Department of Computer Science

Elaborating inductive definitions in Curry-style polymorphic type theory

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE COMPREHENSIVE EXAM OF THE PH.D. PROGRAM

by

Christopher Jenkins

✉ christopher-jenkins@uiowa.edu

Advisor

Aaron Stump, Ph.D.

✉ aaron-stump@uiowa.edu

May 14, 2020

Abstract

This report considers the rôle of *elaboration* in providing a translational semantics of programming language features to polymorphic lambda calculi, using as case studies *bidirectional type inference* and *inductive definitions*. *Elaboration* is a term of art describing the process by which programs written in a practical, feature-rich surface language (the *source language*) are translated down into a minimal core calculus or sub-language (the *target language*). It allows the object of study (the target language) to be small, aiding in meta-theoretic analysis, without sacrificing programmer conveniences that may require complex features be implemented in the surface language. Meta-theoretic properties of the target language such as logical soundness and termination can then be imported into the surface language via a *static and dynamic soundness* theorem for the translation. A discipline of design by elaboration can inform the way new features are added, clarifying what is merely notational convenience and what requires true extension to the core theory.

1 Introduction

Programmers love language features: algebraic datatypes, pattern matching and recursion, subtyping, and many others provide notational convenience for problem solving. But, programming languages are themselves complex software artifacts and seemingly unrelated features may have subtle and undesirable interactions. For general-purpose languages these interactions may result in unintuitive edge cases, regressions, or at the worst even runtime crashes or logical errors. For programming languages that serve also as proof assistants or implementations of some type theory, we interpret via the *Curry-Howard correspondence* [SU06] types as *propositions* and programs as *proofs*, and consequences of bugs in such systems may be even as dire as logical inconsistency and thus loss of trust in proofs carried out within them.

A notable example concerns the language Agda and a feature called *dependent pattern matching* [Coq92]. To make dependent pattern matching compatible with *homotopy type theory* (**HoTT**) [Uni13], Agda supports an optional flag `--without-K`, which disables certain unification rules from which the **K** axiom of Streicher [Str93] (incompatible with **HoTT**) could be derived. An earlier implementation of this flag (described in [NAD12]) was demonstrated by Cockx in [Coc14] to be still incompatible with HoTT, requiring further changes to Agda's implementation. Cockx put dependent pattern matching without **K** back on solid theoretical foundations in his dissertation [Coc17] by formulating *elaborating unification rules*. A less alarming (but sill significant) example of unintended feature interactions concerns the Lean theorem prover. The combination of an impredicative *Prop* sort with a strong form of proof-irrelevance (two very desirable features) results in a failure of strong normalization [AC19].

To avoid scenarios such as these, it is desirable to prove that a given type theory satisfies properties such as logical consistency or normalization. This raises a new difficulty: each time language implementors add or extend a feature, must they re-do all meta-theoretic results for the entire system? The answer is *no!* For some features, language designers may instead choose *definitional* language extensions justified by *elaboration*, which is a term of art describing:

1. a high-level **surface (source) language** with convenient features for users;
2. a low-level **internal (target) language**, usually a small core (sub-)language, which is the object of meta-theoretic study;
3. a **translation** from the surface language to the internal language; and
4. a **soundness proof** showing that this translation preserves meaning between the two languages.

Elaboration can thus be seen as a *translational semantics* to surface-language features, and usually requires that such features be only definitional (notational) extensions to the core theory.

1.1 Raison d'être

This report considers the rôle of elaboration for giving a translational semantics of a feature of primary importance in programming languages: *inductive definitions*, by which we mean declarations of algebraic datatypes (ADTs) and functions defined over them by pattern matching and recursion. For the most part, the target language we consider will be the polymorphic lambda calculus (System F) [Gir72, Rey74], considered to be the "gold standard" of expressivity for functional languages. We specifically focus on the *Curry-style* (extrinsically typed) formulation of System F. Our interest in extrinsic type theories arises from our active research into impredicative encodings of datatypes in the Curry-style *calculus of dependent lambda eliminations* (**CDLE**), which is discussed informally in Section 6.

This report is organized as follows. In Section 2, we recall some basic notions of the untyped lambda calculus and System F. In Section 3, we use the relatively simple case study of *bidirectional type inference* to motivate a general formulation of elaborating type inference rules and the *static and dynamic soundness property* of the translation, terminology we have found convenient to coin. In preparation for giving a translational semantics of inductive definitions, in Section 4 we recall their mathematical (algebraic, categorical) semantics and their encodings in polymorphic lambda calculi. In Section 5, we combine all that proceeded to present an extended case study on elaborating polynomial and recursive types, the basic ingredients for inductive definitions. We state the static and dynamic soundness property for the translation, and demonstrate how some meta-theoretic properties of the statics and dynamics of target language (System F) are imported into the source language. Finally, in Section 6 we conclude with a discussion of related work and gesture toward future investigations.

2 Background: Lambda calculi

2.1 Untyped lambda calculus

Grammar The *lambda calculus* was invented (or discovered) by Alonzo Church [Chu41] as part of his research on foundational mathematics, and is often the subject of study in programming language theory as it serves as a minimal specification of a functional language. For untyped lambda calculus, terms u are inductively defined using three constructors (Figure 1): *variables* x, *lambda abstractions* (or *function* abstractions) $\lambda\, x.\, u$, and *applications* $u_1\ u_2$.

We assume that the set of variables V is (countably) infinite, and that given some finite set $X \subseteq V$ there is always a way to pick a variable $x \in V \setminus X$; if we have an x satisfying this criterion, we say that x *is fresh w.r.t* X. When reading lambda terms we adopt the convention that the scope of the lambda extends as far to the right as possible, e.g., the term $\lambda\, x.\, x\ \lambda\, x.\, x$ should be read as $\lambda\, x.\, x\ (\lambda\, x.\, x\ x)$, and application associates to the left, e.g., the term $\lambda\, x.\, x\ x\ x$ should be read $\lambda\, x.\, (x\ x)\ x$.

α-equivalence and free and bound variables The identity function can be equivalently expressed as the lambda terms $\lambda\, x.\, x$ and $\lambda\, y.\, y$. In general, we say that x is *locally bound* in an abstraction $\lambda\, x.\, u$, and that x possibly occurs *free* in u. The variable x may also occur bound in a lambda expression $\lambda\, x.\, u'$ occurring as a sub-expression of u, which should be interpreted

$$\begin{array}{ll} \text{variables } x, y, z, x_1, ... & \\ \text{terms } u, u', u_1, u_2, ... & ::= \quad x \mid \lambda\, x.\, u \mid u_1\ u_2 \end{array}$$

Figure 1: Grammar of the untyped lambda calculus

as *name-shadowing* of the x bound in $\lambda\, x.\, u$. If y is fresh w.r.t $FV(u)$ (the free variables of u), then we say $\lambda\, x.\, u$ is α-equivalent to $\lambda\, y.\, (u[y/x])$, where $u[y/x]$ indicates the *capture-avoiding substitution* of x for y in u. Thus, $\lambda\, x.\, x$ is α-equivalent to $\lambda\, y.\, y$.

The meaning of the stipulation *capture-avoiding* is illustrated as follows. Consider the term $x\ \lambda\, y.\, x$. The variable y is not a free variable of this term, but it is locally bound. After applying a substitution $[y/x]$, we do not wish to have as a result $y\ \lambda\, y.\, y$, which would be an instance of *variable capture*. The result of capture-avoiding substitution for this term should instead be $y\ \lambda\, z.\, y$, renaming the locally bound y to avoid capture. A proper definition of bound and free occurrences of variables, α-equivalence, and capture-avoiding substitution can be found in any reference textbook on lambda calculus and programming languages, such as [Stu13].

β-equivalence and operational semantics What is the *semantics*, or meaning, of a program? A common and rather pragmatic answer is that the meaning of a program is given by the way it *runs* (computes, executes, etc); this is known as the *operational semantics* of a language, specifically its *small-step* operational semantics. We describe this for the untyped lambda calculus. The relation $u \dashrightarrow_\beta u'$ between terms u and u' indicates that u *reduces to* u' in a single computational step. The main rule governing reduction of a term to another term in lambda calculus is the β-rule, given by

$$(\lambda\, x.\, u)\ u' \dashrightarrow_\beta u[u'/x]$$

Lambda expressions of the form $(\lambda\, x.\, u)\ u'$ are called *β-reducible expressions*, or *β-redexes* for short. The reduction relation $\dashrightarrow_\beta$, called *full β-reduction*, is given by the *compatible closure* of the β-reduction rule. That is to say, if u contains a β-redex u_1 as a sub-expression, and $u_1 \dashrightarrow_\beta u_2$ by the β-rule, then we have $u \dashrightarrow_\beta u'$ if replacing that occurrence of the redex u_1 in u with the resulting expression u_2 (the *contractum*) produces u'. For example, in $\lambda\, x.\, \underline{(\lambda\, y.\, y)\ x}$ the underlined sub-expresses is a β-redex, and the whole expression reduces to $\lambda\, x.\, x$ in one step. We notate by $\overset{*}{\dashrightarrow}_\beta$ the reflexive and transitive closure of $\dashrightarrow_\beta$, and by $\cong_\beta$ the symmetric closure of $\overset{*}{\dashrightarrow}_\beta$.

η-equivalence In functional languages we expect that the meaning of a function u is unchanged if we were to wrap u itself within another function which takes an argument and applies u to that argument. An additional useful rule for rewriting lambda terms that captures this intuition is the η-rule,

$$\lambda\, x.\, u\ x \dashrightarrow_\eta u$$

provided $x \notin FV(u)$. Throughout this paper, we will write $\dashrightarrow$ as the compatible closure of the union of the β and η reduction rules presented, write $\overset{*}{\dashrightarrow}$ for the reflexive and transitive closure of $\dashrightarrow$, and $\cong$ for the symmetric closure of $\overset{*}{\dashrightarrow}$.

2.2 Polymorphic type theory

Elaboration in programming languages is closely tied to *type systems*, a popular and effective form of static analysis in programming languages. Even the simplest of type systems can guarantee, for example, that a function which expects an `int` argument is not given a `string` instead. More generally, guarantees such as this are instances of *type safety*, the mantra for which is *well-typed programs cannot "go wrong"* [Mil78]. Type systems may provide other guarantees: in general, a term having any type at all might guarantee *termination* (executing the program with any strategy is guaranteed to produce a value); in type systems rich enough to be interpreted as logics under the Curry-Howard isomorphism, a term of type T corresponds to a proof of the logical interpretation of T. The downside to type systems (at least, non-trivial ones) is that

$$
\begin{array}{lcl}
\text{term variables } x... & & \\
\text{type variables } X, Y, Z, R, X_1, ... & & \\
\text{types } S, T, U, V, T_1, ... & ::= & X \mid S \to T \mid \forall\, X.\, T \\
\text{terms } t, t_1, t_2, t', ... & ::= & x \mid \lambda\, x{:}T.\, t \mid \Lambda\, X.\, t \mid t_1\ t_2 \mid t \cdot T \\
\\
\text{typing contexts } \Gamma, \Delta, ... & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, X
\end{array}
$$

(a) Grammar and typing context formation

$$
\boxed{\Gamma \vdash t : T} \qquad
\frac{FV(T) \subseteq DV(\Gamma) \quad \Gamma, x{:}T \vdash t : S}{\Gamma \vdash \lambda\, x{:}T.\, t : T \to S}\ \textsc{Abs} \qquad
\frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : T}\ \textsc{App}
$$

$$
\frac{(x{:}T) \in \Gamma}{\Gamma \vdash x : T}\ \textsc{Var} \qquad
\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda\, X.\, t : \forall\, X.\, T}\ \textsc{Poly} \qquad
\frac{\Gamma \vdash t : \forall\, X.\, S \quad FV(T) \subseteq DV(\Gamma)}{\Gamma \vdash t\ \cdot T : S[T/X]}\ \textsc{Inst}
$$

(b) Type inference

$$
\begin{array}{rclcrcl}
|x| & = & x, & & |\lambda\, x{:}T.\, t| & = & \lambda\, x.\, |t| \\
|t_1\ t_2| & = & |t_1|\ |t_2|, & & |\Lambda\, X.\, t| & = & |t| \\
|t \cdot T| & = & |t| & & & &
\end{array}
$$

(c) Erasure

Figure 2: Fully annotated System F

sometimes it is either not clear how to type some program of interest, or even possible that an interesting program *cannot* be typed within some system!

It is standard to present type systems using *judgments* [ML96], which are relations on the language constructs. For example, the judgment $\Gamma \vdash t : T$ is read "under a typing context Γ, the term t has type T", and can be understood as the assertion that the components of the triple (Γ, t, T) are in a three-place relation $(\_ \vdash \_ : \_) \subseteq \mathit{Contexts} \times \mathit{Terms} \times \mathit{Types}$. Judgments are inductively defined by *inference rules*, a two-dimensional notation whose vertical dimension is separated by a horizontal line for deriving some conclusion (given below the horizontal bar) from zero or more premises (given above the horizontal bar).

We now give such a presentation for System F [Gir72,Rey74]. System F is of great interest to programming language theorists: although its formal description is quite small, it is expressive enough to encode all of second-order Peano arithmetic [Gir71]. This expressivity comes at a cost: the type-assignment problem for System F is undecidable [Wel99]. In order to make type inference algorithmic, a system of typing annotations for terms is required. We provide one such system, the *fully annotated* System F, in Figure 2.

2.2.1 Type-annotated language

Grammar The grammar of annotated System F is given in Figure 2a. As before, $x, y, z, ...$ indicate term variables. In addition, we use meta-variables $X, Y, Z, ...$ to denote *type variables*, and meta-variables $t, t_1, t_2, ...$ to denote terms of annotated System F. Types $S, T, ...$ are comprised of variables, function types $S \to T$, and polymorphic types $\forall\, X.\, T$. Terms of annotated System F are variables (as before), function abstractions $\lambda\, x{:}T.\, t$ wherein the bound variable x is given with a type annotation $(x{:}T)$, type abstractions $\Lambda\, X.\, t$ for introducing polymorphic

terms, applications $t_1\ t_2$ (as before), and type instantiation $t\ \cdot T$. Typing contexts Γ are not in the term or type language of annotated System F, but are required for the type inference rules: a typing context can be empty ($\emptyset$) or can contain term variables of a given type $\Gamma, x:T$ (where in formulating inference rules we should take care that the free type variables of T are declared within the context Γ), and type variables (Γ, X); we assume that all such variables in a context Γ, called the *declared variables of* Γ or $DV(\Gamma)$, are distinct.

Typing rules The typing rules for System F are given in Figure 2b. In listings of inference rules, we use a box to indicate the judgment being inductively defined by those rules. Read rule VAR as saying that a variable x has type T if x is declared to have type T in Γ. Rule ABS says function $\lambda\, x{:}\, T.\, t$ has type $T \to S$ if we can type its body t at type S by extending the given typing context Γ with the assumption $x:T$, assuming the free variables of T are declared in Γ. Rule APP says simply that if function t_1 has type $S \to T$ and t_2 has type S, the application $t_1\ t_2$ has type T. Rule POLY is similar to ABS, and concerns the introduction of polymorphic terms. Rule INST says that if t is a polymorphic term of type $\forall\, X.\, S$, and the free variables of T are declared in Γ, then $t \cdot T$ has type $S[T/X]$, where capture-avoiding substitution replaces all free occurrences of X in S with T.

We illustrate with a simple example: the polymorphic identity function $\Lambda\, X.\, \lambda\, x{:}\, X.\, x$ has type $\forall\, X.\, X \to X$. We can take this identity function, instantiate variable X to $\forall\, X.\, X \to X$, and apply the result to the identity function again: $(\Lambda\, X.\, \lambda\, x{:}X.\, x) \cdot (\forall\, X.\, X \to X)\ \Lambda\, X.\, \lambda\, x{:}X.\, x$, which also has type $\forall\, X.\, X \to X$.

2.2.2 Operational semantics (erasure and Curry-style theories)

What we have just described is the "statics" of System F. From this, it should be clear that the polymorphic lambda calculus corresponds to the implicative fragment of second-order intuitionistic propositional logic. To be considered as a programming language, we need to breathe life into the system by specifying its operational semantics, i.e., the "dynamics". Usually, this is done by reformulating the operational semantics for the untyped lambda calculus for the term language of annotated System F and adding a new reduction rule:

$$(\Lambda\, X.\, t) \cdot T \dashrightarrow t[T/X]$$

However, we find it more pleasant to provide a semantics by *subtraction*. We take the point-of-view that types should play no computational rôle in terms, and that the system of type annotations is merely a syntactic mechanism for algorithmically determining a type assignment for some term of the untyped lambda calculus.

Our erasure rules for annotated System F are given in Figure 2c. Type erasure should be viewed as a method of *program extraction* for proofs. The notion of some "code" underlying a proof in intuitionistic logic goes back to Kleene's *realizability semantics* for intuitionistic number theory [Kle45]. In this view, we have a semantic interpretation of types as sets of ($\alpha\beta\eta$-equivalences classes of) terms of the untyped lambda calculus which realize them. This interpretation underlies Tait's method [Tai75] for proving termination of System F (which is itself a modification of Girard's original "candidats de reductibilité" method [Gir72]). More recently, type erasure has been uses as a feature of Curry-style type theories (rather than being a meta-theoretic notion) like the *implicit calculus of constructions* (**ICC**) [BB08] and **CDLE** [Stu17]. These theories are dependently typed, and thus type-checking may require determining whether two terms are equal; a notion of erasure permits definitional equality of terms "up to erasure" of typing annotations.

Elaborating the language of dynamics We remark that, with respect to designing elaborating inference rules, there is some technical overhead introduced by this point-of-view. We now have *two* source languages (one for the statics, one for the dynamics), and thus *two* target languages. For the design of our bidirectional type system, which is concerned only with the placement of type annotations, this will not prove to be a significant burden. In general, one can extract the rules for the judgment $\vdash u \rightsquigarrow u'$ for elaborating the language of dynamics from the rules given for the language of statics by removing from typing judgments all typing contexts and types, and by replacing the subject t of the judgment and its elaboration t' with their erasures. For the bidirectional system, the resulting elaboration rules for the dynamic language are a no-op.

3 Bidirectional type inference

We now consider a relatively simple case of using elaboration to guide the design of a new language feature, *bidirectional type inference*. Annotated System F, though powerful, is not a language in which one wishes to write programs directly, as it requires excessive discipline of type annotation. For example, if we know that f has type $(S \to S) \to T$ (for some S and T), we should like to write $f\ \lambda\, x.\, x$, rather than $f\ \lambda\, x{:}\, S.\, x$ as would currently be required, because the type that should be associated with the bound variable x in $\lambda\, x.\, x$ is determined already by the type of the function f. Additionally, it would be desirable to have notation for giving a more top-level typing annotation $\chi\ T$ - t to better separate the type (i.e., specification) of, and the term (i.e., implementation) for, a program.

A popular technology for inferring missing type annotations is *Damas-Hindley-Milner type inference* (DHM) [Dam84, Hin69, Mil78]. In its simplest form, DHM guarantees *complete* type inference for a restriction of the polymorphic lambda calculus. This means no type annotations in source programs are ever required. However, and as discussed previously, unrestricted System F is beyond the power of any complete method of inference.

Considerations such as these lead one to search for *partial* methods of type inference (meaning that some annotations will still be required in the source language) such as *bidirectional typechecking*. Bidirectional typechecking, described by Pierce and Turner in [PT00], uses the typing information available *locally* (that is, from neighboring nodes of the abstract syntax tree of the language) to infer missing type annotations in programs. The utility of bidirectional typechecking goes well beyond the modest use we will give here. For example, it can be employed to infer missing type arguments to polymorphic functions [PT00, OZZ01, DK13], to support overloading of datatype constructors [APTS13], and even for giving algorithmic elaboration rules for language constructs like algebraic datatypes [DM12].

Bidirectional typechecking is elegantly described with inference rules by splitting the typing judgment into two forms, *synthesis mode* written $\Gamma \vdash e \in T$ indicating the type T is coming "up and out of" term e, and *checking mode* written $\Gamma \vdash T \ni e$ indicating that T is being pushed "down and into" e. Following the convention of [DM12], for elaborating type inference rules we extend these judgments to $\Gamma \vdash e \rightsquigarrow t \in T$ and $\Gamma \vdash T \ni e \rightsquigarrow t$, where we call t the *elaboration* of e. The intended reading in inference rules is that inputs to type inference go to the left of $\rightsquigarrow$ and outputs go to the right.

3.1 Elaboration

Grammar Recall from Section 1 that elaboration means designating an internal, minimal language and an external, extended language. Our chosen internal language is annotated System F (Figure 2), and our external language, *bidirectional System F*, is given in Figure 3. The language of types, and typing context formation, is the same in both, so we specify only the

$$\text{external terms } e, e_1, e_2, f, \ldots \quad ::= \quad x \mid \lambda\, x{:}T.\, e \mid \lambda\, x.\, e \mid \Lambda\, X.\, e \mid f\ e \mid e \cdot T \mid \chi\ T\ \text{-}\ e$$

(a) Grammar

$$\boxed{\Gamma \vdash e \rightsquigarrow t \in T}$$

$$\frac{FV(T) \subseteq DV(\Gamma) \quad \Gamma, x{:}T \vdash e \rightsquigarrow t \in S}{\Gamma \vdash \lambda\, x{:}T.\, e \rightsquigarrow \lambda\, x{:}T.\, t \in T \to S}\ \textsc{Abs}\in \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow t_1 \in S \to T \quad \Gamma \vdash S \ni e_2 \rightsquigarrow t_2}{\Gamma \vdash e_1\ e_2 \rightsquigarrow t_1\ t_2 \in T}\ \textsc{App}$$

$$\frac{\Gamma, X \vdash e \rightsquigarrow t \in T}{\Gamma \vdash \Lambda\, X.\, e \rightsquigarrow \Lambda\, X.\, t \in \forall\, X.\, T}\ \textsc{Poly}\in \qquad \frac{\Gamma \vdash e \rightsquigarrow t \in \forall\, X.\, S \quad FV(T) \subseteq DV(\Gamma)}{\Gamma \vdash e \cdot T \rightsquigarrow t \cdot T \in [T/X]S}\ \textsc{Inst}$$

$$\frac{}{\Gamma \vdash x \rightsquigarrow x \in \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma \vdash T \ni e \rightsquigarrow t \quad FV(T) \subseteq DV(\Gamma)}{\Gamma \vdash \chi\ T\ \text{-}\ e \rightsquigarrow t \in T}\ \textsc{Ann}$$

$$\boxed{\Gamma \vdash T \ni e \rightsquigarrow t}$$

$$\frac{T_1 = T_2 \quad \Gamma, x{:}T_2 \vdash S \ni e \rightsquigarrow t}{\Gamma \vdash T_1 \to S \ni \lambda\, x{:}T_2.\, e \rightsquigarrow \lambda\, x{:}T_2.\, t}\ \textsc{Abs}\ni \qquad \frac{\Gamma, x{:}T \vdash S \ni e \rightsquigarrow t}{\Gamma \vdash T \to S \ni \lambda\, x.\, e \rightsquigarrow \lambda\, x{:}T.\, t}\ \textsc{Bare}$$

$$\frac{\Gamma, X \vdash T \ni e \rightsquigarrow t}{\Gamma \vdash \forall\, X.\, T \ni \Lambda\, X.\, e \rightsquigarrow \Lambda\, X.\, t}\ \textsc{Poly}\ni \qquad \frac{\neg \mathit{Abs}(e) \quad \Gamma \vdash e \rightsquigarrow t \in T_2 \quad T_1 = T_2}{\Gamma \vdash T_1 \ni e \rightsquigarrow t}\ \textsc{Sub}$$

$$\mathit{Abs}(e) = (\exists\, X, e'.\, e = \Lambda\, X.\, e') \vee (\exists\, x, T, e'.\, e = \lambda\, x{:}T.\, e')$$

(b) Type inference

$$\begin{array}{rclcrcl} |x| & = & x, & \quad & |\lambda\, x{:}T.\, e| & = & \lambda\, x.\, |e|, \quad |f\ e| = |f|\ |e| \\ |\chi\ T\ \text{-}\ e| & = & |e|, & & |\lambda\, x.\, e| & = & \lambda\, x.\, |e|, \quad |e \cdot T| = |e| \\ & & & & |\Lambda\, X.\, e| & = & |e| \end{array}$$

(c) Erasure

Figure 3: Bidirectional System F

grammar of the term language. The external term language (Figure 3a) simply adds two new term constructs: "bare" function abstractions $\lambda\, x.\, e$ and type annotations $\chi\ T$ - e.

Typing rules Figure 3b gives the type inference rules for bidirectional System F. Read the judgment $\Gamma \vdash e \rightsquigarrow t \in T$ as "under context Γ, external term e elaborates to internal term t and synthesizes type T"; read $\Gamma \vdash T \ni e \rightsquigarrow t$ as "under Γ, use T to check e and elaborate t". Associated with the subjects Γ, e, t, and T of both judgments is a **mode**: in both judgments, Γ and t are considered inputs, and t is an output; in the synthesis judgment T is also an output, but in the checking judgment it is an input. We remark on some interesting rules.

For type synthesis, rule App says we synthesize type T from an application $e_1\ e_2$ and elaborate $t_1\ t_2$ if e_1 synthesizes type $S \to T$ and elaborates to t_1, and we can check e_2 with type S and have it elaborate to t_2 – S is contextually available from the type of e_1. For rule Var, we remove the possibly confusing notation $(x{:}T \in \Gamma)$ and say instead that a variable x elaborates to itself and synthesizes $\Gamma(x)$, the type associated to it in the context Γ (if this exists). Rule Ann demonstrates a more drastic change between internal and external terms: top-level annotations with χ are not part of annotated System F, and so elaboration removes them.

For type checking, rule Bare shows how elaboration handles bare abstractions $\lambda\, x.\, e$. It is not obvious from the syntax alone what type should be associated with the bound variable x, so we require that this be provided contextually in the form of a checked type $T \to S$. Then, if the body e can be checked against the type S and elaborates to t under a context extended with the assumption $x : T$, the bare abstraction elaborates to $\lambda\, x{:}\, T.\, t$. Finally, rule Sub is a "mode-switch" rule: terms for which there is no other checking rule (function application, polymorphic instantiation, variables, and top-level annotations) can be checked against type T_1 if they synthesize type T_2, and these two types are equal (up to α-conversion). In order to avoid ambiguity for checkable annotated function abstractions (Abs$\ni$) and checkable polymorphic terms (Poly$\ni$), wherein type information is available both (partially) in the syntax of the term and (fully) contextually, we require that the subject e of rule Sub not have the form of a λ- or Λ-abstraction with the premise $\neg Abs(e)$.

Erasure The erasure rules for bidirectional System F (Figure 3c) are straightforward. For bare abstractions we simply erase the body; for type annotations, the ascribed type T plays no rôle in the computation of $|e|$, and so is erased.

3.2 Properties

Having designed elaborating type inference rules for bidirectional System F, we are now interested in verifying that certain essential properties hold of the system. In this report, the properties we state serve to illustrate the principles that language designers should keep in mind when formulating elaborating type inference rules. In particular, those properties labeled as "conjecture" had not been formally proven. When appropriate, we point to instances of similar properties in the literature on elaboration.

To begin, not every set of inference rules determines and algorithm, but to claim that we have given a translational semantics of the external language to the internal one, it is essential that there is an algorithmic reading of the inference rules comprising the judgments for type synthesis and checking. Specifically, we must have a *bottom-up proof search* reading of the inference rules wherein the form of the goal (the desired typing judgment) uniquely determines which rule (if any) could have it as a conclusion, and each rule must be **mode-correct** [Pfe01, War77]. The recipe for mode-correctness is:

1. **Assume** that the inputs of the judgment in the conclusion are given

2. **Show** that the inputs of the judgment in the premise can be constructed
3. **Assume** that the outputs of the judgment in the premise are given (if there are multiple premises, read them left to right and propagate outputs to premises further to the right)
4. **Show** that the output of the judgment in the conclusion can be constructed

Proposition 3.1 (Algorithmic type inference)**.**

1. *Every rule is mode-correct for a moding* $\Gamma^- \vdash e^- \rightsquigarrow t^+ \in T^+$ *and* $\Gamma^- \vdash T^- \ni e^- \rightsquigarrow t^+$ *of the two judgments (where* $-$ *indicates input and* $+$ *indicates output).*
2. *For all* Γ*,* e*,* t*, and* T*, there is at most one rule whose conclusion matches the syntactic form of the judgment* $\Gamma \vdash e \rightsquigarrow t \in T$
3. *For all* Γ*,* e*,* t*, and* T*, there is at most one rule whose conclusion matches the syntactic form of the judgment* $\Gamma \vdash T \ni e \rightsquigarrow t$ *(where we consider rule* Sub *as abbreviating multiple rules, one for each of the possible ways* e *could be constructed such that* $\neg App(e)$*).*

Proof sketch. By inspection of the type inference rules. □

An easy consequence of Proposition 3.1 is that type inference is deterministic.

Corollary 3.1.1 (Determinism)**.** *For all* Γ*,* e*,* t*,* t'*,* T*, and* T'*:*

1. *if* $\Gamma \vdash e \rightsquigarrow t \in T$ *and* $\Gamma \vdash e \rightsquigarrow t' \in T'$ *then* $t = t'$ *and* $T = T'$
2. *if* $\Gamma \vdash T \ni e \rightsquigarrow t$ *and* $\Gamma \vdash T \ni e \rightsquigarrow t'$ *then* $t = t'$

Where '$=$*' indicates formal (syntactic) equality.*

Proof sketch. By mutual induction on typing derivations of $\Gamma \vdash e \rightsquigarrow t \in T$ and $\Gamma \vdash T \ni e \rightsquigarrow t$, appealing to Proposition 3.1(2,3) at each step. □

The next property of interest comes from the observation that every term of the internal language is also a term of the external language. We might like to verify, as a kind of sanity check, that the type inference rules for bidirectional System F are *sufficient* for typing the terms of annotated System F, and furthermore that elaboration for such a term re-produces that term. We expect this because our internal language is a *sublanguage* of the external one. We borrow the term *sufficiency* from Swamy et al. [SHB09], who use it to state a similar property concerning the elaboration of subtyping by the insertion of type coercions.

Proposition 3.2 (Sufficiency)**.** *For all* Γ*,* t*, and* T*, if* $\Gamma \vdash t : T$ *then* $\Gamma \vdash t \rightsquigarrow t \in T$

Proof sketch. By induction on the assumed typing derivation, where in the case that t is an application we use Proposition!3.5 after appealing to the inductive hypothesis for the argument in that application. □

Soundness of our bidirectional type inference rules states that well-typed terms e of the external language are elaborated to well-typed (at the same type) terms t of the internal language. We call this *static soundness* of our elaboration rules. In literature on elaboration, such as [DM12, SHB09, JS18], this is often simply referred to as "soundness"; in [CA18], it is called "type preservation" which is more descriptive but uses overloaded terminology.

For our purposes, however, it is not enough to specify only that e and t have the same (or related) type in their respective systems. For example, if e is the Boolean value *true* (using standard techniques to encode the type of Booleans in System F), we certainly do not wish that its elaboration t is *false*. Thus, we need also *dynamic soundness* of elaboration, which here means that e and t are equal "up to erasure". Examples of dynamic soundness properties appear in the work of Gougen et al. [GMM06] and [CA18] in their translation of dependent pattern matching to eliminators, and by Jenkins et al. [JMS19] in unpublished work on elaborating simple pattern-matching and course-of-values induction to lambda encodings in **CDLE**.

Conjecture 3.3 (Static and dynamic soundness)**.**
For all Γ*,* e*,* T*, and* t*,*

- *if* $\Gamma \vdash e \rightsquigarrow t \in T$ *then* $\Gamma \vdash t : T$ *and* $|e| = |t|$
- *if* $\Gamma \vdash T \ni e \rightsquigarrow t$ *then* $\Gamma \vdash t : T$ *and* $|e| = |t|$

As our bidirectional type inference rules describe a partial typechecking method, we do not have a general completeness theorem stating that *all terms* of the external language can have their types inferred. However, we can instead define what it means to be complete assuming some restriction $\mathcal{R}$ relating internal and external terms whose erasures are the same.

Definition 3.4 ($\mathcal{R}$-restricted completeness)**.** Given some relation $\mathcal{R}$ between external and internal terms of the static language, we say that the inference system of Figure 3b is $\mathcal{R}$-complete if for any Γ, t, e, and T, if $\Gamma \vdash t : T$ and $|e| = |t|$ and $\mathcal{R}(e, t)$ then $\Gamma \vdash e \rightsquigarrow t \in T$.

This definition of completeness may also be called the *annotation character* of the bidirectional system [DK19]. As a trivial example, take $\mathcal{R}(e, t)$ to be the property that e and t are formally identical. Then, Proposition 3.2 gives us $\mathcal{R}$-restricted completeness (this was referred to as "trivial completeness" in [JS18]). A more useful definition of $\mathcal{R}$, typical of bidirectional systems, is the property that e require type annotations only on lambda abstractions in β-redexes (c.f. Dunfield and Krishnaswami [DK13]).

Finally, a simple property that bidirectional typechecking systems should enjoy is that any term whose type we can synthesize is able to be checked against that type. We expect also that the elaborated term in both cases is the same:

Proposition 3.5 (Synthesis and checking)**.**
For all Γ*,* e*,* t*, and* T*, if* $\Gamma \vdash e \rightsquigarrow t \in T$ *then* $\Gamma \vdash T \ni e \rightsquigarrow t$

Proof sketch. By mutual induction on the assumed typing derivation. The restriction in the premise of Sub means that we must use the inductive hypothesis in the cases given by rules Abs$\in$ and Poly$\in$. □

3.3 Related Work

There is much literature on elaborating type inference systems. Regarding the design of typechecking systems for inferring missing type annotations, Pierce and Turner [PT00] formulated the problem generally in three parts. We quote (emphasis theirs):

- Syntax, typing rules, and semantics for a fully annotated *internal language*.
- An *external language* in which some type annotations are made optional or omitted entirely. This is the language that the programmer actually uses.
- Some specification of a *type inference* relation between the external language and the internal one.

This formulation is especially useful for studying partial typechecking methods such as *local type inference*, as it facilitates a specification of the annotation character $\mathcal{R}$. Ibid. work within the theory $F_{\leqslant}$, an extension of System F with subtyping, and give an analogue to Conjecture 3.3 and a combined from of Propositions 3.2 and 3.5 relating checking mode and type subsumption. Jenkins and Stump [JS18] presented a similar bidirectional system called *spine-local type inference*, using elaboration for similar ends. Jones et al. [JVWS07] studied a (non-local, unification-based) bidirectional type inference method for higher-rank polymorphism (i.e., arbitrarily left-nested type quantification), providing a "type-directed translation" (i.e., elaborating type inference rules) from their system to (predicative) System F. Their system features a form of subtyping derived from the "more polymorphic than" relation of [OL96], and elaboration of subtyping derivations produces retyping functions *a lá* Mitchell [Mit88] – type coercions whose underlying computational content (given by the complete erasure of type abstractions, annotations, and instantiation) is β-equivalent to the identity function.

4 Algebraic Semantics of Inductive Definitions

We now turn to a feature of great importance in programming languages – user-defined datatypes. In functional programming languages, this feature is usually made available via *algebraic* datatype declarations, wherein a type D is defined by specifying the collection of (necessarily disjoint) constructors c_i which generate it. These constructors may take arguments, the types of which may furthermore recursively refer to the datatype D being defined. For example, in the Cedille programming language [Stu18b, JMS19], the type of natural numbers `Nat` and the type of lists `List ·A` (where here the center dot denotes application of a type to a type, which we have not defined) of some element type `A` can be defined using the syntax given in Figure 4.

```
data Nat : ⋆                          data List (A: ⋆): ⋆
= zero : Nat                          = nil : List
| suc  : Nat → Nat                    | cons : A → List → List
```

Figure 4: Example datatype declarations in Cedille

Algebraic datatypes first appeared in the HOPE programming language [BMS80], and since then have become a popular feature in functional programming languages such as Haskell, ML, and Scala, as well as in implementations of dependent type theories like Agda, Coq, Idris, and Lean. Before we attempt to translate ADTs and functions over them ourselves, however, it behooves us to review the literature on the translational and algebraic semantics of inductive definitions.

4.1 Lambda encodings

4.1.1 Church encoding

One of the most venerable semantical accounts of datatypes is *lambda encoding*, which can be understood as identifying datatypes with some scheme for defining functions over them. The most famous such encoding is known as *Church-encoding*, named after Alonzo Church who demonstrated the technique for simple datatypes such as natural numbers and Booleans [Chu41]. The *Church encoding* identifies datatypes with their *iteration scheme*. In the case of `Nat`, this can be given by expressed by the following typing and reduction rules for $\texttt{iter}^{\texttt{Nat}}$:

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \to T}{\Gamma \vdash \texttt{iter}^{\texttt{Nat}} \cdot T\ t_1\ t_2 : \texttt{Nat} \to T}$$

$$\texttt{iter}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ \texttt{zero} \dashrightarrow t_1, \quad \texttt{iter}^{\texttt{Nat}} \cdot T\ t_2\ t_2\ (\texttt{succ}\ t) \dashrightarrow t_2\ (\texttt{iter}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ t)$$

This method of lambda-encoding is alternatively called the *Böhm-Berraduci* encoding, as [BB85] gave a systematic translation of datatype systems expressed in the language of universal algebras into lambda expressions typeable in System F, meaning that the foregoing type inference and reduction rules need not be added as primitives to System F, but are derivable for the definitions of `Nat`, `zero`, and `succ` (replacing $\dashrightarrow$ with $\overset{*}{\dashrightarrow}$) that can be algorithmically determined by the data declaration of `Nat`. We give this below, as well as the corresponding schematic definition for $\texttt{List} \cdot T$.

For the types `Nat` and $\texttt{List} \cdot T$:

$$\texttt{Nat} =_{\mathbf{df}} \forall X.\, X \to (X \to X) \to X, \quad \texttt{List} \cdot T =_{\mathbf{df}} \forall X.\, X \to (T \to X \to X) \to X$$

for their respective constructors:

$$\begin{aligned}
\texttt{zero} &=_{\mathbf{df}} \chi\ \texttt{Nat}\ \texttt{-}\ \Lambda X.\, \lambda z.\, \lambda s.\, z \\
\texttt{suc} &=_{\mathbf{df}} \chi\ (\texttt{Nat} \to \texttt{Nat})\ \texttt{-}\ \lambda n.\, \Lambda X.\, \lambda z.\, \lambda s.\, s\ (n \cdot X\ z\ s) \\
\\
\texttt{nil} &=_{\mathbf{df}} \chi\ (\texttt{List} \cdot T)\ \texttt{-}\ \Lambda X.\, \lambda n.\, \lambda c.\, n \\
\texttt{cons} &=_{\mathbf{df}} \chi\ (T \to \texttt{List} \cdot T \to \texttt{List} \cdot T)\ \texttt{-}\ \lambda hd.\, \lambda tl.\, \Lambda X.\, \lambda n.\, \lambda c.\, c\ hd\ (tl \cdot X\ n\ c)
\end{aligned}$$

for their iteration schemes:

$$\begin{aligned}
\texttt{iter}^{\texttt{Nat}} &=_{\mathbf{df}} \chi\ (\forall X.\, X \to (X \to X) \to \texttt{Nat} \to X)\ \texttt{-} \\
&\qquad \Lambda X.\, \lambda x_1.\, \lambda x_2.\, \lambda n.\, n \cdot X\ x_1\ x_2 \\
\texttt{iter}^{\texttt{List}\cdot T} &=_{\mathbf{df}} \chi\ (\forall X.\, X \to (T \to X \to X) \to \texttt{List} \cdot T \to X)\ \texttt{-} \\
&\qquad \Lambda X.\, \lambda x_1.\, \lambda x_2.\, \lambda l.\, l \cdot X\ x_1\ x_2
\end{aligned}$$

Almost as well-known as the method of Church encoding itself is its primary deficiency (shown by Parigot in [Par90]) as a representation of datatypes in lambda calculi: subdata accessors, such as the predecessor function for naturals or the tail function for lists, can take no better than linear time to compute. Other methods of encoding exist which address this deficiency, and come with their own trade-offs. We discuss two of these briefly (the *Scott* and *Parigot* encoding). A third encoding, the *Mendler* encoding, appears to have the same deficiency in System F, and receives much more attention in this section. See [SF16] for a more detailed analysis of the efficiency of lambda encodings in type theory (they do not remark on the Mendler encoding).

4.1.2 Scott encoding

The Scott encoding first appeared in unpublished lecture notes by Dana Scott [Sco62], and was studied (without attribution) by Michel Parigot in [Par88], wherein it is called the "stack" representation. The Scott encoding identifies datatypes with their *case-distinction* scheme [Geu14], which for the type `Nat` is given by type inference and reduction rules for the operator $\texttt{case}^{\texttt{Nat}}$:

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \texttt{Nat} \to T}{\Gamma \vdash \texttt{case}^{\texttt{Nat}} \cdot T\ t_1\ t_2 : \texttt{Nat} \to T}$$

$$\texttt{case}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ \texttt{zero} \dashrightarrow t_1, \quad \texttt{case}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ (\texttt{succ}\ t) \dashrightarrow t_2\ t$$

It is immediate that one can define a constant-time predecessor function over `Nat` using this case scheme. What is not clear, however, is whether there is a solution for `Nat` in System F satisfying this specification. Spławski and Urzyczyn in [SU99] point to a negative result: in brief, they show that datatype systems with *recursion* schemes (of which the case-distinction scheme above is a degenerate instance) can be efficiently simulated in System F extended with recursive types, but that an arbitrary well-typed term of System F with recursive types *cannot* be efficiently simulated in System F. If we seek to define `Nat` in terms of its case scheme, the occurrence of `Nat` in the premises of the typing rule becomes a recursive one, and it seems (but is not proven to be) inevitable that we must reach for recursive types to do so, or else face a linear-time penalty in iterative rebuilding the predecessor [ACP93]. With recursive types, the definition of `Nat` is easily given (for brevity we omit the corresponding definition of the constructors and case scheme):

$$\texttt{Nat} =_{\mathbf{df}} \mu\, N.\, \forall\, X.\, X \to (N \to X) \to X$$

where μ is a recursive type former equipped with operations $\texttt{roll} : T[\mu\, X.\, T/X] \to \mu\, X.\, T$ and $\texttt{unroll} : \mu\, X.\, T \to T[\mu\, X.\, T/X]$ forming an isomorphism.

Adding unrestricted recursive types has the dire consequence of ruining the meta-theoretic property of *termination* [Men87]. We thus require some form of restriction on the formation, introduction, or elimination of recursive types to preserve termination, such as a syntactic criterion of positive occurrences of type variables or internalized witnesses of positivity as pursued by [Mat02]. Concerning the Scott encoding, it is clear that in the presence of general recursion (which would be given by unrestricted recursive types [MJ69]) an iteration scheme can be given for them, but it is rather less clear how to do so without it. In [Par88], Parigot gave a strongly normalizing iterator for Scott-encoded naturals, using meta-theoretic reasoning to argue that it could safely be assigned a suitable type. Taking advantage of this argument *within* a type theory seems to require rather exotic features: Lepigre and Raffalli [LR19] accomplish this in a theory with recursive types and a subtyping judgment with "circular but well-founded" derivations; Jenkins and Stump [JS20] use a method related to theirs with derivable (monotone) recursive types and an internalized notion of subtyping amenable to proof by induction (though they failed to observed that a Parigot's original proof can be carried out *directly* in Cedille).

Finally, the Scott encoding has an unexpected trade-off in comparison to the Church encoding. Concretely, for the Church encoding of `Nat` it is possible to define addition in **constant time**, and (symmetrically to the case of predecessor) it is provably impossible to do so for the Scott encoding [Par90].

4.1.3 Parigot encoding

When considering the Church and Scott encoding, a natural question arises: is there some way to combine these two encodings such that the benefits of both can be enjoyed? Put another way: if the Church encoding identifies datatypes with their iterators, and the Scott encoding identifies datatypes with their case schemes, what is the encoding that identifies datatypes with there *recursion scheme*, which subsumes these? Such considerations lead one to the *Parigot* encoding, first described in [Par88] for natural numbers. In [Geu14], Geuvers calls this method the "Church-Scott" encoding, and connects them to the general form of the recursion scheme for datatypes. For the type `Nat`, the recursion scheme is given by type inference and reduction rules for the operator $\texttt{rec}^{\texttt{Nat}}$:

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \mathtt{Nat} \to T \to T}{\Gamma \vdash \mathtt{rec}^{\mathtt{Nat}} \cdot T\ t_1\ t_2 : \mathtt{Nat} \to T}$$

$$\mathtt{rec}^{\mathtt{Nat}} \cdot T\ t_1\ t_2\ \mathtt{zero} \dashrightarrow t_1, \quad \mathtt{rec}^{\mathtt{Nat}} \cdot T\ t_1\ t_2\ (\mathtt{succ}\ t) \dashrightarrow t_2\ t\ (\mathtt{rec}^{\mathtt{Nat}} \cdot T\ t_1\ t_2\ t)$$

As with the Scott encoding, expressing the type of Parigot-encoded data appears to require some form of recursive types. The type of Parigot-encoded naturals is given by:

$$\mathtt{Nat} =_{\mathbf{df}} \mu\, N. \forall\, X.\, X \to (N \to X \to X) \to X$$

Parigot-encoded datatypes enjoy both constant-time destructors (e.g., predecessor for `Nat`), impossible for the Church encoding, and ease in the definition of recursive functions over them, which is not the case for the Scott encoding without more exotic type features. However, these conveniences come with a price: the space complexity of the closed normal form of a natural number n (i.e., the representation of n and a term of the lambda calculus for which there are no further β-redexes) is **exponential** in the size of n [Par88]. Additionally, there is another, more subtle issue plaguing the Parigot encoding: its type is not precise enough, and it admits "bogus numbers" as well as the intended set of naturals [Geu14]. To illustrate, let $\underline{n}$ denote the Parigot encoding of the number n. Then, the term $\mathtt{roll}\ (\Lambda\, X.\, \lambda\, z{:}X.\, \lambda\, s{:}\mathtt{Nat} \to X \to X.\, s\ \underline{0}\ z)$ corresponds to a natural number (specifically, 1), but the term $\mathtt{roll}\ (\Lambda\, X.\, \lambda\, z{:}X.\, \lambda\, s{:}\mathtt{Nat} \to X \to X.\, s\ \underline{10}\ z)$ does not. This difficulty does note arise Parigot's original presentation of the encoding, as the language of types he uses, second-order predicate logic, is strong enough to rule out such cases.

4.1.4 Mendler encoding

Barring concerns of efficiency, in System F the Church encoding is adequate for expressing many of the desired recursive functions over datatypes. As it identifies datatypes with their associated iteration scheme, using them is similar to the "Squigoll style" of programming [BW88, MFP91], where for example (in the simplest case) all recursive functions over natural numbers are written using $\mathtt{iter}^{\mathtt{Nat}}$. We illustrate with the definition of a function computing the sum of two numbers:

$$\mathtt{add}\ t_1\ t_2 =_{\mathbf{df}}\ \mathtt{iter}^{\mathtt{Nat}} \cdot \mathtt{Nat}\ t_1\ (\lambda\, x.\, \mathtt{succ}\ x)\ t_2$$

The Squiggol style of programming enjoys many benefits: it facilitates automatic program calculation from specification; it can be used to guarantee program termination; and it distills equational reasoning for recursive programs into a handful of general properties (*cancellation*, *reflection*, and *fusion*, which we discuss in Section 4.2.2). However, it arguably suffers from a rather pragmatic drawback: programmers accustomed to defining functions using *general recursion* can find the insistence on working only with the previous recursively computed values rather unidiomatic. For example, in the definition of `add` above confusion possibly arises for the uninitiated as to whether the bound variable `x` refers to the previously computed sum (which it does) or to some predecessor of the number over which we are recursing (which it does not).

In [UV00], Uustalu and Vene propose an alternative "Mendler style" of programming which enjoys the same benefits of the Squiggol style but which more closely resembles the idiomatic approach of making explicit recursive calls when defining functions over datatypes. Intuitively (and somewhat informally), rather than working with previously computed results the programmer instead works with "approximate versions" of the recursively defined function itself. More concretely, if $f : \mathtt{Nat} \to T$ is the recursive function we seek to define, then we can do so assuming some $f' : R \to T$ in a context extended by a type variable R. One considers R to be a sort of "subtype" of `Nat`, since when defining how f computes over an argument of the form $\mathtt{succ}\ t$, the

predecessor t has type R; the only type-correct application of f' is to this predecessor, entitling us to interpret R as *a subtype of* `Nat` *for which recursive calls are guaranteed to be well-founded.*

We make the foregoing more precise by giving the type inference and reduction rules for $\texttt{miter}^{\texttt{Nat}}$, the Mendler-style combinator for iteration.

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \forall\, R.\, (R \to T) \to R \to T}{\Gamma \vdash \texttt{miter}^{\texttt{Nat}} \cdot T\ t_1\ t_2 : \texttt{Nat} \to T}$$

$$\texttt{miter}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ \texttt{zero} \dashrightarrow t_1, \quad \texttt{miter}^{\texttt{Nat}} \cdot T\ t_1\ t_2\ (\texttt{succ}\ t) \dashrightarrow t_2 \cdot \texttt{Nat}\ (\texttt{miter}^{\texttt{Nat}} \cdot T\ t_1\ t_2)\ t$$

Notice that for the computation rules in the case of successor, the universally quantified type R of t_2 is instantiated to the type `Nat`, and the argument which is intended to serve as an approximation to function being recursively defined is none other than $\texttt{miter}^{\texttt{Nat}} \cdot T\ t_1\ t_2$. Additionally, in the Mendler style the confusion in the definition of `add` mentioned earlier disappears:

$$\texttt{add}\ t_1\ t_2 =_{\mathbf{df}} \texttt{miter}^{\texttt{Nat}} \cdot \texttt{Nat}\ t_2\ (\Lambda\, R.\, \lambda\, f'.\, \lambda\, x.\, \texttt{succ}\ (f'\ x))\ t_1$$

where we now interpret the lambda-bound f' as the approximation (in the sense of having its domain restricted to the type R) of the recursively defined function which adds t_1 to its argument, and x as some given predecessor of t_1 having type R.

As with the Church encoding, types for Mendler-encoded data can be defined in System F with no further extension:

$$\texttt{Nat} =_{\mathbf{df}} \forall\, X.\, X \to (\forall\, R.\, (R \to X) \to R \to X) \to X$$

with constructors:

$$\begin{array}{lll} \texttt{zero} & =_{\mathbf{df}} & \chi\ \texttt{Nat}\ \texttt{-}\ \Lambda\, X.\, \lambda\, z.\, \lambda\, s{:}\, z. \\ \texttt{succ} & =_{\mathbf{df}} & \chi\ (\texttt{Nat} \to \texttt{Nat})\ \texttt{-}\ \lambda\, n.\, \Lambda\, X.\, \lambda\, z.\, \lambda\, s.\, s \cdot \texttt{Nat}\ (\lambda\, x.\, x \cdot X\ z\ s)\ n \end{array}$$

and iterator

$$\begin{array}{lll} \texttt{miter}^{\texttt{Nat}} & =_{\mathbf{df}} & \chi\ (\forall\, X.\, X \to (\forall\, R.\, (R \to X) \to R \to X) \to \texttt{Nat} \to X)\ \texttt{-} \\ & & \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, \lambda\, n.\, n \cdot X\ x_1\ x_2 \end{array}$$

The categorical account of the Church and Mendler encoding we consider in Section 4.2.2 tells us that they are equally expressive as encodings (for the categorically disinclined reader, we leave it as an exercise to show these two definitions of `Nat` are isomorphic). We are unaware of any formal proof concerning the efficiency of predecessor for the Mendler encoding, but conjecture that the situation is the same as for the Church encoding: when defining a recursive function using Mendler-style iteration, there is no way to "reveal" the fact that some term of type R "really is" a term of type `Nat` (since R introduced by universal abstraction), and so one appears left with no choice but to iteratively rebuild such a term if that is required.

4.2 Algebraic semantics of inductive definitions

4.2.1 Universal algebras

The moniker "algebraic datatype" comes from a semantic understanding of inductive definitions in programming languages as *universal algebras* [BG82]. We do not require universal algebras in their full generality, so for simplicity we present them here in a restricted form (specifically, we only require *free universal algebras*). Our presentation is adapted from [BB85].

Definition 4.1 (Algebra)**.** An *algebra* is a pair $(\mathcal{C}, \mathcal{G})$ where

- $\mathcal{C} = \{C_k\}_{k \in K}$ is a family of sets C_k (called the *carriers* of the algebra) indexed by some finite set K;
- $\mathcal{G} = \{g_j\}_{j \in J}$ is a family of finitary operations (called the *actions* of the algebra) indexed by some finite set J where each g_j defines a mapping

$$g_j : C_{\delta(1,j)} \times ... \times C_{\delta(\alpha(j),j)} \to C_{\iota(j)}$$

with $\alpha(j) \in \mathbb{N}$ giving the arity of g_j, $\delta(i,j) \in K$ giving the index for the carrier set of the ith component (for $i \in \{1, ..., \alpha(j)\}$) of the domain of g_j, and $\iota(j) \in K$ giving the index for the carrier set of the codomain of g_j.

Definition 4.2 (Term algebra)**.** A *term algebra* is an algebra $(\mathcal{C}, \mathcal{G})$ such that, for any two operators g_1 and g_2 and tuples $(x_1, ..., x_n)$ and $(y_1, ..., y_m)$ in the respective domains of g_1 and g_2, if $g_1(x_1, ..., x_n) = g_2(y_1, ..., y_n)$ then $m = n$, $(x_1, ..., x_m) = (y_1, ..., y_n)$, and $g_1 = g_2$, where '=' indicates formal identity (syntactic equality).

The following two examples express natural numbers and lists as term algebras:

Example. Let $\texttt{Nat}$ be the least set generated by uninterpreted constant $\texttt{zero} \in \texttt{Nat}$ and uninterpreted function $\texttt{succ} : \texttt{Nat} \to \texttt{Nat}$. Then $(\{\texttt{Nat}\}, \{\texttt{zero}, \texttt{succ}\})$ is a term algebra.

Example. Let A be an arbitrary non-empty set. Let $\texttt{List}_A$ be the least set generated by uninterpreted constant $\texttt{nil} \in \texttt{List}_A$ and uninterpreted function $\texttt{cons} : A \times \texttt{List}_A \to \texttt{List}_A$. Then $(\{A, \texttt{List}_A\}, \{\texttt{nil}, \texttt{cons}\})$ is a term algebra.

At this point, one may be tempted to think that the notion of "term algebra" is enough to express what intuition tells us an inductive datatype should be. The following example demonstrates that this is mistaken.

Example. For any element $x \in \texttt{Nat}$, $(\{\texttt{Nat}\}, \{x, \texttt{succ}\})$ is a term algebra.

Expressing inductive datatypes as universal algebras requires some subtlety. For a term algebra $(\mathcal{C}, \mathcal{G})$, if we were to require that for every carrier $C \in \mathcal{C}$ its elements are precisely those elements in the images of operators $g \in \mathcal{G}$ such that $C = cod(g)$ (the codomain of g), this would exclude the list datatype – the carrier A of the term algebra $(\{A, \texttt{List}_A\}, \{\texttt{nil}, \texttt{cons}\})$ representing lists of elements of type A is **not** a codomain of **any** action, because we wish to consider it as a parameter to the definition of lists, and not part of the definition itself! In [BB85], Böhm and Berarducci overcome this technical difficulty by carefully distinguishing between parametric and non-parametric carriers. Rather than detail this, we shall instead reach for greater conceptual (and notational) clarity by turning to *category theory* for an understanding of algebras and datatypes.

To ease the passage from universal algebras to the categorical notion of F-algebras, it is useful to note in the special case of an algebra $(\{C\}, \{g_j\}_{j \in J})$ with a single carrier C that the set of actions $\{g_j\}_{j \in J}$ can be replaced with a *single* action a whose domain is the disjoint union of the domains of each $g \in \mathcal{G}$. We express this disjoint union as $\coprod_{j \in J} dom(g_j)$ and its J-indexed set of injections as $\texttt{inj}_{j'} : dom(g_{j'}) \to \coprod_{j \in J} dom(g_j)$. Then, the generalized algebra action $a : (\coprod_{j \in J} dom(g_j)) \to C$ is defined by

$$a(\texttt{inj}_j(x_1, ..., x_{\alpha j})) =_{\mathbf{df}} g_j(x_1, ..., x_{\alpha j}) \quad \text{for all } j \in J$$

Thus arises the phrase "sum-of-products" when describing the general form of defining algebraic datatypes. One result of the next section will be the complete removal of any need mention the set of actions $\{g_j\}_{j \in J}$, allowing us to work directly with the generalized action a.

4.2.2 F-algebras

We begin with some definitions of basic concepts in category theory. For a more thorough treatment of category theory and its applications to computer science, consult [Pie91].

Definition 4.3 (Category)**.** A *category* $\mathcal{C}$ consists of

1. a collection of *objects* (denoted with upper-case Latin letters $A, B, C, ...$);
2. a collection of *morphisms* (denoted with lower-case Latin letters $f, g, h, ...$), where each arrow f is associated with a domain object A and codomain object B written $f : A \to B$.
3. an associative composition operator $\circ$ such that for all objects A, B, and C and morphisms $f : A \to B$ and $g : B \to C$, $g \circ f : A \to C$.
4. for every object A there is an identity morphism id_A which is a neutral element wrt $\circ$.

For the discussion of this section, we are interested in those categories which can provide semantics for type theories, with the objects corresponding to types and morphisms corresponding to functions. As with type theory and set theory, in category theory we can express what it means for an object $A \times B$ to be a binary product of objects A and B (corresponding resp. to 2-tuples and Cartesian products), a nullary product $\top$ (resp. the empty tuple and $\{\emptyset\}$), binary coproducts $A + B$ of objects A and B (resp. a binary sum type and disjoint union of sets), and nullary coproducts $\bot$ (resp. the uninhabited type and the empty set). These constructions appear only in the examples of this section, so for space considerations we omit their formal definitions and rely on the reader's intuition of their counterparts in type theory and set theory.

Definition 4.4 (Functor)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor F between $\mathcal{C}$ and $\mathcal{D}$ (written $F : \mathcal{C} \to \mathcal{D}$) is a mapping that

- takes each object C of $\mathcal{C}$ to an object $F(C)$ of $\mathcal{D}$
- takes each morphism $f : A \to B$ of $\mathcal{C}$ to a morphism $F(f) : F(A) \to F(C)$ of $\mathcal{D}$

and that satisfies the following laws (called the functor identity and composition laws)

- for all objects C of $\mathcal{C}$, $F(id_C) = id_{F(C)}$
- for all morphisms $f : A \to B$ and $g : B \to C$ of $\mathcal{C}$, $F(g \circ f) = F(g) \circ F(f)$

In Section 4.2.1, we said that for a universal algebra $(\{C\}), \{g_j\}_{j \in J})$ we could specify a generalized algebra action without reference to the actions g_j. *Endofunctors* (functors whose source and target categories are the same) are the technical device that allows us to do so; when a functor F is used in this fashion, we call it the *signature functor* (or *pattern functor*) of the algebra. Signature functors allow us to express the family of algebras whose actions share a certain "shape" independent of the definitions of those actions and the particular carrier they are defined over.

Example. In a category with binary coproducts and a nullary product, the signature functor for the universal algebras $(\{\texttt{Nat}\}, \{\texttt{zero}, \texttt{succ}\})$ and $(\{\texttt{Nat}\}, \{x, \texttt{succ}\})$ (for any $x \in \texttt{Nat}$) is given by $X \mapsto 1 + X$ (we omit the morphism mapping). We arrive at this definition by interpreting constants x and `zero` as morphisms from 1 to `Nat`, forming the coproduct $1 + \texttt{Nat}$ from the domains of the actions of each algebra, and abstracting away with X all occurrences of the carrier `Nat`.

Example. In a category with binary and nullary products, and binary coproducts, the signature functor for the universal algebra $(\{A, \mathtt{List}_A\}, \{\mathtt{nil}, \mathtt{cons}\})$ is given by $X \mapsto 1 + (A \times X)$, where object A is some suitable representative of the set A. We arrive at this definition in a fashion similar to the example above, but additionally we now designate A not as a carrier but as a *parameter* to the definition of the signature functor: for any object A, $X \mapsto 1 + (A \times X)$ is the signature functor for lists of elements of A.

With the foregoing preliminaries, we can now express F-algebras – algebras with a signature functor F – in purely category-theoretic terms.

Definition 4.5 (F-algebras)**.** Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$, an *$F$-algebra* is a pair (X, a) where X is an object of $\mathcal{C}$ (called the *carrier* of the algebra) and $a : F(X) \to X$ is a morphism of $\mathcal{C}$ (called the *action* of the algebra).

An F-algebra corresponds to a "plan" or "scheme" for giving an inductive definition. To see this, notice that if F is the signature functor for $\mathtt{Nat}$, then the type inference and reduction rules given for $\mathtt{iter}^{\mathtt{Nat}}$ for defining functions by iteration in Section 4.1.1 can be rephrased in terms of F-algebras, where below $\langle\rangle$ is the single element of 1, and inj_1 and inj_2 are resp. the left and right injections for coproducts.

$$\frac{\Gamma \vdash a : (1 + T) \to T}{\Gamma \vdash \mathtt{iter}^{\mathtt{Nat}} \cdot T\ a : \mathtt{Nat} \to T}$$

$$\mathtt{iter}^{\mathtt{Nat}} \cdot T\ a\ \mathtt{zero} \dashrightarrow a\ (inj_1\ \langle\rangle), \quad \mathtt{iter}^{\mathtt{Nat}} \cdot T\ a\ (\mathtt{succ}\ t) \dashrightarrow a\ (inj_2\ (\mathtt{iter}^{\mathtt{Nat}} \cdot T\ a\ t))$$

This suggests the following definition for datatypes when we consider them as algebras.

Definition 4.6 (Initial F-algebra)**.** An *initial F-algebra* $(\mu F, in)$ is one such that for any other F-algebra (X, a) there exists a unique morphism $(\!|\, a \,|\!) : \mu F \to X$ (called the *catamorphism* of a) such that $(\!|\, a \,|\!) \circ in = a \circ F((\!|\, a \,|\!))$. This equation can be alternatively expressed as a commuting categorical diagram:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{in} & \mu F \\ \downarrow{\scriptstyle F((\!|\, a \,|\!))} & & \downarrow{\scriptstyle (\!|\, a \,|\!)} \\ F(X) & \xrightarrow{a} & X \end{array}$$

Example. Consider the inductive datatype $\mathtt{Nat}$ of natural numbers with constructors $\mathtt{zero} : 1 \to \mathtt{Nat}$ and $\mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$. Let F be the functor defined by the mapping $X \mapsto 1 + X$. Then, $(\mathtt{Nat}, [\mathtt{zero}, \mathtt{succ}])$ is an F-algebra (where $[\_, \_]$ is the eliminator for binary sum types; alternatively, a case distinction for disjoint unions). It is also initial, with the catamorphism for an arbitrary algebra (X, a) given by $\mathtt{iter}^{\mathtt{Nat}} \cdot X\ a$.

There are two parts to the definition of an initial F-algebra: an *existence* claim for the catamorphism and a *uniqueness* claim [JR11]. The existence claim entitles us to a definition principle for iterative functions over datatypes; the uniqueness claim entitles us to a proof principle for equational reasoning over terms, just as in the Squigoll-style of programming. Of particular note are the *cancellation*, *reflection*, and *fusion* laws (c.f [Ven00], Cor. 2.1)

Proposition 4.7 (Cancellation, reflection, fusion)**.** *Let $(\mu F, in)$ be an initial F-algebra.*

- ***Cancellation:*** *For any F-algebra (X, a), $(\!|\, a \,|\!) \circ in = a \circ F((\!|\, a \,|\!))$*

  (this expresses how $(\!|\, a \,|\!)$ computes over elements constructed with in, and holds by the definition of initiality for F-algebras)

- ***Reflection:*** $(\!|\, in \,|\!) = id_{\mu F}$

  (this says iteratively rebuilding data with its constructors produces the original data)

- ***Fusion:*** *For any F-algebras (X, a) and (Y, b) and morphism $f : X \to Y$, if $f \circ a = b \circ F(f)$ then $f \circ (\!|\, a \,|\!) = (\!|\, b \,|\!)$*

  (this provides a useful optimization rule for "fusing" f with the iterative function $(\!|\, a \,|\!)$)

The notation μF for the carrier of an initial algebra suggests that is least a fixpoint[1] of F. That this is so – i.e., that there exists a generalized destructor $in^{-1} : \mu F \to F(\mu F)$ which is an inverse to in – is a result due to Lambek [Lam68] (see also [Ven00], Thm. 2.2).

Proposition 4.8 (Lambek's lemma)**.** *The action* in *of the initial algebra $(\mu F, in)$ is an isomorphism, with the inverse $in^{-1} : \mu F \to F(\mu F)$ given by $(\!|\, F(in) \,|\!)$.*

Finally, we remark that the initial F-algebras can be expressed in polymorphic type theory, and that this expression is essentially the type of Church-encoded data. It is folklore knowledge [Wad90] (and a generalization of Tarski's fixpoint theorem [T$^+$55, Mat02]) that for any positive type scheme F, its least fixpoint μF can be expressed by the type $\forall X.\, (F \cdot X \to X) \to X$. In the case that $F \cdot X = 1 + X$, it is easy to see that the type $(1 + X) \to X$ is isomorphic to $X \times (X \to X)$, and thus (by currying) $\mu F \cong \forall X.\, X \to (X \to X) \to X$. Lambek's lemma can therefore be understood as a generalized version of Kleene's solution for the predecessor function for the Church encoding of natural numbers [Cro75].

4.2.3 Mendler-style F-algebras

As discussed in Section 4.1.4, the Mendler style of coding recursive functions provides an arguably more natural alternative for programmers to the Squiggol style by making recursive calls explicit. As commented upon by [UV00], part of Mendler's motivation for formulating the lambda encoding underling this style [Men87, Men91] was a type-theoretic dissatisfaction with the conventional style. Unlike in category theory, in type theory a type scheme F is first defined only by its mapping of types to types, and only after this might also a function mapping be defined. However, in the conventional (Squiggol) style, the morphism-mapping component of the signature functor F is needed in the defining equation of the iterative combinator (i.e., the right-hand side of the *cancellation law* explicitly mentions it).

A surprising consequence of Mendler's approach, that not even Mendler himself was aware of at the time, is that the Mendler-style encoding applies also to mixed-variant type schemes. In [UV99], Uustalu and Vene give a thorough category-theoretic treatment of the Mendler style using initial *Mendler-style F-algebras* (where F is an endodifunctor, the categorical representation of a mixed-variant type scheme) and show that datatypes represented in this way are equivalent to initial (ordinary) F-algebras, with the reduction from initial Mendler algebras to initial classical ones requiring the presence of "restricted existential types". For simplicity, we shall consider Mendler-style F-algebras where F is required to be a functor.

Recall that the typing rule of the combinator for Mendler-style iteration over natural numbers (Section 4.1.4) required that the inductive case be given as a polymorphic higher-order function $\forall R.\, (R \to T) \to R \to T$. To arrive at a suitable expression for Mendler-style inductive types in category theory, we first need to express the notion of a polymorphic function and a function between sets of functions. These are resp. *natural transformations*[2] and *hom-sets*. A

[1] Leastness is given by *initiality* of *in*: for every F-algebra (X, a), including ones where a is an isomorphism, there is a unique F-algebra homomorphism from $(\mu F, in)$

[2] *Dinatural transformations* express polymorphic functions in their full generality, but these are not needed for the discussion.

more detailed explanation of these concepts can be found in any textbook on category theory (c.f. [Pie91, Awo10]).

Definition 4.9 (Natural transformation)**.** Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\Phi : F \dashrightarrow G$ between them is a function mapping objects A of $\mathcal{C}$ to morphisms $\Phi_A : F(A) \to G(A)$ in $\mathcal{D}$ such that for every $f : A \to B$ in $\mathcal{C}$ we have $\Phi_B \circ F(f) = G(f) \circ \Phi_A$. This can be depicted diagrammatically as:

$$\begin{array}{ccc} F(A) & \xrightarrow{\Phi_A} & G(A) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow{\Phi_B} & G(B) \end{array}$$

The equational constraint given for every $f : A \to B$ can be understood in type theory as a (meta-theoretic) *parametricity condition*: inputs $F(A)$ and $F(B)$ related by $F(f)$ are mapped by the polymorphic function Φ to outputs $G(A)$ and $G(B)$ related by $G(f)$. In standard polymorphic lambda calculi, such a condition holds for any definable polymorphic function and can thus be considered a "theorem for free" *a lá* Wadler [Wad89].

Definition 4.10 (Opposite category)**.** If $\mathcal{C}$ is a category, then $\mathcal{C}^{op}$ is the category whose objects are the objects of $\mathcal{C}$ and whose morphisms $f : D \to C$ are the morphisms $f : C \to D$ in $\mathcal{C}$.

Opposite categories can be used to express "contravariant" functors (type-theoretically, negative type schemes), which we shall need for expressing *hom-functors*.

Definition 4.11 (Hom-set, hom-functor)**.** Let $\mathcal{C}$ be a category. For every two objects A and B of $\mathcal{C}$, the *hom-set* $\mathcal{C}(A, B)$ is the set of all morphisms $f : A \to B$ in $\mathcal{C}$.

Furthermore, for every object C of $\mathcal{C}$, the (contravariant) hom-functor $\mathcal{C}(-, C) : \mathcal{C}^{op} \to \mathbf{Sets}$ (where **Sets** is the category of sets) maps objects B to the hom-set $\mathcal{C}(B, C)$ and morphisms $f : A \to B$ of $\mathcal{C}$ (which is a morphism $f : B \to A$ in $\mathcal{C}^{op}$) to morphisms $\mathcal{C}(f, B) : \mathcal{C}(B, C) \to \mathcal{C}(A, C)$ in **Sets** defined by $(h : B \to C) \mapsto h \circ f$.

Definition 4.12 (Mendler-style F-algebras)**.** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. A *Mendler-style F-algebra* (or *F-malg* for short) is a pair (X, Φ) where X is an object of $\mathcal{C}$ and $\Phi : \mathcal{C}(-, X) \dashrightarrow \mathcal{C}(F-, X)$ is a natural transformation. ($\mathcal{C}(F-, X)$ denotes the hom-functor whose object mapping takes R to $\mathcal{C}(F(R), X)$).

In requiring F to be a functor, F-malgs are isomorphic to ordinary F-algebras, which tells us that F-malgs give a suitable alternative semantics for inductive definitions.

Proposition 4.13. *The following transformations between F-malgs and F-algebras form an isomorphism:*

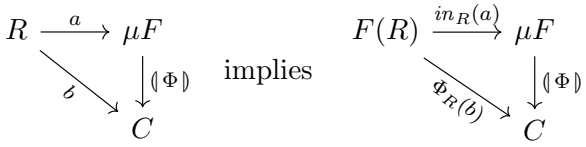
1. *Given an F-malg (X, Φ), produce F-algebra $(X, \Phi_X(id_X))$*
2. *Given an F-algebra (X, a), produce F-malg (X, Ψ), where for all R and $f : R \to X$, $\Psi_R(f) = a \circ F(f)$*

Proof. This is a consequence of the Yoneda lemma [Mac98].

In the forward direction, we wish to show that the given F-malg (X, Φ) is equal to the produced (X, Ψ), where for all R and $b : R \to X$, $\Psi_R(b) = \Phi_X(id_X) \circ F(b)$. Assume an arbitrary R and $b : R \to X$, and we therefore wish to show $\Phi_R(b) = \Phi_X(id_X) \circ F(b)$. This is a direct consequence of assuming Φ is a natural transformation between hom-functors.

In the other direction, we wish to show that the given F-algebra (X, a) is equal to the produced $(X, \Phi_X(id_X))$, where for all R and $b : R \to X$, $\Phi_R(b) = a \circ F(b)$. Instantiating with the given argument id_X, we wish to show that $a = a \circ F(id_X)$, which is given by the functor identity law. □

Definition 4.14 (Initial Mendler-style F-algebras)**.** An *initial F-malg* $(\mu F, in)$ is one such that, for every other F-malg (X, Φ) there exists a unique morphism $(\!|\, \Phi \,|\!) : \mu F \to X$ such that for every object R and morphisms $a : R \to \mu F$ and $b : R \to X$, if $b = (\!|\, \Phi \,|\!) \circ a$ then $(\!|\, \Phi \,|\!) \circ in_R(a) = \Phi_R(b)$. This is depicted diagrammatically below by saying that commutation of the left diagram implies commutation of the right.



In concordance with the discussion at the beginning of this section, notice that Definition 4.14 does not contain any reference the morphism-mapping component of the functor F; this only appears in the naturality condition associated with each F-malg action. However, to define $out : \mu F \to F(\mu F)$, the inverse of $in_{\mu F}(id_{\mu F})$, we use the morphism-mapping component directly: $out = F((\!|\, in \,|\!))$. The reflects the observation by Mendler in [Men87] that, even without explicit language features for recursion, case analysis for datatypes with negative recursive occurrences in the types of constructor arguments is enough to write diverging programs (see also Section 2 of [AS11]) – the assumption of a morphism-mapping component of F corresponds to a requirement that the type scheme it represents is positive.

5 Elaborating Inductive Definitions

In this section, we combine the proceeding sections on polymorphic lambda calculi, elaborating type inference rules, and the algebraic semantics of inductive definitions. For simplicity of presentation, we make the following design choices for our source and target languages. First, we shall choose as our target language *bidirectional System F* (Figure 3), as the lighter annotation requirements improve the readability of inference rules. Next, we will not consider elaborating the convenient syntactic declarations for inductive datatypes, as done in [DM12,JMS19]. Instead, we show the elaboration of *polyrec bidirectional System F*, which extends bidirectional System F with a certain recursive type constructor and type constructors corresponding to polynomial functors.

To avoid concerning ourselves with elaborating also *positivity witnesses* for the type schemes used in recursive types, as undertaken in [Mat02, Mat98], we shall diverge from the usual presentation of recursive types and use a Mendler-style formulation for the *unrolling* operation. For technical reasons concerning the meta-theoretic properties of datatype constructors discussed in Section 5.4 (specifically Proposition 5.8), we will maintain the classical formulation for *rolling* recursive types. This means that while nominally *polyrec bidirectional System F* supports recursive types using arbitrary type schemes, as would a fully Mendler-style formulation (see remarks by [UV02]), in practice the conditions under which they may be introduced are quite restricted. Additionally, the choice of Mendler-style unroll means that the programmer is burdened with manually defining the case-distinction scheme for positive datatypes.

Grammar The grammar for the statics of polyrec bidirectional System F is given in Figure 5a; the grammar of its dynamics, which extends the untyped lambda calculus, is given in Figure 5b. We omit the erasure rules, as these can be inferred directly from the two grammars and the discussion in Section 2.2.2. Similarly, we do not give a separate listing for the elaboration rules of the language of dynamics, as these can be inferred from the elaboration of the language of statics (remove context and types from judgments, and replace terms with their erasures).

To the language of types we add the unitary type $\top$, the empty type $\bot$, product types $S \times T$, sum (variant) types $S + T$, and recursive types $\mu\, R.\, T$ (where μ binds R in T). For terms, we

$$
\begin{array}{lcl}
\text{types } S, T, U, \ldots & ::= & \ldots \mid \top \mid \bot \mid S \times T \mid S + T \mid \mu\, R.\, T \\
\text{terms } e, e_1, e_2, e', \ldots & ::= & \ldots \mid \mathtt{trv} \mid \mathtt{falso}(e, S) \mid \\
& & \mathtt{pair}(e_1, e_2) \mid \mathtt{prj}_1(e) \mid \mathtt{prj}_2(e) \mid \\
& & \mathtt{inj}_1(e) \mid \mathtt{inj}_2(e) \mid \mathtt{case}(e, U)[x.e_1 \;;\; y.e_2] \mid \\
& & \mathtt{in}(e) \mid \mathtt{fold}(e_1, U)[R.x.y.e_2]
\end{array}
$$

(a) Static language (extends Figure 3a)

$$
\begin{array}{lcl}
\text{terms } u, u_1, u_2, u', \ldots & ::= & \ldots \mid \mathtt{trv} \mid \mathtt{falso}(u) \mid \\
& & \mathtt{pair}(u_1, u_2) \mid \mathtt{prj}_1(u) \mid \mathtt{prj}_2(u) \\
& & \mathtt{inj}_1(u) \mid \mathtt{inj}_2(u) \mid \mathtt{case}(u)[x.u_1 \;;\; u.t_2] \mid \\
& & \mathtt{in}(u) \mid \mathtt{fold}(u_1)[x.y.u_2]
\end{array}
$$

(b) Dynamic language (extends Figure 1)

Figure 5: Grammar of polyrec bidirectional System F

have: the single element `trv` ("trivial") of $\top$; the eliminator `falso` for $\bot$; pair introduction `pair` and first and second projections $\mathtt{prj}_1$ and $\mathtt{prj}_2$ for $S \times T$; first and second injections $\mathtt{inj}_1$ and $\mathtt{inj}_2$ and case distinction `case` for sum types; and the constructor `in` and eliminator `fold` for recursive types $\mu\, R.\, T$. These new term constructs are better understood in the context of their type inference and reduction rules, discussed in Sections 5.1 and 5.2.

The entirety of the system we present in this section has been broken down to ease digestion. It spans the grammar in Figure 5, the rules of Figure 6 for polynomial type constructors, the rules in Figure 7 for recursive types, and the rules in Figure 8 for elaboration of typing contexts and the bidirectional System F subsystem. The operational semantics of the dynamic language is given by the compatible closure of the β- and η-reduction rules in Section 2.1 augmented with the additional reduction rules for redexes in Figures 6c and 7c. As a whole, polyrec bidirectional System F minus its bidirectionality is essentially the language **EMIT** studied by Matthes in [Mat98] §4.1.

5.1 Polynomial functors

Polynomial functors are named as such because they arise from the translation to category theory of the notion of a polynomial in ring theory. Specifically, addition, multiplication, and exponentiation correspond resp. to disjoint union ($+$), pairs ($\times$), and function types ($\to$), with $\top$ the neutral element for $\times$ and $\bot$ the neutral element for $+$. Our choice to include these new type constructors is not arbitrary: axiomatic type theory *a lá* Martin-Löf [ML75] uses *W-types* as the basis of inductive datatypes, which are precisely the class of initial algebras for polynomial functors [Abb03] when these are suitably generalized to dependent types. Polynomial functors as presented here are easily representable in System F with impredicative encodings – as these types are not recursive, they share the same representation across all impredicative encodings we have considered so far.

5.1.1 Elaboration rules

Type formation rules Figure 6a gives the inference rules for judgment $\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T'$ *type* for just those cases where the subject T has been formed by a type constructor corresponding to a polynomial functor. We use label "I" on the turnstile $\vdash_{\mathrm{I}}$ to distinguish the system from that of bidirectional System F. The inference rules for recursive types are listed in Figure 7a, and the cases involving the type constructors present already in System F in Figure 8.

$$\boxed{\Gamma \vdash T \rightsquigarrow T' \ \mathit{type}}$$

$$\frac{\Gamma \vdash S \rightsquigarrow S' \ \mathit{type} \quad \Gamma \vdash T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash S + T \rightsquigarrow \forall X.\, (S' \to X) \to (T' \to X) \to X \ \mathit{type}} \textsc{Sum} \qquad \frac{}{\Gamma \vdash \bot \rightsquigarrow \forall X.\, X \ \mathit{type}} \textsc{Bot}$$

$$\frac{\Gamma \vdash S \rightsquigarrow S' \ \mathit{type} \quad \Gamma \vdash T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash S \times T \rightsquigarrow \forall X.\, (S' \to T' \to X) \to X \ \mathit{type}} \textsc{Prod} \qquad \frac{}{\Gamma \vdash \top \rightsquigarrow \forall X.\, X \to X \ \mathit{type}} \textsc{Top}$$

(a) Type formation

$$\boxed{\Gamma \vdash e \rightsquigarrow e' \in T} \quad \boxed{\Gamma \vdash T \ni e \rightsquigarrow e'}$$

$$\frac{\Gamma \vdash S \ni e \rightsquigarrow e'}{\Gamma \vdash S + T \ni \mathtt{inj}_1(e) \rightsquigarrow \Lambda\, X.\, \lambda\, x.\, \lambda\, y.\, x\ e'} \textsc{Inj}_1 \qquad \frac{\Gamma \vdash T \ni e \rightsquigarrow e'}{\Gamma \vdash S + T \ni \mathtt{inj}_2(e) \rightsquigarrow \Lambda\, X.\, \lambda\, x.\, \lambda\, y.\, y\ e'} \textsc{Inj}_2$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' \in S + T \quad \Gamma \vdash U \rightsquigarrow U' \ \mathit{type} \quad \Gamma, x{:}S \vdash U \ni e_1 \rightsquigarrow e_1' \quad \Gamma, y{:}T \vdash U \ni e_2 \rightsquigarrow e_2'}{\Gamma \vdash \mathtt{case}(e, U)[x.e_1\ ;\ y.e_2] \rightsquigarrow e' \cdot U'\ (\lambda\, x.\, e_1')\ (\lambda\, y.\, e_2') \in U} \textsc{Case}$$

$$\frac{\Gamma \vdash S \ni e_1 \rightsquigarrow e_1' \quad \Gamma \vdash T \ni e_2' \rightsquigarrow e_2'}{\Gamma \vdash S \times T \ni \mathtt{pair}(e_1, e_2) \rightsquigarrow \Lambda\, X.\, \lambda\, x.\, x\ e_1'\ e_2'} \textsc{Pair}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' \in S \times T \quad \Gamma \vdash S \rightsquigarrow S' \ \mathit{type}}{\Gamma \vdash \mathtt{prj}_1(e) \rightsquigarrow e' \cdot S'\ (\lambda\, x.\, \lambda\, y.\, x) \in S} \textsc{Prj}_1 \qquad \frac{\Gamma \vdash e \rightsquigarrow e' \in S \times T \quad \Gamma \vdash T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash \mathtt{prj}_2(e) \rightsquigarrow e' \cdot T'\ (\lambda\, x.\, \lambda\, y.\, y) \in S} \textsc{Prj}_2$$

$$\frac{}{\Gamma \vdash \top \ni \mathtt{trv} \rightsquigarrow \Lambda\, X.\, \lambda\, x.\, x} \textsc{Trv} \qquad \frac{\Gamma \vdash e \rightsquigarrow e' \in \bot \quad \Gamma \vdash U \rightsquigarrow U' \ \mathit{type}}{\Gamma \vdash \mathtt{falso}(e, U) \rightsquigarrow e' \cdot U' \in U} \textsc{Falso}$$

(b) Introduction and elimination

$$\boxed{u_1 \dashrightarrow u_2}$$

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{case}(\mathtt{inj}_1(u))[x.u_1\ ;y.u_2] & \dashrightarrow & u_1[u/x], & \mathtt{case}(\mathtt{inj}_2(u))[x.u_1\ ;y.u_2] & \dashrightarrow & u_2[u/y], \\ \mathtt{prj}_1(\mathtt{pair}(u_1, u_2)) & \dashrightarrow & u_1, & \mathtt{prj}_2(\mathtt{pair}(u_1, u_2)) & \dashrightarrow & u_2 \end{array}$$

(c) Reduction rules

Figure 6: Polynomial types

Read the judgment $\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T'$ *type* as "under a typing context Γ, the type T elaborates to T'". In inference rules and prose, we shall maintain as a notational invariant that T' and every other primed type meta-variable is formed only from type constructors of System F (see Lemma 5.3). These new types are elaborated to the types of their impredicative encodings in System F, e.g., in rule Sum if S elaborates to S' and T elaborates to T', then the type $S + T$ elaborates to the type of functions polymorphic in X and taking two function arguments – one of type $S' \to X$ and one of type $T' \to X$ – and returning a value of type X. This corresponds to the type of the case-distinction scheme for sum types.

Introduction and elimination rules Figure 6b gives the inference rules comprising the bidirectional judgments $\Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T$ (for type synthesis) and $\Gamma \vdash_{\mathrm{I}} T \ni e \rightsquigarrow e'$ (for type checking). Similar to type elaboration, we intend for primed term meta-variables to denote terms formed from bidirectional System F constructs only. Following the proof-theoretic recipe of bidirectional type-checking [WCPW04,DK19], our inference rules have the types for introduction forms *checked* and the types for elimination forms *synthesized*.

For sum types, the constructors $\mathtt{inj}_1$ and $\mathtt{inj}_2$ (rules $\textsc{Inj}_1$ and $\textsc{Inj}_2$) must be given with an argument e and checked against a type $S + T$; if e can be checked against type S (resp. T), the whole expression elaborates to a polymorphic lambda term where the first argument x (resp. second argument y) is applied to e', the elaboration of e. Elaborating to the bidirectional term language spares us from annotating for example the bound variable x with the type $S' \to X$. The eliminator `case` in rule Case is given four arguments: some term e which synthesizes a type of the form $S + T$; a type U to eliminate into; and case branches for handling the two ways e was constructed ($\mathtt{inj}_1$ or $\mathtt{inj}_2$), where the syntax $x.e_1$ indicates the binding of free occurrences of x in e_1. The body of each case branch is *checked* against type U under a context extended with the assumption x has type S (resp. y has type T). Finally, we elaborate the term given by applying e' (the elaboration of e) to the type U' and elaborated case bodies e'_1 and e'_2, where lambda abstractions bind the free variables x and y.

A similar reading can be given for the constructor $\mathtt{pair}(e_1, e_2)$ and eliminators $\mathtt{prj}_1$ and $\mathtt{prj}_2$ for pairs. We remark upon a minor asymmetry in the presentation of sums and products: for the former, we did not need to appeal to the judgment $\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T'$ *type* in the rule Case for elimination, whereas we do require this for the rules $\textsc{Prj}_1$ and $\textsc{Prj}_2$. The discrepancy arises from the fact that the impredicative encoding of $S \times T$ corresponds to the *positive* presentation of product types, in which there is a single eliminator $\mathtt{unpack}(e_1)[x.y.e_2]$ in which the first and second components of the pair e_1 are resp. bound by x and y in e_2. For our extended language, we choose instead the more familiar *negative* presentation (c.f. [nLa12] for the equivalence of the two presentations).

Finally, the unitary type $\top$ has a single constructor `trv` (which elaborates to the polymorphic identity function) and no eliminator, and the empty type $\bot$ has no constructor and single eliminator $\mathtt{falso}(e, U)$ which has type U, where U is an arbitrary type well-formed under the current typing context Thus, `falso` embodies the principle of *ex falso quodlibet*.

Reduction rules When axiomatically introducing new type constructors to a (constructive) type theory with associated introduction and elimination forms, one must also at least say what the *computation* (β) laws are for them – that is, how eliminators compute over constructors. Figure 6c gives this for the counterparts in the dynamic language of the term constructs we have been discussing in the static language. As our dynamic language is intended to be a definitional extension to the untyped lambda calculus, we must take care that these reduction rules agree with the ordinary notion of β-reduction for the elaborations of these untyped terms (though perhaps with some constant overhead in the number of steps taken). For case-distinction on sum types, we take the left or right branch depending on whether the scrutinee was constructed

with $\mathtt{inj}_1(u)$ or $\mathtt{inj}_2(u)$, replacing all occurrences of the bound variable for that branch with the term u. The computation rules for the first and second projections for terms $\mathtt{pair}(u_1, u_2)$ are as expected.

Example: Booleans With polynomial type constructors, we can give definitions for many non-recursive datatypes. To illustrate, consider the definition of Booleans and their *if-then-else* elimination principle (`ite`):

$$
\begin{array}{rcl}
\mathtt{Bool} & =_{\mathbf{df}} & 1 + 1 \\
\\
\mathtt{tt} & =_{\mathbf{df}} & \chi\ \mathtt{Bool}\ -\ \mathtt{inj}_1(\mathtt{trv}) \\
\mathtt{ff} & =_{\mathbf{df}} & \chi\ \mathtt{Bool}\ -\ \mathtt{inj}_2(\mathtt{trv}) \\
\\
\mathtt{ite} & =_{\mathbf{df}} & \chi\ (\forall X.\, X \to X \to \mathtt{Bool} \to X)\ - \\
 & & \quad \Lambda X.\, \lambda x.\, \lambda y.\, \lambda z.\, \mathtt{case}(z, X)[\, x'.\, x\ ;\ \ y'.\, y] \\
\\
|\mathtt{ite}|\ u_1\ u_2\ |\mathtt{tt}| & \overset{*}{\dashrightarrow} & u_1 \\
|\mathtt{ite}|\ u_1\ u_2\ |\mathtt{ff}| & \overset{*}{\dashrightarrow} & u_2
\end{array}
$$

5.2 Recursive types

The origin of recursive types in programming languages can be traced back to the work by Dana Scott [Sco76] to give a denotational semantics for the untyped lambda calculus. Internalized within a language, such as System F extended with recursive types, if T is a type in which $R \in FV(T)$ then the type $\mu\, R.\, T$ is to be interpreted as a self-referential type wherein the free occurrences of R in T stand in for recursive occurrences of $\mu\, R.\, T$ itself. For example, with polynomial types the type of natural numbers can be defined as $\mu\, R.\, 1 + R$ – natural numbers are either zero, corresponding to the left disjunct 1 wherein there is no useful subdata, or the successor of another natural number, corresponding to the right disjunct R which is to be interpreted as the type $\mu\, R.\, 1 + R$ itself. As another example, lists of some element type S are given by the type $\mu\, R.\, 1 + (S \times R)$.

True recursive types $\mu\, R.\, T$ can be characterized by their relation to their one-step unfolding $T[\mu\, R.\, T/R]$. In one formulation, there is an isomorphism given by $\mathtt{roll} : T[\mu\, R.\, T/R] \to \mu\, R.\, T$ and its inverse `unroll`. Alternatively, these two types may be seen as *equal by definition*, and no new term constructs are introduced. These two formulations are respectively called *iso-recursive types* and *equi-recursive types*; this terminology was introduced by Crary et al. in [CHP99], though the formulations themselves existed much earlier (c.f. [Pie02] §20.4).

In this report, we are interested only in definitional extensions of System F. However, recursive types as just discussed enable the definition of a general fixpoint combinator [MJ69], which is not possible in System F. Therefore (and at the cost of somewhat misleading terminology), this report takes a rather different approach by using as a semantics the Mendler-style version of "free" recursive types in System F [Wad90, Men91]. In particular, this means that an unfolding operation $\mu\, R.\, T \to T[\mu\, R.\, T/R]$ is *not* given, and only definable for types T in which R only occurs positively. The type schemes for natural numbers and lists fall in this class; the type $\mu\, R.\, \underline{R} \to R$ does not (with the offending negative occurrence underlined).

5.2.1 Elaboration rules

Type formation rules Figure 7a gives the inference rule for the judgment $\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T'$ *type* for just that case where the subject is of the form $\mu\, R.\, T$. This definition is derived from the type inference rule for `miter`, the Mendler-style iteration scheme (Section 4.1.4): a function

$$\boxed{\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}$$

$$\frac{\Gamma, R \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash_{\mathrm{I}} \mu\, R.\, T \rightsquigarrow \forall\, X.\, (\forall\, R.\, (R \to X) \to T' \to X) \to X \ \mathit{type}}\ \textsc{Rec}$$

(a) Recursive types: type formation

$$\boxed{\Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T} \quad \boxed{\Gamma \vdash_{\mathrm{I}} T \ni e \rightsquigarrow e'}$$

$$\frac{\Gamma \vdash_{\mathrm{I}} T[\mu\, R.\, T/R] \ni e \rightsquigarrow e' \quad \Gamma \vdash_{\mathrm{I}} \mu\, R.\, T \rightsquigarrow S \ \mathit{type}}{\Gamma \vdash_{\mathrm{I}} \mu\, R.\, T \ni \mathtt{in}(e) \rightsquigarrow \Lambda\, X.\, \lambda\, a.\, a \cdot S\ (\lambda\, x.\, x \cdot X\ a)\ e'}\ \textsc{In}$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e_1 \rightsquigarrow e_1' \in \mu\, R.\, T \quad \Gamma \vdash_{\mathrm{I}} U \rightsquigarrow U' \ \mathit{type} \quad \Gamma, R, x : R \to U, y : T \vdash_{\mathrm{I}} U \ni e_2 \rightsquigarrow e_2'}{\Gamma \vdash_{\mathrm{I}} \mathtt{fold}(e_1, U)[R.x.y.e_2] \rightsquigarrow e_1' \cdot U'\ (\Lambda\, R.\, \lambda\, x.\, \lambda\, y.\, e_2') \in U}\ \textsc{Fold}$$

(b) Recursive types: introduction and elimination

$$\boxed{u_1 \dashrightarrow u_2}$$

$$\mathtt{fold}(\mathtt{in}(u_1))[x.y.u_2] \quad \dashrightarrow \quad u_2[(\lambda\, z.\, \mathtt{fold}(z)[x.y.u_2])/x, u_1/y]$$

(c) Recursive types: reduction rules

Figure 7: Recursive types

polymorphic in X, which takes another function polymorphic in R mapping functions $R \to X$ to functions $T' \to X$ (where T' is the elaboration of T, both of which having R as a free variable), and returning a value of type X.

Introduction and elimination rules Figure 7b gives the introduction rule In and elimination rule Fold for our formulation of recursive types. In In, the typing rule for the constructor `in` is the same as that expected for `roll` for iso-recursive types: $\mathtt{in}(e)$ can be checked against the type $\mu\, R.\, T$ if e can be checked with type $T[\mu\, R.\, T/R]$. To keep the type inference rule brief, we use the premise $\Gamma \vdash \mu\, R.\, T \rightsquigarrow S \ \mathit{type}$ to make the meta-variable S a "local definition" of the rule (the definition of S is determined by the rule Rec and by inversion of the type formation judgment, as well as by the fact that the system of inference rules is algorithmic). The expression $\mathtt{in}(e)$ then elaborates to a function polymorphic in X taking an argument a of type $\forall\, R.\, (R \to X) \to T' \to X$ (where T' is the elaboration of T) and: instantiates the type argument of a to S; provides a function of type $(S \to X)$ by applying the given S argument to a itself $(\lambda\, x.\, x \cdot X\ a)$; and provides as the final argument e', the elaboration of e.

The elimination rule Fold is the point of divergence from classical formulations of recursive types. Rather than an unrolling operator, it facilitates the definition of iterative functions in the Mendler style. The expression $\mathtt{fold}(e_1, U)[R.x.y.e_2]$ synthesizes type U when the term argument e_1 synthesizes type $\mu\, R.\, T$, the given type U is well-formed under Γ, and the body e_2 of the function being recursively defined can be checked against type U assuming that x has type $R \to U$ (for a fresh type variable R) and y has type T (wherein R may occur free). If e_1 elaborates to e_1' and e_2 elaborates to e_2' (under a suitably extended typing context), then $\mathtt{fold}(e_1, U)(R.x.y.e_2)$ elaborates to an application where: the head is e_1'; the type argument

given is U' (the elaboration of U); and the term argument is a function with body e_2' where the free variables R, x, and y of e_2' are bound by lambda abstractions.

Reduction rules We now address how the eliminator `fold` computes over values constructed with `in` in the dynamic language. The expression $\texttt{fold}(\texttt{in}(u_1))[x.y.u_2]$ reduces to u_2 with free variables x (the handle for making recursive calls) and y (our unfolded data) substituted resp. for $\lambda\, z.\, \texttt{fold}(z)[x.y.u_2]$ and u_1. This corresponds to the commuting diagram of Definition 4.14 (instantiating the object R with μF and morphism $a : R \to \mu F$ with $id_{\mu F}$).

Example: natural numbers With both polynomial and recursive types, we can give definitions for many inductive datatypes. We illustrate with the example of natural numbers.

$$
\begin{array}{rcl}
\texttt{Nat} & =_{\mathbf{df}} & \mu\, R.\, 1 + R \\
\\
\texttt{zero} & =_{\mathbf{df}} & \chi\ \texttt{Nat}\ -\ \texttt{in}(\texttt{inj}_1(\texttt{trv})) \\
\texttt{succ} & =_{\mathbf{df}} & \chi\ (\texttt{Nat} \to \texttt{Nat})\ -\ \lambda\, x.\, \texttt{in}(\texttt{inj}_2(x)) \\
\\
\texttt{iter} & =_{\mathbf{df}} & \chi\ (\forall\, X.\, X \to (X \to X) \to \texttt{Nat} \to X)\ - \\
 & & \Lambda\, X.\, \lambda\, x_1.\, \lambda\, x_2.\, \lambda\, y.\, \texttt{fold}(y, X)[R.z.y'.\texttt{case}(y', X)[y_1.x_1\ ;\ y_2.x_2\ (z\ y_2)]]
\end{array}
$$

$$
\begin{array}{rcl}
|\texttt{iter}|\ u_1\ u_2\ |\texttt{zero}| & \overset{*}{\dashrightarrow} & u_1 \\
|\texttt{iter}|\ u_1\ u_2\ (|\texttt{succ}|\ u) & \overset{*}{\dashrightarrow} & u_2\ (\texttt{fold}(u)[z.y'.\texttt{case}(y')[y_1.u_1\ ;\ y_2.u_2\ (z\ y_2)]]) \\
 & & \overset{*}{\dashleftarrow}\ u_2\ (|\texttt{iter}|\ u_1\ u_2\ u)
\end{array}
$$

Notice that for the reduction rule involving `succ`, we do not have the more desirable behavior $|\texttt{iter}|\ u_1\ u_2\ (|\texttt{succ}|\ u) \overset{*}{\dashrightarrow} u_2\ (|\texttt{iter}|\ u_1\ u_2\ u)$ expected for the iteration scheme for natural numbers (Section 4.1.1). Instead, this holds only up to *joinability* of the reduction relation: there exists some term u_3 (given above) to which both expressions can reduce.

5.3 Contexts and the bidirectional System F subsystem

To complete our discussion of the elaboration rules for polyrec bidirectional System F, we must also give a (mostly tedious and unsurprising) account of how the bidirectional System F subsystem elaborates. This is done for types and terms in Figure 8. In addition, in order even to state certain properties of the inference system, such as claiming that the elaborations of well-typed terms are themselves well-typed in bidirectional System F, we require a notion of elaboration for *typing contexts* – after all, typing contexts for the external language are not necessarily valid typing contexts of the internal language, as they may contain declarations of term variables with polynomial and recursive types. This is also given in Figure 8.

Typing contexts In Figure 8a, read the judgment $\vdash_{\mathrm{I}} \Gamma \rightsquigarrow \Gamma'$ as "the typing context Γ elaborates to Γ'". The three inference rules are straight-forward: the empty context elaborates to itself, type variables elaborate to themselves, and term variable declarations $x : T$ elaborate to $x : T'$ if T elaborates to T'.

Types In addition to elaborating polynomial and recursive types, the judgment $\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T'$ *type* ensures the type T is well-formed, i.e., that its free variables are declared in the typing context Γ (replacing the explicit checks we used in Figure 3b). Type variables elaborate to themselves, and arrows and type quantification are first decomposed, the corresponding subexpressions elaborated, and re-composed with the original type constructor.

$$\boxed{\vdash_{\mathrm{I}} \Gamma \rightsquigarrow \Gamma'}$$

$$\frac{}{\vdash_{\mathrm{I}} \emptyset \rightsquigarrow \emptyset} \quad \frac{\vdash_{\mathrm{I}} \Gamma \rightsquigarrow \Gamma'}{\vdash_{\mathrm{I}} \Gamma, X \rightsquigarrow \Gamma', X} \quad \frac{\vdash_{\mathrm{I}} \Gamma \rightsquigarrow \Gamma' \quad \Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}{\vdash_{\mathrm{I}} \Gamma, x{:}T \rightsquigarrow \Gamma', x{:}T'}$$

(a) Typing contexts

$$\boxed{\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}$$

$$\frac{X \in DV(\Gamma)}{\Gamma \vdash_{\mathrm{I}} X \rightsquigarrow X \ \mathit{type}} \quad \frac{\Gamma \vdash_{\mathrm{I}} S \rightsquigarrow S' \ \mathit{type} \quad \Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash_{\mathrm{I}} S \to T \rightsquigarrow S' \to T' \ \mathit{type}} \quad \frac{\Gamma, X \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash_{\mathrm{I}} \forall\, X.\, T \rightsquigarrow \forall\, X.\, T' \ \mathit{type}}$$

(b) Types

$$\boxed{\Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T}$$

$$\frac{\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type} \quad \Gamma, x{:}T \vdash_{\mathrm{I}} e \rightsquigarrow e' \in S}{\Gamma \vdash_{\mathrm{I}} \lambda\, x{:}T.\, e \rightsquigarrow \lambda\, x{:}T'.\, e' \in T \to S} \quad \frac{\Gamma \vdash_{\mathrm{I}} e_1 \rightsquigarrow e_1' \in S \to T \quad \Gamma \vdash_{\mathrm{I}} S \ni e_2 \rightsquigarrow e_2'}{\Gamma \vdash_{\mathrm{I}} e_1\ e_2 \rightsquigarrow e_1'\ e_2' \in T}$$

$$\frac{\Gamma, X \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T}{\Gamma \vdash_{\mathrm{I}} \Lambda\, X.\, e \rightsquigarrow \Lambda\, X.\, e' \in \forall\, X.\, T} \quad \frac{\Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in \forall\, X.\, S \quad \Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type}}{\Gamma \vdash_{\mathrm{I}} e \cdot T \rightsquigarrow e \cdot T' \in [T/X]S}$$

$$\frac{}{\Gamma \vdash_{\mathrm{I}} x \rightsquigarrow x \in \Gamma(x)} \quad \frac{\Gamma \vdash_{\mathrm{I}} T \rightsquigarrow T' \ \mathit{type} \quad \Gamma \vdash_{\mathrm{I}} T \ni e \rightsquigarrow e'}{\Gamma \vdash_{\mathrm{I}} \chi\ T \text{ - } e \rightsquigarrow \chi\ T' \text{ - } e' \in T}$$

(c) Synthesizing terms

$$\boxed{\Gamma \vdash_{\mathrm{I}} T \ni e \rightsquigarrow e'}$$

$$\frac{T_1 = T_2 \quad \Gamma \vdash_{\mathrm{I}} T_2 \rightsquigarrow T_2' \ \mathit{type} \quad \Gamma, x{:}T_2 \vdash_{\mathrm{I}} S \ni e \rightsquigarrow e'}{\Gamma \vdash_{\mathrm{I}} T_1 \to S \ni \lambda\, x{:}T_2.\, e \rightsquigarrow \lambda\, x{:}T_2'.\, e'} \quad \frac{\Gamma, x{:}T \vdash_{\mathrm{I}} S \ni e \rightsquigarrow e'}{\Gamma \vdash_{\mathrm{I}} T \to S \ni \lambda\, x.\, e \rightsquigarrow \lambda\, x.\, e'}$$

$$\frac{\Gamma, X \vdash_{\mathrm{I}} T \ni e \rightsquigarrow e'}{\Gamma \vdash_{\mathrm{I}} \forall\, X.\, T \ni \Lambda\, X.\, e \rightsquigarrow \Lambda\, X.\, e'} \quad \frac{\neg \mathit{Abs}(e) \quad \Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T_2 \quad T_1 = T_2}{\Gamma \vdash_{\mathrm{I}} T_1 \ni e \rightsquigarrow e'}$$

(d) Checkable terms

$$\boxed{\vdash_{\mathrm{I}} u \rightsquigarrow u'}$$

(e) Untyped terms (rules omitted)

Figure 8: Additional elaboration rules

Synthesizing and checkable terms The elaborating type inference rules for synthesizing terms in Figure 8c are mostly the same as those discussed in Section 3 (Figure 3b) for elaborating bidirectional System F to annotated System F, modulo the change in the target language. Notable exceptions are the rule for polymorphic type instantiation $e \cdot T$, in which the elaborated term $e' \cdot T'$ is an instantiation using the elaborated type T', and the rule for explicit type annotations wherein the annotation remains in the elaborated term and the type is replaced with its elaboration. The situation is similar for checkable terms in Figure 8d. Since bare abstractions are also term constructors of our internal language, the rule for them in polyrec bidirectional System F elaborates a bare abstraction.

Untyped terms The last judgment we need, $\vdash_{\mathrm{I}} u \rightsquigarrow u'$, is for elaborating the dynamic terms of polyrec bidirectional System F. As discussed earlier, we omit the rules comprising this judgment as they consist of a repetition of the rules for elaborating typed terms where typing contexts and well-formedness checks for types have been removed and the subjects and their elaborations have been replaced by their erasures. This also means removing from the judgment those rules in which the erasure of the subject does not change between the conclusion and the premise, such as the rules for explicit type annotations and polymorphic instantiation. To give an example of a rule that remains, the rule for elaborating untyped `in` would thus be:

$$\frac{\vdash_{\mathrm{I}} u \rightsquigarrow u'}{\vdash_{\mathrm{I}} \mathtt{in}(u) \rightsquigarrow \lambda\, a.\, a\ (\lambda\, x.\, x\ a)\ u'}$$

5.4 Properties

Recall from Section 3.2 that in order for the system of inference rules given above to sensibly be considered *elaborating type inference rules* – which is to say that they provide a translational semantics for the source language in the target language – it must satisfy certain properties:

- the inference rules must describe an **algorithm** for the translation;
- the inference rules must be **sufficient** for typing terms of the internal language;
- the translation of the external language must be **sound** with respect to the statics and dynamics of the internal language.

Unlike the case for bidirectional type inference, we have extended the type and dynamic term language, so the statement of our theorems must be adjusted accordingly.

Proposition 5.1 (Algorithmic type inference)**.**

1. *Every rule is mode-correct for a moding* $\Gamma^- \vdash_I e^- \rightsquigarrow e'^+ \in T^+$ *and* $\Gamma^- \vdash_I T^- \ni e^- \rightsquigarrow e'^+$ *(where* $-$ *and* $+$ *indicate resp. input and output modes).*
2. *For all* Γ*,* e*,* t*, and* T
   - *there is at most one inference rule whose conclusion matches the syntactic form of the judgment* $\Gamma \vdash_I e \rightsquigarrow e' \in T$
   - *there is at most one inference rule whose conclusion matches the syntactic form of the judgment* $\Gamma \vdash_I T \ni e \rightsquigarrow e'$

Proof sketch. By inspection of the type inference rules. □

Proposition 5.2 (Sufficiency)**.** *For all* Γ*,* e*, and* T *in the language bidirectional System F, and for all* t *in the language of fully-annotated System F:*

1. *if* $\Gamma \vdash e \rightsquigarrow t \in T$ *then* $\Gamma \vdash_I e \rightsquigarrow e \in T$

2. *if* $\Gamma \vdash T \ni e \rightsquigarrow t$ *then* $\Gamma \vdash_I T \ni e \rightsquigarrow e$

Proof sketch. By mutual induction on the assumed typing derivation, replacing each rule used in the derivation with the corresponding rule of the bidirectional System F subsystem. □

Lemmas 5.3 and 5.4 are useful for showing static and dynamic soundness. They state resp. that elaborated types and terms really are in the grammar of bidirectional System F, and that terms in the external language whose type can be synthesized and which are not abstractions elaborate to terms which are not abstractions (this is needed when showing type soundness for the rule corresponding to Sub in polyrec bidirectional System F).

Lemma 5.3 (Grammar soundness)**.**

1. *For all* Γ, T, *and* T', *if* $\Gamma \vdash_I T \rightsquigarrow T'$ *type then* T' *is formed entirely by the type constructors of Figure 2a.*

2. *For all* Γ, e, e', *and* T, *if* $\Gamma \vdash_I e \rightsquigarrow e' \in T$ *(or if* $\Gamma \vdash_I T \ni e \rightsquigarrow e'$*) then* e' *is formed entirely by the term constructors of Figure 3a.*

3. *For all* u *and* u', *if* $\vdash_I u \rightsquigarrow u'$ *then* u' *is a term of untyped lambda calculus.*

Proof sketch. By mutual induction on the assumed typing derivation, using the fact that each rule of the static language introduces only type and term constructors of bidirectional System F, and each rule of the dynamic language introduces only constructs of untyped lambda calculus. □

Lemma 5.4. *For all* Γ, T, e, *and* e', *if* $\Gamma \vdash_I e \rightsquigarrow e' \in T$ *and* $\neg Abs(e)$ *then* $\neg Abs(e')$.

Prook sketch. By case analysis on the assumed derivation. □

We now elaborate on the difference between the statement of elaboration soundness in Conjecture 5.5 and that of Conjecture 3.3. Concerning the statics, if we have a derivation of $\Gamma \vdash_\mathrm{I} e \rightsquigarrow e' \in T$, we cannot expect that e' synthesizes T, but rather that it synthesizes the type T' to which T elaborates under Γ. Furthermore, we cannot expect to be able to synthesize T' from e' under Γ, but only under the context Γ' to which Γ elaborates. Concerning the dynamics, every reduction step $u_1 \dashrightarrow u_2$ in the source language must correspond to number of steps (at least one, and preferably bounded by some constant) for the elaborated terms in the target language.

Conjecture 5.5 (Static and dynamic soundness)**.**

1. *For all* Γ, T, *and* T', *if* $\Gamma \vdash_I T \rightsquigarrow T'$ type *and* $\vdash_I \Gamma \rightsquigarrow \Gamma'$ *then* T' *is a well-formed type under* Γ', *i.e.,* $FV(T') \subseteq DV(\Gamma')$

2. *For all* Γ, e, e', *and* T,

   - *if* $\Gamma \vdash_I e \rightsquigarrow e' \in T$ *(resp.* $\Gamma \vdash_I T \ni e \rightsquigarrow e'$*)*
   - *and* $\vdash_I \Gamma \rightsquigarrow \Gamma'$
   - *and* $\Gamma \vdash_I T \rightsquigarrow T'$ *type*

   then $\Gamma' \vdash e' \rightsquigarrow t \in T'$ *(resp.* $\Gamma' \vdash T' \ni e' \rightsquigarrow t$*) for some* t.

3. *For all* u_1 *and* u_2 *of the dynamic external language, and for all* u_1' *and* u_2' *of the dynamic internal language:*

- *if* $u_1 \dashrightarrow u_2$
- *and* $\vdash_I u_1 \rightsquigarrow u_1'$
- *and* $\vdash_I u_2 \rightsquigarrow u_2'$

then $u_1' \dashrightarrow \circ \overset{*}{\dashrightarrow} u_2'$.

An easy consequence of dynamic soundness is that equivalence of terms by the operational semantics in the dynamic external language implies equivalence of their elaborations in the internal language.

Corollary 5.5.1. *For all u_1 and u_2 of the dynamic external language and for all u_1' and u_2' of the dynamic internal language such that $\vdash_I u_1 \rightsquigarrow u_1'$ and $\vdash_I u_2 \rightsquigarrow u_2'$, if $u_1 \cong u_2$ then $u_1' \cong u_2'$.*

Proof sketch. By induction on derivation of $u_1 \cong u_2$, appealing to Proposition 5.5(3) for each revealed reduction step. □

Importing meta-theoretic properties The main benefit to considering only definitional extensions of some internal language is that, after having established that our elaboration rules provide a sound translational semantics of the external language, we can re-use meta-theoretic results of the internal language. In particular, static soundness of the translation to bidirectional System F gives us *logical soundness* of our extended type system (not all types are inhabited), and static and dynamic soundness gives us *termination* for well-typed programs.

Corollary 5.5.2 (Logical soundness)**.**
There are no closed terms e and e' such that $\emptyset \vdash_I e \rightsquigarrow e' \in \forall\, X.\, X$

Proof (of negation). Assume that there is some e and e' such that $\emptyset \vdash_{\mathrm{I}} e \rightsquigarrow e' \in \forall\, X.\, X$. It is clear by inspection of the rules of Figures 8b and 8b that we can construct derivations of $\vdash_{\mathrm{I}} \emptyset \rightsquigarrow \emptyset$ and $\emptyset \vdash_{\mathrm{I}} \forall\, X.\, X \rightsquigarrow \forall\, X.\, X$ *type*. Thus, by static soundness $\emptyset \vdash e' \rightsquigarrow t \in \forall\, X.\, X$ for some t. But, this contradicts logic soundness for bidirectional System F (which is itself a consequence of logical soundness of annotated System F and static soundness of the translation into it from bidirectional System F). □

Corollary 5.5.3 (Termination)**.**
For all Γ, e, e', T, if $\Gamma \vdash_I e \rightsquigarrow e' \in T$ (resp. $\Gamma \vdash_I T \ni e \rightsquigarrow e'$), then $|e|$ is terminating.

Proof (of negation). Assume there exists a Γ, e, e', and T such that $\Gamma \vdash_{\mathrm{I}} e \rightsquigarrow e' \in T$. Furthermore, assume there is an infinite sequence of reductions $|e| \dashrightarrow u_1 \dashrightarrow u_2 \dashrightarrow \ldots$. By coinduction on this infinite sequence we can produce an infinite reduction sequence for $|e'|$, contradicting termination for bidirectional System F (since e' is a well-typed term of bidirectional System F by static soundness), as follows.

First, we make use of the implicit fact that every term of dynamic polyrec System F elaborates to a term of untyped lambda calculus. So, for each term $(u_i)_{i \in \mathbb{N}}$ of the infinite reduction sequence starting from $u_0 = |e|$, $\vdash_{\mathrm{I}} u_i \rightsquigarrow u_i'$ for some u_i'. Next, observe that by dynamic soundness $u_i' \dashrightarrow \circ \overset{*}{\dashrightarrow} u_{i+1}'$. We thus have an infinite reduction sequence starting from $u_0' = |e'|$. □

Some meta-theoretic properties of interest for the source language are not so easily imported from the internal language. Consider *confluence*[3], the property that any two terms produced from reducing a term u are joinable. Because our operational semantics is untyped, we could easily add reduction rules to the source language that take introduction forms (e.g., $\mathtt{inj}_1(u)$) to their lambda representations (e.g., $\lambda\, x.\, \lambda\, y.\, x\ u$). This would preserve dynamic soundness, and

[3]Matthes, [Mat98] §8.2: "Unfortunately there is no result on preservation of confluence via embeddings."

destroy confluence of the system. Fortunately, polyrec bidirectional System F is essentially the system **EMIT** studied by Matthes in §4.1.1 of [Mat98], for which confluence was claimed in §8.2. We take that as license to assert the following:

Conjecture 5.6 (Confluence)**.** *For all u, u_1, and u_2 in the external dynamic language, if $u \dashrightarrow^* u_1$ and $u \dashrightarrow^* u_2$ then there exists some u_3 such that $u_1 \dashrightarrow^* u_3$ and $u_2 \dashrightarrow^* u_3$.*

Meta-theoretic properties of new features The final sort of property we will consider are those concerning the nature of inductive types themselves. By verifying expected characteristics of features we have added (where our expectations are informed by the feature's mathematical semantics), we can be assured that we have not soundly translated rubbish.

Inductive types are expected to enjoy the following properties [MGM04]:

1. the ability to write (terminating) programs over data using **case-analysis and structural recursion** [Gim94];
2. **disjointness** and **injectivity** of constructors [CT95]; and
3. **acyclicity** (for example, there are no terms u such that $u \cong \mathtt{in}(u)$)

For 1. we content ourselves with Mendler-style iteration (as discussed above, the case-distinction scheme is not in general available for recursive types formed from arbitrary type schemes). The properties listed in 2. and 3., in the context of the works cited, are considered for interactive proof assistants based on dependent type theory, and the goal of those works is to establish such properties *within* the theory automatically (and even better, generically). Our concern here is to formulate these properties meta-theoretically with respect to our translational semantics.

Definition 5.7 (Constructors)**.** Meta-linguistically, a constructor $\mathbf{c}$ of our dynamic external language is an element of the set of pairs $\{(\mathtt{trv}, 0), (\mathtt{pair}, 2), (\mathtt{inj}_1, 1), (\mathtt{inj}_2, 1), (\mathtt{in}, 1)\}$, where the first component is a string *identifier* for the constructor and the second is the *arity* of the constructor, written $\mathit{arity}(\mathbf{c})$. If $\vec{u}$ is a sequence of terms whose length $\#\vec{u}$ is equal to $\mathit{arity}(\mathbf{c})$, then we write $\mathbf{c}(\vec{u})$ to denote the term whose outermost formation is the identifier of $\mathbf{c}$ with arguments $\vec{u}$. For example, if $\mathbf{c} = (\mathtt{trv}, 0)$, then $\mathbf{c}()$ denotes (the term) $\mathtt{trv}$; if $\mathbf{c} = (\mathtt{pair}, 2)$, then $\mathbf{c}(u_1, u_2)$ denotes the term $\mathtt{pair}(u_1, u_2)$.

Proposition 5.8 (Injectivity of constructors)**.** *For all Γ, e_1, e_2, e_1', e_2', T_1, T_2, $\vec{u_1}$, $\vec{u_2}$, and $\mathbf{c}$*

- *if $\Gamma \vdash_I e_1 \rightsquigarrow e_1' \in T_1$ and $\Gamma \vdash_I e_2 \rightsquigarrow e_2' \in T_2$*
- *and $\#\vec{u_1} = \#\vec{u_2} = \mathit{arity}(\mathbf{c})$*
- *and $|e_1| = \mathbf{c}(\vec{u_1})$ and $|e_2| = \mathbf{c}(\vec{u_2})$*
- *and $\mathbf{c}(\vec{u_1}) \cong \mathbf{c}(\vec{u_2})$*

then $\vec{u_1} \cong \vec{u_2}$ (the corresponding terms of the two sequences are $\beta\eta$-equivalent).

Proof. By an appeal to termination (Corollary 5.5.3) and confluence (Conjecture 5.6), from $\mathbf{c}(\vec{u_1}) \cong \mathbf{c}(\vec{u_2})$ we have that both reduce to the same value v. By inspection of the reduction rules, v has the form $\mathbf{c}(\vec{v})$ and the reduction sequence producing v consists entirely of reducing $\vec{u_1}$ and $\vec{u_2}$ to $\vec{v}$. Thus, $\vec{u_1} \cong \vec{u_2}$. □

Proposition 5.9 (Disjointness of constructors)**.** *Assume $\mathbf{c}_1$ and $\mathbf{c_2}$ such that $\mathbf{c}_1 \neq \mathbf{c}_2$. For all sequences of terms $\vec{u_1}$ and $\vec{u_2}$ such that $\mathit{arity}(\mathbf{c}_1) = \#\vec{u_1}$ and $\mathit{arity}(\mathbf{c}_2) = \#\vec{u_2}$, and for all u_1' and u_2' such that $\vdash_I \mathbf{c_1}(\vec{u_1}) \rightsquigarrow u_1'$ and $\vdash_I \mathbf{c_2}(\vec{u_2}) \rightsquigarrow u_2'$, $u_1' \ncong u_2'$.*

Proof (of negation) sketch. By inspection of the elaboration rules for introduction forms, making use of the fact that they each produce lambda abstractions whose body is an application whose head is a bound variable. We consider a particularly tricky case:

Case $\mathbf{c_1} := \texttt{pair}$, $\mathbf{c_2} = \texttt{in}$: Assume arbitrary u_1, u_2, and u_3 in the external dynamic language. From rules PAIR and IN, it is clear that $\vdash_{\mathrm{I}} \texttt{pair}(u_1, u_2) \rightsquigarrow \lambda\, x.\, x\ u_1'\ u_2'$ and $\vdash_{\mathrm{I}} \texttt{in}(u_3) \rightsquigarrow \lambda\, a.\, a\ (\lambda\, x.\, x\ a)\ u_3'$ for some u_1', u_2', and u_3' such that $(\vdash_{\mathrm{I}} u_i \rightsquigarrow u_i')_{i \in \{1,2,3\}}$. Without loss of generality, the bound variable a is assumed to be fresh with respect to $FV(u_1')$. Since the term $\lambda\, x.\, x\ a$ is normal, $u_1' \ncong \lambda\, x.\, x\ a$ and thus that the two elaborations of the introduction forms are $\beta\eta$-inequivalent. □

To show acyclicity in full generality would require us to develop a notion that a term u_1 is *constructor-guarded* in a term u_2. Informally, this means that u_2 is formed by a nesting of constructors in which u_1 occurs as an argument, for example $u_2 = \texttt{pair}(u_3, \texttt{inj}_1(u_1))$. For simplicity, we consider only the special case in which u_1 is *shallowly constructor-guarded* in u_2.

Definition 5.10 (Shallow constructor-guarding)**.** Let u_1 and u_2 be dynamic external terms. We say that u_1 is *shallowly constructor-guarded* in u_2 if there is some constructor $\mathbf{c}$ and sequence of terms $\vec{u}$ such that $u_2 = \mathbf{c}(\vec{u})$ and $u_1 \in \vec{u}$.

Proposition 5.11 (Shallow acyclicity)**.** *For all* Γ, e_1, e_2, e_1', e_2', T_1, *and* T_2*:*

- *if* $\Gamma \vdash_I e_1 \rightsquigarrow e_1' \in T_1$
- *and* $\Gamma \vdash_I e_2 \rightsquigarrow e_2' \in T_2$
- *and* $|e_1|$ *is shallowly constructor-guarded in* $|e_2|$

then $|e_1'| \ncong |e_2'|$.

Proof (of negation) sketch. Assuming $|e_1'| \cong |e_2'|$, by confluence (of untyped lambda calculus) and termination (of bidirectional System F, using static soundness and the assumption that e_1 and e_2 are well-typed terms of polyrec bidirectional System F), these both reduce to the same value v. By inspection of the elaboration rules for introduction forms, $|e_2'|$ is formed by some number of lambda abstractions whose body is a variable-headed application with $|e_1'|$ as an argument, meaning v must contain itself as a sub-expression, which is impossible. □

6 Conclusion and Future Work

In this report, we have studied the rôle of *elaborating type inference rules* for giving a translational semantics for two language features, *bidirectional type inference* and *inductive definitions*. Having fixed some core *internal language*, such as annotated or bidirectional System F, we can use elaboration to guide the design of a more sophisticated (and more convenient) *external language* with features given as *definitional extensions*. This setup allows us to import meta-theoretic results of the internal language by establishing *static and dynamic soundness* of the translation. Inductive definitions are rather complex machinery in any language, and so in preparation for giving a translational semantics for these we studied several mathematical (*algebraic*) semantics first. We chose a Mendler-style semantics for simplicity in designing the resulting external language, as it relieves us (the designers) from the burden of elaborating positivity witnesses for type schemes – at the cost of burdening others (language users) with manually defining the case-distinction scheme for inductive types when this is required.

CDLE With a feature as massive as inductive definitions, we might wish to consider also several enticing features in its orbit. For the discussion of future work, we turn away from System F and definitional extensions of it and instead consider these features informally in the context of the *calculus of dependent lambda eliminations* (**CDLE**) [Stu17, Stu18b]. **CDLE** is an impredicative Curry-style pure type system that extends the *calculus of constructions* (**CC**) [CH88]. As **CC** contains System F as a subsystem, inductive datatypes as we have considered them here can also be defined using impredicative encodings in **CC** (and therefore also in **CDLE**). Unlike **CC**, in **CDLE** it is possible to *generically* derive the induction principle for these encodings [Stu18a, FS18]. By generically, we mean that the derivation is carried for an arbitrary (positive) datatype signature. The impossibility of deriving induction for impredicative encodings in second-order dependent type theory was shown by Geuvers [Geu01], and the higher-order nature of **CC** does not appear change the situation.

Derivable induction for datatypes within the core theory means, for example, that rather than giving meta-theoretic proofs of constructor disjointness and injectivity and for datatype acyclicity (as we did in Section 5.4), we could consider equipping inductive datatypes in the surface language with these reasoning principles *automatically*, justifying them with a proof of the translated property in the internal language. This was the undertaking of Dagand and McBride in [DM12], with their target language being a modest extension of Luo's *extended calculus of constructions* (**ECC**) [Luo94]. Their goal was to remove technically complex properties of datatypes such as these, and positivity checking for datatype declarations, from the trusted computing base of interactive theorem provers, in keeping with the design principles for proof assistants [Geu09] of having a *small kernel* and satisfying the *de Bruijn criterion* (the ability to produce proof objects which are independently checkable).

These very same goals motivate ongoing work on, and within, **CDLE**. Cedille, an implementation of **CDLE**, currently supports declarations for datatypes and definitions of functions over data with pattern-matching and recursion. These surface-language features are justified by a translational semantics to Cedille Core, a minimal (∼1K Haskell LoC) implementation of **CDLE**. The details of this translation, including elaboration of positivity witnesses, as well as proofs of the corresponding static and dynamic soundness, are described by Jenkins et al. in [JMS19].

6.1 An efficient recursion and case-distinction scheme

In this report, we have elected to avoid discussing the elaboration of positivity witnesses for type schemes, and thus the system we presented lacks the crucial feature of a built-in operator for the case-distinction scheme. As previously remarked upon, Matthes [Mat98, Mat02] studied several type theories supporting an efficient recursion scheme for datatypes (with the case-distinction scheme a special instance of this), using positivity witnesses of "functorial strength" to given an embedding (translation) of these theories into System F. This embedding does not preserve what we have called here "dynamic soundness", and the desired efficiency of the recursion scheme is lost in translation to System F.

Recent work by Firsov et al. [FBS18], and Jenkins and Stump in [JS20], established two distinct methods of generically deriving lambda encodings of datatypes enjoying an efficient recursion scheme. In common to both approaches is that positivity of a type scheme is expressed by a much weaker property than that considered by Matthes: rather than there being a lifting, for all types S and T, of *arbitrary* functions $f : S \to T$ to a function of type $F \cdot S \to F \cdot T$, there need only be a lifting defined for an "identity" (or "retyping") function from S to T – meaning, functions $f : S \to T$ that are (propositionally) equal to $\lambda\, x.\, x$. Cedille's type theory is rich enough to express the existence of an identity function between types S and T, which we will notate $S \leqslant T$, and many non-trivial instances of it can be given by a *direct computation* axiom (similar to that described by Allen et al. in [ABC+06], §2.2). This strong computation axiom

allows one, among other things, to assign the type $S \to T$ to $\lambda\, x.\, x$ if there exists some $f : S \to T$ which is *extensionally* equal to the identity function (that is, for every $x : S$, $f\ x$ is equal modulo erasure to x). In Firsov et al. [FBS18], this is utilized to exchange a function that iteratively re-builds data with a retyping function in order to produce a predecessor in *constant-time*. In essence, this guarantees an efficient recursion scheme by proof of the reflection law (Section 4.14) for the initial algebra the datatype represents. Ongoing work by the present author and others [JMS19] seeks to elaborate inductive datatypes with efficient recursion schemes down to their generic derivations in **CDLE**, using the same techniques discussed in this report.

6.2 Subtyping and zero-cost program reuse

Let us reconsider the type $S \leqslant T$ of identity functions in **CDLE**. It is tempting to interpret this type as an *internalized subtyping judgment*, as it states that every term of type S is definitionally equal to a term of type T. This observation goes back (at least) to Mitchell [Mit88] (from which we borrow the terminology *retyping function*) in his study of type containment in polymorphic lambda calculi, and appears also in work by Chen [Che98] on subtyping in **CC**. In [Miq01], Miquel made a similar observation in his presentation of the *implicit calculus of constructions* (**ICC**), which extends **CC** and whose primary novel type constructor, *implicit products*, appears also in **CDLE**. The observation is this: a suitable subtyping judgment might be given as definitional extension of the theory by

$$\Gamma \vdash S \leqslant T =_{\mathbf{df}} \Gamma, x{:}S \vdash x : T$$

from which it is easy to derive $\Gamma \vdash \lambda\, x.\, x : \Pi\, x{:}S.\, T$ (or $\Gamma \vdash \lambda\, x.\, x : S \to T$ if x is not free in T).

We consider the possibilities that this derived notion of subtyping has for *program reuse* in dependently typed programming languages. In such languages, every "ordinary" datatype of functional programming has any number of possible dependently-typed variations, depending upon what particular property the programmer wishes for the type system to track. The standard example is length-indexed lists, referred to as "vectors", and the desire to write a total function `head` taking a non-empty list and returning its first element. While the ability to define such a datatype is certainly useful, it poses a problem: how should one *reuse* other functions over lists for length-indexed lists? Often, there is no better option than defining functions to convert between ordinary and length-indexed lists by iteratively tearing down one structure to build the other, incurring linear-time overhead in each passing between the two representations.

We illustrate with the datatype declarations of `List` and `Vec` in Cedille, given in Figure 9

```
data List (A: ⋆) : ⋆              data Vec (A: ⋆) : Nat → ⋆
= nil  : List                     = vnil  : Vec zero
| cons : A → List → List .        | vcons : ∀ n: Nat. A → Vec n → Vec (succ n) .
```

Figure 9: Lists and length-indexed lists (`Vec`)

The syntax for datatype declarations in Cedille is similar to that for GADTs in Haskell [XCC03]. In the definition of `Vec`, the natural number serves to track the length of the list, with `vnil` having length `zero` and a vector assembled with `vcons` having length `succ n` if the tail of that list has length `n`. To convert from `Vec` to `List`, one could define a recursive function which replaces all occurrences of `vnil` with `nil` and `vcons` with `cons`.

In [DFS18], Diehl et al. established the desirable alternative of *zero-cost* reuse between ordinary datatypes and their dependently-decorated variants. They observed that, in **CDLE**, it is possible for the underlying lambda encodings for the constructors of these datatypes to be definitionally equal (after erasure), meaning that a function replacing e.g. `vnil` with `nil` and `vcons` with `cons` is really doing no work at all – it is extensionally equal to the identity function!

Using the notation introduced earlier, this means it is possible to prove that $\texttt{Vec}\cdot T\ n \leqslant \texttt{List}\cdot T$ and also that $(xs : \texttt{List}\cdot T) \leqslant \texttt{Vec}\cdot T\ (\texttt{length}\ xs)$, with the latter to be interpreted as a *dependent* subtyping judgment where the supertype depends on the given element of the subtype.

The current status of this work still requires programmers to insert explicit type coercions to take advantage of zero-cost reuse. Pointing towards a possible solution for this, ibid. also define several *generic reuse combinators* which refine certain reuse problems to smaller sub-goals, evidence that this technique can be automated with a type-class system similar to Haskell's so that programmers need not insert type coercions explicitly in programs. Another alternative is *ornaments* for datatypes [McB11], a technique in which a type like `Vec` is declared explicitly in terms of `List` by describing how the former refines the latter. Either solution would be in need of justification with respect to the core theory **CDLE** in the form of a translational semantics.

6.3 Dependent pattern matching

Pattern matching in functional programming, originating in the work of Burstall [Bur69] and McBride [McB70], provides a convenient notation for the recursion scheme over data, allowing for conditional branching based on the shape of data and binding of immediate subdata. In languages with dependent types, datatype constructors carry more information, as they may constrain the shape of the indices of the datatype. For example, when performing case-analysis on a term of type $\texttt{Vec}\cdot T\ n$, in the case that this term is `vnil` we have learned that n is equal to `zero`. In [Coq92], Coquand extended the familiar notation of pattern matching to dependent types, presenting a constructor-based unification algorithm enabling constraints on types and terms revealed by pattern-matching to *refine* the current typing context. In particular, if the constraints introduced by a particular constructor are impossible to satisfy, this means that the data *could not have been so constructed*. Dependent pattern matching permits the user to avoid handling such absurd cases entirely. We illustrate with a Cedille-style pseudocode example of a function `vhead` which returns the first element of a non-empty list.

```
vhead : ∀ A: ⋆. ∀ n: Nat. Vec ·A (succ n) → A ;
vhead ·A -.(succ m) (vcons -m x xs) = x         .
```

In the code listing, hyphens prefixing an argument in a function indicate *erased application* (associated with functions whose types are formed by implicit products with $\forall$, the same notation for type quantification); for the purposes of this discussion, hyphens can safely be ignored. The function proceeds by pattern-matching on its vector argument. The case where the vector argument is `vnil` need not be given – it introduces an equational constraint $\texttt{zero} = \texttt{succ}\ n$, which (by constructor disjointness) is impossible. In the case that the vector argument is of the form `vcons` $\texttt{-}m\ x\ xs$, we have a solution $n = \texttt{succ}\ m$, indicated by the forced "dot-pattern" `.(succ m)`.

The theoretical foundations of dependent pattern-matching, given by elaboration to datatype eliminators [GMM06, CA18, Coc17], has been studied extensively in Church-style theories. However, to our best knowledge it has not been studied in the context of Curry-style theories like **CDLE**. One significant difference between the two styles that impacts dependent pattern matching concerns injectivity of constructors for *type arguments*. To illustrate, assume that t_1 has type $S[T_1/X]$ and t_2 has type $S[T_2/X]$, and consider the equation $\mathbf{c}(T_1, t_1) = \mathbf{c}(T_2, t_2)$, where the constructor $\mathbf{c}$ takes a type argument X and a term argument of type S (with X free in S). In a Church-style theory, we can infer from this equation that $T_1 = T_2$ and $t_1 = t_2$. In a Curry-style theory, only $t_1 = t_2$ is a valid conclusion, because the equality holds up to *erasure* of type annotations. For such an example, when using forced "dot-patterns" in clauses of constructor patterns to rewrite occurrences of t_1 to t_2, care must be taken to preserve the information that t_2 is equal to a term that has type $S[T_2/X]$. What we find promising in this line of investigation

is that, in fact, **CDLE** already has a mechanism to do this in the form of *intersection types*: refinement by dependent pattern matching in such a situation could reveal to the user that t_2 has type $S[T_1/X] \wedge S[T_2/X]$.

References

[Abb03] Michael Gordon Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.

[ABC+06] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using NuPRL. *J. Applied Logic*, 4(4):428–469, 2006.

[AC19] Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality, 2019.

[ACP93] Martin Abadi, Luca Cardelli, and Gordon Plotkin. Types for the Scott numerals, 1993. (unpublished notes).

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, volume 48, pages 27–38. ACM, 2013.

[AS11] Ki Yung Ahn and Tim Sheard. A hierarchy of Mendler style recursion combinators: taming inductive datatypes with negative occurrences. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 234–246, New York, NY, USA, 2011. ACM.

[Awo10] Steve Awodey. *Category Theory*. Oxford Logic Guides 52. Oxford University Press, 2nd edition, 2010.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BB08] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[BG82] R. M. Burstall and J. A. Goguen. *Algebras, Theories and Freeness: An Introduction for Computer Scientists*, pages 329–349. Springer Netherlands, Dordrecht, 1982.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language, 1980.

[Bur69] Rod M. Burstall. Proving properties of programs by structural induction. *The Computing Journal*, 12(1):41–48, 1969.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall International series in computer science. Prentice Hall, 1988.

[CA18] Jesper Cockx and Andreas Abel. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.*, 2(ICFP):75:1–75:30, July 2018.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[Che98] Gang Chen. *Subtyping, type conversions and elimination of transitivity.* PhD thesis, Université Paris 7, 1998. PhD thesis.

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 50–63. ACM, 1999.

[Chu41] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[Coc14] Jesper Cockx. Yet another way `--without-K` is incompatible with univalence. URL https://lists.chalmers.se/pipermail/agda/2014/006367.html. On the Agda mailing list, 2014.

[Coc17] Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual Workshop on Logical Frameworks*, volume 92, pages 66–79, Båstad, Sweden, 1992.

[Cro75] John Newsome Crossley. Reminiscences of logicians. In *Algebra and Logic*, pages 1–62. Springer, 1975.

[CT95] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in coq. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 1995.

[Dam84] Luis Damas. *Type assignment in programming languages.* PhD thesis, The University of Edinburgh, 1984.

[DFS18] Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):104:1–104:30, jul 2018.

[DK13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. *SIGPLAN Not.*, 48(9):429–442, sep 2013.

[DK19] Joshua Dunfield and Neel Krishnaswami. Bidirectional typing, 2019.

[DM12] Pierre-Évariste Dagand and Conor McBride. Elaborating inductive definitions. *arXiv preprint arXiv:1210.6390*, 2012.

[FBS18] Denis Firsov, Richard Blair, and Aaron Stump. Efficient Mendler-style lambda-encodings in Cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252, Cham, 2018. Springer International Publishing.

[FS18] Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 215–227, New York, NY, USA, 2018. Association for Computing Machinery.

[Geu01] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *International Conference on Typed Lambda Calculi and Applications*, pages 166–181, Berlin, Heidelberg, 2001. Springer.

[Geu09] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(Part 1):3–25, 2009.

[Geu14] Herman Geuvers. The Church-Scott representation of inductive and coinductive data. (unpublished manuscript), 2014.

[Gim94] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[Gir71] Jean-Yves Girard. Une extension de Ľinterpretation de Gödel a Ľanalyse, et son application a Ľelimination des coupures dans Ľanalyse et la theorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63 – 92. Elsevier, 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540, Berlin, Heidelberg, 2006. Springer.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American mathematical society*, 146:29–60, 1969.

[JMS19] Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive definitions and course-of-values induction in cedille. https://cwjnkins.github.io/assets/JMS20_Elaborating-Inductive-Definitions-and-COV-Induction-Cedille.pdf, 2019. unpublished manuscript.

[JR11] Bart Jacobs and JJMM Rutten. An introduction to (co) algebra and (co) induction. 2011.

[JS18] Christopher Jenkins and Aaron Stump. Spine-local type inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, IFL 2018, pages 37–48, New York, NY, USA, 2018. ACM.

[JS20] Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. Under consideration for J. Mathematically Strucured Computer Science, 2020.

[JVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(1):1–82, 2007.

[Kle45] Stephen Cole Kleene. On the interpretation of intuitionistic number theory. *J. Symb. Log.*, 10(4):109–124, 1945.

[Lam68] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161, 1968.

[LR19] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58, feb 2019.

[Luo94] Zhaohui Luo. *Computation and reasoning - a type theory for computer science*, volume 11 of *International series of monographs on computer science*. Oxford University Press, 1994.

[Mac98] Saunders MacLane. The Yoneda lemma. *Mathematica Japonica*, 47:156–156, 1998.

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Universität München, 1998.

[Mat02] Ralph Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133(1-2):107–129, 2002.

[McB70] Frederick Valentine McBride. *Computer aided manipulation of symbols*. PhD thesis, Queen's University Belfast, 1970.

[McB11] Conor McBride. Ornamental algebras, algebraic ornaments. Manuscript, 2011.

[Men87] Nax P Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, volume 87, pages 30–36, 1987.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Berlin, Heidelberg, 1991. Springer.

[MGM04] Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, page 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2/3):211–249, 1988.

[MJ69] James Hiram Morris Jr. *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic journal of philosophical logic*, 1(1):11–60, 1996.

[NAD12] Ulf Norell, Andreas Abel, and Nils Anders Danielsson. Release notes for Agda 2 version 2.3.2. URL http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main. Version-2-3-2, 2012.

[nLa12] The nLab. Product types. https://ncatlab.org/nlab/show/product+type, 2012. Accessed 2020 Mar 18.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67, 1996.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 41–53, New York, NY, USA, 2001. ACM.

[Par88] Michel Parigot. Programming with proofs: A second order type theory. In H. Ganzinger, editor, *ESOP '88*, pages 145–159, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[Par90] Michel Parigot. On the representation of data in lambda-calculus. In Egon Börger, Hans Kleine Büning, and Michael M. Richter, editors, *CSL '89*, pages 309–321, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[Pfe01] Frank Pfenning. On bidirectional type checking (lecture notes). https://www.cs.cmu.edu/~fp/courses/15312-f04/handouts/15-bidirectional.pdf, October 2001.

[Pie91] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, jan 2000.

[Rey74] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[Sco62] Dana Scott. A system of functional abstraction (1968). lectures delivered at university of california, berkeley. *Cal*, 63, 1962.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

[SF16] Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *J. Funct. Program.*, 26:e3, 2016.

[SHB09] Nikhil Swamy, Michael W. Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 329–340. ACM, 2009.

[Str93] Thomas Streicher. *Investigations into intensional type theory*. Habilitation, Ludwig Maximilian Universität, 1993.

[Stu13] Aaron Stump. *Programming language foundations*. John Wiley & Sons, 2013.

[Stu17] Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017.

[Stu18a] Aaron Stump. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic*, 169(7):637–655, 2018.

[Stu18b] Aaron Stump. Syntax and semantics of Cedille. *CoRR*, abs/1806.04709, 2018.

[SU99] Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, page 102–113, New York, NY, USA, 1999. Association for Computing Machinery.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[T+55] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Tai75] William W Tait. A realizability interpretation of the theory of species. In *Logic Colloquium*, pages 240–251. Springer, 1975.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[UV99] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, sep 1999.

[UV00] Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In *Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19*, pages 69–85. Dept. of Computer Science, Utrecht University, 2000.

[UV02] Tarmo Uustalu and Varmo Vene. Least and greatest fixed points in intuitionistic natural deduction. *Theor. Comput. Sci.*, 272(1-2):315–339, 2002.

[Ven00] Varmo Vene. *Categorical programming with inductive and coinductive types*. PhD thesis, University of Tartu, 2000.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. ACM.

[Wad90] Philip Wadler. Recursive types for free! unpublished, 1990.

[War77] DHD Warren. *Applied Logic – Its Use and Implementation as a Programming Language Tool*. PhD thesis, Univ. of Edinburgh, 1977. PhD. Thesis.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 355–377, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Wel99] Joe B Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure and Applied Logic*, 98(1-3):111–156, 1999.

[XCC03] 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, page 224–235, New York, NY, USA, 2003. Association for Computing Machinery.