon'_{+*\to*}F$ exists for any $F : +(+* \to *) \to (+* \to *)$. Proofs for such conjectures at higher kinds are also reserved for future work.

10.4 KIND POLYMORPHISM AND KIND INFERENCE

We support rank-1 polymorphism at the kind level in our Nax implementation. (e.g., *Path* in Figure 7.3 on page 248, *Env* in Figure 7.9 on page 258). However, our theories (System F_i and Fix_i) do not have kind polymorphism. We strongly believe that rank-1 polymorphism at the kind level does not affect normalisation and consistency, but it needs further investigation to confirm our belief.

In the Nax implementation, types are mostly inferred but kinds are always annotated. For example, we must annotate κ in $\mu_{[\kappa]}$ and $\mathsf{In}_{[\kappa]}$, in addition to the kind annotations in datatype declarations (**data** $F : \kappa$ **where** $\cdots$). We believe we only need kind annotations in datatype declarations. We can omit the kind annotations in μ because μ is always followed by a type constructor, μF; we can always infer the kind of F. Similarly, we might be able omit or simplify the kind annotation in In, because In is always followed by a term, $\mathsf{In}\ t$; we can infer the type of t, and we might have enough information to infer the kind for In.

We are also working on a new implementation of Nax with better syntax that supports better kind inference and non-ambiguous fixpoint derivation. In our new implementation, the kind is inferred for μ without any annotation. We have not found a good way to completely infer the kind for In, but we found out that it is enough to specify the arity of the kind. That is, write In_n instead of $\mathsf{In}_{[\kappa]}$ where n is the arity of κ. For example, in the new syntax, we write In_3 instead of $\mathsf{In}_{[*\to*\to*\to*]}$, $\mathsf{In}_{[*\to(*\to*)\to*\to*]}$, or $\mathsf{In}_{[*\to\{\mathtt{Nat}\}\to*\to*]}$, which is much more succinct, especially for larger arities. Another change to the syntax is on the datatype declaration of the GADT form. The syntax in our dissertation (**data** $F : \kappa$ **where** $\cdots$) can be ambiguous for deriving the fixpoint of F. For example, when $F : (* \to *) \to * \to *$, we can either take fixpoint of $\mu F : * \to *$, or $\mu(F\ t) : *$ for some $t : *$. In the current syntax, we simply choose the longest match, that is $\mu F : * \to *$. In the new syntax, we change the datatype declaration syntax, similar to the syntax of

Agda's, to clearly distinguish parameter arguments from index arguments. For instance, (**data** $F_1\ r : * \to *$ **where** $\cdots$) or (**data** $F_0\ t\ r : *$ **where** $\cdots$), where parameter arguments (t, r) appear on the left of the colon (:). Then, we can derive fixpoints without ambiguity, always on the last parameter argument, for instance, we would derive $\mu F_1 : * \to *$ and $\mu(F_0\ t) : *$.

Chapter 11

CONCLUSIONS

In this final chapter, we conclude the dissertation by restating our thesis and summarizing the research in prior chapters (Section 11.1) that support it. Moreover, we emphasize the significance of our contributions (Section 11.2) and outline the limitations of our research (Section 11.3).

11.1 SUMMARY

Our thesis is that a language design based on *Mendler-style recursion schemes* and *term-indexed types* leads to a system in *the sweet spot* that seamlessly unifies functional programming and logical reasoning (via the Curry–Howard correspondence). In Chapter 1, we characterized *the sweet spot* based on the four features that the unified language system should support. They are: (1) a *convenient programming style*, (2) an *expressive logic*, (3) a *small theory*, and (4) a *simple implementation framework*.

In Chapter 2, we reviewed the two well-known polymorphic calculi, System F and System F_ω, to prepare the reader for our term-indexed calculi, System F_i and System Fix_i, in later chapters. We formalized these polymorphic calculi with Curry-style terms and dividing typing contexts into two parts to show that our term-indexed calculi are extensions of System F_ω. We focused on the strong normalization proofs of these systems because the strong normalization proof of System F_i in Chapter 4 relies on the strong normalization of System F_ω. In addition, we reviewed the Hindley-Milner type system to prepare the reader for our discussion

of type inference in Nax (Chapter 8).

Chapter 3 explores Mendler-style recursion schemes, their hierarchical organization, and their termination behaviors. We use Haskell to model the behavior of the recursion schemes, write examples that illustrate characteristics of each of the recursion schemes, and provide a semi-formal termination proof for some of them. We used Haskell for two purposes.

The first is the availability of a type-correct syntax, an executable platform for fast prototyping of examples, and a mature development environment of GHC for experimenting with new ideas. We use a certain subset of Haskell that conforms to the Mendler-style conventions. The discovery of our new Mendler-style recursion combinator (**msfit**) was suggested by this method of experimentation.

The second purpose is the use of a subset of Haskell as an implementation of System F_ω. We illustrated a semi-formal termination proof of $\mathbf{mit}_*$ and $\mathbf{msfit}_*$ by embedding them into this subset.

We organized the hierarchy of Mendler-style recursion schemes based on two aspects: (1) the abstract operations they support and (2) the kind of a based datatype they operate on.

The first aspect, the abstract operations, categorizes the family of Mendler-style recursion schemes. All Mendler-style recursion schemes support the abstract recursive call, which enables recursive calls over direct subcomponents of the argument value. Mendler-style iteration (**mit**) is the most basic family, supporting only this one abstract operation. Other families of Mendler-style recursion schemes additionally support their own characteristic operations. Mendler-style primitive recursion (**mpr**) additionally supports an abstract cast operation, which enables the programmer to access direct sub-components by casting from abstract recursive values to concrete recursive values. Mendler style course-of-values iteration and recursion (**mcvit** and **mcvpr**) additionally support the abstract out operation,

which enables access to deeper subcomponents.

We also discovered a new Mendler-style recursion scheme **msfit**, iteration with syntactic inverses, which additionally supports abstract inverse operation. To support this abstract inverse operation, we needed to augment the fixpoint type operator with a syntactic inverse. We denoteds this abstract inverse as $\breve{\mu}$, to distinguish it from the standard fixpoint type operator μ. We have discussed the ramifications of having two different fixpoint types in Chapter 10.

The second aspect, the kinds of datatypes operated on, categorizes the indexing structure of the recursive datatype within each family. Each family of Mendler-style recursion schemes is a collection of many recursion combinators, one at each kind. For instance, $\mathbf{mit}_{*}$ iterates over regular datatypes with no type index (i.e., $\mu_{*}F$ where $F : * \to *$), $\mathbf{mit}_{*\to*}$ iterates over datatypes with one type index (i.e., $\mu_{*\to*}F$ where $F : (* \to *) \to (* \to *)$), and, more generally, $\mathbf{mit}_{\kappa}$ iterates over recursive datatypes of the form $\mu_{\kappa}F$ where $F : \kappa \to \kappa$. Mendler-style recursion schemes are uniformly defined regardless of the kinds of datatypes they operate on. That is, the definition of $\mathbf{mit}_{\kappa}$ is identical regardless of κ, only its type signatures depend on κ. Uniform definitions, regardless of indexing structure, is one of the advantages of Mendler style over conventional (or, Squiggol) style. This advantage allowed us to discover that simply-typed Higher-Order Abstract Syntax (HOAS) evaluation can be expressed within System F_{ω}. We were able to write a simply-typed HOAS evaluator using $\mathbf{msfit}_{*\to*}$.

The indexing structure discussed in Chapter 3 is restricted to type indices (as opposed to term indices). To formulate Mendler-style recursion schemes over term-indexed datatypes, we need to extend kinds. For instance, $\mathbf{mit}_{A\to*}$, where A is a type, cannot be expressed in System F_{ω} because $A \to *$ is not a valid F_{ω}-kind. In later chapters, we extend Mendler-style recursion schemes over term-indexed datatypes by formalizing two term-indexed calculi, which extend System F_{ω} with term indices.

The termination behaviors of Mendler-style recursion schemes can depend on the particular recursion scheme. Some recursion schemes (**mit**, **mpr**, **msfit**) terminate for arbitrary datatypes, while others (**mcvpr**, **mcvit**) terminate only for positive (or, monotone) datatypes. One of our contributions to the study of Mendler style is finding a counterexample for the termination of the course-of-values iteration (and also recursion) over negative datatypes. In Chapter 5, we discussed how to embed **mcvpr** into a strongly normalizing calculi, which is another original contribution to the study of Mendler style.

Chapters 4 and 5 present two term-indexed calculi, System F_i and System Fix_i. Our term-indexed calculi serve as the theoretical basis for understanding the Mendler-style recursion schemes over recursive types with term indices. By embedding Mendler-style recursion schemes in our term-indexed calculi, we know that those recursion schemes always terminate, because our term-indexed calculi are strongly normalizing.

System F_i (Chapter 4) extends System F_ω (which supports type indices) with term indices. Term indices in System F_i are erasable[1] unlike term indices in the dependently-typed calculi. We establish strong normalization and logical consistency of System F_i by term-index erasure, which projects a typing judgement in System F_i into a typing judgement in System F_ω. We have extended the understanding of Mendler-style recursion schemes over term-indexed datatypes. All Mendler-style recursion schemes that are embeddable in System F_ω, (e.g., **mit**, **msfit**), can also be embedded into System F_i.

Similarly, System Fix_i (Chapter 5) extends System Fix_ω with erasable term indices. System Fix_ω is an extension of System F_ω with polarized kinds and equi-recursive fixpoint types. By term-index erasure, well-typed terms in System Fix_i are well-typed in System Fix_ω. Because Fix_ω is known to be strongly normalizing and

[1] Well-typed terms in (Curry style) F_i are well-typed in F_ω

logically consistent, we know that Fix_i is also strongly normalizing and logically consistent. There exists a reduction preserving embedding of the Mendler-style primitive recursion (**mpr**) in System Fix_i. This follows from the embedding of **mpr** in System Fix_ω. In addition, we discovered an embedding of **mcvpr** (although not a reduction preserving embedding) in System Fix_i for monotone (or positive) datatypes. Our embedding of **mcvpr** motivates further research into the open question of whether a monotonicity witness is derivable from the polarized kinds of type constructors.

In Chapters 6, 7, and 8, we introduce the Nax programming language, which is based on our term-indexed calculi, Systems F_i and Fix_i.

Chapter 6 provides a tutorial of Nax. We introduce the syntax and features of the language using small example Nax programs. Nax supports language constructs, which are not directly part of the term-indexed calculi. For example, Nax supports non-recursive datatype declarations and pattern matching over those non-recursive datatypes, a fixpoint type operator ($\mu_{[\kappa]}$) and its constructor ($\mathsf{In}_{[\kappa]}$), and several Mendler-style recursion schemes (**mit**, **mpr**, **mcvpr**, **msfit**) as primitive constructs. Adding these constructs to the language would not invalidate strong normalization or logical consistency, because these constructs are known to be embeddable into term-indexed calculi.

Chapter 7 highlights the design principles of the type system of Nax. Extending the kind syntax with the type indexed arrow kinds ($\{A\} \to \kappa$) is the key design element in Nax for supporting term indices. We compare our approach (Nax, System F_i) of adding term indices with an alternative approach (GHC's datatype promotion) of adding term indices to a polymorphic language. In the alternative approach, types are promoted to kinds (i.e., $\{A\}$ is itself a kind) and terms are promoted to types. Our approach has the advantage of allowing nested term indices (i.e., term indices can have term-indexed types). The alternative approach has the

advantage of allowing data structures that contain types as elements.

We also compare these two approaches with the approach taken in Agda, a dependently-typed language with both universe subtyping and universe polymorphism. Our approach is closely related to Agda's universe subtyping, and the alternative approach (GHC's datatype promotion) is closely related to Agda's universe polymorphism. We made these comparisons by programming extended examples of a type-preserving evaluator and a stack-safe compiler in three different languages: Nax, Haskell (with datatype promotion), and Agda. These examples also show that Nax supports certain levels of programming convenience. Each of the programs written in each of the three different languages were about the same size, despite the fact that Nax must define recursive types in two levels by taking a fixpoint of a non-recursive datatype.

Chapter 8 describes the type inference in Nax. To support Hindley–Milner-style type inference, Nax only allows rank-1 type polymorphism. One cannot generally infer types in System F_i or Fix_i since they allow higher-rank polymorphism. For programs involving only regular (i.e., non-indexed) datatypes, type inference is exactly the same as Hindley–Milner type inference, requiring no type annotations. For programs involving indexed datatypes, we require type annotations on datatype declarations, case expressions, and Mendler-style recursion combinators, but nowhere else. Our current implementation requires kind annotations on the fixpoint type operator ($\mu_{[\kappa]}$) and its constructor ($\mathsf{In}_{[\kappa]}$), but we believe these kind annotations can be inferred.

Chapter 9 discusses five categories of related work: Mendler-style co-recursion schemes for possibly infinite structures, Mendler-style recursion schemes over multiple recursive values, dependently-typed Mendler-style induction, sized-types and Mendler style, and a comparison of our approach with logical frameworks.

Chapter 10 summarizes some ongoing and future work. We are designing a

new Mendler-style recursion scheme useful for negative datatypes and studying the relationship between two different fixpoint types (μ and $\breve{\mu}$). We plan to investigate the derivation of monotonicity from polarized kinds. Moreover, we want to find a rigorous proof for the assertion that rank-1 kind polymorphism does not break logical consistency. Finally we would like to implement kind inference in Nax.

11.2 SIGNIFICANCE

Our main contribution is a logically consistent language design that supports all recursive datatypes available in functional programming languages such as Haskell, and in addition, term-indexed types. Our design steers the narrow path between convenient programming and strong guarantees of program invariants, while taking advantages of the strong points of both. Our language, Nax, is based on a small theory and admits a simple implementation framework.

Our investigations into Mendler style uncovered two new aspects. First, we discovered a useful recursion scheme (**msfit**) for negative datatypes The discovery of **msfit** lead to the novel discovery that simply-typed HOAS evaluation is expressible within System F_ω. Second, we generalized Mendler-style recursion schemes over term-indexed datatypes. Generalization over term-indexed datatypes were established by the formalization of our term-indexed calculi (System F_i and Fix_i), which extend the polymorphic calculi (System F_ω and Fix_ω).

Our term-indexed calculi are small theories that can embed indexed datatypes and their (Mendler-style) recursion schemes. That is, we do not need to extend the calculi with primitive datatypes for theoretically modeling a practical language. Datatypes and Mendler-style recursion schemes in Nax can be embedded into our term-indexed calculi.

The Nax language implementation does not need an extra termination checker because its termination is type-based. Once the program type checks, we know that it terminates because Nax programs can be embedded into the term-indexed

calculi, which are strongly normalizing.

In addition, Nax supports a conservative extension of the Hindley-Milner type inference in the presence of both type- and term-indices. This is made possible by clarifying the required annotation sites in the programming language syntax, rather than by ad-hoc type reconstruction from optional annotations appearing at arbitrary locations. For the programs involving indexed types, we require annotations on datatype declarations and their eliminators (i.e., case expressions and Mendler-style recursion combinators), but nowhere else.

11.3 LIMITATIONS AND FUTURE WORK

We summarise several limitations of our term-indexed calculi and the Nax language design.

We implemented rank-1 *kind* polymorphism, for the type constructors defined by the top level datatype declarations, in the Nax language implementation. However, our term-indexed calculi do not have any form of kind polymorphism. We strongly believe that rank-1 *kind* polymorphism for those type constructors should not cause inconsistency, but further investigation is needed.

Nax does not yet have type equality built in. We know that we can encode Leibniz equality over both types and terms in System F_i (see Section 4.2.3). However, we cannot define Leibniz equality as a user defined datatype in Nax because the definition of Leibniz equality requires higher-rank polymorphism. We can, of course, have a built-in Leibniz equality as a primitive construct in Nax. We know that Leibniz equality over types have been useful in the context of higher-order polymorphic lambda calculus [95]. Leibniz equality over term indices is definable in the same manner as Leibniz equality over types, and can be built-in to Nax. However, we are not yet sure how useful Leibniz equality over term indices would be because it does not automatically give us induction principles, which are usually expected for a more powerful provable equality over terms (e.g., proving symmetry

of natural number addition). Further studies are needed for such powerful term index equality.

Mendler-style course-of-values recursion only terminates for monotone recursive types. We conjecture that we can derive monotonicity from kinds, but it is an open question as to whether doing so is a sound method. We need further theoretical investigation into this difficult problem. Meanwhile, we plan to support ad-hoc methods of deriving monotonicity by analyzing the syntactic structure of datatype definitions in Nax.

Nax does not support datatypes that contain types (e.g. $[Int, Bool]$). This is often useful for datatype generic programming. We can work around this by reflecting a certain subset of types as term representations of types (a.k.a. type universes). We plan to investigate whether we can extend Nax with first-class datatype descriptions [27] that enable representing arbitrary types as terms.

Nax currently does not support any syntax for optional type annotations. Because types can be completely inferred (with the exception of for index transformer annotations required on case expressions and Mendler-style recursion combinators), including such support did not seem necessary. However, optional type annotations can be useful for documentation purposes, especially for global definitions, which are often reused as library functions in many other places.

We are thinking about supporting some implicit coercions (e.g. *cast* abstract operation of **mpr**) in Nax to make the code more concise. This would allow Nax programs to look even more similar to the code using general recursion. Similarly, we can also support implicit conversion from the indexed augmented recursive types ($\breve{\mu}$-values) to the standard recursive types (μ-values).

BIBLIOGRAPHY

[1] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types, February 15 2012. URL `http://arxiv.org/abs/1202.3496`. Comment: In Proceedings FICS 2012, arXiv:1202.3174.

[3] Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004. ISBN 3-540-23024-6. URL `http://dx.doi.org/10.1007/978-3-540-30124-0_17`.

[4] 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.

[5] 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. doi: DOI:10.1016/j.tcs.2004.10.017.

[6] Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011.

ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034807. URL http://doi.acm.org/10.1145/2034773.2034807.

[7] Ki Yung Ahn, Tim Sheard, Marcelo Fiore, and Andrew M. Pitts. System Fi: a higher-order polymorphic lambda calculus with erasable term indices. In *Proceedings of the 11th international conference on Typed lambda calculi and applications*, TLCA '13, 2013.

[8] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[9] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004. URL http://dx.doi.org/10.1017/S0960129503004122.

[10] Bird and Meertens. Nested datatypes. In *MPC: 4th International Conference on Mathematics of Program Construction*. LNCS, Springer-Verlag, 1998.

[11] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:11–2, 1999.

[12] Richard S Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[13] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[14] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[15] Edwin Brady. IDRIS —: systems programming meets full dependent types. In *PLPV*, pages 43–54. ACM, 2011. ISBN 978-1-4503-0487-0.

[16] Edwin Brady and Kevin Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundam. Inform*, 102(2):145–176, 2010.

[17] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 90–104. ACM, 2002. ISBN 1-58113-605-6. doi: 10.1145/581690.581698.

[18] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[19] 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. ISBN 978-1-59593-919-7. doi: http://doi.acm.org/10.1145/1411204.1411226. URL `http://doi.acm.org/10.1145/1411204.1411226`.

[20] Alonzo Church. A set of postulates for the foundation of logic (2nd paper). *Annals of Mathematics*, 34(4):839–864, October 1933.

[21] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[22] Alonzo Church. The calculi of lambda-conversion. *Annals of Mathematical Studies*, 6, 1941.

[23] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, August 1986.

[24] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP '07*, volume 4421 of *LNCS*. Springer, 2007. ISBN 978-3-540-71314-2.

[25] 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.

[26] Thierry Coquand and Gérard Huet. The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France, May 1986.

[27] Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 103–114, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364544. URL `http://doi.acm.org/10.1145/2364527.2364544`.

[28] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Also published as Technical Report CST-33-85, Department of Computer Science.

[29] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: 10.1145/582153.582176. URL `http://doi.acm.org/10.1145/582153.582176`.

[30] 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.

[31] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 117–130. ACM, 2012. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364522.

[32] 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, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237792. URL `http://doi.acm.org/10.1145/237721.237792`.

[33] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 112–121. ACM, 2007. ISBN 978-1-59593-620-2. doi: 10.1145/1244381.1244400. URL `http://doi.acm.org/10.1145/1244381.1244400`.

[34] Jacques Garrigue and Jacques Le Normand. Adding GADTs to OCaml: the direct approach. In *ML '11: Proceedings of the 2011 ACM SIGPLAN workshop on ML*. ACM, 2011.

[35] Gerhard Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in [36].

[36] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969. Translation of [35].

[37] 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.

[38] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer, 1994. ISBN 3-540-60579-7. URL `http://dx.doi.org/10.1007/3-540-60579-7_2`.

[39] Herman Geuvers. Induction is not derivable in second order dependent type theory. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 166–181, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3-540-41960-8. URL `http://dl.acm.org/citation.cfm?id=1754621.1754639`.

[40] 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.

[41] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de lâĂŹarithmétique dâĂŹordre supérieur*. PhD thesis, PhD thesis, Universit e Paris VII, 1972.

[42] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225 – 230, 1981. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(81)90040-2. URL `http://www.sciencedirect.com/science/article/pii/0304397581900402`.

[43] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[44] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[45] Susumu Hayashi. Singleton, union and intersection types for program extraction. In *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in LNCS, pages 701–730. Springer, September 1991.

[46] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, April 1993. ISSN 0164-0925. doi: 10.1145/169701.169692. URL `http://doi.acm.org/10.1145/169701.169692`.

[47] Jacques Herbrand. *Recherches sur la Théorie de la Démonstration*. PhD thesis, University of Paris, 1930.

[48] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[49] 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.

[50] W. A. Howard. To H.B. Curry: The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1969.

[51] Antonius J. C. Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculus and Applications*, pages 266–278, 1995.

[52] Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *POPL*, pages 297–308, 2008.

[53] Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci*, 174(7):79–104, 2007.

[54] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488.

[55] Chuan-kai Lin. *Practical Type Inference for GADT Type System*. PhD thesis, Department of Computer Science, Portland State University, Portland, Oregon, USA, 2010.

[56] Yitzhak Mandelbaum and Aaron Stump. GADTs for the OCaml masses. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*. ACM, 2009.

[57] Clare E. Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1): 19–35, 2004.

[58] 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.

[59] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians Universität, May 1998.

[60] 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`.

[61] Ralph Matthes. An induction principle for nested datatypes in intensional type theory. *Journal of Functional Programming*, 19(3-4):439–468, June 2009.

[62] Conor Thomas McBride. Agda-curious?: an exploration of programming with dependent types. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 1–2. ACM, 2012. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364529.

[63] 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, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: http://doi.acm.org/10.1145/224164.224225. URL `http://doi.acm.org/10.1145/224164.224225`.

[64] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, pages 30–36, 1987.

[65] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.

[66] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[67] Alexandre Miquel. A model for impredicative type systems, universes, intersection types and subtyping. In *LICS*, pages 18–29. IEEE Computer Society, 2000.

[68] Alexandre Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001.

[69] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure

type systems. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[70] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, UK, 1984. Springer-Verlag. ISBN 3-540-12925-1. URL `http://dl.acm.org/citation.cfm?id=647326.721798`.

[71] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[72] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly, August 2008.

[73] Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: Mixing dependent types and Hindley-Milner type inference. Technical report, Rice University, 2006. URL `http://www.metaocaml.org/concoqtion/`.

[74] Ross Paterson. Control structures from types. Unpublished draft, 1993.

[75] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *LNCS*, pages 328–345. Springer, 1993. ISBN 3-540-56517-5.

[76] 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, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54010. URL `http://doi.acm.org/10.1145/53990.54010`.

[77] Frank Pfenning. Logical frameworks—A brief introduction. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, volume 62 of *NATO Science Series II*, pages 137–166. Kluwer Academic Publishers, 2002. Lecture notes from the Marktoberdorf Summer School, July 2001.

[78] Brigitte Pientka. Beluga: programming with dependent types, contextual data, and contexts. In *Proceedings of the 10th international conference on Functional and Logic Programming*, FLOPS'10, pages 1–12, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12250-7, 978-3-642-12250-7. doi: 10.1007/978-3-642-12251-4_1. URL `http://dx.doi.org/10.1007/978-3-642-12251-4_1`.

[79] Adam Brett Poswolsky. *Functional programming with logical frameworks*. PhD thesis, New Haven, CT, USA, 2008. AAI3342732.

[80] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study*, volume 3 of *Stockholm Studies in Philosophy*. Almqvist & Wiksell, Stockholm, 1965.

[81] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[82] Chung-Chieh Shan. A static simulation of dynamic delimited control. *Higher Order Symbol. Comput.*, 20:371–401, December 2007. ISSN 1388-3690. doi: 10.1007/s10990-007-9010-4.

[83] Tim Sheard. Languages of the future. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 116–119. ACM, 2004. ISBN 1-58113-833-4. doi: 10.1145/1028664.1028711.

[84] Tim Sheard. Putting curry-howard to work. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 74–85, New York, NY,

USA, 2005. ACM. ISBN 1-59593-071-X. doi: http://doi.acm.org/10.1145/1088348.1088356. URL http://doi.acm.org/10.1145/1088348.1088356.

[85] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, September 2004. ISSN 0956-7968. doi: 10.1017/S095679680300488X. URL http://dx.doi.org/10.1017/S095679680300488X.

[86] Tim Sheard, Jim Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. URL http://cs.pdx.edu/~sheard/.

[87] W. W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lectures Notes in Mathematics*, pages 240–251, Boston, 1975. Springer-Verlag.

[88] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.0.1*, 2010. URL http://www.haskell.org/ghc/.

[89] Tarmo Uustalu. *Natural Deduction for Intuitionistic Least and Greatest Fixedpoint Logics, with an Application to Program Construction*. PhD thesis (Dissertation TRITA-IT AVH 98:03), Dept. of Teleinformatics, Royal Inst. of Technology, Stockholm, May 1998.

[90] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

[91] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica, Lith. Acad. Sci*, 10(1):5–26, 1999.

[92] Tarmo Uustalu and Varmo Vene. Coding recursion à la Mendler (extended

abstract). In Johan Jeuring, editor, *Proc. of 2nd Workshop on Generic Programming*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., pages 69–85. 2000.

[93] Tarmo Uustalu and Varmo Vene. The recursion scheme from the cofree recursive comonad. *Electr. Notes Theor. Comput. Sci*, 229(5):135–157, 2011. URL `http://dx.doi.org/10.1016/j.entcs.2011.02.020`.

[94] 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.

[95] Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming*, 20(02):175–210, 2010.

[96] 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, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944728. URL `http://doi.acm.org/10.1145/944705.944728`.

[97] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 249–257. ACM, 1998. ISBN 0-89791-987-4. doi: 10.1145/277650.277732.

[98] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 224–235. ACM, 2003. ISBN 1-58113-628-5. doi: 10.1145/604131.604150.

[99] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Petro Magalh aes. Giving Haskell a promotion. In *Typed Languages Design and Implementation (TLDI'12)*, Jan 2012.

INDEX

Appendix A

THE PROOF FOR COMPLETENESS OF W

Proof of Theorem 2.4.4:

For any Γ and t, there exist S', where $\mathrm{dom}(S') \subseteq \mathrm{FV}(\Gamma)$, and A' such that

$$\frac{S'\Gamma \vdash_{\mathsf{s}} t : A'}{W(\Gamma, t) \rightsquigarrow (S, A_W) \ \wedge \ \exists R. \left(S'\Gamma = R(S\Gamma) \ \wedge \ R(\overline{S\Gamma}(A_W)) \sqsubseteq A'\right)}$$

Proof. By induction on recursive call step of the algorithm W.

case (x) From the $\textsc{Var}_s$ rule, we know that $S'\sigma \in S'\Gamma$, where $\sigma \in \Gamma$, and $S'\sigma \sqsubseteq A'$. By definition of $\sqsubseteq$, A' has the form $S'A[B_1/X_1]\cdots[B_n/X_n]$ where $\sigma = \forall X_1 \ldots X_n.A$.

From $\textsc{Var}_W$ rule, we know that $S = \emptyset$ and $A_W = A[X'_1/X_1]\cdots[X'_n/X_n]$ where $X'_1, \ldots, X'_n$ are fresh.

Let $R = S'$, then, we are done.

case $(\lambda x.t)$ We want to show that

$$\frac{S'\Gamma \vdash_{\mathsf{s}} \lambda x.t : A \to B}{\begin{array}{c} W(\Gamma, \lambda x.t) \\ \rightsquigarrow (S, SX \to B_W) \end{array} \ \wedge \ \exists R. \begin{pmatrix} S'\Gamma = R(S\Gamma) \ \wedge \\ R(\overline{S\Gamma}(SX \to B_W)) \sqsubseteq A \to B \end{pmatrix}}$$

Without loss of generality, we can choose $A = X$, since we can choose S' accordingly such that $S'X = A$. Then, we have

$$\frac{S'\Gamma \vdash_{\mathsf{s}} \lambda x.t : S'X \to B}{\begin{array}{c} W(\Gamma, \lambda x.t) \\ \rightsquigarrow (S, SX \to B_W) \end{array} \ \wedge \ \exists R. \begin{pmatrix} S'\Gamma = R(S\Gamma) \ \wedge \\ R(\overline{S\Gamma}(SX \to B_W)) \sqsubseteq S'X \to B \end{pmatrix}}$$

By induction, we know that

$$\frac{S'(\Gamma, x : X) \vdash_{\mathsf{s}} t : B}{W((\Gamma, x : X), t) \rightsquigarrow (S, B_W) \ \wedge \ \exists R. \begin{pmatrix} S'(\Gamma, x : X) = R(S(\Gamma, x : X)) \\ \wedge \ R(\overline{S(\Gamma, x : X)}(B_W)) \sqsubseteq B \end{pmatrix}}$$

By Proposition 2.4.3, $S'\Gamma \vdash \lambda x.t : S'X \to B$ is sufficient to assume that $S'(\Gamma, x : X) \vdash_{\mathsf{s}} t : B$.

By applying $\textsc{Abs}_W$ rule to $W((\Gamma, x : X), t) \rightsquigarrow (S, B_W)$ where X fresh, we have $W(\Gamma, \lambda x.t) \rightsquigarrow (S, SX \to B_W)$.

From $S'(\Gamma, x : X) = R(S(\Gamma, x : X))$, we know that $S'\Gamma = R(S\Gamma)$ and $S'X = R(SX)$.

If we can show that $R(\overline{S\Gamma}(SX \to B_W)) \sqsubseteq S'X \to B$, we are done. Since $R(\overline{S\Gamma}(SX \to B_W)) = R(\overline{S\Gamma}(SX)) \to R(\overline{S\Gamma}(B_W))$, what we need to show are $R(\overline{S\Gamma}(SX)) \sqsubseteq S'X$ and $R(\overline{S\Gamma}(B_W)) \sqsubseteq B$. The former is true by Proposition 2.4.2 and the facts that $S'X = R(SX)$ and $X \notin \mathrm{dom}(\Gamma)$ since X is fresh: $R(\overline{S\Gamma}(SX)) = R(S(\overline{\Gamma}(X))) = R(S(X)) = S'X \sqsubseteq S'X$. The latter is true since $R(\overline{S\Gamma}(B_W)) \sqsubseteq R(\overline{S(\Gamma, x : X)}(B_W)) \sqsubseteq B$.

case $(t\ s)$ We want to show that

$$\frac{S'\Gamma \vdash_{\mathsf{s}} t\ s : B}{\begin{array}{c} W(\Gamma, t\ s) \\ \rightsquigarrow (S_3 \circ S_2 \circ S_1, S_3X) \end{array} \ \wedge \ \exists R. \begin{pmatrix} S'\Gamma = R((S_3 \circ S_2 \circ S_1)\Gamma) \ \wedge \\ R(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3X)) \sqsubseteq B \end{pmatrix}}$$

Note that we can use S_3X instead of $(S_3 \circ S_2 \circ S_1)X$ since $X \notin \mathrm{dom}(S_2) \cup \mathrm{dom}(S_1)$ because X has been picked fresh after S_2 and S_1 has been computed in $\textsc{App}_W$ rule. So, $(S_3 \circ S_2 \circ S_1)X = S_3X$.

Since $S'\Gamma = R((S_3 \circ S_2 \circ S_1)\Gamma)$, we can replace $S'\Gamma$ with $S''((S_3 \circ S_2 \circ S_1)\Gamma)$ without loss of generality. Then, what we want to show is

$$\frac{S''((S_3 \circ S_2 \circ S_1)\Gamma) \vdash_{\mathsf{s}} t\ s : B'}{\begin{array}{c} W(\Gamma, t\ s) \\ \rightsquigarrow (S_3 \circ S_2 \circ S_1, S_3X) \end{array} \ \wedge \ \exists R. \begin{pmatrix} S''((S_3 \circ S_2 \circ S_1)\Gamma) = R((S_3 \circ S_2 \circ S_1)\Gamma) \\ \wedge \ R(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3X)) \sqsubseteq B' \end{pmatrix}}$$

(A.0.1)

By induction and ($\textsc{App}_s$), we know that

$$\frac{S_1'\Gamma \vdash_{\mathsf{s}} t : A_t}{\begin{array}{c} W(\Gamma, t) \\ \rightsquigarrow (S_1, A_1) \end{array} \ \wedge \ \exists R_1. \begin{pmatrix} S_1'\Gamma = R_1(S_1\Gamma) \ \wedge \\ R_1(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_t \end{pmatrix}} \tag{A.0.2}$$

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_s}{\begin{array}{c} W(S_1\Gamma, s) \\ \rightsquigarrow (S_2, A_2) \end{array} \ \wedge \ \exists R_2. \begin{pmatrix} S_2'(S_1\Gamma) = R_2(S_2(S_1\Gamma)) \ \wedge \\ R_2(\overline{S_2(S_1\Gamma)}(A_2)) \sqsubseteq A_s \end{pmatrix}} \tag{A.0.3}$$

From $S_1'\Gamma = R_1(S_1\Gamma)$ in the conclusion of (A.0.2), we can replace $S_1'\Gamma$ with $S_2'(S_1\Gamma)$ in (A.0.2) without loss of generality, as follows:

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} t : A_t}{\begin{array}{c} W(\Gamma, t) \\ \rightsquigarrow (S_1, A_1) \end{array} \ \wedge \ \exists R_1. \begin{pmatrix} S_2'(S_1\Gamma) = R_1(S_1\Gamma) \ \wedge \\ R_1(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_t \end{pmatrix}}$$

From $S_2'(S_1\Gamma) = R_1(S_1\Gamma)$, R_1 must be a substitution equivalent to S_2' for all free type variables of $S_1\Gamma$. That is, $\mathrm{dom}(S_2') \subseteq \mathrm{dom}(R_1)$ and $S_2'X = R_1X$ for any $X \in \mathrm{dom}(S_2')$. Note that $S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq R_1(\overline{S_1\Gamma}(A_1))$. So, we can choose $R_1 = S_2'$ without loss of generality, as follows:

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} t : A_t}{\begin{array}{c} W(\Gamma, t) \\ \rightsquigarrow (S_1, A_1) \end{array} \ \wedge \ \begin{pmatrix} S_2'(S_1\Gamma) = S_2'(S_1\Gamma) \ \wedge \\ S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_t \end{pmatrix}}$$

Removing the trivial equation $S_2'(S_1\Gamma) = S_2'(S_1\Gamma)$ from above, we have

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} t : A_t}{W(\Gamma, t) \rightsquigarrow (S_1, A_1) \ \wedge \ S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_t} \tag{A.0.4}$$

Similarly, from (A.0.2), we know that

$$\frac{S_3'(S_2(S_1\Gamma)) \vdash_{\mathsf{s}} s : A_s}{W(S_1\Gamma, s) \rightsquigarrow (S_2, A_2) \wedge S_3'(\overline{S_2(S_1\Gamma)})A_2)) \sqsubseteq A_s} \tag{A.0.5}$$

We can choose $S_3' = S''' \circ S_3$ and $S_2' = S''' \circ S_3 \circ S_2$. Here, we rely on the fact that S_3 is a most general unifier. Recall that unify(A, B) succeeds when

the two types A and B are unifiable and the resulting subsitutiion is a most general unifier for those two types. If S_3 were not a most general unifier, it might make the closures of A_1 and A_2 too specific so that $\sqsubseteq$ relations A.0.5 no longer hold. So our choice $S'_3 = S''' \circ S_3$ for A.0.5 is a most probable candidate – that is, nothing else could work if this choice doesn't work. The choice $S'_2 = S''' \circ S_3 \circ S_2$ for A.0.4 is made accordingly to match $S'_3 = S''' \circ S_3$. Then, by the syntax drived typing rule ($\textsc{App}_s$), $A_t = A_s \to B'$. Thus, the premises of (A.0.4) and (A.0.5) are sufficient to assume the premise of what we want to prove, by Proposition 2.4.4. Note that left-hand sides of the logical conjuctions in the conclusions, $W(\Gamma, t) \rightsquigarrow (S_1, A_1)$ and $W(S_1\Gamma, s) \rightsquigarrow (S_2, A_2)$, cocincides with the recursive call in the W algorithm ($\textsc{App}_W$), since we are proving by induction on the recursive call step of the algorithm W. All we need to check is that the right-hand sides of ($\wedge$) in the conclusions of (A.0.4) and (A.0.5) are neccessary conditions for the right-hand side of ($\wedge$) in the conclusion of what we want to prove.

Consider the right-hand side of ($\wedge$) in the conclusion of (A.0.4), replacing S'_2 with our choice of $S'_2 = S''' \circ S_3 \circ S_2$:

$$(S''' \circ S_3 \circ S_2)(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_s \to B'$$

We can replace A_1 in terms of A_2 and X as follows:

$$\begin{aligned}
& (S''' \circ S_3 \circ S_2)(\overline{S_1\Gamma}(A_1)) \\
=\ & S'''(\overline{S_3(S_2(S_1\Gamma))}(S_3(S_2A_1))) && \text{by Proposition 2.4.2} \\
=\ & S'''(\overline{S_3(S_2(S_1\Gamma))}(S_3A_2 \to S_3X)) && \text{by unification in } (\textsc{App}_W) \\
=\ & S'''(\overline{S_3(S_2(S_1\Gamma))}(S_3A_2 \to S_3X)) \\
=\ & S'''(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3A_2 \to S_3X)) && \sqsubseteq \quad A_s \to B'
\end{aligned}$$

Since closure operation and substitutions distribute over ($\to$), we have

$$S'''(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3A_2)) \sqsubseteq A_s \wedge S'''(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3X) \sqsubseteq B' \quad \text{(A.0.6)}$$

Consider the right-hand side of ($\wedge$) in the conclusion of (A.0.5):

$$\begin{aligned}
& (S''' \circ S_3)(\overline{S_2(S_1\Gamma)}(A_2)) \\
= \;& S'''(\overline{S_3(S_2(S_1\Gamma))}(S_3A_2))) && \text{by Proposition 2.4.2} \\
= \;& S'''(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3A_2)) && \sqsubseteq \quad A_s
\end{aligned}$$

Note that above is exaclty the same as the left-hand side of ($\wedge$) in A.0.6, which is expected due to the nature the unification.

We are done by choosing $S'' = S'''$ and $R = S'''$ in what we want to show (A.0.1). Consider the right-hand side of ($\wedge$) in the conclusion, replacing both S'' and R with S''':

$$\begin{pmatrix} S'''((S_3 \circ S_2 \circ S_1)\Gamma) = S'''((S_3 \circ S_2 \circ S_1)\Gamma) \\ \wedge \;\; S'''(\overline{(S_3 \circ S_2 \circ S_1)\Gamma}(S_3X)) \sqsubseteq B' \end{pmatrix}$$

Note that left-hand side of ($\wedge$) is trivially true and the right-hand side exactly matches the right-hand side of (A.0.6).

case ($\mathbf{let}\ x = s\ \mathbf{in}\ t$) We want to show that

$$\frac{S'\Gamma \vdash_{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2'}{W(\Gamma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \rightsquigarrow (S_2 \circ S_1, A_2) \;\wedge\; \exists R. \begin{pmatrix} S'\Gamma = R((S_2 \circ S_1)\Gamma) \;\wedge \\ R(\overline{(S_2 \circ S_1)\Gamma}(A_2)) \sqsubseteq A_2' \end{pmatrix}}$$

By induction, we know that

$$\frac{S_1'\Gamma \vdash_{\mathsf{s}} s : A_1'}{W(\Gamma, s) \rightsquigarrow (S_1, A_1) \;\wedge\; \exists R_1. \left(S_1'\Gamma = R_1(S_1\Gamma) \;\wedge\; R_1(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_1'\right)} \tag{A.0.7}$$

$$\frac{S_2'(S_1\Gamma, x : \overline{S_1\Gamma}(A_1)) \vdash_{\mathsf{s}} t : A_2'}{\substack{W((S_1\Gamma, x:\overline{S_1\Gamma}(A_1)), t) \\ \rightsquigarrow (S_2, A_2)} \;\wedge\; \exists R_2. \begin{pmatrix} S_2'(S_1\Gamma, x:\overline{S_1\Gamma}(A_1)) = R_2(S_2(S_1\Gamma, x:\overline{S_1\Gamma}(A_1))) \\ \wedge \quad R_2(\overline{S_2(S_1\Gamma, x:\overline{S_1\Gamma}(A_1))}(A_2)) \sqsubseteq A_2' \end{pmatrix}} \tag{A.0.8}$$

From $S_1'\Gamma = R_1(S_1\Gamma)$ in the conclusion of (A.0.7), we can replace $S_1'\Gamma$ with $S_2'(S_1\Gamma)$ in (A.0.7) without loss of generality, as follows:

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1'}{W(\Gamma, s) \rightsquigarrow (S_1, A_1) \;\wedge\; \exists R_1. \left(S_2'(S_1\Gamma) = R_1(S_1\Gamma) \;\wedge\; R_1(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_1'\right)}$$

From $S_2'(S_1\Gamma) = R_1(S_1\Gamma)$, R_1 must be a substitution equivalent to S_2' for all free type variables of $S_1\Gamma$. That is, $\mathrm{dom}(S_2') \subseteq \mathrm{dom}(R_1)$ and $S_2'X = R_1X$ for any $X \in \mathrm{dom}(S_2')$. Note that $S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq R_1(\overline{S_1\Gamma}(A_1))$. So, we can choose $R_1 = S_2'$ without loss of generality, as follows:

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1'}{W(\Gamma, s) \rightsquigarrow (S_1, A_1) \ \wedge \ \left(S_2'(S_1\Gamma) = S_2'(S_1\Gamma) \ \wedge \ S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_1'\right)}$$

Removing the trivial equation $S_2'(S_1\Gamma) = S_2'(S_1\Gamma)$ from above, we have

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1'}{W(\Gamma, s) \rightsquigarrow (S_1, A_1) \ \wedge \ S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_1'}$$

Using above and Lemma 2.4.1, we have

$$\frac{S_2'(S_1\Gamma), x : A_1' \vdash_{\mathsf{s}} t : A_2' \qquad \dfrac{\dfrac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1'}{S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq A_1'}}{S_2'(S_1\Gamma), x : S_2'(\overline{S_1\Gamma}(A_1)) \sqsubseteq S_2'(S_1\Gamma), x : A_1'}}{S_2'(S_1\Gamma), x : S_2'(\overline{S_1\Gamma}(A_1)) \vdash_{\mathsf{s}} t : A_2'}$$

which can be summarized as

$$\frac{S_2'(S_1\Gamma), x : A_1' \vdash_{\mathsf{s}} t : A_2' \qquad S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1'}{S_2'(S_1\Gamma), x : S_2'(\overline{S_1\Gamma}(A_1)) \vdash_{\mathsf{s}} t : A_2'}$$

By Proposition 2.4.5, we have

$$\frac{\dfrac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2'}{\exists A_1'.\, (S_2'(S_1\Gamma), x : A_1' \vdash_{\mathsf{s}} t : A_2' \ \wedge \ S_2'(S_1\Gamma) \vdash_{\mathsf{s}} s : A_1')}}{S_2'(S_1\Gamma), x : S_2'(\overline{S_1\Gamma}(A_1)) \vdash_{\mathsf{s}} t : A_2'} \tag{A.0.9}$$

Note that the assumption of (A.0.9), $S_2'(S_1\Gamma) \vdash_{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2'$, implies both the assumption of (A.0.7) instantiated by $S_1' = S_2' \circ S_1$ and the assumption (A.0.8). So, we can merge the conclusion of (A.0.7) and the conclusion of (A.0.8) instantiated by $S_1' = S_2' \circ S_1$ in order to synthesize what we want to prove.

Applying $\textsc{Let}_W$ rule to left-hand arguments of $\wedge$ in the conclusions of (A.0.7) and (A.0.8), we get $W(\Gamma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \rightsquigarrow (S_2 \circ S_1, A_2)$.

Let $R_2 = R$ in the right-hand side in the conclusion of (A.0.8). Then, we get $\exists R.\left(S_2'(S_1\Gamma) = R((S_2 \circ S_1)\Gamma) \ \wedge \ R(\overline{(S_2 \circ S_1)\Gamma}(A_2)) \sqsubseteq A_2'\right)$ by similar steps we took for the case ($\textsc{Abs}_s$).

In summary, we get

$$\frac{S_2'(S_1\Gamma) \vdash_{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2'}{\begin{array}{c} W(\Gamma, \mathbf{let}\ x = s\ \mathbf{in}\ t) \\ \rightsquigarrow (S_2 \circ S_1, A_2) \end{array} \ \wedge \ \exists R. \begin{pmatrix} S_2'(S_1\Gamma) = R((S_2 \circ S_1)\Gamma) \ \wedge \\ R(\overline{(S_2 \circ S_1)\Gamma}(A_2)) \sqsubseteq A_2' \end{pmatrix}}$$

which is almost exactly what we want to prove, except that $S_2'(S_1\Gamma)$ is used in place of $S'\Gamma$.

Without loss of generality, we can use $S_2'(S_1\Gamma)$ instead of $S'\Gamma$. By Proposition 2.4.5, $S'\Gamma \vdash_{\mathsf{s}} \mathbf{let}\ x = s\ \mathbf{in}\ t : A_2'$ implies $S'\Gamma \vdash_{\mathsf{s}} s : A_1'$ for some A_1'. Applying (A.0.7) to $S'\Gamma \vdash_{\mathsf{s}} s : A_1'$ with $S_1' = S'$, we have $S'\Gamma = R_1(S_1\Gamma)$ for some R_1. $\square$

Appendix B

PROOFS IN THE METATHEORY OF SYSTEM F_i

This appendix contains proofs of propositions in Section 4.3.

Proof of Proposition 4.3.1:

$$\frac{\vdash \Delta \qquad \Delta \vdash F : \kappa}{\vdash \kappa : \Box}$$

Proof. By induction on the derivation.

case (Var) Trivial by the second well-formedness rule of Δ.

case $(Conv)$ By induction and Lemma 4.3.1.

case (λ) By induction, we know that $\vdash \kappa : \Box$.

By the second well-formedness rule of Δ, we know that $\vdash \Delta, X^{\kappa}$ since we already know that $\vdash \kappa : \Box$ and $\vdash \Delta$ from the property statement.

By induction, we know that $\vdash \kappa' : \Box$ since we already know that $\vdash \Delta, X^{\kappa}$ and that $\Delta, X^{\kappa} \vdash F : \kappa'$ from induction hypothesis.

By the sorting rule (R), we know that $\vdash \kappa \to \kappa' : \Box$ since we already know that $\vdash \kappa : \Box$ and $\vdash \kappa' : \Box$.

case $(@)$ By induction, easy.

case (λi) By induction we know that $\cdot \vdash A : *$. By the third well-formedness rule of Δ, we know that $\vdash \Delta, i^A$ since we already know that $\cdot \vdash A : *$ and that $\vdash \Delta$ from the property statement.

By induction, we know that $\vdash \kappa : \Box$ since we already know that $\vdash \Delta, i^A$ and that $\Delta, i^A \vdash F : \kappa$ from the induction hypothesis.

By the sorting rule (Ri), we know that $\vdash A \to \kappa : \Box$ since we already know that $\cdot \vdash A : *$ and $\vdash \kappa : \Box$.

case $(@i)$ By induction and Proposition 4.3.2, easy.

case $(\to)$ Trivial since $\vdash * : \Box$.

case $(\forall)$ Trivial since $\vdash * : \Box$.

case $(\forall i)$ Trivial since $\vdash * : \Box$. □

The basic structure of the proof for Proposition 4.3.2 on typing derivations is similar to above. So, we illustrate the proof for most of the cases, which can be done by applying the induction hypothesis, rather briefly. We elaborate more on interesting cases $(\forall E)$ and $(\forall Ei)$ which involve substitutions in the types resulting from the typing judgments.

Proof of Proposition 4.3.2:

$$\frac{\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash t : A}{\Delta \vdash A : *}$$

Proof. By induction on the derivation.

case $(:)$ Trivial by the second well-formedness rule of Γ.

case $(: i)$ Trivial by the third the well-formedness rule of Δ.

case $(=)$ By induction and Lemma 4.3.2.

case $(\to I)$ By induction and well-formedness of Γ.

case $(\to E)$ By induction.

case $(\forall I)$ By induction and well-formedness of Δ.

case $(\forall E)$ By induction we know that $\Delta \vdash \forall X^{\kappa}.B : *$.

By the kinding rule $(\forall)$, which is the only kinding rule able to derive $\Delta \vdash \forall X^{\kappa}.B : *$, we know that $\Delta, X^{\kappa} \vdash B : *$.

Then, we use the type substitution lemma (Lemma 4.3.4(1)).

case $(\forall Ii)$ By induction and well-formedness of Δ.

case $(\forall Ei)$ By induction we know that $\Delta \vdash \forall i^A.B : *$.

By the kinding rule $(\forall i)$, which is the only kinding rule able to derive $\Delta \vdash \forall i^A.B : *$, we know that $\Delta, i^A \vdash B : *$.

Then, we use the index substitution lemma (Lemma 4.3.4(2)). $\square$