Focusing on Liquid Refinement Typing

DIMITRIOS J. ECONOMOU, Queen's University, Canada
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom
JANA DUNFIELD, Queen's University, Canada

We present a foundation systematizing, in a way that works for any evaluation order, the variety of mechanisms for SMT constraint generation found in index refinement and liquid type systems. Using call-by-push-value, we design a polarized subtyping relation allowing us to prove that our logically focused typing algorithm is sound, complete, and decidable, even in cases seemingly likely to produce constraints with existential variables. We prove type soundness with respect to an elementary domain-theoretic denotational semantics. Soundness implies, relatively simply, our system's totality and logical consistency.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; **Denotational semantics**; **Program reasoning**; *Proof theory*; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: refinement types, bidirectional typechecking, polarity, call-by-push-value

1 INTRODUCTION

True, "well-typed programs cannot 'go wrong' " [Milner 1978], *but only relative to a given semantics* (if the type system is proven sound with respect to it). Unfortunately, well-typed programs *do* go wrong, in many ways that matter, but about which the given semantics cannot speak: division by zero, out-of-bounds array accesses, information leaks. If we want a type system to prevent more things from going wrong, then we must refine its semantics. However, there is often not enough information in the types themselves with which to refine the semantics. So, we must refine our types with more information, information that tends to be related to programs. Great care is needed, though, because incorporating too much information from programs (like non-terminating programs themselves) can spoil good properties of the type system, such as type soundness or the decidability of type checking and inference.

Consider the inductive type List A of lists with entries of type A. Such a list is either nil ([]) or a term x of type A together with a tail list xs (that is, $x :: xs$). In a typed functional language like Haskell or OCaml, the programmer can declare such an inductive type by specifying its constructors:

$$\begin{array}{l}\texttt{data}\ \mathsf{List}\ A\ \texttt{where}\\ \quad [\,] : \mathsf{List}\ A\\ \quad (::) : A \to \mathsf{List}\ A \to \mathsf{List}\ A\end{array}$$

Suppose we define, by pattern matching, the function get, that takes a list xs and a natural number n, and returns the nth element of xs (where the first element is numbered 0):

$$\begin{array}{l}\mathsf{get}\ \texttt{[]}\ y = \mathsf{error}\ \text{“Out of bounds”}\\ \mathsf{get}\ (x :: xs)\ \mathsf{zero} = x\\ \mathsf{get}\ (x :: xs)\ (\mathsf{succ}\ n) = \mathsf{get}\ xs\ n\end{array}$$

A conventional type system has no issue checking get against, say, the type List $A \to$ Nat $\to A$ (for any type A), but get is unsafe because it throws an out-of-bounds error when the input number is greater than or equal to the length of the input list. If we want to make it impossible for get to throw

Authors' addresses: Dimitrios J. Economou, Queen's University, Goodwin Hall 557, Kingston, ON, K7L 3N6, Canada, d.economou@queensu.ca; Neel Krishnaswami, University of Cambridge, Computer Laboratory, William Gates Building, Cambridge, CB3 0FD, United Kingdom, nk480@cl.cam.ac.uk; Jana Dunfield, Queen's University, Goodwin Hall 557, Kingston, ON, K7L 3N6, Canada, jd169@queensu.ca.

2022. XXXX-XXXX/2022/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

such an error, then get must check against a type where the input number is restricted to natural numbers strictly less than the length of the input list. Ideally, the programmer would simply refine the type of get, while leaving the program alone (except, perhaps, for removing the first clause). This is the chief aim of *refinement types* [Freeman and Pfenning 1991]: to increase the expressive power of a given (unrefined) type system, while keeping the latter's good properties, like type soundness and decidability of typing, and without placing much or any burden on programmers to refactor their code or manually provide tedious proofs.

To refine get's type so as to rule out, statically, run-time out-of-bounds errors, we need to compare numbers against list lengths. Thus, we refine the type of lists by their length: $\{v : \mathsf{List}\ A \mid \mathsf{len}\ v = n\}$, the type of lists v of length n. This is worrying, though, because the measurement, $\mathsf{len}\ v = n$, seems to involve the recursive program len. The structurally recursive

$$\begin{aligned}&\mathsf{len}\ [] = 0\\ &\mathsf{len}\ (x :: xs) = 1 + \mathsf{len}\ xs\end{aligned}$$

happens to terminate, but it's well-known that there is no general algorithm for deciding whether an arbitrary recursive program terminates [Turing 1936]. As such, we would prefer not to use ordinary recursive programs at all in our type refinements. Indeed, doing so would violate a *phase distinction* [Moggi 1989a; Harper et al. 1990] between *static* specification and *dynamic* program, which has proven almost indispensable for *decidable* typing. From our perspective, it seems liquid types [Rondon et al. 2008] achieve highly expressive, yet sound and decidable *recursive refinements* [Kawaguchi et al. 2009] of inductive types by a kind of phase distinction: namely, by restricting the recursive predicates of specifications to terminating *measures* that soundly characterize, in a theory decidable by off-the-shelf tools like SMT solvers, the static structure of inductive types.

Distinguishing between index terms, which can always safely appear in types, and program terms, which cannot, we want to check get against a more informative type like

$$\forall l : \mathbb{N}.\ \{v : \mathsf{List}\ A \mid \mathsf{len}\ v = l\} \to \{v : \mathsf{Nat} \mid v < l\} \to A$$

which quantifies over *indexes* l of *sort* $\mathbb{N}$ (natural numbers). Technically, this type isn't quite right, however, because Nat is a type and $\mathbb{N}$ is a sort, so writing "$v < l$" confounds the phase distinction between programs and indexes. Instead, the type should look more like

$$\forall l : \mathbb{N}.\ \{v : \mathsf{List}\ A \mid \mathsf{len}\ v = l\} \to \{v : \mathsf{Nat} \mid \mathsf{index}\ v < l\} \to A$$

where

$$\begin{aligned}&\mathsf{index}\ \mathsf{zero} = 0\\ &\mathsf{index}\ (\mathsf{succ}\ n) = 1 + \mathsf{index}\ n\end{aligned}$$

computes the index term of sort $\mathbb{N}$, associated with a program term of type Nat, by a structural recursion homologous to that of len. This isn't mere pedantry, because it is precisely this homologous recursion into a decidable logic, we think, that facilitates the decidability of typing. The third clause of get has a nonempty list as input, so its index (length) must be $1 + m$ for some m; the type checker assumes $\mathsf{index}\ (\mathsf{succ}\ n) < 1 + m$; by the aforementioned homology, these constraints are again satisfied at the recursive call ($n < m$), until the second clause returns ($0 < 1 + m'$). The first clause is now impossible, because no natural number is negative; we can therefore safely remove this clause, or replace error with `unreachable`:

$$\begin{aligned}&\mathsf{get} : \forall l : \mathbb{N}.\ \{v : \mathsf{List}\ A \mid \mathsf{len}\ v = l\} \to \{v : \mathsf{Nat} \mid \mathsf{index}\ v < l\} \to A\\ &\mathsf{get}\ \texttt{[]}\ y = \texttt{unreachable} \qquad\qquad \text{-- } l = 0\\ &\mathsf{get}\ (x :: xs)\ \mathsf{zero} = x\\ &\mathsf{get}\ (x :: xs)\ (\mathsf{succ}\ n) = \mathsf{get}\ xs\ n\end{aligned}$$

While this kind of reasoning about get's type refinement may seem straightforward, how do we generalize it to recursion over any algebraic datatype (ADT)? What are its key logical and semantic ingredients? And, most importantly for this paper, how do we use these ingredients to concoct a type system with decidable typing, good (localized) error messages and so on, while also keeping its metatheory relatively stable or maintainable under various extensions or different evaluation strategies?

Type systems that can do this kind of reasoning automatically, especially in a way independent of evaluation strategy, are hard to design correctly. Indeed, the techniques used in the original liquid type system(s) [Rondon et al. 2008; Kawaguchi et al. 2009] had to be modified for Haskell, essentially because of the latter's *call-by-name* (rather than *call-by-value*) evaluation order [Vazou et al. 2014]. The basic issue was that binders can bind in (static) refinement predicates, which is fine when binders only bind values (as in call-by-value), but not when they bind (dynamic) computations that may or may not terminate (as in call-by-name). Liquid Haskell regained type soundness by imposing restrictions that involve approximating binders as terminating or non-terminating, and using the refinement logic to verify termination. But the theoretical waters are nonetheless murky.

Levy [2004] designed and built, from the ground up, the paradigm and calculus *call-by-push-value* (CBPV) to put both call-by-name and call-by-value on equal footing, even in the storm of computational effects such as nontermination. CBPV subsumes both call-by-name and call-by-value functional languages, because it allows us to encode both evaluation strategies via modification of type discipline. Further, CBPV can be viewed logically in terms of *focusing* [Espírito Santo 2017]—a technique in proof theory that dramatically reduces the search space for proofs of logical formulas. From a Curry–Howard perspective, therefore, CBPV is a good foundation for a refinement typing algorithm.

Further still, bidirectional typing, which systematizes the difference between input and output, seems to fit nicely with focused systems [Dunfield and Krishnaswami 2021]. Bidirectional typing has its own practical virtues: it is easy to implement (if designed properly); it scales well (to refinement types, higher-rank polymorphism [Dunfield and Krishnaswami 2019], subtyping, effects—and so does CBPV); it leads to localized error messages; and it clarifies where type annotations are needed, typically in reasonable positions (such as at the top level) that tend to be helpful as machine-checked documentation. A focused and bidirectional approach therefore appears more than suitable, both theoretically and practically, for systematically designing and implementing such an expressive language of type refinement that can handle any evaluation strategy and effect (at least in principle).

Contributions. Our two key contributions are a logical *and* an algorithmic account of recursive, index-based refinement of inductive types. For the logical account, we design a declarative type system in a *bidirectional* and *focused* style, resulting in a system that's theoretically convenient. For the algorithmic account, we design a fully *algorithmic*, *bidirectional* type system, and prove it is decidable, as well as sound and complete with respect to the declarative system. We demonstrate that bidirectional typing and logical focusing work very well together to manage the complex flow of information pertaining to indexes of recursive data. We contribute:

- A polarized type system, including (polarized) subtyping, universals, existentials, and index refinements with ordinary SMT constraints, as well as recursive predicates on inductive data (which can be extended to model types such as lists of increasing integer elements). We prove the system satisfies substitution, which requires proving, among other things, that subsumption is admissible and subtyping is transitive.
- A polarized subtyping algorithm, together with proofs that it is sound, complete and decidable.
- A focused typing algorithm, together with proofs that it is sound, complete and decidable.

- A type soundness proof with respect to an elementary domain-theoretic denotational semantics, which implies, relatively easily, both the refinement system's logical consistency and enforcement of program termination—even if the programs are *non-structurally* recursive.

All proofs are in the appendix.

2 OVERVIEW

Before going into the details of our type system's inference rules and algorithm, we give an overview of the central logical and semantic issues that inform their design.

Refinement typing, evaluation strategy, and computational effects. The interaction between refinement typing (and other fancy typing), evaluation strategy, and computational effects is a source of peril. The combination of parametric polymorphism with effects is often unsound [Harper and Lillibridge 1991]; the value restriction in Standard ML recovers soundness in the presence of mutable references by restricting polymorphic generalization to syntactic values. [Wright 1995]. The issue was also not peculiar to polymorphism: Davies and Pfenning [2000] discovered that a similar value restriction recovers type soundness for intersection refinement types and effects in call-by-value languages. For union types, Dunfield and Pfenning [2003] obtained soundness by an evaluation context restriction on union elimination.

For similar reasons, Liquid Haskell was also found unsound in practice, and had to be patched; we adapt an example [Vazou et al. 2014] demonstrating the discovered unsoundness:

```
diverge :: Nat -> {v:Int | false}
diverge x = diverge x

safediv :: n:Nat -> {d:Nat | 0 < d} -> {v:Nat | v <= n}
safediv n d = if 0 < d then n / d else error "unreachable"

unsafe :: Nat -> Int
unsafe x = let {notused = diverge 1; y = 0} in safediv x y
```

In typechecking `unsafe`, we need to check that the type of `y` (a singleton type of one value: 0) is a subtype of `safediv`'s second argument type (under the context of the let-binding). Due to the refinement of the let-bound `notused`, this subtyping generates a *verification condition* (or *constraint*) of the form "if false is true, then…". This constraint holds vacuously, implying that `unsafe` is safe. But `unsafe` really is unsafe because Haskell has lazy evaluation: since `notused` is never used, `diverge` is never called, and hence `safediv` divides by zero (and crashes). Vazou et al. [2014] recover type soundness and decidable typing by restricting let-binding and subtyping, using an operational semantics to approximate whether or not expressions diverge, and whether or not terminating terms terminate to a *finite* value.

The value and evaluation context restrictions were reactions to the failure of simple typing rules to deal with specific practical problems that arose from interactions between effects and evaluation strategy. But Zeilberger [2009] explained the value and evaluation context restrictions in terms of a broader, more logical view of refinement typing. Not only did this view improve our understanding of these restrictions, it provided theoretical tools and material for designing highly expressive yet practical, safe, and decidable type systems for functional languages with effects. At the heart of Zeilberger's approach is the proof-theoretic technique of *focusing*, which we discuss near the end of this overview. An important question that we address is whether polarization and focusing can also help us understand Liquid Haskell's restrictions on let-binding and subtyping.

Refining CBPV. Call-by-push-value [Levy 2004] was created to have good semantic properties for both call-by-value (CBV) and call-by-name (CBN), even when extended to richer types and computational effects. CBPV subsumes both CBV and CBN by polarizing the usual terms and types of the λ-calculus into a finer structure that can be used to encode both evaluation strategies: *value* (or *positive*) types (classifying terms which "are", that is, values v), *computation* (or *negative*) types (classifying terms which "do", that is, expressions e), and polarity shifts $\uparrow$ and $\downarrow$ between them. An upshift $\uparrow P$ *lifts* a (positive) value type P up to a (negative) computation type of expressions that compute values (of type P). A downshift $\downarrow N$ pushes a (negative) computation type N down into a (positive) value type of *thunked* or *suspended* computations (of type N). We can embed the usual λ-calculus function type $A \to B$, for example, into CBPV (whose function types have the form $P \to N$ for positive P and negative N) so that it behaves like CBV, via the translation ι_{CBV} with $\iota_{\text{CBV}}(A \to B) = \downarrow(\iota_{\text{CBV}}(A) \to \uparrow\iota_{\text{CBV}}(B))$; or so that it behaves like CBN, via the translation ι_{CBN} with $\iota_{\text{CBN}}(A \to B) = (\downarrow\iota_{\text{CBN}}(A)) \to \iota_{\text{CBN}}(B)$.

Evaluation order is made explicit by CBPV type discipline. Therefore, adding a refinement layer on top of CBPV requires directly and systematically dealing with the interaction between type refinement and evaluation order. If we add this layer to CBPV correctly from the very beginning, then we can be (at least more) confident that our type refinement system will be well-behaved when extended to talk about various computational effects.

Type soundness, totality, and logical consistency. The unrefined system that underlies our type refinement system has the computational effect of nontermination. To model nontermination, we give the unrefined system an elementary domain-theoretic denotational semantics. Because the unrefined system is based on CBPV, proving type soundness is relatively straightforward.

The denotational semantics of our refined system is defined in terms of that of its *erasure* (of indexes), that is, its underlying, unrefined system. A refined type denotes a logical subset of what its erasure denotes. An *unrefined* return type $\uparrow P$ denotes either what P denotes, or a diverging expression. A *refined* return type $\uparrow P$ denotes *only* what P denotes. Therefore, our *refined* type soundness result implies that our refined system enforces termination; it also implies logical consistency, because a logically inconsistent refinement type denotes the empty set. We also prove that syntactic substitution is semantically sound.

Inductive types and measures. A novelty of liquid typing is the use of *measures*: functions, defined on inductive types, which may be structurally recursive, but are guaranteed to terminate, and can be safely used to refine the inductive types over which they are defined.

For example, consider the type BinTree A of binary trees with terms of type A at leaves:

$$\begin{array}{l} \texttt{data}\ \mathsf{BinTree}\ A\ \texttt{where} \\ \quad \mathsf{leaf} : A \to \mathsf{BinTree}\ A \\ \quad \mathsf{node} : \mathsf{BinTree}\ A \to \mathsf{BinTree}\ A \to \mathsf{BinTree}\ A \end{array}$$

Suppose we want to refine BinTree A by the height of trees. Perhaps the most direct way to specify this is to measure it using a function hgt defined by structural recursion:

$$\begin{array}{l} \mathsf{hgt} : \mathsf{BinTree}\ A \to \mathbb{N} \\ \mathsf{hgt}\ \mathsf{leaf} = 0 \\ \mathsf{hgt}\ (\mathsf{node}\ t\ u) = 1 + \mathsf{max}(\mathsf{hgt}(t), \mathsf{hgt}(u)) \end{array}$$

As another example, consider an inductive type Expr of expressions in a CBV lambda calculus:

$$\begin{array}{l}\texttt{data}\ \mathsf{Expr}\ \texttt{where}\\ \quad \mathsf{var} : \mathsf{Nat} \to \mathsf{Expr}\\ \quad \mathsf{lam} : \mathsf{Nat} \to \mathsf{Expr} \to \mathsf{Expr}\\ \quad \mathsf{app} : \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}\end{array}$$

Measures need not involve recursion. For example, if we want to refine the type Expr to expressions Expr that are values (in the sense of CBV, not CBPV), then we can use isval:

$$\begin{array}{l}\mathsf{isval} : \mathsf{Expr} \to \mathbb{B}\\ \mathsf{isval}\ (\mathsf{var}\ z) = \mathsf{T}\\ \mathsf{isval}\ (\mathsf{lam}\ z\ \mathit{expr}) = \mathsf{T}\\ \mathsf{isval}\ (\mathsf{app}\ \mathit{expr}\ \mathit{expr}') = \mathsf{F}\end{array}$$

Because isval isn't recursive and returns indexes, it's safe to use it to refine Expr to expressions that are CBV values: $\{v : \mathsf{Expr} \mid \mathsf{isval}\ v = \mathsf{T}\}$. But, as with len (Sec. 1), we may again be worried about using the seemingly dynamic, recursively defined hgt in a static type refinement. Again, we need not worry because hgt, like len, is a terminating function into a decidable logic [Barrett et al. 2009]. We can use it to specify that, say, a height function defined by pattern matching on trees of type $\{v : \mathsf{BinTree}\ A \mid \mathsf{hgt}\ v = n\}$ actually returns (a program value representing) n for any tree of height n. Given the phase distinction between indexes like n and programs, how exactly do we specify such a function type?

Unrolling and singletons. Let's consider a slightly simpler function, length, that takes a list and returns its length:

$$\begin{array}{l}\mathsf{length}\ [] = \mathsf{zero}\\ \mathsf{length}\ (x :: xs) = \mathsf{succ}\ (\mathsf{length}\ xs)\end{array}$$

What should be the type specifying that length actually returns a list's length? The proposal $\forall n : \mathbb{N}.\ \mathsf{List}\ A\ n \to {\uparrow}\mathsf{Nat}$ does not work because Nat has no information about the length n. Something like $\forall n : \mathbb{N}.\ \mathsf{List}\ A\ n \to {\uparrow}(n : \mathsf{Nat})$, read literally as returning the index n, would violate the phase distinction between programs and indexes. Instead, we use a *singleton* type in the sense of Xi [1998] to specify the kind of type we want here: a singleton type contains just those program terms (of the type's erasure), that correspond to exactly one semantic index. For example, given $n : \mathbb{N}$, we define the singleton type Nat n by $\{v : \mathsf{Nat} \mid \mathsf{index}\ v = n\}$ where

$$\begin{array}{l}\mathsf{index} : \mathsf{Nat} \to \mathbb{N}\\ \mathsf{index}\ \mathsf{zero} = 0\\ \mathsf{index}\ (\mathsf{succ}\ x) = 1 + \mathsf{index}(x)\end{array}$$

specifies the indexes (of sort $\mathbb{N}$) corresponding to program values of type Nat.

How do we check length against $\forall n : \mathbb{N}.\ \mathsf{List}\ A\ n \to {\uparrow}(\mathsf{Nat}(n))$? In the first clause, the input $[]$ has type $\mathsf{List}\ A\ n$ for some n, but we need to know $n = 0$ (and that the index of zero is 0). Similarly, we need to know $x :: xs$ has length $n = 1 + n'$ where $n' : \mathbb{N}$ is the length of x. To generate these constraints, we use an *unrolling* judgment (Sec. 4.6) that unrolls a refined inductive type. Unrolling puts the type's refinement constraints, expressed by *asserting* and *existential* types, in the structurally appropriate positions. An asserting type is written $P \wedge \phi$ (read "P with ϕ"), where P is a (positive) type and ϕ is an index proposition. When a term has type $P \wedge \phi$, the term has type P and also ϕ holds. (Dual to asserting types, we have the *guarded* type $\phi \supset N$, which is equivalent to N if ϕ holds, but is otherwise useless.) We use asserting types to express that index equalities like $n = 0$ hold for terms of inductive type. We use existentials to quantify over indexes that characterize the refinements of recursive subparts of inductive types, like n'. For example, modulo a small difference (see Sec. 4.6), $\mathsf{List}\ A\ n$ unrolls to

$$\big(1 \wedge (n = 0)\big) + \big(A \times \exists n' : \mathbb{N}.\ \{v : \mathsf{List}\ A \mid \mathsf{len}\ v = n'\} \wedge (n = 1 + n')\big)$$

That is, to construct an A-list of length n, the programmer (or library designer) can either left-inject a unit value, *provided the constraint* $n = 0$ *holds*, or right-inject a pair of one A value and a tail list, *provided that* n'*, the length of the tail list, is* $n - 1$ (the equations $n = 1 + n'$ and $n - 1 = n'$ are equivalent). These index constraints are not a syntactic part of the construction of the list itself, but they must hold in order for the construction to proceed. By the phase distinction, the programmer need not provide any explicit proof terms showing that these index constraints hold. Dual[1] to constructing refined inductive values, pattern matching on refined inductive values allows us to use the index refinements locally in the bodies of the various clauses for different patterns.

The shape of the refinement types generated by our unrolling judgment (such as the one above) is a judgmental and refined-ADT version of the fact that generalized ADTs (GADTs) can be desugared into types with equalities and existentials that express constraints of the return types for constructors [Cheney and Hinze 2003; Xi et al. 2003]. It would be tedious and inefficient for the programmer to work directly with terms of types produced by our unrolling judgment, but we can implement (in our system) singleton refinements of base types and common functions on them, such as addition, subtraction, multiplication, division, and the modulo operation on integers, and build these into the surface language used by the programmer, similarly to the implementation of Dependent ML [Xi 1998].

Inference and subtyping. For a typed functional language to be practical, it must support some degree of inference, especially for function application: to pass a value to a function, its type must be compatible with the function's argument type, but it would be burdensome to make programmers be too explicit about this compatibility. In our setting, for example, if $x : \mathsf{Nat}(3)$ and $f : \downarrow(\forall a : \mathbb{N}.\ \mathsf{Nat}(a) \to \uparrow P)$, then we would prefer to write $f\ x$ rather than $f\ [3]\ x$, which would quickly make our programs incredibly—and unnecessarily—verbose. Omitting index and type annotations, however, has significant implications. In particular, we need a mechanism to instantiate indexes somewhere in our typing rules: for example, if $g : \downarrow(\downarrow(\mathsf{Nat}(4 + b) \to \uparrow P) \to N)$ and $h : \downarrow((\exists a : \mathbb{N}.\ \mathsf{Nat}(a)) \to \uparrow P)$, then to apply g to h, we need to know $\mathsf{Nat}(4 + b)$ is compatible with $\exists a : \mathbb{N}.\ \mathsf{Nat}(a)$, which requires instantiating the bound a to a term logically equal to $4 + b$. Our system does this kind of instantiation via subtyping.

We polarize subtyping into two, mutually recursive, positive and negative relations $\Theta \vdash P \leq^+ Q$ and $\Theta \vdash N \leq^- M$ (where Θ is a *logical context* including index propositions). The algorithmic versions of these only introduce existential variables in positive supertypes and negative subtypes, guaranteeing that they can always be solved by indexes that do not have any existential variables. We delay checking constraints until the end of certain, logically designed phases (the *focusing* ones, as we will see), when all of their existential variables have been solved.

Well-formedness. To guarantee that our algorithm can always instantiate quantifiers, we restrict quantification to indexes appearing in certain positions within types: namely, those that are uniquely determined (semantically speaking) by values of the type, both according to a measure and before crossing a polarity shift. For example, in $\{v : \mathsf{List}\ A \mid \mathsf{len}\ v = b\}$, the index b is uniquely determined by values of that type: the list $[x, y]$ uniquely determines b to be 2 (by the length measure). We make this restriction in the type well-formedness judgment, which has an output Ξ that tracks value-determined indexes; well-formed types can only quantify over indexes in Ξ. The variables in Ξ

[1]Dual in the sense of verifications and uses.

also pay attention to type structure: for example, a value of a product type is a pair of values, where the first value determines all Ξ_1 (for the first type component) and the second value determines all Ξ_2 (second type component), so the Ξ of the product type is their union $\Xi_1 \cup \Xi_2$. We also take the union for function types $P \to N$, because index information flows through argument types to the return type, marked by a polarity shift.

By emptying Ξ at shift types, we prevent lone existential variables from being introduced at a distance, across polarity shifts. In practice, this restriction on quantification is not onerous, because most functional types that programmers use are, in essence, of the form

$$\forall \cdots . P_1 \to \cdots \to P_n \to \uparrow \exists \cdots . Q$$

where the "$\forall$" quantifies over indexes of argument types P_k that are uniquely determined by argument values, and the "$\exists$" quantifies over indexes of the return type that are determined by (or at least depend on) fully applying the function. (This polarized structure of functions is common in mathematics. For example, in real analysis, the ϵ-δ definition of continuity implicitly has this polarized structure: an input function $f : X \to \mathbb{R}$ on real numbers is *continuous* if, *for all* real inputs $x_0 \in X$ and $\epsilon > 0$, *there exists* a real output $\delta > 0$—that may depend on the nature of inputs f, x_0, and ϵ—such that $|x - x_0| < \delta$ implies $|f(x) - f(x_0)| < \epsilon$.) Our Ξ restriction, together with focusing, is very helpful metatheoretically, because it means that our typing algorithm only introduces—and is guaranteed to solve—existential variables for indexes within certain logical phases.

Focusing and bidirectional typing. In proof theory, the technique of *focusing* [Andreoli 1992] exploits invertibility properties of logical formulas (types), as they appear in inference rules (typing rules), in order to characterize their proofs (programs), both soundly and completely, by way of a focused system that only forecloses redundant proofs. The foreclosure of redundant proofs makes proof search tractable. Brock-Nannestad et al. [2015] and Espírito Santo [2017] study the relation between CBPV and focusing: each work provides a focused calculus that is essentially the same as CBPV, "up to the question of η-expansion" [Brock-Nannestad et al. 2015]. Our system is also a focused variant of CBPV.

An inference rule is *invertible* if its conclusion implies its premises. In a sequent calculus, positive formulas have invertible left rules and negative formulas have invertible right rules. A *weakly* focused sequent calculus eagerly applies non-invertible rules as far as possible (in either *left-* or *right-focusing* phases); a *strongly* focused sequent calculus does too, but also eagerly applies invertible rules as far as possible (in either *left-* or *right-inversion* phases). There are also *stable* phases (or moments) in which a decision has to be made between focusing on the left, or on the right [Espírito Santo 2017]. The decision can be made explicitly via proof terms (specifically, cuts): in our system, a principal task of let-binding is to focus on the left (to process the list of arguments in function application); and a principal task of pattern matching is to focus on the right (to decompose the value being matched against a pattern).

From a Curry–Howard view, let-binding and pattern matching are different kinds of cuts. The cut formula A—basically, the type being matched or let-bound—must be *synthesized* as an output (judgmentally, $\cdots \Rightarrow A$) from *heads h* (variables and annotated values) or *bound expressions g* (function application and annotated returner expressions); and ultimately, the outcomes of these cuts can also be (in our system, are) synthesized. But all other program terms are *checked* against input types A: judgmentally, $\cdots \Leftarrow A \cdots$ or $\cdots [A] \vdash \cdots$. In this sense, both our declarative and algorithmic type systems are *bidirectional* [Dunfield and Krishnaswami 2021].

Our declarative system includes two focusing phases, one for positive types on the right of the turnstile ($\vdash$), and the other for negative types on the left of the turnstile. Our algorithmic system closely mirrors the declarative one, but does not conjure index instantiations or witnesses, and instead systematically introduces and solves existential variables, which we keep in algorithmic contexts Δ. Our algorithmic right-focusing judgment has the form $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \Delta'$, where χ is an output list of typing constraints and Δ' is an output context that includes index instantiations. Similarly, the left-focusing phase is $\Theta; \Delta; \Gamma; [N] \vdash s \gg \uparrow P \;/\; \chi \dashv \Delta'$; it focuses on decomposing N (the type of the function being applied), introducing its existential variables for the arguments in the list s (sometimes called a *spine* [Cervesato and Pfenning 2003]), and outputting its guards to verify at the end of decomposition (an upshift). These existential variables flow to the right-focusing phase (value typechecking) and are solved there, possibly via subtyping. The constraints χ are only checked right at the end of focusing phases, when all their existential variables are solved. For example, consider our rule for (fully) applying a function h to a list of arguments s:

$$\frac{\Theta; \Gamma \rhd h \Rightarrow \downarrow N \qquad \Theta; \cdot; \Gamma; [N] \vdash s \gg \uparrow P \;/\; \chi \dashv \cdot \qquad \Theta; \Gamma \lhd \chi}{\Theta; \Gamma \rhd h(s) \Rightarrow \uparrow P}\;\text{AlgSpineApp}$$

After synthesizing a thunk type $\downarrow N$ for the function h we are applying, we process the entire list of arguments s, until N's return type $\uparrow P$. All value-determined indexes Ξ_N of N are guaranteed to be solved by the time we reach an upshift, and these solutions are eagerly applied to constraints χ, so that χ does not have existential variables and hence does have tractable verification ($\Theta; \Gamma \lhd \chi$).

Our system requires intermediate computations like $h(s)$ to be explicitly named and sequenced by let-binding (this is *A-normal* [Flanagan et al. 1993] or *let-normal form*). Combined with focusing, this allows us to use (within the value typechecking phase) subtyping only in the typing rule for variables. This makes our subsumption rule syntax-directed, simplifying and increasing the efficiency of our algorithm. We nonetheless prove a general subsumption lemma, which is needed to prove that substitution respects typing, a key syntactic or operational correctness property.

Focusing also gives us pattern matching for free [Krishnaswami 2009]: from a Curry–Howard view, the left-inversion phase is pattern matching. The (algorithmic[2]) left-inversion phase in our system is written $\Theta; \Gamma; [P] \rhd \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$: it decomposes the positive P (representing the pattern being matched) to the left of the turnstile (written $\rhd$ to distinguish the algorithmic judgment from the corresponding declarative judgment, which instead uses $\vdash$). Our system is "half" strongly focused: we eagerly apply right-invertible but not left-invertible rules. This makes pattern matching in our system resemble the original presentation of pattern matching in CBPV. From a Curry–Howard view, increasing the strength of focusing would permit nested pattern matching.

A pattern type can have index equality constraints, such as for refined ADT constructors (for example, that the length of an empty list is zero) as output by unrolling. By using these equality constraints, we get a coverage-checking algorithm. For example, consider checking get (introduced in Sec. 1) against the type

$$\forall l, k : \mathbb{N}.\ \{v : \mathsf{List}\ A \mid \mathsf{len}\ v = l\} \to (\{v : \mathsf{Nat} \mid \mathsf{index}\ v = k\} \wedge (k < l)) \to \uparrow A$$

At the clause

$$\texttt{get []}\ y = \texttt{unreachable}$$

we obtain a logically inconsistent context ($l : \mathbb{N}, k : \mathbb{N}, l = 0, k < l$), which entails that `unreachable` checks against any type. Proof-theoretically, this use of equality resembles the elimination rule for Girard–Schroeder-Heister equality [Girard 1992; Schroeder-Heister 1994].

Bidirectional typing controls the flow of type information. Focusing in our system directs the flow of index information. The management of the flow of type refinement information, via the stratification of both focusing and bidirectionality, makes our algorithmic metatheory highly manageable (at the cost, perhaps, of apparent complexity).

[2]We give the algorithmic judgment to note existential variables Δ do not flow through it, or any of the non-focusing phases.

3 EXAMPLE: VERIFYING MERGESORT

We show how to verify a non-structurally recursive mergesort function in our system. We only consider sorting lists of natural numbers $\mathsf{Nat} = (\exists n : \mathbb{N}.\ \mathsf{Nat}(n))$. For clarity, and continuity with the introduction (Sec. 1) and the overview (Sec. 2), we use some syntactic sugar for our system (presented in Sec. 4).

Define the measure

$$\begin{aligned}&\mathsf{ixbool} : \mathsf{Bool} \to \mathbb{B}\\ &\mathsf{ixbool}\ \mathtt{true} = \mathsf{T}\\ &\mathsf{ixbool}\ \mathtt{false} = \mathsf{F}\end{aligned}$$

Given $b : \mathbb{B}$, the singleton type of boolean b is $\mathsf{Bool}(b) = \{\nu : \mathsf{Bool} \mid \mathsf{ixbool}\ \nu = b\}$. Our unrolling judgment (Sec. 4.6) outputs the following type, a refinement of the boolean type encoded as $1 + 1$:

$$\big(1 \land (b = \mathsf{T})\big) + \big(1 \land (b = \mathsf{F})\big)$$

We encode $\mathtt{true}$ by rolling the left injection $\mathtt{inj}_1\ \mathtt{()}$ into $\mathsf{Bool}(\mathsf{T})$, and $\mathtt{false}$ by rolling the right injection $\mathtt{inj}_2\ \mathtt{()}$ into $\mathsf{Bool}(\mathsf{F})$.

Assume we have the following:

$$\begin{aligned}\mathsf{add}\ &:\ \forall m, n : \mathbb{N}.\ \mathsf{Nat}(m) \to \mathsf{Nat}(n) \to \uparrow\mathsf{Nat}(m + n)\\ \mathsf{sub}\ &:\ \forall m, n : \mathbb{N}.\ (n \le m) \supset \mathsf{Nat}(m) \to \mathsf{Nat}(n) \to \uparrow\mathsf{Nat}(m - n)\\ \mathsf{lt}\ &:\ \forall m, n : \mathbb{N}.\ \mathsf{Nat}(m) \to \mathsf{Nat}(n) \to \uparrow\exists b : \mathbb{B}.\ \mathsf{Bool}(b) \land (b = (m < n))\\ \mathsf{length}\ &:\ \forall n : \mathbb{N}.\ \mathsf{List}(\mathsf{Nat})(n) \to \uparrow\mathsf{Nat}(n)\\ []\ &:\ \mathsf{List}(\mathsf{Nat})(0)\\ (::)\ &:\ \forall n : \mathbb{N}.\ \mathsf{Nat} \to \mathsf{List}(\mathsf{Nat})(n) \to \uparrow\mathsf{List}(\mathsf{Nat})(1 + n)\end{aligned}$$

We define the function merge for merging two lists while preserving order. It takes two lists, and one of them always gets smaller at recursive calls. In the *refined* system presented in this paper, we need to reify the sum of the length indexes, which always gets smaller at recursive calls—implying *termination*—into its own type, so we use an auxiliary function:

$$\begin{aligned}&\mathsf{auxmerge} : \forall n, n_1, n_2 : \mathbb{N}.\ \mathsf{List}(\mathsf{Nat})(n_1) \to \mathsf{List}(\mathsf{Nat})(n_2) \to \mathsf{Nat}(n) \land (n = n_1 + n_2)\\ &\qquad\qquad \to \uparrow\mathsf{List}(\mathsf{Nat})(n_1 + n_2)\\ &\mathsf{auxmerge}\ []\ xs_2\ y = \mathtt{return}\ xs_2\\ &\mathsf{auxmerge}\ xs_1\ []\ y = \mathtt{return}\ xs_1\\ &\mathsf{auxmerge}\ (x_1 :: xs_1)\ (x_2 :: xs_2)\ (\mathsf{succ}\ y) =\\ &\quad \mathsf{if}\ \mathsf{lt}(x_1, x_2)\ \mathtt{then}\\ &\qquad \mathtt{let}\ \mathit{recresult} = \mathsf{auxmerge}(xs_1, (x_2 :: xs_2), y);\\ &\qquad \mathtt{let}\ \mathit{result} = x_1 :: \mathit{recresult};\\ &\qquad \mathtt{return}\ \mathit{result}\\ &\quad \mathsf{else}\\ &\qquad \mathtt{let}\ \mathit{recresult} = \mathsf{auxmerge}((x_1 :: xs_1), xs_2, y);\\ &\qquad \mathtt{let}\ \mathit{result} = x_2 :: \mathit{recresult};\\ &\qquad \mathtt{return}\ \mathit{result}\end{aligned}$$

We use auxmerge to define merge:

$$
\begin{aligned}
&\mathsf{merge} : \forall m, n : \mathbb{N}.\ \mathsf{List}(\mathsf{Nat})(m) \to \mathsf{List}(\mathsf{Nat})(n) \to \uparrow\mathsf{List}(\mathsf{Nat})(m+n)\\
&\mathsf{merge}\ xs\ ys =\\
&\quad \mathtt{let}\ lenxs = \mathsf{length}(xs);\\
&\quad \mathtt{let}\ lenys = \mathsf{length}(ys);\\
&\quad \mathtt{let}\ lenxsys = \mathsf{add}(lenxs, lenys);\\
&\quad \mathtt{let}\ result = \mathsf{auxmerge}(xs, ys, lenxsys);\\
&\quad \mathtt{return}\ result
\end{aligned}
$$

Next, we define the function split that takes a list and splits it into two lists. We will need to divide the length of a list by two. Assume we have already implemented a program-level division function divide (the index-level $\div$ on natural numbers rounds down):

$$\mathsf{divide} : \forall m, n : \mathbb{N}.\ (n \neq 0) \supset \mathsf{Nat}(m) \to \mathsf{Nat}(n) \to \uparrow\mathsf{Nat}(m \div n)$$

We implement a function take that takes a natural number n and a list, and returns the first n elements of the list. Notice that the number (and list) get smaller at the recursive call:

$$
\begin{aligned}
&\mathsf{take} : \forall n, l : \mathbb{N}.\ (n \leq l) \supset \mathsf{Nat}(n) \to \mathsf{List}(\mathsf{Nat})(l) \to \uparrow\mathsf{List}(\mathsf{Nat})(n)\\
&\mathsf{take}\ \mathsf{zero}\ xs = \mathtt{return}\ []\\
&\mathsf{take}\ (\mathsf{succ}\ y)\ (x :: xs) =\\
&\quad \mathtt{let}\ recresult = \mathsf{take}\ y\ xs;\\
&\quad \mathtt{let}\ result = x :: recresult;\\
&\quad \mathtt{return}\ result
\end{aligned}
$$

Similarly, the function drop takes a natural number n and a list, and returns the list with the first n elements dropped (we elide the implementation, which is similar to that of take):

$$\mathsf{drop} : \forall n, l : \mathbb{N}.\ (n \leq l) \supset \mathsf{Nat}(n) \to \mathsf{List}(\mathsf{Nat})(l) \to \uparrow\mathsf{List}(\mathsf{Nat})(l - n)$$

We are now ready to implement split:

$$
\begin{aligned}
&\mathsf{split} : \forall n : \mathbb{N}.\ \mathsf{List}(\mathsf{Nat})(n) \to \uparrow(\mathsf{List}(\mathsf{Nat})(n \div 2)) \times (\mathsf{List}(\mathsf{Nat})(n - (n \div 2)))\\
&\mathsf{split}\ xs =\\
&\quad \mathtt{let}\ lenxs = \mathsf{length}\ xs;\\
&\quad \mathtt{let}\ leftlen = \mathsf{divide}(lenxs, \mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}));\\
&\quad \mathtt{let}\ leftxs = \mathsf{take}(leftlen, xs);\\
&\quad \mathtt{let}\ rightxs = \mathsf{drop}(leftlen, xs);\\
&\quad \mathtt{return}\ \langle leftxs, rightxs\rangle
\end{aligned}
$$

Finally, we implement a mergesort that is verified to terminate and to return a list of the same length as the input list. In the last clause of mergesort, we know the input list has a length of at least two, so when we split it, both *left* and *right* have smaller length:

$$
\begin{array}{l}
\mathsf{mergesort} : \forall n : \mathbb{N}.\ \mathsf{List}(\mathsf{Nat})(n) \to {\uparrow}\mathsf{List}(\mathsf{Nat})(n) \\
\mathsf{mergesort}\ \ [] = \mathtt{return}\ [] \\
\mathsf{mergesort}\ \ (x :: []) = \mathtt{return}\ (x :: []) \\
\mathsf{mergesort}\ \ (x :: xs) = \\
\quad \mathsf{let}\ \mathit{splitlist} = \mathsf{split}(x :: xs); \\
\quad \mathtt{match}\ \mathit{splitlist}\ \{ \\
\quad\quad \langle \mathit{left}, \mathit{right} \rangle \Rightarrow \\
\quad\quad\quad \mathsf{let}\ \mathit{sortleft} = \mathsf{mergesort}(\mathit{left}); \\
\quad\quad\quad \mathsf{let}\ \mathit{sortright} = \mathsf{mergesort}(\mathit{right}); \\
\quad\quad\quad \mathsf{let}\ \mathit{result} = \mathsf{merge}(\mathit{sortleft}, \mathit{sortright}); \\
\quad\quad\quad \mathtt{return}\ \mathit{result} \\
\quad \}
\end{array}
$$

In the system of this paper, we cannot verify that mergesort returns a *sorted* list (we plan to extend the system to handle this in future work). But this example is nonetheless interesting because mergesort is *not* structurally recursive (it splits the input list in half), and it showcases the main features of our system, which we turn to next.

4 DECLARATIVE SYSTEM

We present our core declarative calculus and type system.

Program terms. Program terms are defined in Fig. 1. We polarize terms into two main syntactic categories: expressions (which have negative type) and values (which have positive type). Program terms are further distinguished according to whether their (principal) types are synthesized (heads and bound expressions) or checked (spines and patterns).

$$
\begin{array}{lrcl}
\text{Program variables} & x, y, z & & \\
\text{Expressions} & e & ::= & \mathtt{return}\, v \mid \mathtt{let}\ x = g;\ e \mid \mathtt{match}\ h\ \{r_i \Rightarrow e_i\}_{i \in I} \\
& & \mid & \lambda x.\, e \mid \mathtt{rec}\ x : N.\ e \\
\text{Values} & v & ::= & x \mid \langle\rangle \mid \langle v, v \rangle \mid \mathtt{inj}_1\, v \mid \mathtt{inj}_2\, v \mid \mathtt{into}(v) \mid \{e\} \\
\text{Heads} & h & ::= & x \mid (v : P) \\
\text{Bound expressions} & g & ::= & h(s) \mid (e : {\uparrow}P) \\
\text{Spines} & s & ::= & \cdot \mid v, s \\
\text{Patterns} & r & ::= & \mathtt{into}(x) \mid \langle\rangle \mid \langle x, y \rangle \mid \mathtt{inj}_1\, x \mid \mathtt{inj}_2\, x
\end{array}
$$

Fig. 1. Program terms

Expressions e consist of functions $\lambda x.\, e$, recursive expressions $\mathtt{rec}\ x : N.\ e$, let-bindings $\mathtt{let}\ x = g;\ e$, match expressions $\mathtt{match}\ h\ \{r_i \Rightarrow e_i\}_{i \in I}$, and value returners (or producers) $\mathtt{return}\, v$. Bound expressions g, which can be let-bound, consist of expressions annotated with a returner type $(e : {\uparrow}P)$ and applications $h(s)$ of a head h to a spine s. Heads h, which can be applied to a spine or pattern-matched, consist of variables x and positive-type-annotated values $(v : P)$. Spines s are lists of values; we often omit the empty spine $\cdot$, writing (for example) v_1, v_2 instead of $v_1, v_2, \cdot$. In match expressions, heads are matched against patterns r.

Values consist of variables x, the unit value $\langle\rangle$, pairs $\langle v_1, v_2\rangle$, injections into sum type $\mathtt{inj}_k\, v$ where k is 1 or 2, rollings into inductive type $\mathtt{into}(v)$, and thunks (suspended computations) $\{e\}$.

Types. Types are defined in Fig. 2. Types are polarized into positive (value) types P and negative (computation) types N. We write A, B and C for types of either polarity.

$$\begin{array}{lrcl}
\text{Positive types} & P, Q & ::= & 1 \mid P \times Q \mid 0 \mid P + Q \mid \downarrow N \\
 & & & \mid \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\} \\
 & & & \mid \exists a : \tau.\ P \mid P \land \phi \\
\text{Negative types} & N, M & ::= & P \to N \mid \uparrow P \mid \forall a : \tau.\ N \mid \phi \supset N \\
\text{Types} & A, B, C & ::= & P \mid N
\end{array}$$

Fig. 2. Types

Positive types consist of the unit type 1, products $P_1 \times P_2$, the void type 0, sums $P_1 + P_2$, downshifts (of negative types; *thunk* types) $\downarrow N$, *asserting* types $P \land \phi$ (read "P with ϕ"), index-level existential quantifications $\exists a : \tau.\ P$, and refined inductive types $\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}$. We read $\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}$ as the type having values ν of inductive type μF (with signature F) such that the (index-level) *measurement* $(\mathsf{fold}_F\ \alpha)\ \nu =_\tau t$ holds; in Sec. 4.0.1 and Sec. 5, we explain the metavariables F, α, τ, and t, as well as what these and the syntactic parts μ and fold denote. Briefly, μ roughly denotes "least fixed point of" and a fold over F with α (having carrier sort τ) indicates a measure on the inductive type μF into τ.

Negative types consist of function types $P \to N$, upshifts (of positive types; *lift* or *returning* types) $\uparrow P$ (dual to $\downarrow N$), propositionally *guarded* types $\phi \supset N$ (read "ϕ implies N"; dual to $P \land \phi$), and index-level universal quantifications $\forall a : \tau.\ N$ (dual to $\exists a : \tau.\ P$).

In $P \land \phi$ and $\phi \supset N$, the index proposition ϕ has no run-time content. Neither does the a in $\exists a : \tau.P$ and $\forall a : \tau.\ N$, nor the recursive refinement predicate $(\mathsf{fold}_F\ \alpha)\ \nu =_\tau t$ in $\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}$.

Index language: sorts, terms, and propositions. Our type system is parametric in the index domain, provided the latter has certain properties. For our system to be decidable, the index domain must be decidable. It is instructive to work with a specific index domain: Figure 3 defines a quantifier-free logic of linear equality, inequality, and arithmetic, which is decidable [Barrett et al. 2009].

$$\begin{array}{lrcl}
\text{Sorts} & \tau & ::= & \mathbb{B} \mid \mathbb{N} \mid \mathbb{Z} \mid \tau \times \tau \\
\\
\text{Index variables} & a, b, c & & \\
\text{Index terms} & t & ::= & a \mid n \mid t + t \mid t - t \mid (t, t) \mid \pi_1\, t \mid \pi_2\, t \mid \phi \\
\text{Propositions} & \phi, \psi & ::= & t = t \mid t \le t \mid \phi \land \phi \mid \phi \lor \phi \mid \neg\phi \mid \mathsf{T} \mid \mathsf{F}
\end{array}$$

Fig. 3. Index domain

Sorts τ consist of booleans $\mathbb{B}$, natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, and products $\tau_1 \times \tau_2$. Index terms t consist of variables a, numeric constants n, addition $t_1 + t_2$, subtraction $t_1 - t_2$, pairs (t_1, t_2), projections $\pi_1\, t$ and $\pi_2\, t$, and propositions ϕ. Propositions ϕ (the logic of the index domain) are built over index terms, and consist of equality $t_1 = t_2$, inequality $t_1 \le t_2$, conjunction $\phi_1 \land \phi_2$, disjunction $\phi_1 \lor \phi_2$, negation $\neg\phi$, trivial truth T, and trivial falsity F.

4.0.1 Inductive types, functors, and algebras. For convenient semantic metatheory, we encode inductive types (and measures on them) in a way that looks very close to existing (categorical) semantics of inductive types. In the introduction (Sec. 1), to refine the type of A-lists by their length, we defined a recursive function len over the inductive structure of lists. Semantically, we characterize this structural recursion by algebraic folds over polynomial endofunctors; we design our system to resemble this semantics.

We express inductive type structure without reference to constructor names (which is metatheoretically convenient) by syntactic functors resembling their own standard semantics. For example (modulo the difference for simplifying unrolling), we can specify the signature of the inductive type of lists of terms of type A syntactically as a functor $\underline{1} \oplus (\underline{A} \otimes \mathrm{Id})$, where $\underline{C}$ denotes the constant (set) functor (sending any set to the set denoted by type C), Id denotes the identity functor (sending any set to itself), the denotation of $F_1 \otimes F_2$ sends a set X to the product $(\llbracket F_1 \rrbracket\, X) \times (\llbracket F_2 \rrbracket\, X)$, and the denotation of $F_1 \oplus F_2$ sends a set X to the disjoint union $(\llbracket F_1 \rrbracket\, X) \uplus (\llbracket F_2 \rrbracket\, X)$. The idea is that each component of the sum functor $\oplus$ represents a data constructor, so that (for example) $\underline{1}$ represents the nullary constructor $[]$, and $\underline{A}$ represents the head element of a cons cell which is attached (via $\otimes$) to the recursive tail list represented by Id.

A direct grammar F for sums of products (of constant and identity functors) consists of $F ::= \hat{P} \mid F \oplus F$ and $\hat{P} ::= B \mid B \otimes \hat{P}$ and $B ::= \underline{P} \mid \mathrm{Id}$. The grammar $F ::= F \oplus F \mid F \otimes F \mid B$ is semantically equivalent to sums of products, but syntactically inconvenient, because it allows writing products of sums. We do not use either of these grammars, but rather Fig. 4, because it simplifies inductive type unrolling (Sec. 4.6). These grammars have naturally isomorphic interpretations. For example, in our functor grammar (Fig. 4), we instead write $\mathsf{NatF} = I \oplus (\mathrm{Id} \otimes I)$: notice that for any set X, we have $\llbracket I \oplus (\mathrm{Id} \otimes I) \rrbracket\, X = 1 \uplus (X \times 1) \cong 1 \uplus X = \llbracket \underline{1} \oplus \mathrm{Id} \rrbracket\, X$.

$$
\begin{array}{llcl}
\text{Functors} & F, G, H & ::= & \hat{P} \mid F \oplus F \\
 & \hat{P} & ::= & I \mid B \otimes \hat{P} \\
 & B & ::= & \underline{P} \mid \mathrm{Id} \\
 & \mathcal{F} & ::= & F \mid B
\end{array}
$$

Fig. 4. Functors

A functor F (Fig. 4) is a sum ($\oplus$) of products ($\hat{P}$), which multiply ($\otimes$) base functors (B) consisting of identity functors that represent recursive positions (Id) and constant functors ($\underline{P}$) at positive type P. The rightmost factor in a product $\hat{P}$ is the (product) unit functor I. By convention, $\otimes$ has higher precedence than $\oplus$. For convenience in specifying functor well-formedness (appendix Fig. 7) and denotation (appendix Fig. 37), $\mathcal{F}$ is a functor F or a base functor B.

As we will discuss in Sec. 5, every polynomial endofunctor F has a fixed point μF that satisfies a recursion principle for defining measures (on μF) by folds with algebras. We define algebras in Fig. 5. An algebra α is a list of clauses $p \Rightarrow t$ which pattern match on algebraic structure (p, q, and o are patterns) and bind variables in index bodies t. Sum algebra patterns p consist of $\mathtt{inj}_1\, p$ and $\mathtt{inj}_2\, p$ (which match on sum functors $\oplus$). Product algebra patterns q consist of tuples (o, q) (which match on $\otimes$) ending in the unit pattern $()$ (which match on I). Base algebra patterns o consist of wildcard patterns _ (which match on constant functors $\underline{P}$), variable patterns a (which match on the identity functor Id), and pack patterns $\mathtt{pack}(a, o)$ (which match on *existential* constant functors $\underline{\exists a : \tau.\, P}$, where a is also bound in the bodies t of algebra clauses).

Algebras	α, β	::=	$\cdot \mid (p \Rightarrow t \mathrel{\|} \alpha)$
Sum algebra patterns	p	::=	$\mathtt{inj}_1\, p \mid \mathtt{inj}_2\, p \mid q$
Product algebra patterns	q	::=	$() \mid (o, q)$
Base algebra patterns	o	::=	$\_ \mid a \mid \mathtt{pack}(a, o)$

Fig. 5. Algebras

For example, given a type P, consider the functor $I \oplus (\underline{P} \otimes \mathrm{Id} \otimes I)$. To specify the function $\mathsf{length} : \mathsf{List}\ P \to \mathbb{N}$ computing the length of a list of values of type P, we write the algebra $\mathtt{inj}_1\, () \Rightarrow 0 \mathrel{\|} \mathtt{inj}_2\, (\_, (a, ())) \Rightarrow 1 + a$ with which to fold List P.

With the pack algebra pattern, we can use indexes of an inductive type in our measures. For example, given $a : \mathbb{N}$, and defining the singleton type $\mathsf{Nat}(a)$ as $\{\nu : \mu\mathsf{NatF} \mid (\mathsf{fold}_{\mathsf{NatF}}\ \mathsf{ixnat})\ \nu = a\}$ where $\mathsf{NatF} = I \oplus \mathrm{Id} \otimes I$ and $\mathsf{ixnat} = \mathtt{inj}_1\, () \Rightarrow 0 \mathrel{\|} \mathtt{inj}_2\, (a, ()) \Rightarrow 1 + a$, consider lists of natural numbers, specified by $I \oplus \underline{\exists b : \mathbb{N}.\ \mathsf{Nat}(b)} \otimes \mathrm{Id} \otimes I$. Folding such a list with the algebra $\mathtt{inj}_1\, () \Rightarrow 0 \mathrel{\|} \mathtt{inj}_2\, (\mathtt{pack}(b, \_), a, ()) \Rightarrow a + b$ sums all the numbers in the list. (Here, we updated the definitions in Sec. 2 to match our actual functor grammar.)

For measures relating indexes in structurally distinct positions within an inductive type, in future work we plan to extend our system with higher-order sorts $\tau_1 \Rightarrow \tau_2$. Doing so would allow us to refine, for example, integer lists to lists of integers *in increasing order*, because we could then compare the indexed elements of a list.

Contexts. A logical context Θ is an ordered list of propositions ϕ and index sort declarations $a : \tau$. A program variable context Γ is a list of type declarations $x : P$. A value-determined context Ξ is a set of index sort declarations $a : \tau$. In any kind of context, a variable can be declared at most once.

4.1 Index sorting and propositional validity

We have a standard index sorting judgment $\Theta \vdash t : \tau$ (appendix Fig. 4) checking that, under context Θ, index term t has sort τ. For example, $a : \mathbb{N} \vdash \neg(a \leq a + 1) : \mathbb{B}$. This judgment does not depend on propositions in Θ, which only matter when checking propositional validity (only done in subtyping and program typing).

An *index-level semantic substitution* $\vdash \delta : \Theta$ is a list assigning exactly one semantic index value to each index variable in $dom(\Theta)$ such that all propositions in Θ are true (written $\{\bullet\}$):

$$\frac{}{\vdash \cdot : \cdot} \qquad \frac{\vdash \delta : \Theta \qquad d \in [\![\tau]\!] \qquad a \notin dom(\Theta)}{\vdash (\delta, d/a) : (\Theta, a : \tau)} \qquad \frac{\vdash \delta : \Theta \qquad [\![\phi]\!]_\delta = \{\bullet\}}{\vdash \delta : (\Theta, \phi)}$$

We define $[\![\Theta]\!] = \{\delta \mid \vdash \delta : \Theta\}$. Well-sorted index terms $\Theta \vdash t : \tau$ denote functions $[\![t]\!] : \left[\!\left[\overline{\Theta}\right]\!\right] \to [\![\tau]\!]$ where $\overline{\Theta}$ filters propositions out of Θ. For example, $[\![4 + a]\!]_{3/a} = 7$ and $[\![b = 1 + 0]\!]_{1/b} = \{\bullet\}$ (that is, true) and $[\![a = 1]\!]_{2/a} = \emptyset$ (that is, false).

A *propositional validity* or *truth* judgment $\Theta \vdash \phi$ true, which is a *semantic* entailment relation, holds if ϕ is valid under Θ, that is, if ϕ is true under every interpretation of variables in Θ such that all propositions in Θ are true. We say t and t' are *logically equal* under Θ if $\Theta \vdash t = t'$ true.

An *index-level syntactic substitution* σ is a list of index terms to be substituted for index variables: $\sigma ::= \cdot \mid \sigma, t/a$. The metaoperation $[\sigma]O$, where O is an index term, program term, or type, is sequential substitution: $[\cdot]O = O$ and $[\sigma, t/a]O = [\sigma]([t/a]O)$, where $[t/a]O$ is standard capture-avoiding (by α-renaming) substitution. Syntactic substitutions (index-level) are typed ("sorted") in a standard way. Because syntactic substitutions substitute terms that may have free variables, their judgment form includes a context to the left of the turnstile, in contrast to semantic substitution:

$$\frac{}{\Theta_0 \vdash \cdot : \cdot} \qquad \frac{\Theta_0 \vdash \sigma : \Theta \qquad \Theta_0 \vdash [\sigma]t : \tau \qquad a \notin dom(\Theta)}{\Theta_0 \vdash (\sigma, t/a) : (\Theta, a : \tau)} \qquad \frac{\Theta_0 \vdash \sigma : \Theta \qquad \Theta_0 \vdash [\sigma]\phi \text{ true}}{\Theta_0 \vdash \sigma : (\Theta, \phi)}$$

Because our substitution operation $[\sigma]-$ applies sequentially, we type ("sort") the application of the rest of the substitution to the head being substituted.

The decidability of our system depends on the decidability of propositional validity. Our example index domain is decidable [Barrett et al. 2009]. We mentioned earlier that our system is parametric in the index domain, *provided the latter has certain properties*. In particular, propositional validity must satisfy the following properties (Θ ctx is logical context well-formedness).

- *Weaken*: If $\Theta_1, \Theta, \Theta_2$ ctx and $\Theta_1, \Theta_2 \vdash \phi$ true, then $\Theta_1, \Theta, \Theta_2 \vdash \phi$ true.
- *Permute*: If Θ, Θ_1 ctx and Θ, Θ_2 ctx and $\Theta, \Theta_1, \Theta_2, \Theta' \vdash \phi$ true, then $\Theta, \Theta_2, \Theta_1, \Theta' \vdash \phi$ true.
- *Substitution*: If $\Theta \vdash \phi$ true and $\Theta_0 \vdash \sigma : \Theta$, then $\Theta_0 \vdash [\sigma]\phi$ true.
- *Equivalence*: The relation $\Theta \vdash t_1 = t_2$ true is an equivalence relation.
- *Assumption*: We have $\Theta_1, \phi, \Theta_2 \vdash \phi$ true.
- *Consequence*: If $\Theta_1 \vdash \psi$ true and $\Theta_1, \psi, \Theta_2 \vdash \phi$ true, then $\Theta_1, \Theta_2 \vdash \phi$ true.
- *Consistency*: It is not the case that $\cdot \vdash \mathsf{F}$ true.

Our example index domain satisfies all these properties. We also assume (and know for our example index domain) that $\Theta \vdash t : \tau$ is decidable and satisfies weakening and substitution.

4.2 Well-formedness

Type well-formedness $\Theta \vdash A$ type$[\Xi]$ (read "under Θ, type A is well-formed with value-determined indexes Ξ") has Ξ in output mode, which tracks index variables appearing in the type A that are uniquely[3] determined by values of refined inductive types in A, particularly by their folds. (See Lemma 5.4 in Sec. 5.) Consider the following type well-formedness rule:

$$\frac{\Theta \vdash F \text{ functor}[\Xi] \qquad \cdot; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau \qquad (b : \tau) \in \Theta}{\Theta \vdash \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau b\} \text{ type}[\Xi \cup b : \tau]}\ \text{DeclTp}\mu\text{Var}$$

The index b is uniquely determined by a value of the conclusion type, so we add it to Ξ. For example, the value one $= \mathtt{into}(\mathtt{inj}_2\ \langle \mathtt{into}(\mathtt{inj}_1\ \langle\rangle), \langle\rangle\rangle)$ determines the variable b appearing in the value's type $\mathsf{NatF}(b) = \{\nu : \mu\mathsf{NatF} \mid (\mathsf{fold}_{\mathsf{NatF}}\ \mathsf{ixnat})\ \nu = b\}$ to be one. (We have a similar rule where the b-position metavariable is not an index variable, adding nothing to Ξ.) We use set union ($\Xi \cup b : \tau$) as b may already be value-determined in F (that is, $(b : \tau)$ may be in Ξ). The algebra well-formedness premise $\cdot; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau$ requires the algebra α to be closed (that is, the first context is empty, $\cdot$). This premise ensures that existential variables never appear in algebras, which is desirable because folds with algebras solve existential variables when typechecking a value (see Sec. 6).

Because ultimately Ξ tracks only measure-determined indexes, DeclTpμVar is the only rule that adds to Ξ. The index propositions of asserting and guarded types do not track anything beyond what is tracked by the types to which they are connected.

$$\frac{\Theta \vdash P \text{ type}[\Xi] \qquad \Theta \vdash \phi : \mathbb{B}}{\Theta \vdash P \land \phi \text{ type}[\Xi]}\ \text{DeclTp}\land \qquad \frac{\Theta \vdash N \text{ type}[\Xi] \qquad \Theta \vdash \phi : \mathbb{B}}{\Theta \vdash \phi \supset N \text{ type}[\Xi]}\ \text{DeclTp}\supset$$

We restrict quantification to value-determined index variables in order to guarantee we can always solve them algorithmically. For example, in checking one against the type $\exists a : \mathbb{N}. \mathsf{Nat}(a)$, we solve a to an index semantically equal to $1 \in \mathbb{N}$. If $\Theta, a : \tau \vdash P$ type$[\Xi]$, then $\exists a : \tau. P$ is well-formed

[3] Semantically speaking.

if and only if $(a : \tau) \in \Xi$, and similarly for universal quantification (which we'll restrict to the argument types of function types; argument types are positive):

$$\frac{\Theta, a : \tau \vdash P \ \mathsf{type}[\Xi, a : \tau]}{\Theta \vdash \exists a : \tau.\, P \ \mathsf{type}[\Xi]}\ \text{DeclTp}\exists \qquad \frac{\Theta, a : \tau \vdash N \ \mathsf{type}[\Xi, a : \tau]}{\Theta \vdash \forall a : \tau.\, N \ \mathsf{type}[\Xi]}\ \text{DeclTp}\forall$$

We read commas in value-determined contexts such as Ξ_1, Ξ_2 as set union $\Xi_1 \cup \Xi_2$ together with the fact that $dom(\Xi_1) \cap dom(\Xi_2) = \emptyset$, so these rules can be read top-down as removing a.

A value of product type is a pair of values, so we take the union of what each component value determines:

$$\frac{\Theta \vdash P_1 \ \mathsf{type}[\Xi_1] \qquad \Theta \vdash P_2 \ \mathsf{type}[\Xi_2]}{\Theta \vdash P_1 \times P_2 \ \mathsf{type}[\Xi_1 \cup \Xi_2]}\ \text{DeclTp}\times$$

We also take the union for function types $P \to N$, because to use a function, due to focusing, we must provide values for all its arguments. The Ξ of $\mathsf{Nat}(a) \to {\uparrow}\mathsf{Nat}(a)$ is $a : \mathbb{N}$, so $\forall a : \mathbb{N}.\, \mathsf{Nat}(a) \to {\uparrow}\mathsf{Nat}(a)$ is well-formed. In applying a head of this type to a value, we must instantiate a to an index semantically equal to what that value determines; for example, if the value is one, then a gets instantiated to an index semantically equal to $1 \in \mathbb{N}$.

However, a value of sum type is either a left- or right-injected value, but we don't know which, so we take the intersection of what each injection determines:

$$\frac{\Theta \vdash P_1 \ \mathsf{type}[\Xi_1] \qquad \Theta \vdash P_2 \ \mathsf{type}[\Xi_2]}{\Theta \vdash P_1 + P_2 \ \mathsf{type}[\Xi_1 \cap \Xi_2]}\ \text{DeclTp+}$$

The unit type 1 and void (empty) type 0 both have empty Ξ. We also empty out value-determined indexes at shifts, preventing certain quantifications over shifts. For example, $\forall a : \mathbb{N}.\, {\uparrow}\mathsf{Nat}(a)$ (which is void anyway) is not well-formed. Crucially, we will see that this restriction, together with focusing, guarantees indexes will be algorithmically solved by the end of certain phases.

We define functor and algebra well-formedness in Fig. 7 of the appendix.

Functor well-formedness $\Theta \vdash \mathcal{F} \ \mathsf{functor}[\Xi]$ is similar to type well-formedness: constant functors output the Ξ of the underlying positive type, the identity and unit functors Id and I have empty Ξ, the product functor $B \otimes \hat{P}$ takes the union of the component Ξs, and the sum functor $F_1 \oplus F_2$ takes the intersection. The latter two reflect the fact that unrolling inductive types (Sec. 4.6) generates + types from $\oplus$ functors and $\times$ types from $\otimes$ functors. That I has empty Ξ reflects that 1 (unrolled from I) does too, together with the fact that asserting and guarded types do not affect Ξ.

Algebra well-formedness $\Xi; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau$ (read "under Ξ and Θ, algebra α is well-formed and has type $F(\tau) \Rightarrow \tau$") has two contexts: Ξ is for α (in particular, the index bodies of its clauses) and Θ is for F (in particular, the positive types of constant functors); we maintain the invariant that $\Xi \subseteq \Theta$. We have these separate contexts to prevent existential variables from appearing in α (as explained with respect to DeclTpμVar) while still allowing them to appear in F. For example, consider $\exists b : \mathbb{N}.\, \{ \nu : \mu F(b) \mid (\mathsf{fold}_{F(b)}\ \alpha)\ \nu = n \}$ where $F(b) = (\underline{\mathsf{Nat}(b)} \otimes I) \oplus (\underline{\mathsf{Nat}(b)} \otimes \mathrm{Id} \otimes I)$ and $\alpha = \mathsf{inj}_1\, (\_, ()) \Rightarrow 0 \mid \mathsf{inj}_2\, (\_, a, ()) \Rightarrow 1 + a$.

Refined inductive type well-formedness initializes the input Ξ to $\cdot$, but index variables can be bound in the body of an algebra:

$$\frac{\Xi, a : \tau; \Theta, a : \tau \vdash q \Rightarrow t : \hat{P}(\tau) \Rightarrow \tau}{\Xi; \Theta \vdash (a, q) \Rightarrow t : (\mathrm{Id} \otimes \hat{P})(\tau) \Rightarrow \tau} \qquad \frac{\Xi, a : \tau'; \Theta, a : \tau' \vdash (o, q) \Rightarrow t : (\underline{Q} \otimes \hat{P})(\tau) \Rightarrow \tau}{\Xi; \Theta \vdash (\mathsf{pack}(a, o), q) \Rightarrow t : (\underline{\exists a : \tau'.\, Q} \otimes \hat{P})(\tau) \Rightarrow \tau}$$

where the right rule simultaneously binds a in both t and Q, and the left rule only binds a in t (but we add a to both contexts to maintain the invariant $\Xi \subseteq \Theta$). We sort algebra bodies only when a product ends at a unit (possible by design of the functor grammar), and merely under Ξ; constant functors depend on Θ:

$$\frac{\Xi \vdash t : \tau}{\Xi; \Theta \vdash () \Rightarrow t : I(\tau) \Rightarrow \tau} \qquad \frac{\Xi; \Theta \vdash q \Rightarrow t : \hat{P}(\tau) \Rightarrow \tau \qquad \Theta \vdash Q \ \mathsf{type}[\_]}{\Xi; \Theta \vdash (\_, q) \Rightarrow t : (\underline{Q} \otimes \hat{P})(\tau) \Rightarrow \tau}$$

For algebras α of "type" $(F_1 \oplus F_2)\ \tau \Rightarrow \tau$, we use a straightforward judgment $\alpha \circ \mathsf{inj}_k \doteq \alpha_k$ (appendix Fig. 5) that outputs the kth clause α_k of input algebra α:

$$\frac{\alpha \circ \mathsf{inj}_1 \doteq \alpha_1 \qquad \alpha \circ \mathsf{inj}_2 \doteq \alpha_2 \qquad \Xi; \Theta \vdash \alpha_1 : F_1(\tau) \Rightarrow \tau \qquad \Xi; \Theta \vdash \alpha_2 : F_2(\tau) \Rightarrow \tau}{\Xi; \Theta \vdash \alpha : (F_1 \oplus F_2)(\tau) \Rightarrow \tau}$$

By restricting the bodies of algebras to *index terms* t and the carriers of our F-algebras to *index sorts* τ, we uphold the phase distinction: we can therefore safely refine inductive types by folding them with algebras, and also manage decidable typing.

4.3 Equivalence

We have equivalence judgments for propositions $\Theta \vdash \phi \equiv \psi$ (appendix Fig. 11), logical contexts $\Theta \vdash \Theta_1 \equiv \Theta_2$ (appendix Fig. 12), functors $\Theta \vdash \mathcal{F} \equiv \mathcal{G}$ (appendix Fig. 13), and types $\Theta \vdash A \equiv^{\pm} B$ (appendix Fig. 14), which use $\Theta \vdash \phi$ true to verify logical equality of index terms. Basically, two entities are equivalent if their respective, structural subparts are equivalent (under the logical context). For space reasons, we do not show all their rules here (see appendix), only the ones we think are most likely to surprise.

Refined inductive types are equivalent only if they use syntactically the same algebra (but the algebra must be well-formed at both functors F and G; this holds by inversion on the conclusion's presupposed type well-formedness judgments):

$$\frac{\Theta \vdash F \equiv G \qquad \Theta \vdash t = t' \ \mathsf{true}}{\Theta \vdash \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\} \equiv^+ \{\nu : \mu G \mid (\mathsf{fold}_G\ \alpha)\ \nu =_\tau t'\}}$$

Two index equality propositions (respectively, two index inequalities) are equivalent if their respective sides are logically equal:

$$\frac{\Theta \vdash t_1 = t_1' \ \mathsf{true} \qquad \Theta \vdash t_2 = t_2' \ \mathsf{true}}{\Theta \vdash t_1 = t_2 \equiv t_1' = t_2'} \qquad \frac{\Theta \vdash t_1 = t_1' \ \mathsf{true} \qquad \Theta \vdash t_2 = t_2' \ \mathsf{true}}{\Theta \vdash t_1 \leq t_2 \equiv t_1' \leq t_2'}$$

We use logical context equivalence in proving subsumption admissibility (see Sec. 4.8) and the completeness of algorithmic typing (see Sec. 7.3). Two logical contexts are judgmentally equivalent under Θ if they have exactly the same variable declarations (in the same list positions) and logically equivalent (under Θ) propositions, in the same order. The most interesting rule is the one for propositions, where, in the second premise, we filter out propositions from Θ_1 because we want each respective proposition to be logically equivalent under the propositions (and indexes) of Θ, but variables in Θ_1 (or Θ_2) may appear in ϕ_1 (or ϕ_2):

$$\frac{\Theta \vdash \Theta_1 \equiv \Theta_2 \qquad \Theta, \overline{\Theta_1} \vdash \phi_1 \equiv \phi_2}{\Theta \vdash \Theta_1, \phi_1 \equiv \Theta_2, \phi_2}$$

(Note that it is equivalent to use $\overline{\Theta_2}$ rather than $\overline{\Theta_1}$ in the second premise above.)

All equivalence judgments satisfy reflexivity (appendix Lemmas B.74 and B.75), symmetry (appendix Lemmas B.90 and B.91), and transitivity (appendix Lemma B.77).

4.4 Extraction

The judgment $\Theta \vdash A \rightsquigarrow^{\pm} A' \ [\Theta_A]$ (Fig. 6) extracts quantified variables and $\wedge$ and $\supset$ propositions from the type A, outputting the type A' without these, and the context Θ_A with them. We call A' and Θ_A the type and context *extracted* from A. For negative A, everything is extracted up to an upshift. For positive A, everything is extracted up to any connective that is not $\exists$, $\wedge$, or $\times$. For convenience in program typing (Sec. 4.7), $\Theta \vdash A \not\rightsquigarrow$ abbreviates $\Theta \vdash A \rightsquigarrow A \ [\cdot]$ (we sometimes omit the polarity label from extraction judgments). If $\Theta \vdash A \not\rightsquigarrow$, then we say A is *simple*.

$\boxed{\Theta \vdash A \rightsquigarrow^{\pm} A' \ [\Theta_A]}$ Under Θ, type A extracts to A' and Θ_A

$$\frac{P \neq \exists, \wedge, \text{ or } \times}{\Theta \vdash P \rightsquigarrow^{+} P \ [\cdot]} \rightsquigarrow^{+}\neq \qquad \frac{\Theta \vdash P \rightsquigarrow^{+} P' \ [\Theta_P]}{\Theta \vdash P \wedge \phi \rightsquigarrow^{+} P' \ [\phi, \Theta_P]} \rightsquigarrow^{+}\wedge$$

$$\frac{\Theta, a : \tau \vdash P \rightsquigarrow^{+} P' \ [\Theta_P]}{\Theta \vdash \exists a : \tau.\, P \rightsquigarrow^{+} P' \ [a : \tau, \Theta_P]} \rightsquigarrow^{+}\exists \qquad \frac{\Theta \vdash P_1 \rightsquigarrow^{+} P_1' \ [\Theta_{P_1}] \qquad \Theta \vdash P_2 \rightsquigarrow^{+} P_2' \ [\Theta_{P_2}]}{\Theta \vdash P_1 \times P_2 \rightsquigarrow^{+} P_1' \times P_2' \ [\Theta_{P_1}, \Theta_{P_2}]} \rightsquigarrow^{+}\times$$

$$\frac{}{\Theta \vdash {\uparrow}P \rightsquigarrow^{-} {\uparrow}P \ [\cdot]} \rightsquigarrow^{-}\neq \qquad \frac{\Theta \vdash N \rightsquigarrow^{-} N' \ [\Theta_N]}{\Theta \vdash \phi \supset N \rightsquigarrow^{-} N' \ [\phi, \Theta_N]} \rightsquigarrow^{-}\supset$$

$$\frac{\Theta, a : \tau \vdash N \rightsquigarrow^{-} N' \ [\Theta_N]}{\Theta \vdash \forall a : \tau.\, N \rightsquigarrow^{-} N' \ [a : \tau, \Theta_N]} \rightsquigarrow^{-}\forall \qquad \frac{\Theta \vdash P \rightsquigarrow^{+} P' \ [\Theta_P] \qquad \Theta \vdash N \rightsquigarrow^{-} N' \ [\Theta_N]}{\Theta \vdash P \to N \rightsquigarrow^{-} P' \to N' \ [\Theta_P, \Theta_N]} \rightsquigarrow^{-}\to$$

Fig. 6. Declarative extraction

4.5 Subtyping

Declarative subtyping $\Theta \vdash A \leq^{\pm} B$ is defined in Fig. 7.

Subtyping is polarized into mutually recursive positive $\Theta \vdash P \leq^{+} Q$ and negative $\Theta \vdash N \leq^{-} M$ relations. The design of inference rules for subtyping is guided by sequent calculi, perhaps most clearly seen in the left and right rules pertaining to quantifiers ($\exists$, $\forall$), asserting types ($\wedge$), and guarded types ($\supset$). This is helpful to establish key properties such as *reflexivity* and *transitivity* (viewing subtyping as a sequent system, we might instead say that the structural *identity* and *cut* rules, respectively, are admissible[4]). We interpret types as sets with some additional structure (Sec. 5), but considering only the sets, we prove that a subtype denotes a subset of the set denoted by any of its supertypes. That is, membership of a (semantic) value in the subtype *implies* its membership in any supertype of the subtype. We may also view subtyping as implication.

Instead of $\leq^{+}\rightsquigarrow$L, one might reasonably expect these two rules (the brackets around the rule names indicate that these rules are *not* in our system):

$$\frac{\Theta, \phi \vdash P \leq^{+} Q}{\Theta \vdash P \wedge \phi \leq^{+} Q} [\leq^{+}\wedge\text{L}] \qquad \frac{\Theta, a : \tau \vdash P \leq^{+} Q}{\Theta \vdash \exists a : \tau.\, P \leq^{+} Q} [\leq^{+}\exists\text{L}]$$

Similarly, one might expect to have $[\leq^{-}\supset\text{R}]$ and $[\leq^{-}\forall\text{R}]$, dual to the above rules, instead of the dual rule $\leq^{-}\rightsquigarrow$R. Reading, for example, the above rule $[\leq^{+}\wedge\text{L}]$ logically and top-down, if Θ and ϕ implies that P implies Q, then we can infer that Θ implies that P and ϕ implies Q. We can also read rules as a bottom-up decision procedure: given $P \wedge \phi$, we know ϕ, so we can assume it;

[4]A proposed inference rule is *admissible* with respect to a system if, whenever the premises of the proposed rule are derivable, we can derive the proposed rule's conclusion using the system's inference rules.

$\boxed{\Theta \vdash A \leq^{\pm} B}$ Under Θ, type A is a subtype of B

$$\frac{}{\Theta \vdash 1 \leq^+ 1}\ {\leq^+}1 \qquad \frac{}{\Theta \vdash 0 \leq^+ 0}\ {\leq^+}0$$

$$\frac{\Theta \vdash P_1 \leq^+ Q_1 \qquad \Theta \vdash P_2 \leq^+ Q_2}{\Theta \vdash P_1 \times P_2 \leq^+ Q_1 \times Q_2}\ {\leq^+}{\times} \qquad \frac{\Theta \vdash P_1 \equiv^+ Q_1 \qquad \Theta \vdash P_2 \equiv^+ Q_2}{\Theta \vdash P_1 + P_2 \leq^+ Q_1 + Q_2}\ {\leq^+}{+}$$

$$\frac{\Theta \vdash P \rightsquigarrow^+ P' \ [\Theta_P] \qquad \Theta_P \neq \cdot \qquad \Theta, \Theta_P \vdash P' \leq^+ Q}{\Theta \vdash P \leq^+ Q}\ {\leq^+}{\rightsquigarrow}\mathrm{L}$$

$$\frac{\Theta \vdash P \leq^+ Q \qquad \Theta \vdash \phi \ \text{true}}{\Theta \vdash P \leq^+ Q \land \phi}\ {\leq^+}{\land}\mathrm{R} \qquad \frac{\Theta \vdash P \leq^+ [t/a]Q \qquad \Theta \vdash t : \tau}{\Theta \vdash P \leq^+ \exists a : \tau.\, Q}\ {\leq^+}{\exists}\mathrm{R}$$

$$\frac{\Theta \vdash F \equiv G \qquad \Theta \vdash t = t' \ \text{true}}{\Theta \vdash \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\} \leq^+ \{\nu : \mu G \mid (\mathsf{fold}_G\ \alpha)\ \nu =_\tau t'\}}\ {\leq^+}\mu$$

$$\frac{\Theta \vdash N \leq^- M}{\Theta \vdash {\downarrow}N \leq^+ {\downarrow}M}\ {\leq^+}{\downarrow} \qquad \frac{\Theta \vdash P \leq^+ Q}{\Theta \vdash {\uparrow}P \leq^- {\uparrow}Q}\ {\leq^-}{\uparrow}$$

$$\frac{\Theta \vdash N \leq^- M \qquad \Theta \vdash \phi \ \text{true}}{\Theta \vdash \phi \supset N \leq^- M}\ {\leq^-}{\supset}\mathrm{L} \qquad \frac{\Theta \vdash [t/a]N \leq^- M \qquad \Theta \vdash t : \tau}{\Theta \vdash \forall a : \tau.\, N \leq^- M}\ {\leq^-}{\forall}\mathrm{L}$$

$$\frac{\Theta \vdash M \rightsquigarrow^- M' \ [\Theta_M] \qquad \Theta_M \neq \cdot \qquad \Theta, \Theta_M \vdash N \leq^- M'}{\Theta \vdash N \leq^- M}\ {\leq^-}{\rightsquigarrow}\mathrm{R}$$

$$\frac{\Theta \vdash Q \leq^+ P \qquad \Theta \vdash N \leq^- M}{\Theta \vdash P \to N \leq^- Q \to M}\ {\leq^-}{\to}$$

Fig. 7. Declarative subtyping

given $\exists a : \tau.\, P$, we know there exists an index of sort τ such that P, but we don't have a specific index term. However, these rules are not powerful enough to derive reasonable judgments such as $a : \mathbb{N} \vdash 1 \times (1 \land a = 3) \leq^+ (1 \land a \geq 3) \times 1$: subtyping for the first component requires verifying $a \geq 3$, which is impossible under no logical assumptions. However, from a logical perspective, $1 \times (1 \land a = 3)$ implies $a \geq 3$. Reading ${\leq^+}{\rightsquigarrow}\mathrm{L}$ bottom-up, in this case, we extract $a = 3$ from the subtype, which we later use to verify that $a \geq 3$. The idea is that, for a type in an assumptive position, it does not matter which product component (products are viewed conjunctively) or function argument (in our system, functions must be fully applied to values) to which index data is attached. Moreover, as we'll explain at the end of Sec. 6, the weaker rules by themselves are incompatible with algorithmic completeness. We emphasize that we do *not* include [${\leq^+}{\land}\mathrm{L}$], [${\leq^+}{\exists}\mathrm{L}$], [${\leq^-}{\supset}\mathrm{R}$] or [${\leq^-}{\forall}\mathrm{R}$] in the system.

For the unit type and the void type, rules ${\leq^+}1$ and void ${\leq^+}0$ are simply reflexivity. Product subtyping ${\leq^+}{\times}$ is covariant subtyping of component types: a product type is a subtype of another if each component of the former is a subtype of the respective component of the latter. We have covariant shift rules ${\leq^+}{\downarrow}$ and ${\leq^-}{\uparrow}$. Function subtyping ${\leq^-}{\to}$ is standard: contravariant (from conclusion to premise, the subtyping direction flips) in the function type's domain and covariant in the function type's codomain.

Rule $\leq^+\wedge$R and its dual rule $\leq^-\supset$L verify the validity of the attached proposition. In rule $\leq^+\exists$R and its dual rule $\leq^-\forall$L, we assume that we can conjure a suitable index term t; in practice (that is, algorithmically), we must introduce an existential variable $\hat{a}$ and then solve it.

Rule $\leq^+ +$ says a sum is a subtype of another sum if their respective subparts are (judgmentally) *equivalent*. The logical reading of subtyping begins to clarify why we don't extract anything under a sum connective: $(1 \wedge \mathsf{F}) + 1$ does not imply F. However, using equivalence here is a conservative restriction: for example, $(1 \wedge \mathsf{F}) + (1 \wedge \mathsf{F})$ does imply F, but making the general idea behind this work in our system would add complexity; we leave it for potential future work. Regardless, we don't expect this to be very restrictive in practice because programmers tend not to work with sum types themselves, but rather algebraic inductive types (like μF), and don't need to directly compare, via subtyping, (the unrolling of) different such types (such as the type of lists and the type of natural numbers).

In rule $\leq^+\mu$, just as in the refined inductive type equivalence rule (Sec. 4.3), a refined inductive type is a subtype of another type if they have judgmentally equivalent functors, they use syntactically the same algebra (that agrees with both subtype and supertype functors), and the index terms on the right-hand side of their measurements are equal under the logical context. As we discuss in Sec. 9, adding polymorphism to the language (future work) might necessitate replacing type and functor equivalence in subtyping with subtyping and "subfunctoring".

In the appendix, we prove that subtyping is reflexive (Lemma B.76) and transitive (Lemma B.82).

Subtyping and type equivalence. We prove that type equivalence implies subtyping (appendix Lemma B.95). To prove that, we use the fact that if Θ_1 is logically equivalent to Θ_2 under their prefix context Θ (judgment $\Theta \vdash \Theta_1 \equiv \Theta_2$) then we can swap Θ_1 and Θ_2 in derivations (appendix Lemma B.94). We use appendix Lemma B.95 to prove subsumption admissibility (Sec. 4.8) and a subtyping constraint verification transport lemma (mentioned in Sec. 7.2). Conversely, mutual subtyping does not imply type equivalence: $1 \wedge \mathsf{T} \leq 1$ and $1 \leq 1 \wedge \mathsf{T}$ but $1 \not\equiv 1 \wedge \mathsf{T}$.

4.6 Unrolling

Given $a : \mathbb{N}$, in our system, the type $\mathsf{List}\ A\ a$ of a-length lists of elements of type A is defined as $\left\{\nu : \mu\mathsf{ListF}_A \;\middle|\; (\mathsf{fold}_{\mathsf{ListF}_A}\ \mathsf{lenalg})\ \nu = a\right\}$ where $\mathsf{ListF}_A = I \oplus (\underline{A} \otimes \mathrm{Id} \otimes I)$ and $\mathsf{lenalg} = \mathtt{inj}_1\ () \Rightarrow 0 \,\|\, \mathtt{inj}_2\ (\_, (a, ())) \Rightarrow 1 + a$. Assuming we have $\mathsf{succ} : \forall a : \mathbb{N}.\ \mathsf{Nat}(a) \to \uparrow\mathsf{Nat}(1 + a)$ for incrementing a (program-level) natural number by one, we define length in our system as follows:

```
rec length : (∀a : ℕ. List A a → ↑Nat(a)). λx. match x. {
  into(x′) ⇒ match x′ {
    inj₁ ⟨⟩ ⇒                        -- a = 0
      return into(inj₁ ⟨⟩)
  | inj₂ ⟨_, ⟨y, ⟨⟩⟩⟩ ⇒              -- a = 1 + a′ such that a′ is the length of y
      let z′ = length(y);
      let z = succ(z′);
      return z
  }
}
```

Checking this program against its type annotation, the lambda rule assumes $x : \mathsf{List}\ A\ a$ for an arbitrary $a : \mathbb{N}$. Upon matching x against the pattern $\mathtt{into}(x')$, we know x' should have the *unrolled* type of $\mathsf{List}\ A\ a$. Ignoring refinements, we know that the erasure of this unrolling should be a sum type where the left component represents the empty list and the right component represents a head element together with a tail list. However, in order to verify the refinement that length does what we intend, we need to know more about the length index associated with x—that is, a—in the case where x is nil and in the case where x is a cons cell. Namely, the unrolling of $\mathsf{List}\ A\ a$ should know that $a = 0$ when x is the empty list, and that $a = 1 + a'$ where a' is the length of the tail of x when x is a nonempty list. This is the role of the unrolling judgment, to output just what we need here:

$$\begin{aligned}&\cdot;\cdot \vdash \left\{v : \mathsf{ListF}_A[\mu\mathsf{ListF}_A] \;\middle|\; \mathsf{lenalg}\left(\mathsf{ListF}_A\ (\mathsf{fold}_{\mathsf{ListF}_A}\ \mathsf{lenalg})\ v\right) =_{\mathbb{N}} a\right\}\\ &\qquad \doteq (1 \wedge a = 0) + \left(A \times (\exists a' : \mathbb{N}.\ \left\{v : \mu\mathsf{ListF}_A \;\middle|\; (\mathsf{fold}_{\mathsf{ListF}_A}\ \mathsf{lenalg})\ v =_{\mathbb{N}} a'\right\} \times (1 \wedge a = 1 + a'))\right)\end{aligned}$$

That is, the type of A-lists of length a unrolls to either the unit type 1 (representing the empty list) together with the fact that a is 0, or the product of A (the type of the head element) and A-lists (representing the tail) of length a' such that a' is a minus one.

Refined inductive type unrolling $\Xi; \Theta \vdash \{v : G[\mu F] \mid \beta\,(G\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq P$ is defined in Fig. 8. There are two contexts: Ξ is for β and Θ is for G, F, and t. Similarly to algebra well-formedness, we maintain the invariant in unrolling that $\Xi \subseteq \Theta$. The (non-contextual) *input* metavariables are G, F, β, α, τ, and t. The type P, called the *unrolled* type, is an output. As in the list example above, inductive type unrolling is always initiated with $\Xi = \cdot$ and $G = F$ and $\beta = \alpha$.

$$\boxed{\Xi; \Theta \vdash \{v : G[\mu F] \mid \beta\,(G\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq P} \quad \begin{array}{l}\text{Abbreviated}\\ \Xi;\Theta \vdash \mathsf{unroll}_{F;\alpha}(G;\beta;\tau;t) \doteq P\end{array}$$

$$\frac{\begin{array}{ll}\beta \circ \mathsf{inj}_1 \doteq \beta_1 & \Xi;\Theta \vdash \{v : G[\mu F] \mid \beta_1\,(G\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq P\\ \beta \circ \mathsf{inj}_2 \doteq \beta_2 & \Xi;\Theta \vdash \{v : H[\mu F] \mid \beta_2\,(H\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq Q\end{array}}{\Xi;\Theta \vdash \{v : (G \oplus H)[\mu F] \mid \beta\,((G \oplus H)\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq P + Q}\ \text{Unroll}\oplus$$

$$\frac{\Xi, a : \tau; \Theta, a : \tau \vdash \left\{v : \hat{P}[\mu F] \mid (q \Rightarrow t')\left(\hat{P}\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\} \doteq Q}{\begin{array}{l}\Xi;\Theta \vdash \left\{v : (\mathrm{Id} \otimes \hat{P})[\mu F] \mid ((a, q) \Rightarrow t')\left((\mathrm{Id} \otimes \hat{P})\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\}\\ \qquad \doteq \exists a : \tau.\ \{v : \mu F \mid (\mathsf{fold}_F\ \alpha)\ v =_\tau a\} \times Q\end{array}}\ \text{UnrollId}$$

$$\frac{\Xi, a : \tau'; \Theta, a : \tau' \vdash \left\{v : (\underline{Q} \otimes \hat{P})[\mu F] \mid ((o, q) \Rightarrow t')\left((\underline{Q} \otimes \hat{P})\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\} \doteq Q'}{\begin{array}{l}\Xi;\Theta \vdash \left\{v : (\underline{\exists a : \tau'.\ Q} \otimes \hat{P})[\mu F] \mid ((\mathsf{pack}(a, o), q) \Rightarrow t')\left((\underline{\exists a : \tau'.\ Q} \otimes \hat{P})\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\}\\ \quad \doteq \exists a : \tau'.\ Q'\end{array}}\ \text{Unroll}\exists$$

$$\frac{\Xi;\Theta \vdash \left\{v : \hat{P}[\mu F] \mid (q \Rightarrow t')\left(\hat{P}\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\} \doteq Q'}{\Xi;\Theta \vdash \left\{v : (\underline{Q} \otimes \hat{P})[\mu F] \mid ((\_, q) \Rightarrow t')\left((\underline{Q} \otimes \hat{P})\ (\mathsf{fold}_F\ \alpha)\ v\right) =_\tau t\right\} \doteq Q \times Q'}\ \text{UnrollConst}$$

$$\frac{}{\Xi;\Theta \vdash \{v : I[\mu F] \mid (() \Rightarrow t')\,(I\ (\mathsf{fold}_F\ \alpha)\ v) =_\tau t\} \doteq 1 \wedge (t = t')}\ \text{Unroll}I$$

Fig. 8. Unrolling

Unroll$\oplus$ unrolls each branch and then sums the resulting types. UnrollId outputs the product of the original inductive type but with a measurement given by the recursive result of the fold (over which we existentially quantify), together with the rest of the unrolling. Unroll$\exists$ pushes the packed index variable a onto the context and continues unrolling, existentially quantifying over the result; in the conclusion, a is simultaneously bound in Q and t'. UnrollConst outputs a product of the type of the constant functor and the rest of the unrolling. Unroll$I$ simply outputs the unit type together with the index term equality given by the (unrolled) measurement.

If our functor and algebra grammars were instead more direct, like those implicitly used in the introduction (Sec. 1) and overview (Sec. 2), and explicitly discussed in Sec. 4.0.1, then we would have to modify the unrolling judgment, and it would need two more rules. We expect everything would still work, but we prefer having to consider fewer rules when proving metatheory.

Unrolling, equivalence and subtyping. Substituting judgmentally equivalent types, functors and indexes for the inputs of unrolling generates an output type that is both a subtype and supertype of the original unrolling output:

Lemma 4.1 (Unroll to Mutual Subtypes). *(Lemma B.96 in appendix)*
If $\Xi;\Theta \vdash \{\nu : G[\mu F] \mid \beta\,(G\ (\mathsf{fold}_F\ \alpha)\ \nu) =_\tau t\} \doteq P$
and $\Theta \vdash G \equiv G'$ *and* $\Theta \vdash F \equiv F'$ *and* $\Theta \vdash t = t'$ true,
then there exists Q *such that* $\Xi;\Theta \vdash \{\nu : G'[\mu F'] \mid \beta\,(G'\ (\mathsf{fold}_{F'}\ \alpha)\ \nu) =_\tau t'\} \doteq Q$
and $\Theta \vdash P \leq^+ Q$ *and* $\Theta \vdash Q \leq^+ P$.

We use this to prove subsumption admissibility (see Sec. 4.8) for the cases that involve constructing and pattern matching inductive values.

4.7 Typing

Declarative bidirectional typing rules are given in Figures 9, 10, and 11. By careful design, guided by logical principles, all typing rules are syntax-directed. That is, when deriving a conclusion, at most one rule is compatible with the syntax of the input program term and the principal input type.

To manage the interaction between subtyping and program typing, types in a well-formed (under Θ) program context Γ must be invariant under extraction: for all $(x : P) \in \Gamma$, we have $\Theta \vdash P \rightsquigarrow^+ P\ [\cdot]$ (that is, $\Theta \vdash P \not\rightsquigarrow$). We maintain this invariant in program typing by extracting before adding any type declaration to the context.

$\boxed{\Theta;\Gamma \vdash h \Rightarrow P}$ Under Θ and Γ, head h synthesizes type P

$$\frac{(x : P) \in \Gamma}{\Theta;\Gamma \vdash x \Rightarrow P}\ \text{Decl}{\Rightarrow}\text{Var} \qquad \frac{\Theta \vdash P\ \mathsf{type}[\Xi] \qquad \Theta;\Gamma \vdash v \Leftarrow P}{\Theta;\Gamma \vdash (v : P) \Rightarrow P}\ \text{Decl}{\Rightarrow}\text{ValAnnot}$$

$\boxed{\Theta;\Gamma \vdash g \Rightarrow \uparrow P}$ Under Θ and Γ, bound expression g synthesizes type $\uparrow P$

$$\frac{\Theta;\Gamma \vdash h \Rightarrow \downarrow N \qquad \Theta;\Gamma;[N] \vdash s \gg \uparrow P}{\Theta;\Gamma \vdash h(s) \Rightarrow \uparrow P}\ \text{Decl}{\Rightarrow}\text{App}$$

$$\frac{\Theta \vdash P\ \mathsf{type}[\Xi] \qquad \Theta;\Gamma \vdash e \Leftarrow \uparrow P}{\Theta;\Gamma \vdash (e : \uparrow P) \Rightarrow \uparrow P}\ \text{Decl}{\Rightarrow}\text{ExpAnnot}$$

Fig. 9. Declarative head and bound expression type synthesis

The judgment $\Theta;\Gamma \vdash h \Rightarrow P$ (Fig. 9) synthesizes the type P from the head h. This judgment is synthesizing, because it is used in what are, from a Curry–Howard perspective, kinds of cut rules: Decl$\Rightarrow$App and Decl$\Leftarrow$`match`, discussed later. The synthesized type is the cut type, which does not appear in the conclusion of Decl$\Rightarrow$App or Decl$\Leftarrow$`match`. For head variables, we look up the

$\boxed{\Theta; \Gamma \vdash v \Leftarrow P}$ Under Θ and Γ, value v checks against type P

$$\frac{P \neq \exists, \wedge \qquad (x : Q) \in \Gamma \qquad \Theta \vdash Q \leq^+ P}{\Theta; \Gamma \vdash x \Leftarrow P}\ \text{Decl}{\Leftarrow}\text{Var} \qquad \frac{}{\Theta; \Gamma \vdash \langle\rangle \Leftarrow 1}\ \text{Decl}{\Leftarrow}1$$

$$\frac{\Theta; \Gamma \vdash v_1 \Leftarrow P_1 \qquad \Theta; \Gamma \vdash v_2 \Leftarrow P_2}{\Theta; \Gamma \vdash \langle v_1, v_2 \rangle \Leftarrow P_1 \times P_2}\ \text{Decl}{\Leftarrow}{\times} \qquad \frac{\Theta; \Gamma \vdash v \Leftarrow P_k}{\Theta; \Gamma \vdash \mathsf{inj}_k\, v \Leftarrow P_1 + P_2}\ \text{Decl}{\Leftarrow}{+_k}$$

$$\frac{\Theta; \Gamma \vdash v \Leftarrow [t/a]P \qquad \Theta \vdash t : \tau}{\Theta; \Gamma \vdash v \Leftarrow (\exists a : \tau.\, P)}\ \text{Decl}{\Leftarrow}\exists \qquad \frac{\Theta; \Gamma \vdash v \Leftarrow P \qquad \Theta \vdash \phi\ \mathsf{true}}{\Theta; \Gamma \vdash v \Leftarrow P \wedge \phi}\ \text{Decl}{\Leftarrow}\wedge$$

$$\frac{\cdot; \Theta \vdash \{\nu : F[\mu F] \mid \alpha\,(F\,(\mathsf{fold}_F\,\alpha)\,\nu) =_\tau t\} \doteq P \qquad \Theta; \Gamma \vdash v \Leftarrow P}{\Theta; \Gamma \vdash \mathsf{into}(v) \Leftarrow \{\nu : \mu F \mid (\mathsf{fold}_F\,\alpha)\,\nu =_\tau t\}}\ \text{Decl}{\Leftarrow}\mu$$

$$\frac{\Theta; \Gamma \vdash e \Leftarrow N}{\Theta; \Gamma \vdash \{e\} \Leftarrow {\downarrow} N}\ \text{Decl}{\Leftarrow}{\downarrow}$$

$\boxed{\Theta; \Gamma \vdash e \Leftarrow N}$ Under Θ and Γ, expression e checks against type N

$$\frac{\Theta; \Gamma \vdash v \Leftarrow P}{\Theta; \Gamma \vdash \mathsf{return}\, v \Leftarrow {\uparrow} P}\ \text{Decl}{\Leftarrow}{\uparrow}$$

$$\frac{\Theta \vdash N \not\rightsquigarrow \qquad \Theta; \Gamma \vdash g \Rightarrow {\uparrow} P \qquad \Theta \vdash P \rightsquigarrow^+ P'\ [\Theta_P] \qquad \Theta, \Theta_P; \Gamma, x : P' \vdash e \Leftarrow N}{\Theta; \Gamma \vdash \mathsf{let}\ x = g;\ e \Leftarrow N}\ \text{Decl}{\Leftarrow}\mathsf{let}$$

$$\frac{\Theta \vdash N \not\rightsquigarrow \qquad \Theta; \Gamma \vdash h \Rightarrow P \qquad \Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}{\Theta; \Gamma \vdash \mathsf{match}\ h\ \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}\ \text{Decl}{\Leftarrow}\mathsf{match}$$

$$\frac{\Theta \vdash P \to N \not\rightsquigarrow \qquad \Theta; \Gamma, x : P \vdash e \Leftarrow N}{\Theta; \Gamma \vdash \lambda x.\, e \Leftarrow P \to N}\ \text{Decl}{\Leftarrow}\lambda$$

$$\frac{\begin{array}{l}\Theta \vdash N \not\rightsquigarrow \\ \Theta \vdash \forall a : \mathbb{N}.\, M \leq^- N \qquad \Theta, a : \mathbb{N}; \Gamma, x : {\downarrow}\big(\forall a' : \mathbb{N}.\, (a' < a) \supset [a'/a]M\big) \vdash e \Leftarrow M\end{array}}{\Theta; \Gamma \vdash \mathsf{rec}\ (x : \forall a : \mathbb{N}.\, M).\ e \Leftarrow N}\ \text{Decl}{\Leftarrow}\mathsf{rec}$$

$$\frac{\Theta \vdash N \rightsquigarrow N'\ [\Theta_N] \qquad \Theta_N \neq \cdot \qquad \Theta, \Theta_N; \Gamma \vdash e \Leftarrow N'}{\Theta; \Gamma \vdash e \Leftarrow N}\ \text{Decl}{\Leftarrow}{\rightsquigarrow}$$

Fig. 10. Declarative value and expression type checking

variable's type in the context Γ (Decl$\Rightarrow$Var). For annotated values, we synthesize the annotation (Decl$\Rightarrow$ValAnnot).

The judgment $\Theta; \Gamma \vdash g \Rightarrow {\uparrow} P$ (Fig. 9) synthesizes the type ${\uparrow} P$ from the bound expression g. Similarly to the synthesizing judgment for heads, this judgment is synthesizing because it is used in a cut rule Decl$\Leftarrow$let (the synthesized type is again the cut type). Bound expressions only synthesize an upshift because of their (lone) role in rule Decl$\Leftarrow$let, discussed later. For an application of a head to a spine (Decl$\Rightarrow$App, an auxiliary cut rule), we first synthesize the head's type (which must be a downshift), and then check the spine against the thunked computation type, synthesizing the

$\boxed{\Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}$ Under Θ and Γ, patterns r_i match against (input) type P and branch expressions e_i check against type N

$$\frac{\Theta, a:\tau; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}{\Theta; \Gamma; [\exists a:\tau.\, P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}\ \text{DeclMatch}\exists$$

$$\frac{\Theta, \phi; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}{\Theta; \Gamma; [P \land \phi] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N}\ \text{DeclMatch}\land \qquad \frac{\Theta; \Gamma \vdash e \Leftarrow N}{\Theta; \Gamma; [1] \vdash \{\langle\rangle \Rightarrow e\} \Leftarrow N}\ \text{DeclMatch1}$$

$$\frac{\begin{array}{l}\Theta \vdash P_1 \rightsquigarrow P_1' \ [\Theta_1] \\ \Theta \vdash P_2 \rightsquigarrow P_2' \ [\Theta_2] \qquad \Theta, \Theta_1, \Theta_2; \Gamma, x_1 : P_1', x_2 : P_2' \vdash e \Leftarrow N\end{array}}{\Theta; \Gamma; [P_1 \times P_2] \vdash \{\langle x_1, x_2\rangle \Rightarrow e\} \Leftarrow N}\ \text{DeclMatch}\times$$

$$\frac{\begin{array}{ll}\Theta \vdash P_1 \rightsquigarrow P_1' \ [\Theta_1] & \Theta, \Theta_1; \Gamma, x_1 : P_1' \vdash e_1 \Leftarrow N \\ \Theta \vdash P_2 \rightsquigarrow P_2' \ [\Theta_2] & \Theta, \Theta_2; \Gamma, x_2 : P_2' \vdash e_2 \Leftarrow N\end{array}}{\Theta; \Gamma; [P_1 + P_2] \vdash \left\{\mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\right\} \Leftarrow N}\ \text{DeclMatch+}$$

$$\frac{}{\Theta; \Gamma; [0] \vdash \{\} \Leftarrow N}\ \text{DeclMatch0}$$

$$\frac{\begin{array}{c}\cdot; \Theta \vdash \{\nu : F[\mu F] \mid \alpha\,(F\,(\mathsf{fold}_F\,\alpha)\,\nu) =_\tau t\} \doteq Q \\ \Theta \vdash Q \rightsquigarrow Q' \ [\Theta_Q] \qquad \Theta, \Theta_Q; \Gamma, x : Q' \vdash e \Leftarrow N\end{array}}{\Theta; \Gamma; [\{\nu : \mu F \mid (\mathsf{fold}_F\,\alpha)\,\nu =_\tau t\}] \vdash \{\mathsf{into}(x) \Rightarrow e\} \Leftarrow N}\ \text{DeclMatch}\mu$$

$\boxed{\Theta; \Gamma; [N] \vdash s \gg \uparrow P}$ Under Θ and Γ, if a head of type $\downarrow N$ is applied to the spine s, then it will return a result of type $\uparrow P$

$$\frac{\Theta \vdash t : \tau \qquad \Theta; \Gamma; [[t/a]N] \vdash s \gg \uparrow P}{\Theta; \Gamma; [\forall a:\tau.\, N] \vdash s \gg \uparrow P}\ \text{DeclSpine}\forall$$

$$\frac{\Theta \vdash \phi \ \mathsf{true} \qquad \Theta; \Gamma; [N] \vdash s \gg \uparrow P}{\Theta; \Gamma; [\phi \supset N] \vdash s \gg \uparrow P}\ \text{DeclSpine}\supset$$

$$\frac{\Theta; \Gamma \vdash v \Leftarrow Q \qquad \Theta; \Gamma; [N] \vdash s \gg \uparrow P}{\Theta; \Gamma; [Q \to N] \vdash v, s \gg \uparrow P}\ \text{DeclSpineApp} \qquad \frac{}{\Theta; \Gamma; [\uparrow P] \vdash \cdot \gg \uparrow P}\ \text{DeclSpineNil}$$

Fig. 11. Declarative pattern matching and spine typing

latter's return type. (Function applications must always be fully applied, but we can simulate partial application via η-expansion. For example, given $x : P$ and $h \Rightarrow \downarrow(P_1 \to P_2 \to \uparrow Q)$, to partially apply h to x we can write $\lambda y.\, \mathsf{let}\ z = h(x, y); \cdots$.) For annotated expressions, we synthesize the annotation (Decl$\Rightarrow$ExpAnnot), which must be an upshift. If the programmer wants, say, to verify guard constraints in N of an expression e of type N whenever it is run, then they must annotate it: $(\mathsf{return}\,\{e\} : \uparrow\downarrow N)$. If an e of type N is intended to be a function to be applied (as a head to a spine; Decl$\Rightarrow$App) only if the guards of N can be verified and the universally quantified indexes of N can be instantiated, then the programmer must thunk and annotate it: $(\{e\} : \downarrow N)$. The two annotation rules have explicit type well-formedness premises to emphasize that type annotations are provided by the programmer.

The judgment $\Theta; \Gamma \vdash v \Leftarrow P$ (Fig. 10) checks the value v against the type P. From a Curry–Howard perspective, this judgment corresponds to a *right-focusing* phase. According to rule Decl$\Leftarrow\exists$, a value checks against an existential type if there is an index instantiation it checks against (declaratively, an index is conjured, but algorithmically we will have to solve for one). For example, as discussed in Sec. 4.2, checking the program value one representing 1 against type $\exists a : \mathbb{N}.\, \mathsf{Nat}(a)$ solves a to an index semantically equal to 1. According to rule Decl$\Leftarrow\wedge$, a value checks against an asserting type if it the asserted proposition ϕ holds (and the value checks against the type to which ϕ is connected). Instead of a general value type subsumption rule like

$$\frac{\Theta; \Gamma \vdash v \Leftarrow Q \qquad \Theta \vdash Q \leq^{+} P}{\Theta; \Gamma \vdash v \Leftarrow P}$$

we restrict subsumption to (value) variables, and prove that subsumption is admissible (see Section 4.8). This is easier to implement efficiently because the type checker would otherwise have to guess Q (and possibly need to backtrack), whereas Decl$\Leftarrow$Var need only look up the variable. Further, the $P \neq \exists, \wedge$ constraint on Decl$\Leftarrow$Var means that any top-level $\exists$ or $\wedge$ constraints must be verified before subtyping, eliminating nondeterminism of verifying these in subtyping or typing. Rule Decl$\Leftarrow\mu$ checks the unrolled value against the unrolled inductive type. Rule Decl$\Leftarrow$1 says $\langle\rangle$ checks against 1. Rule Decl$\Leftarrow\times$ says a pair checks against a product if each pair component checks against its corresponding component type. Rule Decl$\Leftarrow+_k$ say a k-injected value checks against a sum if it can be checked against the sum's kth component. Rule Decl$\Leftarrow\downarrow$ checks the thunked expression against the computation type N under the given thunk type $\downarrow N$.

The judgment $\Theta; \Gamma \vdash e \Leftarrow N$ (Fig. 10) checks the expression e against the type N. From a Curry–Howard perspective, this judgment is a *right-inversion* phase with *stable* moments (Decl$\Leftarrow$`let` and Decl$\Leftarrow$`match`, which enter left- or right-focusing phases, respectively). Instead of Decl$\Leftarrow\rightsquigarrow$, one might expect two rules (one for $\forall$ and one for $\supset$) that simply put the universal index variable or proposition into logical context, but these alone are less compatible with subsumption admissibility (see Sec. 4.8) due to the use of extraction in subtyping rule $\leq^{-}\rightsquigarrow$R. However, the idea is still the same: here we are using indexes, not verifying them as in the dual left-focusing phase. To reduce Decl$\Leftarrow\rightsquigarrow$ nondeterminism, and to enable a formal correspondence between our system and (a variant of) CBPV (which has a general $\downarrow$ elimination rule), the other (expression) rules must check against a type N that's invariant under extraction: $\Theta \vdash N \not\rightsquigarrow$. In practice, we eagerly apply (if possible) Decl$\Leftarrow\rightsquigarrow$ immediately when type checking an expression; extracted types are invariant under extraction.

All applications $h(s)$ must be named and sequenced via Decl$\Leftarrow$`let`, which we may think of as monadic binding, and is a key cut rule. Other computations—annotated returner expressions $(e : \uparrow P)$—must also be named and sequenced via Decl$\Leftarrow$`let`. It would not make sense to allow arbitrary negative annotations because that would require verifying constraints and instantiating indexes that should only be done when the annotated expression is applied, which does not occur in Decl$\Leftarrow$`let` itself.

Heads, that is, head variables and annotated values, can be pattern matched via Decl$\Leftarrow$`match`. From a Curry–Howard perspective, the rule Decl$\Leftarrow$`match` is a cut rule dual to the cut rule Decl$\Leftarrow$`let`: the latter binds the result of a computation to a (sequenced) computation, whereas the former binds the deconstruction of a value to, and directs control flow of, a computation. Rule Decl$\Leftarrow\lambda$ is standard (besides the check that $P \rightarrow N$ is simple). Rule Decl$\Leftarrow$`rec` requires an annotation that universally quantifies over the argument that must be smaller at each recursive call, as dictated by its annotation in the last premise, ensuring that *refined* recursive functions are well-founded. Rule

Decl$\Leftarrow\uparrow$ checks that the value being returned has the positive type under the given returner type ($\uparrow$); this may be thought of as a monadic return operation.

The judgment $\Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$ (Fig. 11) decomposes P, according to patterns r_i (if $P \neq \wedge$ or $\exists$, which have no computational content; if $P = \wedge$ or $\exists$, the index is put in logical context for use), and checks that each branch e_i has type N. The rules are straightforward. Indexes from matching on existential and asserting types are used, not verified (as in value typechecking); we deconstruct heads, and to synthesize a type for a head, its indexes must hold, so within the pattern matching phase itself, we may assume and use them. From a Curry–Howard perspective, this judgment corresponds to a *left-inversion* phase. However, it is not *strongly* focused, that is, it does not decompose P eagerly and as far as possible; therefore, "phase" might be slightly misleading. If our system were more strongly focused, we would have nested patterns, at least for all positive types except inductive types; it's unclear how strong focusing on inductive types would work.

The judgment $\Theta; \Gamma; [N] \vdash s \gg \uparrow P$ (Fig. 11) checks the spine s against N, synthesizing the return type $\uparrow P$. From a Curry–Howard perspective, this judgment corresponds to a *left-focusing* phase. The rules are straightforward: decompose the given N, checking index constraints (DeclSpine$\forall$ and DeclSpine$\supset$) and values (DeclSpineApp) until an upshift, the return type, is synthesized (DeclSpineNil). Similarly to dual rule Decl$\Leftarrow\exists$, the declarative rule DeclSpine$\forall$ conjures an index measuring a value, but in this case an argument value in a spine. For example, in applying a head of type $\forall a : \mathbb{N}.\ \mathsf{Nat}(a) \to \uparrow \mathsf{Nat}(a)$ to the spine with program value one representing 1, we must instantiate a to an index semantically equal to 1; we show how this works algorithmically later in Sec. 6.4. All universal quantifiers (in the input type of a spine judgment) are solvable algorithmically, because in a well-formed return type, the set of value-determined indexes Ξ is empty.

4.8 Substitution

We now extend the index-level syntactic substitutions (and the sequential substitution operation) introduced in Sec. 4.1. A *syntactic substitution* $\sigma ::= \cdot \mid \sigma, t/a \mid \sigma, v : P/x$ is essentially a list of terms to be substituted for variables. Substitution application $[\sigma]-$ is a sequential substitution metaoperation on types and terms. On program terms, it is a kind of *hereditary substitution*[5] [Watkins et al. 2004; Pfenning 2008] in the sense that, at head variables (note the h superscript in the Fig. 12 definition; we elide h when clear from context), an annotation is produced if the value and the head variable being replaced by it are not equal—*thereby modifying the syntax tree of the substitutee.* Otherwise, substitution is standard (homomorphic application) and does not use the value's associated type given in σ: see Fig. 12.

In the definition given in Fig. 12, an annotation is not produced if $v = x$ so that $x : P/x$ is always an *identity* substitution: that is, $[x : P/x]^{\mathsf{h}} x = x$. As usual, we assume variables are α-renamed to avoid capture by substitution.

The judgment $\Theta_0; \Gamma_0 \vdash \sigma : \Theta; \Gamma$ (appendix Fig. 9) means that, under Θ_0 and Γ_0, we know σ is a substitution of index terms and program values for variables in Θ and Γ, respectively. The key rule of this judgment is for program value entries (the three elided rules are similar to the three rules for syntactic substitution typing at index level, found near the start of Sec. 4.1, but adds program contexts Γ where appropriate):

$$\frac{\Theta_0; \Gamma_0 \vdash \sigma : \Theta; \Gamma \qquad \Theta_0; \Gamma_0 \vdash [\sigma]v \Leftarrow [\lfloor \sigma \rfloor]P \qquad x \notin dom(\Gamma)}{\Theta_0; \Gamma_0 \vdash (\sigma, v : P/x) : \Theta; \Gamma, x : P}$$

We apply the rest of the syntactic substitution—that is, the σ in the rule—to v and P because the substitution operation is sequential; v may mention variables in Γ and Θ, and P may mention

[5]Typically, hereditary substitution reduces terms after substitution, modifying the syntax tree.

$$
\begin{aligned}
[v : P/x]^{\mathsf{h}} y &= y \quad (\text{if } y \neq x) \\
[v : P/x]^{\mathsf{h}} x &= \begin{cases} x & \text{if } v = x \\ (v : P) & \text{else} \end{cases} \\
[v : P/x]^{\mathsf{h}} (v_0 : P_0) &= ([v : P/x] v_0 : P_0)
\end{aligned}
$$

$$
\begin{aligned}
[v : P/x](h(s)) &= ([v : P/x]^{\mathsf{h}} h)([v : P/x] s) \\
[v : P/x](e : \uparrow Q) &= ([v : P/x] e : \uparrow Q) \\
[v : P/x] y &= y \quad (\text{if } y \neq x) \\
[v : P/x] x &= v \\
[v : P/x]\langle v_1, v_2 \rangle &= \langle [v : P/x] v_1, [v : P/x] v_2 \rangle \\
&\vdots \\
[v : P/x](\mathtt{match}\ h\ \{r_i \Rightarrow e_i\}_{i \in I}) &= \mathtt{match}\ \left([v : P/x]^{\mathsf{h}} h\right) ([v : P/x]\{r_i \Rightarrow e_i\}_{i \in I}) \\
&\vdots
\end{aligned}
$$

Fig. 12. Definition of syntactic substitution on program terms

variables in Θ. The metaoperation $\lfloor - \rfloor$ filters out program variable entries (program variables cannot appear in types, functors, algebras or indexes).

That substitution respects typing is an important correctness property of the type system. We state only two parts now, but those of the remaining program typing judgments are similar; all six parts are mutually recursive.

Lemma 4.2 (Syntactic Substitution). *(Lemma B.106 in appendix)*
Assume $\Theta_0; \Gamma_0 \vdash \sigma : \Theta; \Gamma$.

(1) If $\Theta; \Gamma \vdash h \Rightarrow P$*, then there exists* Q *such that* $\Theta_0 \vdash Q \leq^+ [\lfloor \sigma \rfloor] P$ *and* $\Theta_0; \Gamma_0 \vdash [\sigma] h \Rightarrow Q$.
(2) If $\Theta; \Gamma \vdash e \Leftarrow N$*, then* $\Theta_0; \Gamma_0 \vdash [\sigma] e \Leftarrow [\lfloor \sigma \rfloor] N$.

In part (1), if substitution creates a head variable with stronger type, then the stronger type is synthesized. The proof relies on other structural properties such as weakening. It also relies on subsumption admissibility, which captures what we mean here by "stronger type". We show only one part; the mutually recursive parts for the other five program typing judgments are similar.

Lemma 4.3 (Subsumption Admissibility). *(Lemma B.105 in appendix)*
Assume $\Theta \vdash \Gamma' \leq \Gamma$ *(pointwise subtyping).*

(1) If $\Theta; \Gamma \vdash v \Leftarrow P$ *and* $\Theta \vdash P \leq^+ Q$*, then* $\Theta; \Gamma' \vdash v \Leftarrow Q$.

Subtypes are stronger than supertypes. That is, if we can check a value against a type, then we know that it also checks against any of the type's supertypes; similarly for expressions. Pattern matching is similar, but it also says we can match on a stronger type. A head or bound expression can synthesize a stronger type under a stronger context. Similarly, with a stronger input type, a spine can synthesize a stronger return type.

5 TYPE SOUNDNESS

We prove type (and substitution) soundness of the declarative system with respect to an elementary domain-theoretic denotational semantics. Refined type soundness implies the refined system's totality and logical consistency.

The semantics of our refined system is defined in terms of that of its underlying, unrefined system, which we discuss next (after a brief note on notation).

Notation: We define the disjoint union $X \uplus Y$ of sets X and Y by $X \uplus Y = (\{1\} \times X) \cup (\{2\} \times Y)$ and define $inj_k : X_k \to X_1 \uplus X_2$ by $inj_k(d) = (k, d)$. Semantic values are usually named d, f, g, or V.

5.1 Unrefined System

For space reasons, we do not fully present the unrefined system and its semantics here (see appendix Sec. A.4). The unrefined system is basically just the refined system with indexes erased. The unrefined system satisfies a substitution lemma (appendix Lemma C.1) similar to that of the refined system, but its proof is simpler and does not rely on subsumption admissibility, because the unrefined system has no subtyping.

In CBPV, nontermination is regarded as an effect, so value and computation types denote different kinds of mathematical things: predomains and domains, respectively [Levy 2004], which are both sets with some structure. Therefore, because we have recursive expressions, we must model nontermination, an effect. We use elementary domain theory. For our (unrefined) system, we interpret (unrefined) positive types as predomains and (unrefined) negative types as domains. The only effect we consider in this paper is nontermination; we take (chain-)complete partial orders (cpo) as predomains, and pointed (chain-)complete partial orders (cppo) as domains.

Positive types and functors. The grammar for unrefined positive types is similar to that for refined positive types, but lacks asserting and existential types, and unrefined inductive types μF are not refined by predicates. Unrefined inductive types use the unrefined functor grammar, which is the same as the refined functor grammar but uses unrefined types in constant functors.

$$P, Q \quad ::= \; 1 \mid P \times Q \mid 0 \mid P + Q \mid \downarrow N \mid \mu F$$

The denotations of unrefined positive types are standard. We briefly describe their partial orders, then describe the meaning of functors, and lastly return to the meaning of inductive types (which involve functors).

We give (the denotation of) 1 (denoting a terminal object) the discrete order $\{(\bullet, \bullet)\}$. For $P \times Q$ (denoting product) we use component-wise order, for 0 (denoting an initial object) we use empty order, and for $P + Q$ (denoting coproduct, that is, disjoint union $\uplus$) we use injection-wise order. We give $\downarrow N$ the order of N, that is, $\downarrow$ denotes the forgetful functor from the category **Cppo** of cppos and continuous functions to the category **Cpo** of cpos and continuous functions. Finally, $V_1 \sqsubseteq_{[\![\mu F]\!]} V_2$ if $V_1 \sqsubseteq_{[\![F]\!]^{k+1}\emptyset} V_2$ for some $k \in \mathbb{N}$, inheriting the type denotation orders as the functor is applied.

The denotations of unrefined functors are standard **Cpo** endofunctors. We briefly describe them here, but full definitions are in appendix Sec. A.4. The sum functor $\oplus$ denotes a functor that sends a cpo to the disjoint union $\uplus$ of its component applications (with usual injection-wise order), and its functorial action is injection-wise. The product functor $\otimes$ denotes a functor that sends a cpo to the product $\times$ of its component applications (with usual component-wise order), and its functorial action is component-wise. The unit functor I denotes a functor sending any cpo to $1 = \{\bullet\}$ (discrete order), and its functorial action sends all morphisms to $id_{\{\bullet\}}$. The constant (type) functor $\underline{P}$ denotes a functor sending any cpo to the cpo $[\![P]\!]$, and its functorial action sends all morphisms to the identity $id_{[\![P]\!]}$ on $[\![P]\!]$. The identity functor Id denotes the identity endofunctor on **Cpo**. (Forgetting the order structure, functors also denote endofunctors on the category **Set** of sets and functions.)

We now explain the denotational semantics of our inductive types. Semantically, we build an inductive type (such as $[\![\mathsf{List}\ A]\!]$), by repeatedly applying (the denotation of) its functor specification (such as $[\![\mathsf{ListF}_A]\!]$) to the initial object $[\![0]\!] = \emptyset$. For example,

$$[\![\mathsf{List}\ A]\!] = \bigcup_{k \in \mathbb{N}} [\![\underline{1} \oplus \mathrm{Id}]\!]^k\ \emptyset = 1 \uplus \left([\![A]\!] \times \left(1 \uplus \left([\![A]\!] \times \cdots \right) \right) \right)$$

where $1 = \{\bullet\}$ (using the relatively direct functors with more complicated unrolling, discussed in Sec. 4.0.1). We denote the nil list [] by $inj_1\ \bullet$, a list $x :: []$ with one term x by $inj_2([\![x]\!], inj_1\ \bullet)$, and so on. In general, given a (polynomial) **Set** (category of sets and functions) endofunctor F (which, for this paper, will always be the denotation of a well-formed (syntactic) functor, refined or otherwise), we define $\mu F = \cup_{k \in \mathbb{N}} F^k \emptyset$. We then define $[\![\mu F]\!] = \mu\,[\![F]\!]$. In our system, for every well-formed (unrefined) functor F, the set $\mu\,[\![F]\!]$ is a *fixed point* of $[\![F]\!]$ (appendix Lemma C.7): that is, $[\![F]\!]\ (\mu\,[\![F]\!]) = \mu\,[\![F]\!]$ (and similarly for refined functors: appendix Lemma D.11).

Negative types. The grammar for unrefined negative types has unrefined function types $P \to N$ and unrefined upshifts $\uparrow P$, with no guarded or universal types. Unrefined negative types denote cppos.

$$N \quad ::= \ P \to N \mid \uparrow P$$

Function types $P \to N$ denote continuous functions from $[\![P]\!]$ to $[\![N]\!]$ (which we sometimes write as $[\![P]\!] \Rightarrow [\![N]\!]$), where its order is defined pointwise, together with the bottom element (the "point" of "*pointed* cpo") $\bot_{[\![P \to N]\!]}$ that maps every $V \in [\![P]\!]$ to the bottom element $\bot_{[\![N]\!]}$ of $[\![N]\!]$ (that is, $\uparrow$ denotes the lift functor from **Cpo** to **Cppo**). For our purposes, this is equivalent to lifting $[\![P]\!] \in$ **Cpo** to **Cppo** and denoting arrow types by *strict* ($\bot$ goes to $\bot$) continuous functions so that function types denote **Cppo** *exponentials*.

Upshifts $\uparrow P$ denote $[\![P]\!] \uplus \{\bot_\uparrow\}$ with the lift order

$$\sqsubseteq_{[\![\uparrow P]\!]} = \left\{ (inj_1\ d, inj_1\ d') \;\middle|\; d \sqsubseteq_{[\![P]\!]} d' \right\} \cup \left\{ (inj_2\ \bot_\uparrow, d) \;\middle|\; d \in [\![\uparrow P]\!] \right\}$$

and bottom element $\bot_{[\![\uparrow P]\!]} = inj_2\ \bot_\uparrow$. We could put, say, $\bullet$ rather than $\bot_\uparrow$, but we think the latter is clearer in associating it with the *bottom* element of upshifts; or $\bot$ rather than $\bot_\uparrow$ but we often elide the "$[\![A]\!]$" subscript in $\bot_{[\![A]\!]}$ when clear from context.

Appendix Fig. 27 has the full definition of (unrefined) type and functor denotations.

Well-typed program terms. We write $\Gamma \vdash \mathcal{O} \cdots A$ and $\Gamma; [B] \vdash \mathcal{O} \cdots A$ to stand for all six unrefined program typing judgments: $\Gamma \vdash h \Rightarrow P$ and $\Gamma \vdash g \Rightarrow \uparrow P$ and $\Gamma \vdash v \Leftarrow P$ and $\Gamma \vdash e \Leftarrow N$ and $\Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$ and $\Gamma; [N] \vdash s \gg \uparrow P$.

The denotational semantics of well-typed, unrefined program terms of judgmental form $\Gamma \vdash \mathcal{O} \cdots A$ or $\Gamma; [B] \vdash \mathcal{O} \cdots A$ are continuous functions $[\![\Gamma]\!] \to [\![A]\!]$ and $[\![\Gamma]\!] \to [\![B]\!] \to [\![A]\!]$ respectively, where $[\![\Gamma]\!]$ is the set of all semantic substitutions $\vdash \delta : \Gamma$ together with component-wise order. Similarly to function type denotations, the bottom element of a $[\![\Gamma]\!] \to [\![N]\!]$ sends every $\delta \in [\![\Gamma]\!]$ to $\bot_{[\![N]\!]}$ (equivalently for our purposes, we can lift source predomains and consider strict continuous functions). We only interpret typing derivations, but we often only mention the program term in semantic brackets $[\![-]\!]$. For example, if $\Gamma \vdash x \Rightarrow P$, then $[\![x]\!] = (\delta \in [\![\Gamma]\!]) \mapsto \delta(x)$. We write the application of the denotation $[\![E]\!]$ of a program term E (typed under Γ) to a semantic substitution $\delta \in [\![\Gamma]\!]$ as $[\![E]\!]_\delta$. We only mention a few of the more interesting cases of the definition of typing denotations; for the full definition, see appendix Figures 28, 29, and 30. If $\Gamma; [N] \vdash v, s \gg M$, then

$$[\![v, s]\!] = (\delta \in [\![\Gamma]\!]) \mapsto (f \mapsto [\![s]\!]_\delta\,(f([\![v]\!]_\delta)))$$

Returner expressions denote monadic returns:

$$[\![\mathtt{return}\, v]\!]_\delta = inj_1\ [\![v]\!]_\delta$$

Let-binding denotes monadic binding:

$$[\![\mathtt{let}\ x = g;\ e]\!]_\delta = \begin{cases} [\![e]\!]_{(\delta, V/x)} & \text{if } [\![g]\!]_\delta = inj_1\, V \\ \bot_{[\![N]\!]} & \text{if } [\![g]\!]_\delta = inj_2\, \bot_\uparrow \end{cases}$$

A recursive expression denotes a fixed point obtained by taking the least upper bound ($\sqcup$) of all its successive approximations:

$$[\![\Gamma \vdash \mathtt{rec}\ x.\ e \Leftarrow N]\!]_\delta = \bigsqcup_{k \in \mathbb{N}} \left(V \mapsto [\![\Gamma, x : \downarrow N \vdash e \Leftarrow N]\!]_{\delta, V/x}\right)^k \bot_{[\![N]\!]}$$

We will say more about this in Sec. 5.2, but note that the action of rolling and unrolling syntactic values is essentially denoted by $d \mapsto d$:

$$[\![\mathtt{into}(v)]\!]_\delta = [\![v]\!]_\delta$$
$$[\![\{\mathtt{into}(x) \Rightarrow e\}]\!]_\delta = V \mapsto [\![e]\!]_{\delta, V/x}$$

This works due to the fact that unrolling is sound (roughly, the denotations of each side of "$\triangleq$" in the unrolling judgment are *equal*) and the fact that $[\![F]\!]\ (\mu\, [\![F]\!]) = \mu\, [\![F]\!]$ (and similarly for the refined system).

Unrefined soundness. Our proofs of (appendix) Lemma C.28 (Unrefined Type Soundness) and (appendix) Lemma C.30 (Unrefined Substitution Soundness) use standard techniques in domain theory [Gunter 1993].

Unrefined type soundness says that a term typed A under Γ denotes a continuous function $[\![\Gamma]\!] \to [\![A]\!]$. We (partly) state (3 out of 6 parts) this in two mutually recursive lemmas as follows:

Lemma 5.1 (Continuous Maps). *(Lemma C.27 in appendix)*
Suppose $\vdash \delta_1 : \Gamma_1$ *and* $\vdash \delta_2 : \Gamma_2$ *and* $\vdash \Gamma_1, y : Q, \Gamma_2$ ctx.

1. *If* $\Gamma_1, y : Q, \Gamma_2 \vdash h \Rightarrow P$*, then the function* $[\![Q]\!] \to [\![P]\!]$ *defined by* $d \mapsto [\![h]\!]_{\delta_1, d/y, \delta_2}$ *is continuous.*
2. *If* $\Gamma_1, y : Q, \Gamma_2 \vdash e \Leftarrow N$*, then the function* $[\![Q]\!] \to [\![N]\!]$ *defined by* $d \mapsto [\![e]\!]_{\delta_1, d/y, \delta_2}$ *is continuous.*
3. *If* $\Gamma_1, y : Q, \Gamma_2; [N] \vdash s \gg \uparrow P$*, then the function* $[\![Q]\!] \to [\![N]\!] \to [\![\uparrow P]\!]$ *defined by* $d \mapsto [\![s]\!]_{\delta_1, d/y, \delta_2}$ *is continuous.*

Lemma 5.2 (Unrefined Type Soundness). *(Lemma C.28 in appendix)*
Assume $\vdash \delta : \Gamma$.

1. *If* $\Gamma \vdash h \Rightarrow P$*, then* $[\![\Gamma \vdash h \Rightarrow P]\!]_\delta \in [\![P]\!]$.
2. *If* $\Gamma \vdash e \Leftarrow N$*, then* $[\![\Gamma \vdash e \Leftarrow N]\!]_\delta \in [\![N]\!]$.
3. *If* $\Gamma; [N] \vdash s \gg \uparrow P$*, then* $[\![\Gamma; [N] \vdash s \gg \uparrow P]\!]_\delta \in [\![N]\!] \Rightarrow [\![\uparrow P]\!]$.

The proof of unrefined type soundness is standard, and uses the well-known fact that a continuous function in **Cppo** has a least fixed point. Among other things, we also use the fact that $\mu\, [\![F]\!]$ is a fixed point of $[\![F]\!]$ (appendix Lemma C.7). We also use the soundness of unrefined unrolling, which we didn't mention here because it's similar to refined unrolling and its soundness, discussed in the next section.

We interpret an unrefined syntactic substitution (typing derivation) $\Gamma_0 \vdash \sigma : \Gamma$ as a continuous function $[\![\Gamma_0]\!] \to [\![\Gamma]\!]$ that takes a $\delta \in [\![\Gamma_0]\!]$ and uses it to interpret each of the entries in

σ (remembering to apply the rest of the syntactic substitution, because substitution is defined sequentially):

$$\llbracket \Gamma_0 \vdash \cdot : \cdot \rrbracket = (\delta \in \llbracket \Gamma_0 \rrbracket) \mapsto \cdot$$
$$\llbracket \Gamma_0 \vdash \big(\sigma, (v : P/x)\big) : \big(\Gamma, x : P\big) \rrbracket = (\delta \in \llbracket \Gamma_0 \rrbracket) \mapsto \big((\llbracket \sigma \rrbracket_\delta), \llbracket [\sigma]v \rrbracket_\delta /x\big)$$

Similarly to typing derivations, we only consider denotations of typing derivations $\Gamma_0 \vdash \sigma : \Gamma$ of substitutions, but often simply write $\llbracket \sigma \rrbracket$.

Unrefined substitution soundness says that denotation and syntactic substitution (or semantic and syntactic substitution) commute: if E is a program term typed under Γ and $\Gamma_0 \vdash \sigma : \Gamma$ is a substitution, then $\llbracket [\sigma]E \rrbracket = \llbracket E \rrbracket \circ \llbracket \sigma \rrbracket$. Here, we partly show how it is stated in the appendix (1 out of 6 parts):

Lemma 5.3 (Unrefined Substitution Soundness). *(Lemma C.30 in appendix)*
Assume $\Gamma_0 \vdash \sigma : \Gamma$ *and* $\vdash \delta : \Gamma_0$.

(1) If $\Gamma \vdash e \Leftarrow N$, *then* $\llbracket \Gamma_0 \vdash [\sigma]e \Leftarrow N \rrbracket_\delta = \llbracket \Gamma \vdash e \Leftarrow N \rrbracket_{\llbracket \sigma \rrbracket_\delta}$.

We use unrefined type/substitution soundness to prove *refined* type/substitution soundness, discussed next.

5.2 Refined System

Indexes. For any sort τ, we give its denotation $\llbracket \tau \rrbracket$ the discrete order $\sqsubseteq_{\llbracket \tau \rrbracket} = \big\{(d, d) \;\big|\; d \in \llbracket \tau \rrbracket\big\}$, making it a cpo.

Semantic Substitution. We introduced semantic substitutions δ (at the index level) when discussing propositional validity (Sec. 4.1). Here, they are extended to semantic program values:

$$\frac{\vdash \delta : \Theta; \Gamma \qquad V \in \llbracket P \rrbracket_{\lfloor \delta \rfloor} \qquad x \notin dom(\Gamma)}{\vdash (\delta, V/x) : \Theta; \Gamma, x : P}$$

where $\lfloor - \rfloor$ filters out program entries. *Notation*: we define $\llbracket \Theta; \Gamma \rrbracket = \{\delta \mid \vdash \delta : \Theta; \Gamma\}$.

Erasure. The *erasure* metaoperation $|-|$ (appendix Sec. A.5) erases all indexes from (refined) types, program terms (which can have type annotations, but those do not affect program meaning), and syntactic and semantic substitutions. For example, $|\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}| = \mu|F|$ and $|\forall a : \tau.\ N| = |N|$ and $|P \times Q| = |P| \times |Q|$ and so on.

We use many facts about erasure to prove refined type/substitution soundness (appendix lemmas):

- Refined types denote subsets of what their erasures denote: Lemma C.31 (Type Subset of Erasure). Similarly for refined functors and refined inductive types: Lemma C.32 (Functor Application Subset of Erasure) and Lemma C.33 (Mu Subset of Erasure).
- The erasure of both types appearing in extraction, equivalence, and subtyping judgments results in equal (unrefined) types: Lemma C.36 (Extraction Erases to Equality), Lemma C.37 (Equivalence Erases to Equality), and Lemma C.38 (Subtyping Erases to Equality).
- Refined unrolling and typing are sound with respect to their erasure: Lemma C.39 (Erasure Respects Unrolling), Lemma C.40 (Erasure Respects Typing), and Lemma C.42 (Erasure Respects Substitution Typing).
- Erasure commutes with syntactic and semantic substitution: Lemma C.41 (Erasure Respects Substitution) and Lemma C.43 (Erasure Respects Semantic Substitution).

Types, functors, algebras, and folds. The denotations of refined types and functors are defined as logical subsets of the denotations of their erasures (together with their erasure denotations themselves). They are defined mutually with the denotations of well-formed algebras.

In appendix Fig. 36, we inductively define the denotations of well-formed types $\Theta \vdash A$ type[_]. We briefly discuss a few of the cases. The meaning of an asserting type is the set of refined values such that the asserted index proposition holds (read $\{\bullet\}$ as true and $\emptyset$ as false):

$$[\![P \land \phi]\!]_\delta = \left\{V \in [\![|P|]\!] \;\middle|\; V \in [\![P]\!]_\delta \text{ and } [\![\phi]\!]_\delta = \{\bullet\}\right\}$$

Existential and universal types denote elements of their erasure such that the relevant index quantification holds:

$$[\![\exists a : \tau.\, P]\!]_\delta = \left\{V \in [\![|P|]\!] \;\middle|\; \exists d \in [\![\tau]\!].\, V \in [\![P]\!]_{\delta,d/a}\right\}$$
$$[\![\forall a : \tau.\, N]\!]_\delta = \left\{f \in [\![|N|]\!] \;\middle|\; \forall d \in [\![\tau]\!].\, f \in [\![N]\!]_{\delta,d/a}\right\}$$

Guarded types denote elements of their erasure such that they are also in the refined type being guarded if the guard holds ($\{\bullet\}$ means true):

$$[\![\phi \supset N]\!]_\delta = \left\{f \in [\![|N|]\!] \;\middle|\; \text{if } [\![\phi]\!]_\delta = \{\bullet\} \text{ then } f \in [\![N]\!]_\delta\right\}$$

The denotation of refined function types $[\![P \to N]\!]_\delta$ is *not* the set $[\![P]\!]_\delta \Rightarrow [\![N]\!]_\delta$ of (continuous) functions from refined P-values to refined N-values; if it were, then type soundness would break:

$$[\![\cdot;\cdot \vdash \lambda x.\, \mathtt{return}\, x \Leftarrow (1 \land \mathsf{F}) \to \uparrow 1]\!]_\cdot = (\bullet \mapsto inj_1\, \bullet)$$

which is not in $(\emptyset \Rightarrow \{\bullet\} \uplus \{\bot_\uparrow\})$. Instead, the meaning of a refined function type is a set (resembling a unary logical relation)

$$\left\{f \in [\![|P \to N|]\!] \;\middle|\; \forall V \in [\![P]\!]_\delta.\, f(V) \in [\![N]\!]_\delta\right\}$$

of *unrefined* (continuous) functions that take refined values to refined values. The meaning of *refined* upshifts enforces termination (if refined type soundness holds, and we will see it does):

$$[\![\uparrow P]\!]_\delta = \left\{inj_1\, V \;\middle|\; V \in [\![P]\!]_\delta\right\}$$

Note that divergence $inj_2\, \bot_\uparrow$ is *not* in the set $[\![\uparrow P]\!]_\delta$.

In appendix Fig. 37, we inductively define the denotations of well-formed refined functors F and algebras α. The main difference between refined and unrefined functors is that in refined functors, component constant functors produce subsets of their erasure. All functors, refined or otherwise, also (forgetting the partial order structure) denote endofunctors on the category of sets and functions. As with our unrefined functors, our refined functors denote functors with a fixed point (appendix Lemma D.11): $[\![F]\!]_\delta\, (\mu\, [\![F]\!]_\delta) = \mu\, [\![F]\!]_\delta$. Moreover, $\mu\, [\![F]\!]_\delta$ satisfies a recursion principle such that we can inductively define measures on them via $[\![F]\!]_\delta$-algebras (discussed next).

Categorically, given an endofunctor F, an F-*algebra* is an evaluator map $\alpha : F(\tau) \to \tau$ for some carrier set τ. We may think of this in terms of elementary algebra: we form algebraic expressions with F and evaluate them with α. A morphism f from algebra $\alpha : F(\tau) \to \tau$ to algebra $\beta : F(\tau') \to \tau'$ is a morphism $f : \tau \to \tau'$ such that $f \circ \alpha = \beta \circ (F(f))$. If an endofunctor F has an *initial*[6] algebra $into : F(\mu F) \to \mu F$, then it has a recursion principle. By the recursion principle for μF, we can define a recursive function from μF to τ by *folding* μF with an F-algebra $\alpha : F(\tau) \to \tau$ like so:

$$\begin{aligned}&(fold_F\ \alpha) : \mu F \to \tau\\ &(fold_F\ \alpha)\ v = \alpha\ \Big(\big(fmap\ F\ (fold_F\ \alpha)\big)\ \big(out(v)\big)\Big)\end{aligned}$$

[6]An object X in a category $\mathbf{C}$ is *initial* if for every object Y in $\mathbf{C}$, there exists a unique morphism $X \to Y$ in $\mathbf{C}$.

where $out : \mu F \to F(\mu F)$, which by Lambek's lemma exists and is inverse to *into*, embeds (semantic) inductive values into the unrolling of the (semantic) inductive type (we usually elide *fmap*). Conveniently, in our system and semantics, *out* is always $d \mapsto d$, and we almost never explicitly mention it. Syntactic values v in our system must be rolled into inductive types—$\mathtt{into}(v)$—and this is also how (syntactic) inductive values are pattern-matched ("applying *out*" to $\mathtt{into}(v)$), but $\mathtt{into}(-)$ conveniently denotes $d \mapsto d$.

We specify inductive types abstractly as sums of products so that they denote polynomial endofunctors more directly. Polynomial endofunctors always have a "least" (initial) fixed point[7], and hence specify inductive types, which have a recursion principle. For example, we specify (modulo the unrolling simplification) $\mathsf{len} : \mathsf{ListF}_A(\mathbb{N}) \Rightarrow \mathbb{N}$ (Sec. 1) by the (syntactic) algebra

$$\alpha \;=\; \mathtt{inj}_1\ () \Rightarrow 0 \mathrel{\|} \mathtt{inj}_2\ (\_, a) \Rightarrow 1 + a$$

which denotes the (semantic) algebra

$$[\![\alpha]\!] : \underbrace{[\![\mathsf{ListF}_A]\!]\ (\mathbb{N})}_{1 \uplus ([\![A]\!] \times \mathbb{N})} \to \mathbb{N}$$

defined by $[\![\alpha]\!] = [\bullet \mapsto 0, (a, n) \mapsto 1 + n]$. By initiality (the recursion principle), there is a unique function

$$\mathit{fold}_{[\![\mathsf{ListF}_A]\!]}\ [\![\alpha]\!] : \mu\, [\![\mathsf{ListF}_A]\!] \to \mathbb{N}$$

such that $\mathit{fold}_{[\![\mathsf{ListF}_A]\!]}\ [\![\alpha]\!] = [\![\alpha]\!] \circ ([\![F]\!]\ (\mathit{fold}_{[\![\mathsf{ListF}_A]\!]}\ [\![\alpha]\!]))$, which semantically captures len (Sec. 1).

In our system, a refined inductive type is written $\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}$, which looks quite similar to its own semantics:

$$[\![\{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\}]\!]_\delta \;=\; \left\{V \in \mu\, [\![F]\!]_\delta \;\middle|\; (\mathit{fold}_{[\![F]\!]_\delta}\ [\![\alpha]\!]_\delta)\ V = [\![t]\!]_\delta\right\}$$

The type $\mathsf{List}\ A\ n$ of A-lists having length $n : \mathbb{N}$, for example, is defined in our system as:

$$\mathsf{List}\ A\ n = \left\{\nu : \mu\mathsf{ListF}_A \;\middle|\; (\mathsf{fold}_{\mathsf{ListF}_A}\ \mathsf{lenalg})\ \nu =_{\mathbb{N}} n\right\}$$

Syntactic types, functors, and algebras in our system look very similar to their own semantics.

A well-typed algebra $\Xi; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau$ denotes a dependent function $\prod_{\delta \in [\![\Theta]\!]} [\![F]\!]_\delta\ [\![\tau]\!] \to [\![\tau]\!]$. The definition (appendix Fig. 37) is mostly standard, but the unit and pack cases could use some explanation. Because Θ is for F and Ξ ($\subseteq \Theta$) is for α, we restrict δ to Ξ at algebra bodies:

$$[\![\Xi; \Theta \vdash () \Rightarrow t : I(\tau) \Rightarrow \tau]\!]_\delta\ \bullet \;=\; [\![\Xi \vdash t : \tau]\!]_{\delta\restriction_\Xi}$$

The most interesting part of the definition concerns index packing:

$$\begin{aligned}&\left[\!\!\left[\Xi; \Theta \vdash (\mathtt{pack}(a, o), q) \Rightarrow t : (\underline{\exists a : \tau'.\, Q} \otimes \hat{P})(\tau) \Rightarrow \tau\right]\!\!\right]_\delta (V_1, V_2) = \\ &\qquad \left[\!\!\left[\Xi, a : \tau'; \Theta, a : \tau' \vdash (o, q) \Rightarrow t : (\underline{Q} \otimes \hat{P})(\tau) \Rightarrow \tau\right]\!\!\right]_{(\delta, d/a)} (V_1, V_2) \\ &\qquad \text{where } d \in [\![\tau']\!] \text{ is such that } V_1 \in [\![Q]\!]_{\delta, d/a}\end{aligned}$$

The pack clause lets us bind the witness d of τ' in the existential type $\exists a : \tau'.\, Q$ to a in the body t of the algebra. We know d exists since $V_1 \in [\![\exists a : \tau'.\, Q]\!]_\delta$, but it is not immediate that it is unique. However, we prove d is uniquely determined by V_1; we call this property the *soundness of value-determined indexes* (in its statement, below, all parts are mutually recursive):

Lemma 5.4 (Soundness of Value-Determined Indexes). *(Lemma D.14 in appendix)* *Assume* $\vdash \delta_1 : \Theta$ *and* $\delta_2 : \Theta$.

[7]This is not the case for all endofunctors (therefore, not all endofunctors can be said to specify an inductive type). For example, consider the powerset functor.

(1) If $\Theta \vdash P$ type$[\Xi]$ *and* $V \in [\![P]\!]_{\delta_1}$ *and* $V \in [\![P]\!]_{\delta_2}$*, then* $\delta_1\!\upharpoonright_\Xi = \delta_2\!\upharpoonright_\Xi$.
(2) If $\Theta \vdash \mathcal{F}$ functor$[\Xi]$ *and* $X_1, X_2 \in$ **Set** *and* $V \in [\![\mathcal{F}]\!]_{\delta_1} X_1$ *and* $V \in [\![\mathcal{F}]\!]_{\delta_2} X_2$*, then* $\delta_1\!\upharpoonright_\Xi = \delta_2\!\upharpoonright_\Xi$.
(3) If $\Xi; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau$ *and* $\Xi \subseteq \Theta$ *and* $\delta_1\!\upharpoonright_\Xi = \delta_2\!\upharpoonright_\Xi$*, then* $[\![\alpha]\!]_{\delta_1} = [\![\alpha]\!]_{\delta_2}$ *on* $[\![F]\!]_{\delta_1} [\![\tau]\!] \cap [\![F]\!]_{\delta_2} [\![\tau]\!]$.

Therefore, the Ξ in type and functor well-formedness really does track index variables that are *uniquely* determined by values, semantically speaking.

Well-typed program terms. Appendix Fig. 38 specifies the denotations of well-typed refined program terms in terms of the denotations of their erasure. The denotation of a refined program term E typed under $(\Theta; \Gamma)$, at refined semantic substitution $\delta \in [\![\Theta; \Gamma]\!]$, is the denotation $[\![|E|]\!]_{|\delta|}$ of the (derivation of the) term's erasure $|E|$ at the erased substitution $|\delta|$. For example,

$$[\![\Theta; \Gamma \vdash e \Leftarrow N]\!] = (\delta \in [\![\Theta; \Gamma]\!]) \mapsto [\![|e|]\!]_{|\delta|}$$

Unrolling. We prove (appendix Lemma D.15) that unrolling is sound:

Lemma 5.5 (Unrolling Soundness). *(Lemma D.15 in appendix)*
Assume $\vdash \delta : \Theta$ *and* $\Xi \subseteq \Theta$*. If* $\Xi; \Theta \vdash \{\nu : G[\mu F] \mid \beta\,(G\ (\mathsf{fold}_F\ \alpha)\ \nu) =_\tau t\} \doteq P$,
then $\left\{V \in [\![G]\!]_\delta\,(\mu\,[\![F]\!]_\delta) \;\middle|\; [\![\beta]\!]_\delta\,([\![G]\!]_\delta\,(\mathit{fold}_{[\![F]\!]_\delta}\ [\![\alpha]\!]_\delta)\ V) = [\![t]\!]_\delta\right\} = [\![P]\!]_\delta$.

Due to our definition of algebra denotations (specifically, for the pack pattern), we use the soundness of value-determined indexes in the pack case of the proof.

Subtyping. We prove (appendix Lemma D.19) that subtyping is sound:

Lemma 5.6 (Soundness of Subtyping). *(Lemma D.19 in appendix)*
Assume $\vdash \delta : \Theta$*. If* $\Theta \vdash A \leq^\pm B$*, then* $[\![A]\!]_\delta \subseteq [\![B]\!]_\delta$.

Type soundness. Type soundness says that a program term of type A under Θ and Γ denotes a dependent function $\prod_{\delta \in [\![\Theta;\Gamma]\!]} [\![A]\!]_{\lfloor\delta\rfloor}$. Refined types pick out subsets of values of unrefined types. Therefore, by type soundness, if a program has a refined type, then we have learned something more about that program than the unrefined system can verify for us.

Theorem 5.7 (Type Soundness). *(Thm. D.25 in appendix)*
Assume $\vdash \delta : \Theta; \Gamma$*. Then:*
(1) If $\Theta; \Gamma \vdash h \Rightarrow P$*, then* $[\![h]\!]_\delta \in [\![P]\!]_{\lfloor\delta\rfloor}$.
(2) If $\Theta; \Gamma \vdash g \Rightarrow N$*, then* $[\![g]\!]_\delta \in [\![N]\!]_{\lfloor\delta\rfloor}$.
(3) If $\Theta; \Gamma \vdash v \Leftarrow P$*, then* $[\![v]\!]_\delta \in [\![P]\!]_{\lfloor\delta\rfloor}$.
(4) If $\Theta; \Gamma \vdash e \Leftarrow N$*, then* $[\![e]\!]_\delta \in [\![N]\!]_{\lfloor\delta\rfloor}$.
(5) If $\Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$*, then* $[\![\{r_i \Rightarrow e_i\}_{i\in I}]\!]_\delta \in [\![P]\!]_{\lfloor\delta\rfloor} \Rightarrow [\![N]\!]_{\lfloor\delta\rfloor}$.
(6) If $\Theta; \Gamma; [N] \vdash s \gg \uparrow P$*, then* $[\![s]\!]_\delta \in [\![N]\!]_{\lfloor\delta\rfloor} \Rightarrow [\![\uparrow P]\!]_{\lfloor\delta\rfloor}$.

(All parts are mutually recursive.) The proof (appendix Thm. D.25) uses the soundness of unrolling and subtyping. The proof is mostly straightforward. The hardest case is the one for recursive expressions in part (4), where we use an *upward closure* lemma—in particular, part (3) below—to show that the fixed point is in the appropriately refined set:

Lemma 5.8 (Upward Closure). *(Lemma D.22 in appendix)*
Assume $\vdash \delta : \Theta$.
(1) If $\Xi; \Theta \vdash \alpha : F(\tau) \Rightarrow \tau$ *and* $\Xi \subseteq \Theta$ *then* $[\![\alpha]\!]_\delta$ *is monotone.*
(2) If $\Theta \vdash \mathcal{G}$ functor$[_]$ *and* $\Theta \vdash F$ functor$[_]$ *and* $k \in \mathbb{N}$
and $V \in [\![\mathcal{G}]\!]_\delta\,([\![F]\!]_\delta^k\,\emptyset)$ *and* $V \sqsubseteq_{[\![|\mathcal{G}|]\!]\,([\![|F|]\!]^k\,\emptyset)} V'$,
then $V' \in [\![\mathcal{G}]\!]_\delta\,([\![F]\!]_\delta^k\,\emptyset)$.

(3) If $\Theta \vdash A$ type[_] *and* $V \in [\![A]\!]_\delta$ *and* $V \sqsubseteq_{[\![|A|]\!]} V'$*, then* $V' \in [\![A]\!]_\delta$*.*

Out of all proofs in this paper, the proof of upward closure (appendix Lemma D.22) is a top contender for the most interesting induction metric:

Proof. By lexicographic induction on, first, $sz(A)/sz(F)$ (parts (1), (2) and (3), mutually), and, second, $\langle k, \mathcal{G} \text{ structure}\rangle$ (part (2)), where $\langle \dots \rangle$ denotes lexicographic order. □

We define the simple size function $sz(-)$, which is basically a standard structural notion of size, in appendix Fig. 57. This is also the only place, other than unrolling soundness, where we use the soundness of value-determined indexes (again for a pack case, in part (1)).

Substitution soundness. We interpret a syntactic substitution (typing derivation) $\Theta_0; \Gamma_0 \vdash \sigma : \Theta; \Gamma$ as a function $[\![\sigma]\!] : [\![\Theta_0; \Gamma_0]\!] \to [\![\Theta; \Gamma]\!]$ on semantic substitutions (appendix Def. B.1). Similarly to the interpretation of unrefined substitution typing derivations, the interpretation of the head term being substituted (its typing/sorting subderivation) pre-applies the rest of the substitution:

$$\begin{aligned} [\![\cdot]\!]_\delta &= \cdot \\ [\![\sigma, t/a]\!]_\delta &= [\![\sigma]\!]_\delta, [\![[\sigma]t]\!]_{\lfloor\delta\rfloor}/a \\ [\![\sigma, v : P/x]\!]_\delta &= [\![\sigma]\!]_\delta, [\![[\sigma]v]\!]_\delta /x \end{aligned}$$

Substitution soundness holds (appendix Thm. D.28): if E is a program term typed under Θ and Γ, and $\Theta_0; \Gamma_0 \vdash \sigma : \Theta; \Gamma$, then $[\![[\sigma]E]\!] = [\![E]\!] \circ [\![\sigma]\!]$. (Recall we prove a syntactic substitution lemma: Lemma 4.2.) That is, substitution and denotation commute, or (in other words) syntactic substitution and semantic substitution are compatible.

Termination and logical consistency. Our *semantic* type soundness result implies that our system is total and logically consistent.

Corollary 5.9 (Total). *If* $\cdot; \cdot \vdash e \Leftarrow \uparrow P$*, then* $[\![e]\!]_\cdot \neq \bot_{[\![\uparrow P]\!]_\cdot}$*, that is, e does not diverge.*

Proof.

$\vdash \cdot : \cdot; \cdot$	By Emptyδ
$[\![e]\!]_\cdot \in [\![\uparrow P]\!]_{\lfloor\cdot\rfloor}$	By Theorem 5.7 (Type Soundness)
$= [\![\uparrow P]\!]_\cdot$	By definition of $\lfloor - \rfloor$
$= \{ inj_1\, V \mid V \in [\![P]\!]_\cdot \}$	By definition of $[\![-]\!]$.

Therefore, $[\![e]\!]_\cdot \neq inj_2\, \bot_\uparrow = \bot_{[\![\uparrow P]\!]_\cdot}$, that is, e terminates (and returns a value). □

A logically inconsistent type (for example, 0 or $1 \wedge \mathsf{F}$) denotes the empty set, which is uninhabited. Proving logical consistency *syntactically*, say, via progress and preservation lemmas, would require *also* proving that every reduction sequence eventually terminates, which might need a relatively complicated proof using logical relations.

6 ALGORITHMIC SYSTEM

We design our algorithmic system in the spirit of those of Dunfield and Krishnaswami [2013, 2019], but the mechanism of delaying constraint verification, described below, is relatively new. The algorithmic rules closely mimic the declarative rules, except for a few key differences:

- Whenever a declarative rule conjures an index term, the corresponding algorithmic rule adds, to a separate (input) algorithmic context Δ, an existential variable (written with a hat: $\hat{a}$) to be solved.

- As the typing algorithm proceeds, we add index term solutions of the existential variables to the output algorithmic context, increasing knowledge (see Sec. 6.2). We eagerly apply index solutions to input types and output constraints, and pop them off the output context when out of scope.
- Whenever a declarative rule checks propositional validity or equivalence ($\Theta \vdash \phi$ true or $\Theta \vdash \phi \equiv \psi$), the algorithm delays checking the constraint until all existential variables in the propositions are solved. Similarly, subtyping, type equivalence, and expression typechecking constraints are delayed until all existential variables are solved. When an entity has no existential variables, we say that it is *ground*.
- In subtyping, we eagerly extract from assumptive positions immediately under polarity shifts.

Syntactically, objects in the algorithmic system are not much different than corresponding objects of the declarative system. We extend the grammar for index terms with a production of existential variables, which we write as an index variable with a hat $\hat{a}$, $\hat{b}$, or $\hat{c}$:

$$t ::= \cdots \mid \hat{a}$$

We use this (algorithmic) index grammar everywhere in the algorithmic system, using the same declarative metavariables. However, we write algorithmic logical contexts with a hat: $\hat{\Theta}$. Algorithmic logical contexts $\hat{\Theta}$ only appear in output mode, and are like (input) logical contexts Θ, but propositions listed in them may have existential variables (its index variable sorting declarations are universal, so a $\hat{\Theta}$ is only well-formed under a Δ).

Constraints are added to the algorithmic system. Figure 13 gives grammars for subtyping and typing constraints.

$$\begin{array}{llcl}
\text{Subtyping constraints} & W & ::= & \phi \mid \phi \equiv \psi \mid \phi \supset W \mid W \land W \mid \forall a : \tau.W \\
& & & \mid \overline{P <:^{+} Q} \mid \underline{N <:^{-} M} \mid P \equiv^{+} Q \mid N \equiv^{-} M \\
\text{Typing constraints} & \chi & ::= & \cdot \mid (e \Leftarrow N), \chi \mid W, \chi
\end{array}$$

Fig. 13. Typing and subtyping constraints

Checking constraints boils down to checking propositional validity, $\Theta \vdash \phi$ true, which is analogous to checking what are called *verification conditions* in the tradition of imperative program verification initiated by Floyd and Hoare [Floyd 1967; Hoare 1969] (where programs annotated with Floyd–Hoare assertions are analyzed, generating verification conditions whose validity implies program correctness). These propositional validity constraints are the constraints that can be automatically verified by a theorem prover such as an SMT solver. The (algorithmic) W constraint verification judgment is written $\Theta \models W$ and means that W algorithmically holds under Θ. Notice that the only context in the judgment is Θ, which has no existential variables: this reflects the fact that we delay verifying W until W has no existential variables (in which case we say W is *ground*). Similarly, $\Theta; \Gamma \lhd \chi$ is the (algorithmic) χ verification judgment, meaning all of the constraints in χ algorithmically hold under Θ and Γ, and here χ is also ground (by focusing).

6.1 Contexts and Substitution

Algorithmic contexts Δ are lists of solved or unsolved existential variables, and are said to be *complete*, and are written as Ω, if they are all solved:

$$\Delta ::= \cdot \mid \Delta, \hat{a} : \tau \mid \Delta, \hat{a} : \tau{=}t$$
$$\Omega ::= \cdot \mid \Omega, \hat{a} : \tau{=}t$$

We require solutions t of existential variables $\hat{a}$ to be well-sorted under (input) logical contexts Θ, which have no existential variables. To maintain this invariant that every solution in Δ is *ground*, that is, has no existential variables, we exploit type polarity in algorithmic subtyping, and prevent existential variables from ever appearing in refinement algebras.

We will often treat algorithmic contexts Δ as substitutions for existential variables $\hat{a}$ in index terms t (including propositions ϕ), types A, functors $\mathcal{F}$, constraints W and χ, and output logical contexts $\hat{\Theta}$ (whose propositions may have existential variables). The definition is straightforward: homomorphically apply the context to the object O, and further define $[\Delta]O$ by induction on Δ.

$$\begin{aligned} [\cdot]O &= O \\ [\Delta, \hat{a} : \tau]O &= [\Delta]O \\ [\Delta, \hat{a} : \tau{=}t]O &= [\Delta]([t/\hat{a}]O) \end{aligned}$$

The order of substitutions in the definition of context application above does not matter because solutions are ground (we may view $[\Delta]O$ as simultaneous substitution). If O only has existential variables from $dom(\Omega)$, then $[\Omega]O$ is ground.

6.2 Context Extension

The algorithmic context extension judgment $\Theta \vdash \Delta \longrightarrow \Delta'$ says that $dom(\Delta) = dom(\Delta')$ and Δ' has the same solutions as Δ, but possibly solves more (that are unsolved in Δ). All typing and subtyping judgments (under Θ) that have input and output algorithmic contexts Δ and Δ' (respectively) enjoy the property that they increase index information, that is, $\Theta \vdash \Delta \longrightarrow \Delta'$. If $\Theta \vdash \Delta \longrightarrow \Omega$, then Ω *completes* Δ: it has Δ's solutions, but also solutions to all of Δ's unsolved variables.

$$\frac{}{\Theta \vdash \cdot \longrightarrow \cdot} \qquad \frac{\Theta \vdash \Delta \longrightarrow \Delta'}{\Theta \vdash \Delta, \hat{a} : \tau \longrightarrow \Delta', \hat{a} : \tau} \qquad \frac{\Theta \vdash \Delta \longrightarrow \Delta'}{\Theta \vdash \Delta, \hat{a} : \tau{=}t \longrightarrow \Delta', \hat{a} : \tau{=}t} \qquad \frac{\Theta \vdash \Delta \longrightarrow \Delta'}{\Theta \vdash \Delta, \hat{a} : \tau \longrightarrow \Delta', \hat{a} : \tau{=}t}$$

6.3 Subtyping

Algorithmic subtyping $\Theta; \Delta \vdash A <:^{\pm} B \;/\; W \dashv \Delta'$ says that, under logical context Θ and algorithmic context Δ, the type A is algorithmically a subtype of B if and only if output constraint W holds algorithmically (under suitable solutions including those of Δ'), outputting index solutions Δ'. In subtyping and type equivalence, the delayed output constraints W must remember their logical context via $\supset$ and $\forall$. For example, in checking that $\exists a : \mathbb{N}.\ \mathsf{Nat}(a) \wedge (a < 5)$ is a subtype of $\exists a : \mathbb{N}.\ \mathsf{Nat}(a) \wedge (a < 10)$, the output constraint W is $\forall a : \mathbb{N}.\ (a < 5) \supset (a < 10)$.

For space reasons, we don't present all algorithmic subtyping rules here (see appendix Fig. 48), but only enough rules to discuss the key design issues. Further, we don't present algorithmic equivalence here (see appendix Figures 44 and 46), which is similar to and simpler than algorithmic subtyping.

In algorithmic subtyping, we maintain the invariant that positive subtypes and negative supertypes are ground. The rules

$$\frac{\Theta; \Delta, \hat{a} : \tau \vdash P <:^{+} [\hat{a}/a]Q \;/\; W \dashv \Delta', \hat{a} : \tau{=}t}{\Theta; \Delta \vdash P <:^{+} \exists a : \tau.\ Q \;/\; W \dashv \Delta'} \qquad \frac{\Theta; \Delta, \hat{a} : \tau \vdash [\hat{a}/a]N <:^{-} M \;/\; W \dashv \Delta', \hat{a} : \tau{=}t}{\Theta; \Delta \vdash \forall a : \tau.\ N <:^{-} M \;/\; W \dashv \Delta'}$$

are the only subtyping rules which add existential variables (to the side not necessarily ground) to be solved (whereas the declarative system conjures a solution). We pop off the solution as we have the invariant that output contexts are eagerly applied to output constraints and input types.

The rule

$$\frac{t \text{ ground} \qquad \Theta; \Delta \vdash F \equiv G \ / \ W \dashv \Delta_1', \hat{a} : \tau, \Delta_2' \qquad \Delta' = \Delta_1', \hat{a} : \tau{=}t, \Delta_2'}{\Theta; \Delta \vdash \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\ \nu =_\tau t\} <:^+ \{\nu : \mu G \mid (\mathsf{fold}_G\ \alpha)\ \nu =_\tau \hat{a}\} \ / \ W \wedge (t = t) \dashv \Delta'} <:^+/{\dashv}\mu\textsc{Inst}$$

runs the functor equivalence algorithm (which outputs constraint W and solutions $\Delta_1', \hat{a} : \tau, \Delta_2'$), checks that $\hat{a}$ does not get solved there, and then solves $\hat{a}$ to t (yielding Δ') after checking that the latter (which is a subterm of a positive subtype) is ground, outputting the constraint generated by functor equivalence together with the equation $t = t$ (the declarative system can conjure a different but logically equal term for the right-hand side of this equation), and Δ'. Alternatively, there is a rule for when $\hat{a}$ gets solved by functor equivalence, and a rule where a term that is not an existential variable is in place of $\hat{a}$.

The rule

$$\frac{\Theta; \Delta \vdash M \rightsquigarrow M' \ [\hat{\Theta}]}{\Theta; \Delta \vdash {\downarrow}N <:^+ {\downarrow}M \ / \ \left(\hat{\Theta} \supset^* \underline{N <:^- M'}\right) \dashv \Delta}$$

extracts M' and $\hat{\Theta}$ from M and delays the resulting negative subtyping constraint $\underline{N <:^- M'}$, to be verified under its logical setting $\hat{\Theta}$ (whose propositions, which were extracted from the side not necessarily ground, may have existential variables only solved in value typechecking). The metaoperation $\supset^*$ traverses $\hat{\Theta}$, creating universal quantifiers from universal variables and implications from propositions:

$$\begin{aligned} \cdot \supset^* W &= W \\ (\hat{\Theta}, \phi) \supset^* W &= \hat{\Theta} \supset^* (\phi \supset W) \\ (\hat{\Theta}, a : \tau) \supset^* W &= \hat{\Theta} \supset^* (\forall a : \tau.\, W) \end{aligned}$$

The dual shift rule is similar. In the declarative system, $\leq^+{\rightsquigarrow}$L and $\leq^-{\rightsquigarrow}$R are invertible, which means that they can be eagerly applied without getting stuck; algorithmically, we apply them immediately at polarity shifts, so the above rule corresponds to an algorithmic combination of the declarative rules $\leq^-{\rightsquigarrow}$R and $\leq^+{\downarrow}$ (and similarly for its dual rule for $\uparrow$).

For rules with multiple nontrivial premises, such as product subtyping

$$\frac{\Theta; \Delta \vdash P_1 <:^+ Q_1 \ / \ W_1 \dashv \Delta'' \qquad \Theta; \Delta'' \vdash P_2 <:^+ [\Delta'']Q_2 \ / \ W_2 \dashv \Delta'}{\Theta; \Delta \vdash (P_1 \times P_2) <:^+ (Q_1 \times Q_2) \ / \ [\Delta']W_1 \wedge W_2 \dashv \Delta'}$$

we thread solutions through inputs, applying them to the non-ground side ($[\Delta]-$ treats Δ as a substitution of index solutions for existential variables), ultimately outputting both delayed constraints. We maintain the invariant that existential variables in output constraints are eagerly solved, which is why, for example, Δ' is applied to W_1 in the conclusion of the above rule, but not to W_2 (that would be redundant).

6.4 Typing

We now discuss issues specific to algorithmic program typing.

Exploiting polarity, we can restrict the flow of index information to the right- and left-focusing phases: in particular, $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \ / \ \chi \dashv \Delta'$ and $\Theta; \Delta; \Gamma; [N] \vdash s \gg {\uparrow}P \ / \ \chi \dashv \Delta'$, the algorithmic value and spine typechecking judgments. The input types of these judgments can have existential variables, and these judgments synthesize constraints and index solutions, but the algorithmic versions of the other judgments do not; we judgmentally distinguish the latter by replacing the "$\vdash$" in the declarative judgments with "$\rhd$" (for example, $\Theta; \Gamma \rhd g \Rightarrow {\uparrow}P$). Delayed constraints are

$\boxed{\Theta; \Delta; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \Delta'}$ Under inputs contexts Θ, Δ, and Γ, the value v checks against type P, with output computation constraints χ and output context Δ'

$$\frac{P \neq \exists, \wedge \qquad (x : Q) \in \Gamma \qquad \Theta; \Delta \vdash Q <:^{+} P \;/\; W \dashv \Delta'}{\Theta; \Delta; \Gamma \vdash x \Leftarrow P \;/\; W \dashv \Delta'} \;\text{Alg}{\Leftarrow}\text{Var}$$

$$\frac{}{\Theta; \Delta; \Gamma \vdash \langle\rangle \Leftarrow 1 \;/\; \cdot \dashv \Delta} \;\text{Alg}{\Leftarrow}1$$

$$\frac{\Theta; \Delta; \Gamma \vdash v_1 \Leftarrow P_1 \;/\; \chi_1 \dashv \Delta'' \qquad \Theta; \Delta''; \Gamma \vdash v_2 \Leftarrow [\Delta'']P_2 \;/\; \chi_2 \dashv \Delta'}{\Theta; \Delta; \Gamma \vdash \langle v_1, v_2 \rangle \Leftarrow (P_1 \times P_2) \;/\; [\Delta']\chi_1, \chi_2 \dashv \Delta'} \;\text{Alg}{\Leftarrow}\times$$

$$\frac{\Theta; \Delta; \Gamma \vdash v_k \Leftarrow P_k \;/\; \chi \dashv \Delta'}{\Theta; \Delta; \Gamma \vdash \mathsf{inj}_k\, v_k \Leftarrow (P_1 + P_2) \;/\; \chi \dashv \Delta'} \;\text{Alg}{\Leftarrow}{+_k}$$

$$\frac{\Theta; \Delta, \hat{a} : \tau; \Gamma \vdash v \Leftarrow [\hat{a}/a]P \;/\; \chi \dashv \Delta', \hat{a} : \tau{=}t}{\Theta; \Delta; \Gamma \vdash v \Leftarrow (\exists a : \tau.\, P) \;/\; \chi \dashv \Delta'} \;\text{Alg}{\Leftarrow}\exists$$

$$\frac{\Theta; \Delta; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \Delta'' \qquad \Theta; \Delta'' \vdash [\Delta'']\phi \;\mathsf{inst} \dashv \Delta'}{\Theta; \Delta; \Gamma \vdash v \Leftarrow (P \wedge \phi) \;/\; ([\Delta']\phi, [\Delta']\chi) \dashv \Delta'} \;\text{Alg}{\Leftarrow}\wedge$$

$$\frac{\cdot; \Theta; \Delta \vdash \{\nu : F[\mu F] \mid \alpha\,(F\ (\mathsf{fold}_F\ \alpha)\ \nu) =_\tau t\} \doteq P \qquad \Theta; \Delta; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \Delta'}{\Theta; \Delta; \Gamma \vdash \mathsf{into}(v) \Leftarrow \{\nu : \mu F \mid (\mathsf{fold}_F\ \alpha)\, \nu =_\tau t\} \;/\; \chi \dashv \Delta'} \;\text{Alg}{\Leftarrow}\mu$$

$$\frac{}{\Theta; \Delta; \Gamma \vdash \{e\} \Leftarrow {\downarrow}N \;/\; (e \Leftarrow N) \dashv \Delta} \;\text{Alg}{\Leftarrow}{\downarrow}$$

$\boxed{\Theta; \Delta; \Gamma; [N] \vdash s \gg {\uparrow}P \;/\; \chi \dashv \Delta'}$ Under Θ, Δ, and Γ, passing spine s to a head of type ${\downarrow}N$ synthesizes ${\uparrow}P$, with output constraints χ and context Δ'

$$\frac{\Theta; \Delta, \hat{a} : \tau; \Gamma; [[\hat{a}/a]N] \vdash s \gg {\uparrow}P \;/\; \chi \dashv \Delta', \hat{a} : \tau{=}t}{\Theta; \Delta; \Gamma; [\forall a : \tau.\, N] \vdash s \gg {\uparrow}P \;/\; \chi \dashv \Delta'} \;\text{AlgSpine}\forall$$

$$\frac{\Theta; \Delta; \Gamma; [N] \vdash s \gg {\uparrow}P \;/\; \chi \dashv \Delta'}{\Theta; \Delta; \Gamma; [\phi \supset N] \vdash s \gg {\uparrow}P \;/\; [\Delta']\phi, \chi \dashv \Delta'} \;\text{AlgSpine}{\supset}$$

$$\frac{\Theta; \Delta; \Gamma \vdash v \Leftarrow Q \;/\; \chi \dashv \Delta'' \qquad \Theta; \Delta''; \Gamma; [[\Delta'']N] \vdash s \gg {\uparrow}P \;/\; \chi' \dashv \Delta'}{\Theta; \Delta; \Gamma; [Q \to N] \vdash v, s \gg {\uparrow}P \;/\; [\Delta']\chi, \chi' \dashv \Delta'} \;\text{AlgSpineApp}$$

$$\frac{}{\Theta; \Delta; \Gamma; [{\uparrow}P] \vdash \cdot \gg {\uparrow}P \;/\; \mathsf{T} \dashv \Delta} \;\text{AlgSpineNil}$$

Fig. 14. Algorithmic value and spine typing

verified only and immediately after completing a focusing phase, when all their existential variables are necessarily solved.

Consequently, the algorithmic typing judgments for heads, bound expressions, pattern matching, and expressions are essentially the same as their declarative versions, but with a key difference. Namely, in Alg$\Rightarrow$ValAnnot, Alg$\Rightarrow$App, and Alg$\Leftarrow\uparrow$ (below, respectively), focusing phases start with an empty algorithmic context, outputting ground constraints (and an empty output context because solutions are eagerly applied), and a premise is added to verify these constraints:

$$\frac{\Theta; \cdot; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \cdot \qquad \Theta; \Gamma \lhd \chi}{\Theta; \Gamma \rhd (v : P) \Rightarrow P} \qquad \frac{\Theta; \Gamma \rhd h \Rightarrow {\downarrow} N \qquad \Theta; \cdot; \Gamma; [N] \vdash s \gg {\uparrow} P \;/\; \chi \dashv \cdot \qquad \Theta; \Gamma \lhd \chi}{\Theta; \Gamma \rhd h(s) \Rightarrow {\uparrow} P}$$

$$\frac{\Theta; \cdot; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \cdot \qquad \Theta; \Gamma \lhd \chi}{\Theta; \Gamma \rhd \mathsf{return}\, v \Leftarrow {\uparrow} P}$$

Algorithmic typechecking for recursive expressions uses algorithmic subtyping, which outputs a ground constraint W. Because this W is ground, we can verify it ($\Theta \models W$) immediately:

$$\frac{\begin{array}{c}\Theta \vdash N \not\leftrightsquigarrow \qquad \Theta; \cdot \vdash \forall a : \mathbb{N}.\, M <:^- N \;/\; W \dashv \cdot \qquad \Theta \models W \\ \Theta, a : \mathbb{N}; \Gamma, x : {\downarrow}\big(\forall a' : \mathbb{N}.\, (a' < a) \supset [a'/a]M\big) \rhd e \Leftarrow M\end{array}}{\Theta; \Gamma \rhd \mathsf{rec}\; (x : \forall a : \mathbb{N}.\, M).\; e \Leftarrow N}$$

For the full definition of algorithmic typing, see appendix Figures 51, 52, and 53.

Besides the instantiation rules (such as $<:^+/\dashv\mu$Inst) for inductive types in algorithmic subtyping and type equivalence, there are exactly two judgments ($\Theta; \Delta \vdash \phi$ inst $\dashv \Delta'$ and $\Theta; \Delta \vdash \phi \equiv \psi$ inst $\dashv \Delta'$) responsible for index instantiation, both dealing with the output of algorithmic unrolling. Algorithmic unrolling can output indexes of the form $\hat{a} = t$ with t ground, and these equations are solved in either value typechecking, subtyping, or type equivalence. In the former two cases, we can solve $\hat{a}$ as the algebra body t which is necessarily ground (as discussed in Sec. 4.2). The judgment $\Theta; \Delta \vdash \phi$ inst $\dashv \Delta'$ (appendix Fig. 45), used in Alg$\Leftarrow\wedge$, checks whether ϕ has form $\hat{a} = t$ where t is ground. If so, then it solves $\hat{a}$ to t in Δ; otherwise, it does not touch Δ.

$$\frac{\Theta \vdash t : \tau}{\Theta; \Delta_1, \hat{a} : \tau, \Delta_2 \vdash \hat{a} = t \;\mathsf{inst} \dashv \Delta_1, \hat{a} : \tau{=}t, \Delta_2}\;\text{Inst} \qquad \frac{\phi \text{ not of form } \hat{a} = t \text{ where } \Theta \vdash t : \tau}{\Theta; \Delta \vdash \phi \;\mathsf{inst} \dashv \Delta}\;\text{NoInst}$$

For example, suppose a head synthesizes $\forall a : \mathbb{N}.\, \mathsf{Nat}(a) \to {\uparrow}\mathsf{Nat}(a)$ and we wish to apply this head to the spine (containing exactly one argument value) $\mathsf{into}(\mathsf{inj}_2\, \langle \mathsf{into}(\mathsf{inj}_1\, \langle\rangle), \langle\rangle\rangle)$. We generate a fresh existential variable $\hat{a}$ for a (rule AlgSpine$\forall$) and then check the value against $\mathsf{Nat}(\hat{a})$ (rule AlgSpineApp). (Checking the same value against type $\exists a : \mathbb{N}.\, \mathsf{Nat}(a)$ yields the same problem, by dual rule Alg$\Leftarrow\exists$, and the following solution also works in this case.) The type $\mathsf{Nat}(\hat{a})$ has value-determined index $\hat{a}$ (its Ξ is $\hat{a} : \mathbb{N}$), so it is solvable. We unroll (rule Alg$\Leftarrow\mu$) $\mathsf{Nat}(\hat{a})$ to $\big(1 \wedge (\hat{a} = 0)\big) + \big(\exists a' : \mathbb{N}.\, \mathsf{Nat}(a') \times \big(1 \wedge (\hat{a} = 1 + a')\big)\big)$ and check $\mathsf{inj}_2\, \langle \mathsf{into}(\mathsf{inj}_1\, \langle\rangle), \langle\rangle\rangle$ against that (0 and $1 + a'$ are the bodies of the two branches of Nat's algebra). In this unrolled type, $\hat{a}$ is no longer tracked by its Ξ, but we can still solve it.

The value now in question is a right injection, so we must check $\langle \mathsf{into}(\mathsf{inj}_1\, \langle\rangle), \langle\rangle\rangle$ against $\exists a' : \mathbb{N}.\, \mathsf{Nat}(a') \times \big(1 \wedge (\hat{a} = 1 + a')\big)$ (rule Alg$\Leftarrow+_2$). We generate another fresh existential variable $\hat{a}'$ in place of a'. We now check the pair using rule Alg$\Leftarrow\times$. For the first component, we check $\mathsf{inj}_1\, \langle\rangle$ against the unrolled $\mathsf{Nat}(\hat{a}')$, which is $1 \wedge (\hat{a}' = 0) + \cdots$. Now we solve $\hat{a}' = 0$ (rules Alg$\Leftarrow+_1$, Alg$\Leftarrow\wedge$, Inst, and Alg$\Leftarrow 1$). This information flows to the type $1 \wedge (\hat{a} = 1 + \hat{a}')$ against which we need to check the second value component ($\langle\rangle$). By "this information flows," we mean that we apply the context output by type checking the first component, namely $\hat{a} : \mathbb{N}, \hat{a}' : \mathbb{N}{=}0$ (notice $\hat{a}$ is not yet solved), as a substitution to obtain $1 \wedge (\hat{a} = 1 + 0)$ for the second premise of Alg$\Leftarrow\times$. The right-hand side of the equation now has no existential variables, and we solve $\hat{a} = 1 + 0 = 1$ (again using Alg$\Leftarrow\wedge$), as expected. It is worth noting that this solving happens entirely within focusing phases.

Values of inductive type may involve program variables, so existential variables may not be solved by Alg$\Leftarrow\wedge$ (and Inst), but in algorithmic subtyping, using the same instantiation judgment:

$$\frac{\Theta; \Delta \vdash P <:^+ Q \;/\; W \dashv \Delta'' \qquad \Theta; \Delta'' \vdash [\Delta'']\phi \;\mathsf{inst} \dashv \Delta'}{\Theta; \Delta \vdash P <:^+ Q \wedge \phi \;/\; [\Delta']W \wedge [\Delta']\phi \dashv \Delta'}\; <:^+/\dashv\wedge\text{R}$$

Finally, if an equation of the form $\hat{a} = t$ makes its way into type equivalence (by checking a variable value against a sum type), then $\hat{a}$ gets solved, not as t, but rather as the index in the same structural position (including logical structure) of the necessarily ground positive type on the left of the equivalence (see judgment $\Theta; \Delta \vdash \phi \equiv \psi \ \mathsf{inst} \dashv \Delta'$ in appendix Fig. 45, used in appendix Fig. 46). For example, $b : \mathbb{N}; \hat{a} : \mathbb{N}; x : (1 \wedge (b = 0 + 0)) + \exists b' : \mathbb{N}.\ \mathsf{Nat}(b') \wedge (b = b' + 0 + 1) \vdash x \Leftarrow P \ / \ \_ \dashv \hat{a} : \mathbb{N}{=}b$ where $\quad P = (1 \wedge (\hat{a} = 0)) \quad + \exists a' : \mathbb{N}.\ \mathsf{Nat}(a') \wedge (\hat{a} = a' + 1)$.

Next, we cover the algorithmic value and spine typing rules (Fig. 14) in detail.

Typechecking values. Because there is no stand-alone algorithmic version of $\leq^{+}\rightsquigarrow$L (recall that, in algorithmic subtyping, we eagerly extract immediately under polarity shifts), the rule Alg$\Leftarrow$Var clarifies why we require types in contexts to have already been subjected to extraction. With eager extraction in subtyping under polarity shifts, but without eager type extraction for program variables, we would not be able to extract any assumptions from Q in the (algorithmic) subtyping premise.

Rule Alg$\Leftarrow\exists$ generates a fresh existential variable which ultimately gets solved within the same phase. Its solution is eagerly applied to the input type and output constraints, so we pop it off of the output context (as it is no longer needed).

Rule Alg$\Leftarrow\mu$ unrolls the inductive type, checks the inductive type's injected value against the unrolled type, and passes along the constraints and solutions.

Rule Alg$\Leftarrow\wedge$ delays verifying the validity of the conjoined proposition ϕ until it is grounded. As explained in the example above, existential variables can be solved via propositions generated by algorithmic type unrolling. This is the role of the propositional instantiation judgment used in the second premise: it simply checks whether the proposition is of the form $\hat{a} = t$ where t is ground, in which case it solves $\hat{a}$ as t (rule Inst), and otherwise it does nothing (rule NoInst). If the proposition does solve an existential variable, then the $[\Delta']\phi$ part of the constraint is a trivial equation, but ϕ could be a non-ground proposition unrelated to unrolling, in which case $\Delta' = \Delta''$, whose solutions have not yet been applied to the input ϕ.

Rule Alg$\Leftarrow\downarrow$ does not have a premise for typechecking the thunked expression (unlike Decl$\Leftarrow\downarrow$). Instead, the rule delays this typechecking constraint until its existential variables are solved. For example, in

$$\cdot; \cdot; \cdot \vdash \langle \{\mathtt{return}\ \langle\rangle\}, \mathtt{into}(\mathtt{inj}_1\ \langle\rangle) \rangle \Leftarrow \exists a : \mathbb{N}.\ (\downarrow\uparrow(1 \wedge (a = 0))) \times \mathsf{Nat}(a) \ / \ \chi \dashv \cdot$$

the output constraint χ has $[0/\hat{a}](\mathtt{return}\ \langle\rangle \Leftarrow \uparrow(1 \wedge (\hat{a} = 0)))$, where the index solution 0 to the $\hat{a}$ introduced by Alg$\Leftarrow\exists$ is found only in typechecking the second component of the pair.

Rule Alg$\Leftarrow$1 says that $\langle\rangle$ checks against 1, which solves nothing, and there are no further constraints to check.

In rule Alg$\Leftarrow\times$, we check the first component v_1, threading through solutions found there in checking the second component v_2. Checking the second component can solve further existential variables in the first component's constraints χ_1, but solutions are eagerly applied, so in the conclusion we apply all the solutions only to χ_1.

Rule Alg$\Leftarrow+_k$ checks the k-injected value v_k against the sum's kth component type, and passes along the constraints and solutions.

Typechecking spines. Rule AlgSpine$\forall$, similarly to Alg$\Leftarrow\exists$, generates a fresh existential variable that ultimately gets solved in typechecking a value (in this case the spine's corresponding value). As usual, we pop off the solution because solutions are eagerly applied.

In rule AlgSpineApp, we typecheck the value v, outputting constraints χ and solutions Δ''. We thread these solutions through when checking s, the rest of the spine, ultimately outputting constraints χ' and solutions Δ'. The context Δ' may have more solutions than Δ'', and we eagerly apply solutions, so we need only apply Δ' to the first value's constraints χ.

In AlgSpine$\supset$, we check the spine s but add the guarding proposition ϕ to the list of constraints to verify later (applying the solutions Δ' found when checking the spine).

In AlgSpineNil, nothing gets solved, so we output the input algorithmic context Δ. Nothing needs to be verified, so we output the trivial constraint $\top$.

7 ALGORITHMIC METATHEORY

We prove that the algorithmic system (Sec. 6) is decidable, as well as sound and complete with respect to the declarative system (Sec. 4).

7.1 Decidability

We have proved that all algorithmic judgments are decidable (appendix Sec. G). The proofs rely on fairly simple metrics for the various judgments, which involve a simple size function (appendix Figs. 57 and 58) and counting the number of subtyping constraints W in typing constraint lists χ. We show that, for each algorithmic rule, every premise is smaller than the conclusion, according to the metrics. The most interesting lemmas we use state that the constraints output by algorithmic equivalence, subtyping and program typing decrease in size (appendix Lemmas G.9, G.10, and G.12). For example:

Lemma 7.1 (Program Typing Shrinks Constraints). *(Lemma G.12 in appendix)*

(1) If $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \ / \ \chi \dashv \Delta'$, *then* $sz(\chi) \leq sz(v)$.
(2) If $\Theta; \Delta; \Gamma; [N] \vdash s \gg \uparrow P \ / \ \chi \dashv \Delta'$, *then* $sz(\chi) \leq sz(s)$.

7.2 Algorithmic Soundness

We show that the algorithmic system is sound with respect to the declarative system. Since the algorithmic system is designed to mimic the judgmental structure of the declarative system, soundness (and completeness) of the algorithmic system is relatively straightforward to prove (the real difficulty lies in designing the overall system to make this the case).

Soundness of algorithmic subtyping says that, if the subtyping algorithm solves indexes under which its verification conditions hold, then subtyping holds declaratively under the same solutions:

Theorem 7.2 (Soundness of Algorithmic Subtyping). *(Thm. I.4 in appendix)*

(1) If $\Theta; \Delta \vdash P <:^+ Q \ / \ W \dashv \Delta'$ *and* $\Theta; \Delta \vdash Q \ \mathsf{type}[\Xi]$ *and* P ground *and* $\Theta \vdash \Delta' \longrightarrow \Omega$ *and* $\Theta \models [\Omega]W$, *then* $\Theta \vdash P \leq^+ [\Omega]Q$.
(2) If $\Theta; \Delta \vdash N <:^- M \ / \ W \dashv \Delta'$ *and* $\Theta; \Delta \vdash N \ \mathsf{type}[\Xi]$ *and* M ground *and* $\Theta \vdash \Delta' \longrightarrow \Omega$ *and* $\Theta \models [\Omega]W$, *then* $\Theta \vdash [\Omega]N \leq^- M$.

We prove (appendix Thm. I.4) the soundness of algorithmic subtyping by way of two intermediate (sound) subtyping systems: a declarative system that eagerly extracts under shifts, and a semideclarative system that also eagerly extracts under shifts, but outputs constraints W in the same way as algorithmic subtyping, to be checked by the semideclarative judgment $\Theta \rhd W$ (that we prove sound with respect to the algorithmic $\Theta \models W$). We define a straightforward subtyping constraint equivalence judgment $\Theta \rhd W \leftrightarrow W'$, that uses the proposition and type equivalence mentioned in Sec. 4.3, to transport semideclarative to algorithmic subtyping constraint verification (and the other way around for algorithmic subtyping completeness): if $\Theta \rhd W$ and $\Theta \rhd W \leftrightarrow W'$, then $\Theta \rhd W'$ (appendix Lemma H.27).

As a consequence of polarization, the soundness of head, bound expression, expression, and match typing can be stated relatively simply. The typing soundness of values and spines says that if Ω completes the algorithm's solutions such that the algorithm's constraints hold, then the typing of the value or spine holds declaratively with Ω's solutions.

Theorem 7.3 (Alg. Typing Sound). *(Thm. I.5 in appendix)*

(1) If $\Theta; \Gamma \rhd h \Rightarrow P$, then $\Theta; \Gamma \vdash h \Rightarrow P$.
(2) If $\Theta; \Gamma \rhd g \Rightarrow \uparrow P$, then $\Theta; \Gamma \vdash g \Rightarrow \uparrow P$.
(3) If $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \ /\ \chi \dashv \Delta'$ and $\Theta; \Delta \vdash P \ \mathsf{type}[\Xi]$ and $\Theta \vdash \Delta' \longrightarrow \Omega$ and $\Theta; \Gamma \lhd [\Omega]\chi$, then $\Theta; \Gamma \vdash [\Omega]v \Leftarrow [\Omega]P$.
(4) If $\Theta; \Gamma \rhd e \Leftarrow N$, then $\Theta; \Gamma \vdash e \Leftarrow N$.
(5) If $\Theta; \Gamma; [P] \rhd \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$, then $\Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$.
(6) If $\Theta; \Delta; \Gamma; [N] \vdash s \gg \uparrow P \ /\ \chi \dashv \Delta'$ and $\Theta; \Delta \vdash N \ \mathsf{type}[\Xi]$ and $\Theta \vdash \Delta' \longrightarrow \Omega$ and $\Theta; \Gamma \lhd [\Omega]\chi$, then $\Theta; \Gamma; [[\Omega]N] \vdash [\Omega]s \gg [\Omega]\uparrow P$.

We prove (appendix Thm. I.5) algorithmic typing is sound by a straightforward lexicographic induction on, first, program term structure, and, second, input type size. We happen not to use an intermediate system for this proof, but an intermediate system is very helpful if not indispensable for proving algorithmic typing completeness, discussed next.

7.3 Algorithmic Completeness

We show that the algorithmic system is complete with respect to the declarative system. The declarative system can conjure index solutions that are different from the algorithm's solutions, but they must be equal according to the logical context. We capture this with *relaxed* context extension $\Theta \vdash \Delta \xrightarrow{\sim} \Delta'$, similar to (non-relaxed) context extension ($\Theta \vdash \Delta \longrightarrow \Delta'$) but solutions in Δ may change to equal (under Θ) solutions in Δ':

$$\frac{\Theta \vdash \Delta \xrightarrow{\sim} \Delta' \qquad \Theta \vdash t = t' \ \mathsf{true}}{\Theta \vdash \Delta, \hat{a} : \tau{=}t \xrightarrow{\sim} \Delta', \hat{a} : \tau{=}t'}$$

Algorithmic completeness says our subtyping algorithm verifies any declarative subtyping. Since declarative subtyping does not eagerly extract from types inside shifts in assumptive positions, but algorithmic subtyping does, the conclusion involves extraction from the given ground type. For example, the equivalence of the algorithmic solutions Δ' to the indexes in Ω for which subtyping declaratively holds may depend on extracted assumptions like Θ_P in part (1) just below.

Theorem 7.4 (Completeness of Algorithmic Subtyping). *(Thm. J.14 in appendix)*

(1) If $\Theta \vdash P \leq^+ [\Omega]Q$ and $\Theta; \Delta \vdash Q \ \mathsf{type}[\Xi]$ and P ground and $[\Delta]Q = Q$ and $\Theta \vdash \Delta \xrightarrow{\sim} \Omega$, then there exist P', Θ_P, W, and Δ' such that $\Theta, \Theta_P; \Delta \vdash P' <:^+ Q \ /\ W \dashv \Delta'$ and $\Theta, \Theta_P \vdash \Delta' \xrightarrow{\sim} \Omega$ and $\Theta, \Theta_P \models [\Omega]W$ and $\Theta \vdash P \rightsquigarrow^+ P' \ [\Theta_P]$.
(2) If $\Theta \vdash [\Omega]N \leq^- M$ and $\Theta; \Delta \vdash N \ \mathsf{type}[\Xi]$ and M ground and $[\Delta]N = N$ and $\Theta \vdash \Delta \xrightarrow{\sim} \Omega$, then there exist M', Θ_M, W, and Δ' such that $\Theta, \Theta_M; \Delta \vdash N <:^- M' \ /\ W \dashv \Delta'$ and $\Theta, \Theta_M \vdash \Delta' \xrightarrow{\sim} \Omega$ and $\Theta, \Theta_M \models [\Omega]W$ and $\Theta \vdash M \rightsquigarrow^- M' \ [\Theta_M]$.

We prove (appendix Thm. J.14) completeness of algorithmic subtyping by using, in a similar way, the same intermediate systems used to prove soundness. However, it's more complicated. Indexes in semideclarative and algorithmic constraints may be syntactically different but logically and semantically equal. More crucially, to prove the completeness of algorithmic with respect to semideclarative typing, we need to prove that algorithmic subtyping solves all value-determined indexes of input types that are not necessarily ground:

Lemma 7.5 (Sub. Solves Val-det.). *(Lemma J.5 in appendix)*

(1) If $\Theta; \Delta \vdash P <:^+ Q \;/\; W \dashv \Delta'$ *and* P ground *and* $\Theta; \Delta \vdash Q$ type$[\Xi]$,
then for all $(\hat{a} : \tau) \in \Xi$, *there exists* t *such that* $\Theta \vdash t : \tau$ *and* $(\hat{a} : \tau{=}t) \in \Delta'$.
(2) If $\Theta; \Delta \vdash M <:^- N \;/\; W \dashv \Delta'$ *and* N ground *and* $\Theta; \Delta \vdash M$ type$[\Xi]$,
then for all $(\hat{a} : \tau) \in \Xi$, *there exists* t *such that* $\Theta \vdash t : \tau$ *and* $(\hat{a} : \tau{=}t) \in \Delta'$.

We prove (appendix Lemma J.5) this by straightforward induction on the given subtyping derivation, using a similar lemma for type equivalence (appendix Lemma J.4).

We use extraction to achieve a complete subtyping algorithm. For example, the following holds declaratively without extraction but instead using $\leq^+\wedge$L (this rule is not in our system; see Sec. 4.5):

$$a : \mathbb{N}, b : \mathbb{N} \vdash \underbrace{[\hat{c} : \mathbb{N}{=}b]\big(\mathsf{Nat}(\hat{c}) \to (\hat{c} = b) \supset \uparrow 1\big)}_{\mathsf{Nat}(b) \to (b = b) \supset \uparrow 1} \leq^- \big(\mathsf{Nat}(a) \wedge (a = b)\big) \to \uparrow 1$$

However, checking function argument subtyping first, the non-extractive algorithm solves $\hat{c}$ to a (not b) and outputs a verification condition needing $a = b$ to hold under no logical assumptions, which is invalid. Our system instead extracts $a = b$ from the supertype; the algorithmic solution a for $\hat{c}$ and the declarative choice b for $\hat{c}$ are equal under this assumption ($a = b$).

Finally, we prove the completeness of algorithmic typing. Like algorithmic typing soundness, again due to focusing, the head, bound expression, expression, and pattern matching parts are straightforward to state. But, because algorithmic function application may instantiate indexes different but logically equal to those conjured (semi)declaratively, bound expressions may algorithmically synthesize a type (judgmentally) equivalent to the type it synthesizes declaratively.

We introduced logical context equivalence in Sec. 4.3. Other than in proving that type equivalence implies subtyping, logical context equivalence is used in proving the completeness of algorithmic typing (in particular, we use appendix Lemma H.17 to swap logically equivalent logical contexts in semideclarative typing derivations). The type P' in the output of the algorithm in part (6) below can have different index solutions (output Δ') that are logically equal (under Θ) to the solutions in Ω which appear in the declaratively synthesized P. However, P and P' necessarily have the same structure, so $\Theta \vdash P \equiv^+ [\Omega]P'$. Therefore, a bound expression may (semi)declaratively synthesize a type that is judgmentally equivalent to the type synthesized algorithmically. We then extract different but logically equivalent logical contexts from the (equivalent) types synthesized by a bound expression.

As such, algorithmic typing completeness is stated as follows:

Theorem 7.6 (Alg. Typing Complete). *(Thm. J.21 in appendix)*

(1) If $\Theta; \Gamma \vdash h \Rightarrow P$, *then* $\Theta; \Gamma \rhd h \Rightarrow P$.
(2) If $\Theta; \Gamma \vdash g \Rightarrow \uparrow P$, *then there exists* P' *such that* $\Theta; \Gamma \rhd g \Rightarrow \uparrow P'$ *and* $\Theta \vdash P \equiv^+ P'$.
(3) If $\Theta; \Gamma \vdash v \Leftarrow [\Omega]P$ *and* $\Theta; \Delta \vdash P$ type$[\Xi]$ *and* $[\Delta]P = P$ *and* $\Theta \vdash \Delta \overset{\sim}{\longrightarrow} \Omega$,
then there exist χ *and* Δ' *such that* $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \;/\; \chi \dashv \Delta'$
and $\Theta \vdash \Delta' \overset{\sim}{\longrightarrow} \Omega$ *and* $\Theta; \Gamma \lhd [\Omega]\chi$.
(4) If $\Theta; \Gamma \vdash e \Leftarrow N$, *then* $\Theta; \Gamma \rhd e \Leftarrow N$.
(5) If $\Theta; \Gamma; [P] \vdash \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$, *then* $\Theta; \Gamma; [P] \rhd \{r_i \Rightarrow e_i\}_{i \in I} \Leftarrow N$.
(6) If $\Theta; \Gamma; [[\Omega]N] \vdash s \gg \uparrow P$ *and* $\Theta; \Delta \vdash N$ type$[\Xi]$ *and* $[\Delta]N = N$ *and* $\Theta \vdash \Delta \overset{\sim}{\longrightarrow} \Omega$,
then there exist P', χ, *and* Δ' *such that* $\Theta; \Delta; \Gamma; [N] \vdash s \gg \uparrow P' \;/\; \chi \dashv \Delta'$
and $\Theta \vdash \Delta' \overset{\sim}{\longrightarrow} \Omega$ *and* $\Theta; \Gamma \lhd [\Omega]\chi$ *and* $\Theta \vdash P \equiv^+ [\Omega]P'$.

We prove (appendix Thm. J.21) algorithmic typing completeness by way of an intermediate, semideclarative typing system, that is essentially the same as declarative typing in that it conjures indexes, but differs in a way similar to algorithmic typing: it outputs constraints χ and only verifies them (via semideclarative $\Theta; \Gamma \;\widetilde{\lhd}\; \chi$ as opposed to algorithmic $\Theta; \Gamma \lhd \chi$) immediately upon completion of focusing phases. Similarly to the proof of algorithmic subtyping completeness, we transport (appendix Lemma J.18) the semideclarative verification of typing constraints over a straightforward typing constraint equivalence judgment $\Theta; \Gamma \lhd \chi \leftrightarrow \chi'$ that uses the subtyping constraint equivalence ($\Theta \rhd W \leftrightarrow W'$) and type equivalence judgments.

To prove that algorithmic typing is complete with respect to semideclarative typing, we use the fact that typing solves all value-determined indexes in input types of focusing phases. This fact is similar to the fact that subtyping solves the value-determined indexes of non-ground types (used in the algorithmic subtyping completeness proof), but the interaction between value-determined indexes and unrolling introduces some complexity: unrolling a refined inductive type does not preserve the type's Ξ. Therefore, we had to split the value typechecking part into the mutually recursive parts (1) and (2); part (3) depends on parts (1) and (2) but not vice versa.

Lemma 7.7 (Typing Solves Val-det.). *(Lemma J.19 in appendix)*

(1) Suppose $\Delta = \Delta_1, \hat{a} : \tau, \Delta_2$. *If* $\Xi; \Theta; \Delta \vdash \{\nu : G[\mu F] \mid \beta\,(G\ (\mathsf{fold}_F\ \alpha)\ \nu) =_\tau \hat{a}\} \doteq Q$ *and* $\Theta; \Delta \vdash G\ \mathsf{functor}[\Xi_G]$ *and* $(\hat{a} : \tau) \notin \Xi_G$ *and* $\Theta; \Delta; \Gamma \vdash v \Leftarrow Q \ /\ \chi \dashv \Delta'$ *and* $[\Delta]Q = Q$ *and* $\Theta \vdash \Delta' \xrightarrow{\sim} \Omega$ *and* $\Theta; \Gamma \mathrel{\widetilde{\lhd}} [\Omega]\chi$, *then there exists* t *such that* $\Theta \vdash t : \tau$ *and* $(\hat{a} : \tau{=}t) \in \Delta'$.

(2) If $\Theta; \Delta; \Gamma \vdash v \Leftarrow P \ /\ \chi \dashv \Delta'$ *and* $\Theta; \Delta \vdash P\ \mathsf{type}[\Xi_P]$ *and* $[\Delta]P = P$ *and* $\Theta \vdash \Delta' \xrightarrow{\sim} \Omega$ *and* $\Theta; \Gamma \mathrel{\widetilde{\lhd}} [\Omega]\chi$, *then for all* $(\hat{a} : \tau) \in \Xi_P$, *there exists* t *such that* $\Theta \vdash t : \tau$ *and* $(\hat{a} : \tau{=}t) \in \Delta'$.

(3) If $\Theta; \Delta; \Gamma; [N] \vdash s \gg \uparrow P \ /\ \chi \dashv \Delta'$ *and* $\Theta; \Delta \vdash N\ \mathsf{type}[\Xi_N]$ *and* $[\Delta]N = N$ *and* $\Theta \vdash \Delta' \xrightarrow{\sim} \Omega$ *and* $\Theta; \Gamma \mathrel{\widetilde{\lhd}} [\Omega]\chi$, *then for all* $(\hat{a} : \tau) \in \Xi_N$, *there exists* t *such that* $\Theta \vdash t : \tau$ *and* $(\hat{a} : \tau{=}t) \in \Delta'$.

The proof (appendix Lemma J.19) of part (1) boils down to inversion on the propositional instantiation judgment $\Theta; \Delta \vdash \phi\ \mathsf{inst} \dashv \Delta'$ in the unit case of unrolling where ϕ necessarily has the form $\hat{a} = t$ with t ground, due to the invariant that algebras are ground.

8 RELATED WORK

Typing refinement. As far as we know, Constable [1983] was first to introduce the concept of refinement types (though not by that name) as logical subsets of types, writing `{x:A|B}` for the *subset type* classifying terms `x` of type `A` that satisfy proposition `B`. Freeman and Pfenning [1991] introduced type refinement to the programming language ML via *datasort* refinements—inclusion hierarchies of ML-style (algebraic, inductive) datatypes—and intersection types for Standard ML: they showed that full type inference is decidable under a refinement restriction, and provided an algorithm based on abstract interpretation. The dangerous interaction of datasort refinements, intersection types, side-effects, and call-by-value evaluation was first dealt with by Davies and Pfenning [2000] via a value restriction for intersection introduction; they also presented a bidirectional typing algorithm.

Dependent ML (DML) [Xi 1998; Xi and Pfenning 1999] extended the ML type discipline parametrically by *index* domains, which can be restricted to decidable logics. DML used a bidirectional type system with index refinements for a variant of ML, capable of checking properties ranging from in-bound array access [Xi and Pfenning 1998] to program termination [Xi 2002]. DML, similarly to our system, collects constraints from the program and passes them to a constraint solver. This is also the approach of systems like Stardust [Dunfield 2007] (which combines datasort and index refinement) and those with liquid types [Rondon et al. 2008]. The latter are based on a Hindley–Milner approach; typically, Hindley–Damas–Milner inference algorithms [Hindley 1969; Milner 1978; Damas and Milner 1982] generate typing constraints to be verified [Heeren et al. 2002].

Due to issues with existential index instantiation, Xi's approach [Xi 1998] (incompletely) translated programs into a let-normal form [Sabry and Felleisen 1993] before typing them, but Dunfield [2007] provided a complete let-normal translation for similar issues. The system in this paper is already in let-normal form.

Liquid types. Rondon et al. [2008] introduced *logically qualified data types*, that is, *liquid types*, in a system that extends Hindley–Milner type inference to infer (by abstract interpretation) refinements based on built-in or programmer-provided refinement templates. Kawaguchi et al. [2009] introduced *recursive refinement* via sound and terminating *measures* on algebraic inductive data types; they also introduced *polymorphic refinement*. Vazou et al. [2013] generalize recursive and polymorphic refinement into a single, more expressive mechanism: *abstract refinement*. We plan to consider polymorphism and abstract refinements in future work.

The papers cited in the previous paragraph deal with call-by-value languages, and Liquid Haskell was initially unsound due to Haskell's lazy evaluation [Vazou et al. 2014]. Vazou et al. [2014] regained type soundness by imposing semantic restrictions on subtyping and let-binding. In their algorithmic subtyping, there is exactly one rule, $\leq$-Base-D, which pertains to refinements of base types (integers, booleans and so on) and inductive data types; however, these types have a *well-formedness restriction*, namely, that the refinement predicates have the type of boolean expressions that reduce to finite values. But this restriction alone does not suffice for soundness under laziness and divergence. As such, their algorithmic typing rule T-Case-D, which combines let-binding and pattern matching, uses an operational semantics to approximate whether or not the bound expression terminates. If the bound expression might diverge, then so might the entire case expression; otherwise, it checks each branch in a context that assumes the expression reduces to a (potentially infinite Haskell) value.

We also have a type well-formedness restriction, but it is purely syntactic, and only on index quantification, requiring them to be associated with a fold that is necessarily decidable by virtue of a systematic phase distinction between the index level and the program level. Further, via type polarization, our let-binding rule requires the bound expression to return a value, we only allow value types in our program contexts, and we cannot extract index information across polarity shifts (such as in a suspended computation). Therefore, in our system, there is no need to stratify our types according to an approximate criterion; rather, we exploit the systematic distinction between positive (value) types and negative (computation) types, that Levy [2004] designed to be semantically well-behaved. We suspect that liquid types' divergence-based stratification is indirectly grappling with logical polarity. Because divergence-based stratification is peculiar to the specific effect of nontermination, it is unclear how their approach may extend to other effects. Since we are working with CBPV, our system is already in a good position to handle effects other than nontermination.

Contract calculi. Software *contracts* express program properties in the same language as the programs themselves; Findler and Felleisen [2002] introduced contracts for *runtime* verification of higher-order functional programs. These *latent* contracts are not types, but *manifest* contracts are [Greenberg et al. 2010]. Manifest contracts are akin to refinement types. Indeed, Vazou et al. [2013] sketch a proof of type soundness for a liquid type system by translation from liquid types to the manifest contract calculus F_H of Belo et al. [2011]. However, there is no explicit translation back, from F_H to liquid typing. They mention that the translated terms in F_H do not have *upcasts* because the latter in F_H are logically related to identity functions if they correspond to static subtyping (as they do in the liquid type system): an upcast lemma. Presumably, this facilitates a translation from F_H back to liquid types. However, there are technical problems in F_H that break type soundness and the logical consistency of the F_H contract system; Sekiyama et al. [2017] fix these issues, resulting in the system F_H^σ, but do not consider subtyping and subsumption, and do not prove an upcast lemma.

Bidirectional typing. Bidirectional typing [Pierce and Turner 2000] is a popular way to implement a wide variety of systems, including dependent types [Coquand 1996; Abel et al. 2008], contextual types [Pientka 2008], and object-oriented languages [Odersky et al. 2001]. The bidirectional system of Peyton Jones et al. [2007] supports higher-rank polymorphism. Dunfield and Krishnaswami [2013] also present a bidirectional type system for higher-rank polymorphism, but framed more proof theoretically; Dunfield and Krishnaswami [2019] extend it to a richer language with existentials, indexed types, sums, products, equations over type variables, pattern matching, polarized subtyping, and principality tracking. The bidirectional system of this paper uses logical techniques similar to the systems of Dunfield and Krishnaswami, but it does not consider polymorphism. A survey paper [Dunfield and Krishnaswami 2021] includes some discussion of bidirectional typing's connections to proof theory. Basically, good bidirectional systems tend to distinguish checking and synthesis terms or proofs according to their form, such as normal or neutral.

Proof theory, polarization, focusing and analyticity. The concept of polarity most prominent in this paper dates back to Andreoli's work on focusing for tractable proof search [Andreoli 1992] and Girard's work on unifying classical, intuitionistic, and linear logic [Girard 1993]. Logical polarity and focusing have been used to explain many common phenomena in programming languages. We mentioned in the overview that Zeilberger [2009] explains the value and evaluation context restrictions in terms of focusing; and Krishnaswami [2009] explains pattern matching as (proof terms of) the left inversion phase of focused systems (also, its system is bidirectional). More broadly, Downen [2017] discusses many logical dualities common in programming languages.

Brock-Nannestad et al. [2015] study the relation between polarized intuitionistic logic and CBPV. They obtain a bidirectionally typed system of natural deduction related to a variant of the focused sequent calculus *LJF* [Liang and Miller 2009] by η-expansion (for inversion phases). Espírito Santo [2017] does a similar study, but starts with a focused sequent calculus for intuitionistic logic much like the system of Simmons [2014] (but without positive products), proves it equivalent to a natural deduction system (we think the lack of positive products helps establish this equivalence), and defines, also via η-expansion, a variant of CBPV in terms of the natural deduction system. Our system is not in the style of natural deduction, but rather sequent calculus. We think that our system relates to CBPV in a similar way—via η-expansion—but we do not prove it in this paper, because we focus on proving type soundness and algorithmic decidability, soundness and completeness.

Barendregt et al. [1983] discovered that a program that typechecks (in a system with intersection types) using subtyping, can also be checked without using subtyping, if the program is sufficiently η-expanded. An analogous phenomenon involving identity coercion was studied by Zeilberger [2009] in a focused setting. Similarly, our ability to place subtyping solely in (value) variable typechecking is achievable due to the focusing (and let-normality) of our system.

Interpreting Kant, Martin-Löf [1994] considers an *analytic* judgment to be one that is derivable using information found only in its inputs (in the sense of the bidirectional modes, input and output). A *synthetic* judgment, in contrast, requires us to look beyond the inputs of the judgment in order to find a derivation. The metatheoretic results for our algorithmic system demonstrate that our judgments are analytic, *except* the judgment $\Theta \vdash \phi$ true, which is verified by an SMT solver. As such, our system may be said to be analytic modulo an external SMT solver. Focusing, in proper combination with bidirectional typing (annotations), let-normality and our Ξ mechanism, guarantees that all information needed to generate verification conditions suitable for an SMT solver may be found in the inputs to judgments.

Dependent types. Dependent types, introduced by Martin-Löf [1971, 1975], are a key conceptual and historical precursor to index refinement types. Dependent types may depend on arbitrary program terms, not only terms restricted to indexes. This is highly expressive, but undecidable in languages with divergence. The main difference between refinement and dependent typing design is that refinement typing attempts to increase the expressivity of a highly automatic type system, whereas dependent typing attempts to increase the automation of a highly expressive type system.

Many dependent type systems impose their own restrictions for the sake of decidability. In Cayenne [Augustsson 1998], typing can only proceed a given number of steps. All well-typed programs in Epigram [McBride and McKinna 2004] are required to terminate so that its type equivalence is decidable. Epigram, and other systems [Chen and Xi 2005; Licata and Harper 2005], allow programmers to write explicit proofs of type equivalence.

Systems like ATS [Xi 2004] and $F^\star$ [Swamy et al. 2016] can be thought of as combining refinement and dependent types. These systems aim to bring the best of both refinement and dependent types, but ATS is more geared to practical, effectful functional programming (hence refinement types), while $F^\star$ is more geared to formal verification and dependent types. Unlike our system in this paper, they allow the programmer to provide proofs. The overall design of ATS is closer to our system than that of $F^\star$, due to its phase distinction between statics and dynamics; but it allows the programmer to write (in the language itself) proofs in order to simplify or eliminate constraints for the (external) constraint solver: Xi calls this *internalized constraint solving*. It should be possible to internalize constraint solving to some extent in our system.

Both ATS and $F^\star$ have a CBV semantics, which is inherently monadic [Moggi 1989b]. Our system is a variant of CBPV, which subsumes both CBV and CBN. These systems consider effects other than divergence, like exceptions, mutable state and input/output, which we hope to add to our system in future work. The system $F^\star$ allows for termination metrics other than strong induction on naturals, such as lexicographic induction, but we think it would be straightforward to add such metrics to our system; we can also build on the Ξ mechanism, tracking and resolving value-determined dependencies in index propositions.

Data abstraction and category theory. Categorically, inductive types are initial algebras of endofunctors. We only consider certain *polynomial* endofunctors, which specify tree-shaped data structures. Objects (in the sense of object-oriented programming) are dual to inductive types in that, categorically, they are final coalgebras of endofunctors [Cook 2009]. A consideration of categorical duality leads us to a natural (if naïve) question: if we can build a well-behaved system that refines algebraic data types by algebras, could it mean anything to refine objects by coalgebras?

Our rolled refinement types refine *type* constructors μF. Sekiyama et al. [2015], again in work on manifest contracts, compare this to refining (types of) *data* constructors, and provide a translation from type constructor to data constructor refinements. Type constructor refinements (such as our $\{\nu : \mu F \mid (\mathrm{fold}_F\ \alpha)\ \nu =_\tau t\}$) are *easier for the programmer to specify*, but data constructor refinements (such as the output types of our unrolling judgment) are *easier to verify automatically*. Sekiyama et al. [2015] say that their translation from type to data constructor refinements is closely related to the work of Atkey et al. [2012] on refining inductive data in (a fibrational interpretation of) dependently typed languages. Atkey et al. [2012] provide "explicit formulas" that compute inductive characterizations of type constructor refinements. These semantic formulas resemble our syntactic unrolling judgment, which we may view as a translation from type refinements to data constructor refinements.

Ornaments [McBride 2011] describe how inductive types with different logical or ornamental properties can be systematically related using their algebraic and structural commonalities. Practical work in ornaments seems mostly geared toward code reuse [Dagand and McBride 2012], code refactoring [Williams and Rémy 2017] and such. In contrast, this paper focuses on incorporating similar ideas in a foundational liquid refinement typing algorithm.

Melliès and Zeilberger [2015] provide a categorical theory of type refinement in general, where functors are considered to be type refinement systems. This framework is based on Reynolds's distinction between *intrinsic* (or Church) and *extrinsic* (or Curry) views of typing [Reynolds 1998]. We think that our system fits into this framework, but haven't confirmed it formally. This is most readily seen in the fact that the semantics of our refined system is simply the semantics of its erasure (*intrinsic* to unrefined typing derivations), which are unaffected by the indexes that refine the types of program terms, which are about the *extrinsic* properties of (erased) programs.

9 CONCLUSION AND FUTURE WORK

We have presented a declarative system for index-based recursive refinement typing that is logically designed, semantically sound, and theoretically implementable. We have proved that our declarative system is sound with respect to an elementary domain-theoretic denotational semantics, which implies that our system is logically consistent and enforces program termination. We have also presented an algorithmic system and proved that it is decidable, as well as sound and complete with respect to the declarative system.

We plan to add parametric polymorphism in future work. We hope to extend our system with some kind of mechanism for refinement abstraction [Vazou et al. 2013], which would likely include algebra abstraction. It is also of interest to see how other features of liquid typing, like refinement inference with templates, may fit in our system.

Because our system lacks higher-order sorts, it cannot directly specify relations between data in structurally different positions of an inductive type, such as that integers in a list are in increasing order. We plan to add higher-order sorts in future work. We also plan to allow the use of multiple measures on inductive types (so we can specify, for example, the type of length-n lists of naturals in increasing order). It would also be interesting to experiment with and extend our Ξ mechanism. By allowing quantification over indexes in propositions whose only variable dependencies are *value-determined*, we think we can encode more termination metrics purely at the index level. We could then, for example, obviate the function auxmerge that we used to define mergesort (Sec. 3), which has a ghost argument: the third (program level) natural number. We think it would be straightforward to add rules for recursive expressions that have different well-founded orders (such as lexicographic) for verifying termination.

In future work, we hope to apply our type refinement system (or future extensions of it) to various domains, from static time complexity analysis [Wang et al. 2017] to resource management using linear types. Eventually, we hope to be able to express, for example, that a program terminates within a worst-case amount of time. Our system is parametric in the index domain, provided it satisfies some basic properties. Different index domains may be suitable for different applications. We also hope to add more effects, such as input/output and mutable reference cells. CBPV is built for effects, but we expect our refinement layer to add interesting complexities related to the interaction between effects and indexes.

Our system seems complex, but its metatheoretic proofs are largely straightforward, if lengthy (at least as presented). A principal source of this apparent complexity is the proliferation of judgments. But this proliferation is a strength, because it helps us organize different forms of knowledge [Martin-Löf 1996] or (from a Curry–Howard perspective) phases or parts of an implementation.

Our system focuses on the feature of recursive refinement of inductive data, and does not include key typing features expected of a realistic programming language (such as additional effects and polymorphism, which we hope to add in future work). Adding such expressive features tends to significantly affect the metatheory and the techniques used to prove it. We hope to reflect on the development of our proofs (including those for systems with polymorphism [Dunfield and Krishnaswami 2013]) in search of abstractions that may help designers of practical, general-purpose functional languages to establish crucial metatheoretic properties.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through Discovery Grant RGPIN–2018–04352, as well as the European Research Council through Consolidator Grant Typefoundry 10100227.

REFERENCES

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2008. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction (MPC'08) (LNCS)*, Vol. 5133. Springer, 29–56.

Jean-Marc Andreoli. 1992. Logic programming with focusing proofs in linear logic. *J. Logic and Computation* 2, 3 (1992), 297–347.

Robert Atkey, Patricia Johann, and Neil Ghani. 2012. Refining Inductive Types. *Logical Methods in Computer Science* Volume 8, Issue 2 (June 2012). https://doi.org/10.2168/LMCS-8(2:9)2012

Lennart Augustsson. 1998. Cayenne—a Language with Dependent Types. In *ICFP*. 239–250.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symbolic Logic* 48, 4 (1983), 931–940.

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. *Satisfiability modulo theories* (1 ed.). Number 1 in Frontiers in Artificial Intelligence and Applications. IOS Press, 825–885. https://doi.org/10.3233/978-1-58603-929-5-825

João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. 2011. Polymorphic Contracts. In *Proceedings of the 20th European Symposium on Programming (Lecture Notes in Computer Science)*, Gilles Barthe (Ed.), Vol. 6602. Springer International Publishing, 18–37. https://doi.org/10.1007/978-3-642-19718-5_2

Taus Brock-Nannestad, Nicolas Guenot, and Daniel Gustafsson. 2015. Computation in Focused Intuitionistic Logic. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/2790449.2790528

Iliano Cervesato and Frank Pfenning. 2003. A Linear Spine Calculus. *J. Logic and Computation* 13, 5 (2003), 639–688.

Chiyan Chen and Hongwei Xi. 2005. Combining programming with theorem proving. In *ICFP*. 66–77.

James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report CUCIS TR2003-1901. Cornell University. https://hdl.handle.net/1813/5614

Robert L. Constable. 1983. Mathematics as Programming. In *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings (Lecture Notes in Computer Science)*, Edmund M. Clarke and Dexter Kozen (Eds.), Vol. 164. Springer, 116–128. https://doi.org/10.1007/3-540-12896-4_359

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 557–572. https://doi.org/10.1145/1640089.1640133

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming* 26, 1–3 (1996), 167–177.

Pierre-Evariste Dagand and Conor McBride. 2012. Transporting Functions across Ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/2364527.2364544

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *POPL*. ACM Press, 207–212.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *ICFP*. ACM Press, 198–208.

Paul Downen. 2017. *Sequent Calculus: A Logic and a Language for Computation and Duality*. Dissertation Defense. College of Arts and Sciences, University of Oregon. Available at https://www.cs.uoregon.edu/Reports/PHD-201706-Downen.pdf.

Jana Dunfield. 2007. Refined typechecking with Stardust. In *Programming Languages meets Programming Verification (PLPV '07)*. ACM Press, 21–32.

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *ICFP*. ACM Press, 429–442. arXiv:`1306.6032 [cs.PL]`.

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.* 3, POPL (2019), 9:1–9:28. https://doi.org/10.1145/3290322

Jana Dunfield and Neelakantan R. Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2021). https://doi.org/10.1145/3450952

Jana Dunfield and Frank Pfenning. 2003. Type Assignment for Intersections and Unions in Call-by-Value Languages. In *FoSSaCS*. Springer, 250–266.

José Espírito Santo. 2017. The Polarized λ-calculus. *Electronic Notes in Theoretical Computer Science* 332 (2017), 149–168. https://doi.org/10.1016/j.entcs.2017.04.010 LSFA 2016 - 11th Workshop on Logical and Semantic Frameworks with Applications (LSFA).

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *PLDI*. 237–247.

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *PLDI*. ACM Press, 268–277.

Jean-Yves Girard. 1993. On the Unity of Logic. *Annals of Pure and Applied Logic* 59, 3 (1993), 201–217. https://doi.org/10.1016/0168-0072(93)90093-s

Jean-Yves Girard. 1992. A Fixpoint Theorem in Linear Logic. (1992). Post to Linear Logic mailing list, http://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 353–364. https://doi.org/10.1145/1706299.1706341

Carl A. Gunter. 1993. *Semantics of programming languages - structures and techniques*. MIT Press.

Bob Harper and Mark Lillibridge. 1991. ML with callcc is unsound. Post to TYPES mailing list, 8 July 1991, archived at https://web.archive.org/web/20220121203611/https://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html. (1991).

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, San Francisco, California, USA, 341–354. https://doi.org/10.1145/96709.96744

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2002. *Generalizing Hindley-Milner Type Inference Algorithms*. Technical Report UU-CS-2002-031. Department of Information and Computing Sciences, Utrecht University. http://www.cs.uu.nl/research/techreps/repo/CS-2002/2002-031.pdf

R. Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. 2009. Type-Based Data Structure Verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/1542476.1542510

Neelakantan R. Krishnaswami. 2009. Focusing on Pattern Matching. In *POPL*. ACM Press, 366–378.

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, Norwell, MA, US.

Chuck Liang and Dale Miller. 2009. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science* 410, 46 (2009), 4747–4768. https://doi.org/10.1016/j.tcs.2009.07.041 Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

Daniel R. Licata and Robert Harper. 2005. *A formulation of Dependent ML with explicit equality proofs*. Technical Report CMU-CS-05-178. Carnegie Mellon University. https://doi.org/10.1184/R1/6587429.v1

Per Martin-Löf. 1971. A Theory of Types. (1971). Manuscript, Stockholm University. https://raw.githubusercontent.com/michaelt/martin-lof/master/pdfs/martin-loef1971%20-%20A%20Theory%20of%20Types.pdf.

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Per Martin-Löf. 1994. Analytic and Synthetic Judgements in Type Theory. In *Kant and Contemporary Epistemology*, Paolo Parrini (Ed.). Springer Netherlands, 87–99.

Per Martin-Löf. 1996. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic* 1, 1 (1996), 11–60.

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

Conor McBride and James McKinna. 2004. The view from the left. *J. Functional Programming* 14, 1 (2004), 69–111.

Paul-André Melliès and Noam Zeilberger. 2015. Functors Are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/2676726.2676970

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

Eugenio Moggi. 1989a. A Category-Theoretic Account of Program Modules. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 101–117.

Eugenio Moggi. 1989b. Computational Lambda-Calculus and Monads. In *Logic in Computer Science (LICS '89)*. 14–23.

Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored Local Type Inference. In *POPL*. ACM Press, 41–53.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Functional Programming* 17, 1 (2007), 1–82.

Frank Pfenning. 2008. Church and Curry: Combining intrinsic and extrinsic typing. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*. College Publications. http://www.cs.cmu.edu/~fp/papers/andrews08.pdf

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*. ACM Press, 371–382.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Prog. Lang. Sys.* 22 (2000), 1–44.

John C. Reynolds. 1998. *Theories of programming languages.* Cambridge University Press.

Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. ACM Press, 159–169.

Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp Symb. Comput.* 6, 3–4 (Nov. 1993), 289–360. https://doi.org/10.1007/BF01019462

Peter Schroeder-Heister. 1994. Definitional reflection and the completion. In *Extensions of Logic Programming (LNCS)*. Springer, 333–347.

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 3 (Feb. 2017), 36 pages. https://doi.org/10.1145/2994594

Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. 2015. Manifest Contracts for Datatypes. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*. ACM, 195–207. https://doi.org/10.1145/2676726.2676996

Robert J. Simmons. 2014. Structural Focalization. *ACM Trans. Comput. Logic* 15, 3, Article 21 (Sept. 2014). https://doi.org/10.1145/2629678

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Alan M. Turing. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2, 42 (1936), 230–265.

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 209–228. https://doi.org/10.1007/978-3-642-37036-6_13

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133903

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2004. A Concurrent Logical Framework: The Propositional Fragment. In *Types for Proofs and Programs (TYPES 2003)*. Springer LNCS 3085, 355–377.

Thomas Williams and Didier Rémy. 2017. A Principled Approach to Ornamentation in ML. *Proc. ACM Program. Lang.* 2, POPL, Article 21 (Dec. 2017). https://doi.org/10.1145/3158109

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

Hongwei Xi. 1998. *Dependent Types in Practical Programming.* Ph.D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/students/xi.pdf

Hongwei Xi. 2002. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation* 15 (Oct. 2002), 91–131.

Hongwei Xi. 2004. Applied Type System (extended abstract). In *TYPES 2003 (LNCS)*. Springer, 394–408.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL*. ACM Press, 224–235.

Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *PLDI*. 249–257.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL*. ACM Press, 214–227.

Noam Zeilberger. 2009. Refinement Types and Computational Duality. In *Programming Languages meets Programming Verification (PLPV '09)*. ACM Press, 15–26.