





Polarized Subtyping

Zeeshan Lakhani¹ (✉) , Ankush Das³, Henry DeYoung¹, Andreia Mordido² ,
and Frank Pfenning¹

¹ Carnegie Mellon University, Pittsburgh, PA, USA
{zlakhani, hdeyoung, fp}@cs.cmu.edu

² LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal
afmordido@fc.ul.pt

³ Amazon, Cupertino, CA, USA** daankus@amazon.com

Abstract. Polarization of types in call-by-push-value naturally leads to the separation of inductively defined observable values (classified by positive types), and coinductively defined computations (classified by negative types), with adjoint modalities mediating between them. Taking this separation as a starting point, we develop a semantic characterization of typing with step indexing to capture observation depth of recursive computations. This semantics justifies a rich set of subtyping rules for an equirecursive variant of call-by-push-value, including variant and lazy records. We further present a bidirectional syntactic typing system for both values and computations that elegantly and pragmatically circumvents difficulties of type inference in the presence of width and depth subtyping for variant and lazy records. We demonstrate the flexibility of our system by systematically deriving related systems of subtyping for (a) isorecursive types, (b) call-by-name, and (c) call-by-value, all using a structural rather than a nominal interpretation of types.

Keywords: Call-by-push-value · Semantic Typing · Subtyping

1 Introduction

Subtyping is an important concept in programming languages because it simultaneously allows more programs to be typed and more precise properties of programs to be expressed as types. The interaction of subtyping with parametric polymorphism and recursive types is complex and despite a lot of progress and research, not yet fully understood.

In this paper we study the interaction of subtyping with equirecursive types in *call-by-push-value* [53, 54], which separates the language of types into *positive* and *negative* layers. This polarization elegantly captures that positive types classifying observable values are *inductive*, while negative types classifying (possibly recursive) computations are *coinductive*. It lends itself to a particularly simple *semantic definition of typing* using a mixed induction/coinduction [9, 13, 22]. From this definition, we can immediately derive a form of *semantic subtyping* [15, 35, 36].

** work performed prior to joining Amazon

Concretely, we realize the mixed induction/coinduction via step-indexing and carry out our metatheory in Brotherston and Simpson’s system \mathbf{CLKID}^ω of circular proofs [14]. This includes a novel proof that syntactic versions of typing and subtyping are sound with respect to our semantic definitions. While we also conjecture that subtyping is precise (in the sense of [55]), we postpone this more syntactic property to future work.

Because our foundation is call-by-push-value, a paradigm that synthesizes call-by-name and call-by-value based on the logical principle of polarization, we obtain several additional results in relatively straightforward ways. For example, both width and depth subtyping for variant and lazy records are naturally included. Furthermore, following Levy’s interpretation of call-by-value and call-by-name functional languages into call-by-push-value, we extract subtyping relations and algorithms for these languages and prove them sound and complete. We also note that we can directly interpret the *isorecursive* types in Levy’s original formulation of call-by-push-value [53].

We further provide a systematic notion of bidirectional typing that avoids some complexities that arise in a structural type system with variant and lazy records. The resulting decision procedure for typing is quite precise and suggests clear locations for noting failure of typechecking. The combination of equirecursive call-by-push-value with bidirectional typing achieves some of the goals of refinement types [24, 34], which fit a structural system inside a generative type language. Here we have considerably more freedom and less redundancy. However, we do not yet treat intersection types or polymorphism.

We summarize our main contributions:

1. A simple semantics for types and subtyping in call-by-push-value, interpreting positive types inductively and negative types coinductively, realized via step indexing (Sections 3 and 4)
2. A new decidable system of equirecursive subtyping for call-by-push-value including width and depth subtyping for variant and lazy records (Section 4)
3. A novel application of Brotherston and Simpson’s system \mathbf{CLKID}^ω [14] of circular proofs to give a particularly elegant and flexible soundness proof for subtyping (Section 5)
4. A system of bidirectional typing that captures a straightforward and precise typechecking algorithm (Section 6), whose implementation is provided as a publicly available artifact [50]
5. A simple interpretation of Levy’s original isorecursive types for call-by-push-value [53] into our equirecursive setting (Section 7)
6. Subtyping rules for call-by-name and call-by-value, derived via Levy’s translations of such languages into call-by-push-value (Section 8)

These are followed by a discussion of related work and a conclusion. Additional material and proofs are provided in an appendix of the extended paper version [49].

2 Equirecursive Call-by-Push-Value

Call-by-push-value [53, 54] is characterized by a separation of types in *positive* τ^+ and *negative* σ^- layers, with shift modalities going back and forth between them.

The intuition is that positive types classify *observable values* v while negative types classify *computations* e .

$$\begin{aligned}\tau^+, \sigma^+ &::= \tau_1^+ \otimes \tau_2^+ \mid \mathbf{1} \mid \oplus\{\ell: \tau_\ell^+\}_{\ell \in L} \mid \downarrow\sigma^- \mid t^+ \\ \sigma^-, \tau^- &::= \tau^+ \rightarrow \sigma^- \mid \&\{\ell: \sigma_\ell^-\}_{\ell \in L} \mid \uparrow\tau^+ \mid s^-\end{aligned}$$

The usual binary product $\tau \times \sigma$ splits into two: $\tau^+ \otimes \sigma^+$ for eager, observable products inhabited by pairs of values, and $\&\{\ell: \sigma_\ell^-\}_{\ell \in L}$ for lazy, unobservable records with a finite set L of fields we can project out. Binary sums are also generalized to variant record types $\oplus\{\ell: \tau_\ell^+\}_{\ell \in L}$.⁴ These are not just a programming convenience but allow for richer subtyping: lazy and variant record types support both width and depth subtyping, whereas the usual binary products and sums support only the latter. For example, width subtyping means that $\oplus\{\mathbf{false}: \mathbf{1}\}$ is a subtype of $\mathbf{bool}^+ = \oplus\{\mathbf{false}: \mathbf{1}, \mathbf{true}: \mathbf{1}\}$, while $\mathbf{1}$ would not be a subtype of the usual binary $\mathbf{1} + \mathbf{1}$. Neither is $\mathbf{1}$ a subtype of \mathbf{bool}^+ , demonstrating the utility of variant record types with one label, such as $\oplus\{\mathbf{false}: \mathbf{1}\}$. Similar examples exist for lazy record types. This way, we recover some of the benefits of refinement types without the syntactic burden of a distinct refinement layer.

The shift $\downarrow\sigma^-$ is inhabited by an unevaluated computation of type σ^- (a “think”). Conversely, the shift $\uparrow\tau^+$ includes a value as a trivial computation (a “return”). Levy [53] writes $U \underline{B}$ instead of $\downarrow\sigma^-$ and $F A$ instead of $\uparrow\tau^+$.

Finally, we model recursive types not by explicit constructors $\mu\alpha^+. \tau^+$ and $\nu\alpha^-. \sigma^-$ but by *type names* t^+ and s^- which are defined in a global signature Σ . They may mutually refer to each other. We treat these as *equirecursive* (see Section 3) and we require them to be *contractive*, which means the right-hand side of a type definition cannot itself be a type name. Since we would like to directly observe the values of positive types, the definitions of type names $t^+ = \tau^+$ are *inductive*. This allows inductive reasoning about values returned by computations. On the other hand, negative type definitions $s^- = \sigma^-$ are recursive rather than coinductive in the usual sense, which would require, for example, stream computations to be productive. Because we do not wish to restrict recursive computations to those that are productive in this sense, they are “productive” only in the sense that they satisfy a standard progress theorem.

Next, we come to the syntax for values v of a positive type and computations e of a negative type. Variables x always stand for values and therefore have a positive type. We use j to stand for labels, naming fields of variant records or lazy records, where $j \cdot v$ injects value v into a sum with alternative labeled j and $e.j$ projects field e out of a lazy record. When we quantify over a (always finite) set of labels we usually write ℓ as a metavariable for the labels.

$$\begin{aligned}v &::= x \mid \langle v_1, v_2 \rangle \mid \langle \rangle \mid j \cdot v \mid \mathbf{think} \ e \\ e &::= \lambda x. e \mid e v \mid \{\ell = e_\ell\}_{\ell \in L} \mid e.j \mid \mathbf{return} \ v \mid \mathbf{let} \ \mathbf{return} \ x = e_1 \ \mathbf{in} \ e_2 \mid f \\ &\quad \mid \mathbf{match} \ v \ (\langle x, y \rangle \Rightarrow e) \mid \mathbf{match} \ v \ (\langle \rangle \Rightarrow e) \mid \mathbf{match} \ v \ (\ell \cdot x_\ell \Rightarrow e_\ell)_{\ell \in L} \mid \mathbf{force} \ v \\ \Sigma &::= \cdot \mid \Sigma, t^+ = \tau^+ \mid \Sigma, s^- = \sigma^- \mid \Sigma, f: \sigma^- = e\end{aligned}$$

⁴ We borrow the notation \oplus from linear logic even though no linearity is implied.

In order to represent recursion, we use equations $f = e$ in the signature where f is a defined *expression name*, which we distinguish from variables, and all equations can mutually reference each other. An alternative would have been explicit fixed point expressions $\text{fix } f. e$, but this mildly complicates both typing and mutual recursion. Also, it seems more elegant to represent all forms of recursion at the level of types and expressions in the same manner. We also choose to fix a type for each expression name in a signature. Otherwise, each occurrence of f in an expression could potentially be assigned a different type, which strays into the domain of parametric polymorphism and intersection types.

Following Levy, we do not allow names for values because this would add an undesirable notion of computation to values, and, furthermore, circular values would violate the inductive interpretation of positive types. As discussed in [53, Chapter 4], they could be added back conservatively under some conditions.

2.1 Dynamics

For the operational semantics, we use a judgment $e \mapsto e'$ defined inductively by the following rules which may reference a global signature Σ to look up the definitions of expression names f . In contrast, values do not reduce. The dynamics of call-by-push-value are defined as follows:

$$\begin{array}{c}
 \frac{}{(\lambda x. e) v \mapsto [v/x]e} \quad \frac{e \mapsto e'}{e v \mapsto e' v} \quad \frac{}{\text{let return } x = \text{return } v \text{ in } e_2 \mapsto [v/x]e_2} \\
 \\
 \frac{e_1 \mapsto e'_1}{\text{let return } x = e_1 \text{ in } e_2 \mapsto \text{let return } x = e'_1 \text{ in } e_2} \quad \frac{(j \in L) \quad e \mapsto e'}{\{\ell = e_\ell\}_{\ell \in L}. j \mapsto e_j \quad e. j \mapsto e'. j} \\
 \\
 \frac{}{\text{match } \langle v_1, v_2 \rangle (\langle x, y \rangle \Rightarrow e) \mapsto [v_1/x][v_2/y]e} \quad \frac{}{\text{match } \langle \rangle (\langle \rangle \Rightarrow e) \mapsto e} \\
 \\
 \frac{(j \in L)}{\text{match } (j \cdot v) (\ell \cdot x_\ell \Rightarrow e_\ell)_{\ell \in L} \mapsto [v/x_j]e_j} \quad \frac{f : \sigma^- = e \in \Sigma}{\text{force } (\text{think } e) \mapsto e \quad f \mapsto e}
 \end{array}$$

Note that some computations, specifically $\lambda x. e$, $\{\ell = e_\ell\}_{\ell \in L}$, and $\text{return } v$, do not reduce and may be considered values in other formulations. Here, we call them *terminal computations* and use the judgment e *terminal* to identify them.

$$\frac{}{\lambda x. e \text{ terminal}} \quad \frac{}{\{\ell = e_\ell\}_{\ell \in L} \text{ terminal}} \quad \frac{}{\text{return } v \text{ terminal}}$$

We will silently use simple properties of computations in the remainder of the paper which follow by straightforward induction.

Lemma 1 (Computation).

1. If $e \mapsto e'$ and $e \mapsto e''$ then $e' = e''$
2. It is not possible that both $e \mapsto e'$ and e *terminal*.

2.2 Some Sample Programs⁵

Example 1 (Computing with Binary Numbers). We show some example programs for binary numbers in “little endian” representation (least significant bit first) and in standard form, that is, without leading zeros.

$$\begin{aligned} \text{bin}^+ &= \oplus\{\mathbf{e} : \mathbf{1}, \mathbf{b0} : \text{bin}, \mathbf{b1} : \text{bin}\} \\ \text{std}^+ &= \oplus\{\mathbf{e} : \mathbf{1}, \mathbf{b0} : \text{pos}, \mathbf{b1} : \text{std}\} \\ \text{pos}^+ &= \oplus\{\mathbf{b0} : \text{pos}, \mathbf{b1} : \text{std}\} \end{aligned}$$

We expect the subtyping relationships $\text{pos} \leq \text{std} \leq \text{bin}$ to hold, because every positive standard number is a standard number, and every standard number is a binary number. According to our definition and rules in Sections 3 and 5 these will hold semantically as well as syntactically.

We now show some simple definitions $f : \sigma^- = e$.

$$\text{six} : \uparrow\text{pos} = \text{return } \mathbf{b0} \cdot \mathbf{b1} \cdot \mathbf{b1} \cdot \mathbf{e} \cdot \langle \rangle$$

The increment function on binary numbers implements the carry with a recursive call, which has to be wrapped in a let/return.

$$\begin{aligned} \text{inc} : \text{std} &\rightarrow \uparrow\text{pos} \\ &= \lambda x. \text{match } x \left(\begin{array}{l} \mathbf{e} \cdot u \Rightarrow \text{return } \mathbf{b1} \cdot \mathbf{e} \cdot u \\ | \mathbf{b0} \cdot x' \Rightarrow \text{return } \mathbf{b1} \cdot x' \\ | \mathbf{b1} \cdot x' \Rightarrow \text{let return } y' = \text{inc } x' \text{ in return } \mathbf{b0} \cdot y' \end{array} \right) \end{aligned}$$

By subtyping, we also have $\text{inc} : \text{std} \rightarrow \uparrow\text{std}$, for example, but *not* $\text{inc} : \text{bin} \rightarrow \uparrow\text{bin}$ since $\text{bin} \not\leq \text{std}$. However, the definition could be separately checked against this type, which points towards an eventual need for intersection types.

The following incorrect version of the decrement function does *not* have the indicated desired type!

$$\begin{aligned} \text{dec}_0 : \text{pos} &\rightarrow \uparrow\text{std} \quad \% \text{ incorrect!} \\ &= \lambda x. \text{match } x \left(\begin{array}{l} \mathbf{b0} \cdot x' \Rightarrow \text{let return } y' = \text{dec}_0 x' \text{ in return } \mathbf{b1} \cdot y' \\ | \mathbf{b1} \cdot x' \Rightarrow \text{return } \mathbf{b0} \cdot x' \end{array} \right) \end{aligned}$$

The error here is quite precisely located by the bidirectional type checker (see Section 6): When we inject $\mathbf{b0} \cdot x'$ in the second branch it is not the case that $x' : \text{pos}$ as required for standard numbers! And, indeed, $\text{dec}_0 \mathbf{b1} \cdot \mathbf{e} \cdot \langle \rangle \mapsto^* \text{return } \mathbf{b0} \cdot \mathbf{e} \cdot \langle \rangle$ which is not in standard form. On the other hand, the fact that a branch for $\mathbf{e} \cdot u$ is missing is correct because the type pos does not have an alternative for this label.

We can fix this problem by discriminating one more level of the input (which could be made slightly more appealing by a compound syntax for nested pattern matching).

$$\begin{aligned} \text{dec} : \text{pos} &\rightarrow \uparrow\text{std} \\ &= \lambda x. \text{match } x \left(\begin{array}{l} \mathbf{b0} \cdot x' \Rightarrow \text{let return } y' = \text{dec } x' \text{ in return } \mathbf{b1} \cdot y' \\ | \mathbf{b1} \cdot x' \Rightarrow \text{match } x' \left(\begin{array}{l} \mathbf{e} \cdot u \Rightarrow \text{return } \mathbf{e} \cdot u \\ | \mathbf{b0} \cdot x'' \Rightarrow \text{return } \mathbf{b0} \cdot \mathbf{b0} \cdot x'' \\ | \mathbf{b1} \cdot x'' \Rightarrow \text{return } \mathbf{b0} \cdot \mathbf{b1} \cdot x'' \end{array} \right) \end{array} \right) \end{aligned}$$

⁵ These examples and more are captured in our open access implementation artifact [50].

Example 2 (Computing with Streams). We present an example of a type with mixed polarities: a stream of standard numbers with a finite amount of padding between consecutive numbers. Programmer’s intent is for the stream to be lazy and infinite, i.e., no end-of-stream is provided. But because we do not restrict recursion even a well-typed implementation may diverge and fail to produce another number. On the other hand, the padding must always be finite because the meaning of positive types is inductive. We present padded streams as two mutually dependent type definitions, one positive and one negative. Because our type definitions are equirecursive this isn’t strictly necessary, and we could just substitute out the definition of pstream^- .

For our example, we also define a subtype with zero padding, as forcing a single padding label **none** between any two elements could also be expressed.

$$\begin{aligned} \text{pstream}^- &= \uparrow(\text{std} \otimes \text{padding}) \\ \text{padding}^+ &= \oplus\{\mathbf{none} : \text{padding}, \mathbf{some} : \downarrow\text{pstream}\} \\ \text{zstream}^- &= \uparrow(\text{std} \otimes \oplus\{\mathbf{some} : \downarrow\text{zstream}\}) \end{aligned}$$

In zstream , we see the significance of variant record types with just one label: **some**. We exploit this in Section 7 to interpret isorecursive types into equirecursive ones. We have that $\text{zstream} \leq \text{pstream}$, which means we can pass a stream with zero padding into any function expecting one with arbitrary padding.

We now program two mutually recursive functions to create a stream with zero padding from a stream with arbitrary (but finite!) padding.

$$\begin{aligned} \text{compress} &: (\downarrow\text{pstream}) \rightarrow \text{zstream} \\ \text{omit} &: \text{padding} \rightarrow \text{zstream} \\ \text{compress} &= \lambda s. \text{ let return } np = \text{force } s \text{ in} \\ &\quad \text{match } np \langle \langle n, p \rangle \Rightarrow \text{return } \langle n, \mathbf{some} \cdot \text{thunk } (\text{omit } p) \rangle \rangle \\ \text{omit} &= \lambda p. \text{ match } p \left(\mathbf{none} \cdot p' \Rightarrow \text{omit } p' \right. \\ &\quad \left. \mid \mathbf{some} \cdot s \Rightarrow \text{compress } s \right) \end{aligned}$$

Example 3 (Omega). As a final example in this section we consider the embedding of the untyped λ -calculus. The untyped term under consideration is $(\lambda x. x x) (\lambda x. x x)$. The first thing to notice is that this term is not even *syntactically* well-formed because x stands for a value, but in $x x$ the function parts needs to be an expression. Closely related is that the “usual” definition for the embedding of the untyped λ -calculus (see, for example, [42]) $\mathbf{U} = \mathbf{U} \rightarrow \mathbf{U}$ isn’t properly polarized. So, we define it as $\mathbf{U}^- = (\downarrow\mathbf{U}) \rightarrow \mathbf{U}$ instead:

$$\begin{aligned} \omega &: (\downarrow\mathbf{U}) \rightarrow \mathbf{U} & \Omega &: \mathbf{U} \\ \omega &= \lambda x. (\text{force } x) x & \Omega &= \omega (\text{thunk } \omega) \end{aligned}$$

Because our type definitions are equirecursive, both of these definitions are well-typed. Moreover, we also have $\omega : \mathbf{U}$ and in fact the embedding of every untyped λ -term will have type \mathbf{U} . We also observe that $\omega (\text{thunk } \omega) \mapsto^3 \omega (\text{thunk } \omega)$ and therefore represents a well-typed diverging term. Of course, $f : \mathbf{U} = f$ is also well-typed and reduces to itself in one step.

Remarkably, with our notion of *semantic typing* we will see that Ω will have every type σ^- and not just \mathbf{U} [49, Appendix B, Example 9]!

3 Semantic Typing

Our aim is to justify both typing and subtyping by semantic means. We therefore start with *semantic typing* of closed values and computations, written $v \in \tau^+$ and $e \in \sigma^-$. From this we can, for example, define semantic subtyping for positive types $\tau^+ \subseteq \sigma^+$ as $\forall v. v \in \tau^+ \supset v \in \sigma^+$.

Conceptually, semantic typing is a mixed inductive/coinductive definition. Values are typed inductively, which yields the correct interpretation of purely positive types such as natural numbers, lists, or trees, describing finite data structures. Computations are typed coinductively because they include the possibility of infinite computation by unbounded recursion. While we assume we can observe the structure of values, computations e cannot be observed directly. Different notions of observation for computation would yield different definitions of semantic typing. For our purposes, since we want to allow unfettered recursion, we posit we can (a) observe the fact that a computation *steps* according to our dynamics, even if we cannot examine the computation itself, and (b) when a computation is *terminal* we can observe its behavior by applying elimination forms (for types $\tau^+ \rightarrow \sigma^-$ and $\&\{\ell: \sigma_\ell^-\}_{\ell \in L}$) or by observing its returned value (for the type $\uparrow\tau^+$).

Besides capturing a certain notion of observability, our semantics incorporates the usual concept of *type soundness* which is important both for implementations and for interpreting the results of computations. These are:

Semantic Preservation (Theorem 1) If $e \in \sigma^-$ and $e \mapsto e'$ then $e' \in \sigma^-$.

Semantic Progress (Theorem 2) If $e \in \sigma^-$ then either $e \mapsto e'$ for some e' or e is *terminal* (but not both). This captures the usual slogan that “*well-typed programs do not go wrong*” [57]. An implementation will not accidentally treat a pair as a function or try to decompose a function as if it were a pair.

Semantic Observation If $v \in \tau^+$ then the structure of the value v is determined (inductively) by the type τ^+ . Similarly, a *terminal computation* $e \in \uparrow\tau^+$ must have the form $e = \text{return } v$ with $v \in \tau^+$.

These combine to the following: if we start a computation for $e \in \uparrow\tau^+$ then either $e \mapsto^* \text{return } v$ for an observable value $v \in \tau^+$ after a finite number of steps, or e does not terminate.

These are close to their usual syntactic analogues, but the fact that we do not rely on any form of syntactic typing is methodologically significant. For example, if we have a program that does not obey a syntactic typing discipline but behaves correctly according to our semantic typing, our results will apply and this program, in combination with others that are well typed, will both be safe (semantic progress) and return meaningfully observable results (semantic preservation and observation). This point has been made passionately by Dreyer et al. [28] and applied, for example, to trusted libraries in Rust [47]. Another example can be found in gradual typing [38, 60]. As long as we can *prove* by any means that the “dynamically typed” portion of the program is semantically well-typed (even if not syntactically so), the combination is sound and can be executed without worry, returning a correctly observable result. A third example

is provided by *session types* for message-passing concurrency [44]. While it is important to have a syntactic type discipline, processes in a distributed system may be programmed in a variety of languages some of which will have much weaker guarantees. Being able to *prove* their semantic soundness then guarantees the behavioral soundness of the composed system.

Semantic typing in the context of call-by-push-value is well-suited for encoding computational effects, such as input/output, memory mutation, nontermination, etc. Call-by-push-value was designed as a study for the λ -calculus with effects [53, Sec. 2.4], stratifying terms into values (which have no side-effects) and computations (which might). Through the lens of semantic typing, we can ensure behavioral soundness in the presence of effects.

3.1 Semantic Typing with Observation Depth

Despite the extensive work on mixed inductive and coinductive definitions [3, 11, 20, 21, 22, 43, 48, 51, 59, 61, 69], there is no widely accepted style in presenting such definitions and reasoning with them concisely in an mathematical language of discourse. With some regret, we therefore present our semantic definition by turning the coinductive part into an inductive one, following the basic idea underlying *step indexing* [7, 8, 10, 27]. Since the coinduction has priority over the induction, arguments proceed by nested induction, first over the step index and second over the structure of the inductive definition. This representation of mixed definitions implies that reasoning over step indices has lexicographic priority over values.

An alternative point of view is provided by *sized types* [5, 6]. Both sized types and step indexing employ the same concept of observation depth; however, for sized types, we would observe data constructors, whereas for step indexing we observe computation steps. General recursion is supported in our system because “productivity” in the negative layer means that computations can step rather than produce a data constructor. The step index is actually the (universally quantified) observation depth for a coinductively defined predicate. We do not index the (existentially quantified) size of the inductive predicate but use its structure directly since values are finite and become smaller. All step indices k , i and occasionally j range over natural numbers. We use three judgments,

1. $e \in_k \sigma^-$ (e has semantic type σ^- at index k)
2. $e \in_{k+1} \hat{\sigma}^-$ (terminal e has semantic type σ^- at index $k+1$)
3. $v \in_k \tau^+$ (v has semantic type τ^+ at index k)

They should be defined by nested induction, first on k and second on the structure of v/e , where part 2 can rely on part 1 for a computation that is not terminal. We write $v < v'$ when v is a strict subexpression of v' . The clauses of the definition can be found in Figure 1.

A few notes on these definitions. When expanding type definitions $t = \tau^+$ and $s = \sigma^-$ we rely on the assumption that type definitions are contractive, so one of the immediately following cases will apply next. This means that unlike many

$$\begin{aligned}
v \in_k t &\triangleq v \in_k \tau^+ \text{ for } t = \tau^+ \in \Sigma \\
v \in_k \tau_1^+ \otimes \tau_2^+ &\triangleq v = \langle v_1, v_2 \rangle, v_1 \in_k \tau_1^+, \text{ and } v_2 \in_k \tau_2^+ \text{ for some } v_1, v_2 \\
v \in_k \mathbf{1} &\triangleq v = \langle \rangle \\
v \in_k \oplus \{\ell: \tau_\ell^+\}_{\ell \in L} &\triangleq v = j \cdot v_j \text{ and } v_j \in_k \tau_j^+ \text{ for some } j \in L \\
v \in_k \downarrow \sigma^- &\triangleq v = \mathbf{thunk } e \text{ and } e \in_k \sigma^- \text{ for some } e \\
e \in_0 \sigma^- &\text{ always} \\
e \in_{k+1} \sigma^- &\triangleq (e \mapsto e' \text{ and } e' \in_k \sigma^-) \text{ or } (e \text{ terminal and } e \in_{k+1} \sigma^-) \\
e \in_{k+1} s &\triangleq e \in_{k+1} \sigma^- \text{ for } s = \sigma^- \in \Sigma \\
e \in_{k+1} \tau^+ \rightarrow \sigma^- &\triangleq e v \in_{k+1} \sigma^- \text{ for all } i \leq k \text{ and } v \text{ with } v \in_i \tau^+ \\
e \in_{k+1} \&\{\ell: \sigma_\ell^-\}_{\ell \in L} &\triangleq e.j \in_{k+1} \sigma_j^- \text{ for all } j \in L \\
e \in_{k+1} \uparrow \tau^+ &\triangleq e = \mathbf{return } v \text{ for some } v \in_k \tau^+ \\
v \in \tau^+ &\triangleq v \in_k \tau^+ \text{ for all } k \\
e \in \sigma^- &\triangleq e \in_k \sigma^- \text{ for all } k
\end{aligned}$$

Fig. 1. Definition of Semantic Typing

definitions in this style the types do not necessarily get smaller. For the inductive part (typing of values), the values do get smaller and for the coinductive part (typing of computations) the step index will get smaller because in the case of functions and records the constructed expression is not terminal.

A number of variations on this definition are possible. A particularly interesting one avoids decreasing the step index unless recursion is unrolled [8, 27, 60] so sources of nontermination can be characterized more precisely. It may also be possible to keep the step index constant when analyzing a terminal computation of type $\uparrow \tau^+$. Stripping the return constructor constitutes a form of observation and therefore decreasing the index seems both appropriate and simplest.

The quantification over $i \leq k$ in the case of terminal computations of function type seems necessary because we need the relation to be *downward closed* so that it defines a *deflationary fixed point* [4, 41]. Values and computations are then semantically well-typed if they are well-typed for *all* step indices.

Lemma 2 (Downward Closure).

1. $e \in_k \sigma^-$ implies $e \in_i \sigma^-$ for all $i \leq k$
2. $e \in_{k+1} \sigma^-$ implies $e \in_{i+1} \sigma^-$ for all $i \leq k$
3. $v \in_k \tau^+$ implies $v \in_i \tau^+$ for all $i \leq k$

Proof. By a routine nested induction on k and the structure of v/e where part 2 can appeal to part 1 when e is not terminal.

Here are some semantic types that can easily be verified (see [49, Appendix B]).

Example 4 (Semantic Typing).

1. $\lambda x. \text{return } x \in \tau^+ \rightarrow \uparrow\tau^+$ for all τ^+ .
2. Define $s_0 = \mathbf{1} \rightarrow s_0$ and $e_0 = \lambda x. e_0$. Then $e_0 \in s_0$.
3. Define $\omega = \lambda x. (\text{force } x) x$ and $\Omega = \omega(\text{thunk } \omega)$. Then $\Omega \in \sigma^-$ for every σ^- .
4. Define $t_0 = \mathbf{1} \otimes t_0$. Then there is no v such that $v \in t_0$.
5. Assume $e \in \rho^-$ for some ρ^- . Then $e \in t_0 \rightarrow \sigma^-$ for every σ^- .

3.2 Properties of Semantic Typing

The properties of semantic preservation and progress follow immediately just by applying the definitions and Lemma 1, so we elide their proofs.

Theorem 1 (Semantic Preservation). *If $e \in \sigma^-$ and $e \mapsto e'$ then $e' \in \sigma^-$.*

Theorem 2 (Semantic Progress). *If $e \in \sigma^-$ then either $e \mapsto e'$ or e is terminal, but not both.*

4 Subtyping

The semantics of subtyping is quite easy to express using semantic typing.

Definition 1 (Semantic Subtyping).

1. $\tau^+ \subseteq \sigma^+$ iff $v \in \tau^+$ implies $v \in \sigma^+$ for all v .
2. $\tau^- \subseteq \sigma^-$ iff $e \in \tau^-$ implies $e \in \sigma^-$ for all e .

We would now like to give a syntactic definition of subtyping that expresses an algorithm and show it both sound and complete with respect to the given semantic definition. The intuitive rules for subtyping shouldn't be surprising, although to our knowledge our formulation is original.

4.1 Empty and Full Types

A first observation is that $\tau^+ \subseteq \sigma^+$ whenever τ^+ is an empty type, regardless of σ^+ , because the necessary implication holds vacuously. So we need an algorithm to determine *emptiness of a positive type*. For the most streamlined presentation (which is also most suitable for an implementation) we first put the signature into a normal form that alternates between structural types and type names.

$$\begin{aligned} \tau^+ &::= t_1 \otimes t_2 \mid \mathbf{1} \mid \oplus\{\ell: t_\ell\}_{\ell \in L} \mid \downarrow s \\ \sigma^- &::= t \rightarrow s \mid \&\{\ell: s_\ell\}_{\ell \in L} \mid \uparrow t \\ \Sigma &::= \cdot \mid \Sigma, t = \tau^+ \mid \Sigma, s = \sigma^- \mid \Sigma, f : \sigma^- = e \end{aligned}$$

A usual presentation of emptiness maintains a collection of recursive types in a context in order to do a kind of loop detection. For example, the type $t = \mathbf{1} \otimes t$ is empty because we may assume that t is empty while testing $\mathbf{1} \otimes t$. Instead, we

express this and similar kinds of arguments using valid circular reasoning. If one were to formalize it, it would be in \mathbf{CLKID}^ω [14], although the succedent of any sequent is either empty or a singleton (as in \mathbf{CLJID}^ω [12]).

We construct circular derivations for t empty where t is a positive type name. Note that negative types are never empty. We can form a valid cycle when we encounter a goal t empty as a proper subgoal of t empty. Since we fix a signature Σ once and for all before defining each judgment such as emptiness or subtyping, we omit the index Σ since it never changes. The rules can be found in Figure 2.

$$\frac{t = \oplus\{\ell : t_\ell\}_{\ell \in L} \in \Sigma \quad t_j \text{ empty } (\forall j \in L)}{t \text{ empty}} \oplus_{\text{EMP}} \quad (\text{no rules for } t = \mathbf{1} \text{ or } t = \downarrow s)$$

$$\frac{t = t_1 \otimes t_2 \in \Sigma \quad t_1 \text{ empty}}{t \text{ empty}} \otimes_{\text{EMP}_1} \quad \frac{t = t_1 \otimes t_2 \in \Sigma \quad t_2 \text{ empty}}{t \text{ empty}} \otimes_{\text{EMP}_2}$$

Fig. 2. Circular Derivation Rules for Emptiness

Example 5. We continue Example 4, part (4), building a formal circular derivation. We first bring the signature into normal form, $\Sigma = \{u_0 = \mathbf{1}, t_0 = u_0 \otimes t_0\}$, and then construct

$$\frac{\text{CYCLE}() \quad t_0 = u_0 \otimes t_0 \quad t_0 \text{ empty}}{t_0 \text{ empty}} \otimes_{\text{EMP}_2}$$

Theorem 3 (Emptiness). *If t empty then for all k and v , $v \notin_k t$.*

Proof. We interpret the judgment t empty semantically as $v \in_k t \vdash \cdot$ (which expresses $v \notin_k t$ in a sequent), where t is given and k and v are parameters and therefore implicitly universally quantified. The proof of this judgment is carried out in a circular metalogic. We translate each inference rule for t empty into a derivation for $v \in_k t \vdash \cdot$, where each unproven subgoal corresponds to a premise of the rule. When the derivation of t empty is closed by a cycle, the corresponding derivation of $v \in_k t \vdash \cdot$ is closed by a corresponding cycle in the metalogic. The cases can be found in [49, Appendix D].

Next we symmetrically define what it means for a computation type σ^- to be *full*, namely that it is inhabited by *every (semantically well-typed) computation*. A simple example is the type $\&\{\}$, that is, the lazy record without any fields. It contains every well-typed expression because *all* projections (of which there are none) are well-typed. It turns out the fullness is directly defined from emptiness.

$$\begin{array}{c}
\frac{t = t_1 \otimes t_2 \quad u = u_1 \otimes u_2 \quad t_1 \leq u_1 \quad t_2 \leq u_2}{t \leq u} \otimes_{\text{SUB}} \quad \frac{t = \mathbf{1} \quad u = \mathbf{1}}{t \leq u} \mathbf{1}_{\text{SUB}} \\
\frac{t = \oplus\{\ell : t_\ell\}_{\ell \in L} \quad u = \oplus\{k : u_k\}_{k \in K} \quad \forall \ell \in L. t_\ell \text{ empty} \vee (\ell \in K \wedge t_\ell \leq u_\ell)}{t \leq u} \oplus_{\text{SUB}} \\
\frac{t = \downarrow s \quad u = \downarrow r \quad s \leq r}{t \leq u} \downarrow_{\text{SUB}} \quad \frac{s = t_1 \rightarrow s_2 \quad r = u_1 \rightarrow r_2 \quad u_1 \leq t_1 \quad s_2 \leq r_2}{s \leq r} \rightarrow_{\text{SUB}} \\
\frac{s = \uparrow t \quad r = \uparrow u \quad t \leq u}{s \leq r} \uparrow_{\text{SUB}} \\
\frac{s = \&\{\ell : s_\ell\}_{\ell \in L} \quad r = \&\{j : r_j\}_{j \in K} \quad \forall j \in K. j \in L \wedge s_j \leq r_j}{s \leq r} \&_{\text{SUB}} \\
\frac{t \text{ empty} \quad u = \tau^+}{t \leq u} \perp_{\text{SUB}}^+ \quad \frac{s = \uparrow t \quad t \text{ empty} \quad r = \sigma^-}{s \leq r} \perp_{\text{SUB}}^- \quad \frac{s = \sigma^- \quad r \text{ full}}{s \leq r} \top_{\text{SUB}}
\end{array}$$

Fig. 3. Circular Derivation Rules for Subtyping

From a circular derivation we now construct a valid circular proof in an intuitionistic metalogic [12]. For example, $t \leq u$ is interpreted as $t \subseteq u$, that is, every value in t is also a value in u . We actually prove a slightly stronger theorem, namely that for the step index on both sides can remain the same.

Theorem 5 (Soundness of Subtyping).

1. If $t \leq u$ then $v \in_k t \vdash v \in_k u$ for all k and v (and so, $t \subseteq u$).
2. If $s \leq r$ then $e \in_k s \vdash e \in_k r$ for all k and e (and so, $s \subseteq r$).

Proof. We proceed by a compositional translation of the circular derivation of subtyping into a circular derivation in the metalogic. For each rule we construct a derived rule on the semantic side with corresponding premises and conclusion.

When the subtyping proof is closed due to a cycle, we close the proof in the metalogic with a corresponding cycle. In order for this cycle to be valid, it is critical that the judgments in the premises of the derived rule are *strictly smaller* than the judgments in the conclusion. Since our mixed logical relation is defined by nested induction, first on the step index k and second on the structure of the value v or expression e , the lexicographic measure $(k, v/e)$ should strictly decrease. Some sample cases can be found in [49, Appendix F].

Besides soundness, reflexivity and transitivity of syntactic subtyping are two other properties that we prove for assurance that the syntactic subtyping rules are sensible and have no obvious gaps. These proofs can be found in [49, Appendix G]. Ligatti et al. [55] also consider a notion of *preciseness* as a syntactic means for judging the correctness of their syntactic subtyping rules. As they mention in [55,

Sec. 6.2], this property is highly language-sensitive, depending on the choice of evaluation strategy (strict vs. nonstrict), where nonstrict subtyping relies on “which primitives are present in the language, sometimes in nonorthogonal ways.” Moreover, preciseness requires syntactically well-typed counterexamples, whereas we also consider ill-typed terms. We can straightforwardly prove that syntactic subtyping for purely positive types (in relation to strict evaluation) is complete with respect to semantic subtyping. We leave the preciseness of syntactic subtyping of negative types for future consideration.

5 Syntactic Typing and Soundness

We now introduce a syntactic typing judgment, at the moment without regard to decidability. Such a judgment is often called *declarative typing* in contrast with what is *algorithmic typing* in Section 6 (Figure 4). We prove that all syntactically well-typed terms are also semantically well-typed. Conceptually, a declarative system is *unnecessary* because the bidirectional system is very closely related, and there are no problems in justifying the soundness of the the bidirectional system directly with respect to our semantics. Besides the fact that there is a small amount of additional bureaucracy (the rules are divided between four judgments instead of two, and there are two additional rules), it is also the case that the standard versions of call-by-name and call-by-value use a similar form of declarative typing and are therefore easier to relate to our system in Section 8.

Because all declarations in a signature can be mutually recursive, each declaration $f : \sigma^- = e$ is checked assuming all other declarations are valid. The soundness proof below justifies this. The complete set of judgments and rules with their corresponding presuppositions can be found in [49, Appendix H, Figs. 7 and 8]. For these rules, we need contexts Γ , defined as usual with the presupposition that all variables declared in a context are distinct.

$$\Gamma ::= \cdot \mid \Gamma, x : \tau^+$$

The rules for key judgments $\Gamma \vdash v : \tau^+$ and $\Gamma \vdash e : \sigma^-$ can be obtained from the bidirectional rules in Section 6 by replacing both $v \Leftarrow \tau^+$ and $v \Rightarrow \tau^+$ with $v : \tau^+$ and, similarly, $e \Leftarrow \sigma^-$ and $e \Rightarrow \sigma^-$ with $e : \sigma^-$. Moreover, one should drop the two annotation rules ANNO^+ and ANNO^- because these are not in the source language for declarative typing.

We would like to show that the syntactic typing rules are sound with respect to their semantic interpretation. For that, we first define simultaneous substitutions θ of closed values for variables and $\theta \in_k \Gamma$ for the semantic interpretation of contexts as sets of substitutions at step index k .

$$\begin{aligned} \theta &::= \cdot \mid \theta, v/x \\ (\cdot) \in_k (\cdot) &\text{ always} \\ (\theta, v/x) \in_k (\Gamma, x : \tau^+) &\triangleq \theta \in_k \Gamma \text{ and } v \in_k \tau^+ \end{aligned}$$

On the semantic side, we define

1. $\Gamma \models v \in_k \tau^+$ iff for all $\theta \in_k \Gamma$ we have $v[\theta] \in_k \tau^+$
2. $\Gamma \models e \in_k \sigma^-$ iff for all $\theta \in_k \Gamma$ we have $e[\theta] \in_k \sigma^-$

We now can prove a number of lemmas, one for each syntactic typing rule. A representative selection of the lemmas, each written as an admissible rule for semantic typing, can be given by:

$$\begin{array}{c}
\frac{e \in_k \tau^+ \rightarrow \sigma^- \quad v \in_k \tau^+}{e v \in_k \sigma^-} \qquad \frac{x : \tau^+ \models e \in_k \sigma^-}{\lambda x. e \in_k \tau^+ \rightarrow \sigma^-} \qquad \frac{v_1 \in_k \tau_1^+ \quad v_2 \in_k \tau_2^+}{\langle v_1, v_2 \rangle \in_k \tau_1^+ \otimes \tau_2^+} \\
\\
\frac{v \in_k \tau_1^+ \otimes \tau_2^+ \quad x : \tau_1^+, y : \tau_2^+ \models e \in_k \sigma^-}{\text{match } v (\langle x, y \rangle \Rightarrow e) \in_k \sigma^-} \qquad \frac{v \in_k \tau^+}{\text{return } v \in_k \uparrow \tau^+} \qquad \frac{v \in_k \downarrow \sigma^-}{\text{force } v \in_k \sigma^-} \\
\\
\frac{e_1 \in_k \uparrow \tau^+ \quad x : \tau^+ \models e_2 \in_k \sigma^-}{\text{let return } x = e_1 \text{ in } e_2 \in_k \sigma^-} \qquad \frac{e \in_k \sigma^-}{\text{thunk } e \in_k \downarrow \sigma^-} \qquad \frac{v \in_k \tau^+ \quad \tau^+ \leq \sigma^+}{v \in_k \sigma^+}
\end{array}$$

The proofs are somewhat interesting: some require induction on k , others follow more directly by definition. Due to a lack of space, the proofs can be found in [49, Appendix I], each admissible rule formulated as a separate lemma.

Theorem 6 (Soundness of Syntactic Typing). *Assume $\theta \in_k \Gamma$.*

1. If $\Gamma \vdash v : \tau^+$ then $v[\theta] \in_k \tau^+$
2. If $\Gamma \vdash e : \sigma^-$ then $e[\theta] \in_k \sigma^-$

Proof. We construct a circular proof based on the typing derivation, and the typing derivations for all definitions $f : \sigma^- = e \in \Sigma$. There are three kinds of cases (see [49, Appendix I] for samples of each):

1. The case of variables x follows by assumption on θ .
2. In the case of names $f : \sigma^- = e \in \Sigma$ we either expand to e or close the proof with a cycle if we have expanded f already.
3. All other rules follow by the lemmas presented above.

In all these lemmas the step index remains constant for the premises, which is important so we can form a circular proof in the case of names.

Because soundness is stated for all θ , Γ , and k , we can immediately obtain corollaries such as that $\cdot \vdash v : \tau^+$ implies that $v \in \tau^+$, and that $\cdot \vdash e : \sigma^-$ implies that $e \in \sigma^-$.

6 Bidirectional Typing

We now shift from our declarative typing system into an algorithmic one that describes a practical decision procedure. We choose to express it as a bidirectional typechecking algorithm, particularly to avoid inference issues regarding subsumption [45] and our extensive use of type names and variant records, as

$$\begin{array}{c}
\frac{\Gamma \vdash v_1 \Leftarrow \tau_1^+ \quad \Gamma \vdash v_2 \Leftarrow \tau_2^+}{\Gamma \vdash \langle v_1, v_2 \rangle \Leftarrow \tau_1^+ \otimes \tau_2^+} \otimes \text{I} \quad \frac{\Gamma \vdash v \Rightarrow \tau_1^+ \otimes \tau_2^+ \quad \Gamma, x:\tau_1^+, y:\tau_2^+ \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \text{match } v \langle (x, y) \Rightarrow e \rangle \Leftarrow \sigma^-} \otimes \text{E} \\
\\
\frac{x : \tau^+ \in \Gamma}{\Gamma \vdash x \Rightarrow \tau^+} \text{VAR} \quad \frac{}{\Gamma \vdash \langle \rangle \Leftarrow \mathbf{1}} \mathbf{1}\text{I} \quad \frac{\Gamma \vdash v \Rightarrow \mathbf{1} \quad \Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \text{match } v \langle \langle \rangle \Rightarrow e \rangle \Leftarrow \sigma^-} \mathbf{1}\text{E} \\
\\
\frac{\Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \text{thunk } e \Leftarrow \downarrow \sigma^-} \downarrow \text{I} \quad \frac{\Gamma \vdash v \Rightarrow \downarrow \sigma^-}{\Gamma \vdash \text{force } v \Rightarrow \sigma^-} \downarrow \text{E} \quad \frac{(j \in L) \quad \Gamma \vdash v \Leftarrow \tau_j^+}{\Gamma \vdash j \cdot v \Leftarrow \oplus \{ \ell : \tau_\ell^+ \}_{\ell \in L}} \oplus \text{I} \\
\\
\frac{\Gamma \vdash v \Rightarrow \oplus \{ \ell : \tau_\ell^+ \}_{\ell \in L} \quad \forall (\ell \in L) : \Gamma, x_\ell : \tau_\ell^+ \vdash e_\ell \Leftarrow \sigma^-}{\Gamma \vdash \text{match } v \langle \ell \cdot x_\ell \Rightarrow e_\ell \rangle_{\ell \in L} \Leftarrow \sigma^-} \oplus \text{E} \quad \frac{\Gamma, x:\tau^+ \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \lambda x. e \Leftarrow \tau^+ \rightarrow \sigma^-} \rightarrow \text{I} \\
\\
\frac{\Gamma \vdash e \Rightarrow \tau^+ \rightarrow \sigma^- \quad \Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash e v \Rightarrow \sigma^-} \rightarrow \text{E} \quad \frac{\forall (\ell \in L) : \Gamma \vdash e_\ell \Leftarrow \sigma_\ell^-}{\Gamma \vdash \{ \ell = e_\ell \}_{\ell \in L} \Leftarrow \& \{ \ell : \sigma_\ell^- \}_{\ell \in L}} \& \text{I} \\
\\
\frac{\Gamma \vdash e \Rightarrow \& \{ \ell : \sigma_\ell^- \}_{\ell \in L} \quad (j \in L)}{\Gamma \vdash e.j \Rightarrow \sigma_j^-} \& \text{E}_k \quad \frac{f : \sigma^- = e \in \Sigma}{\Gamma \vdash f \Rightarrow \sigma^-} \text{NAME} \quad \frac{\Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash \text{return } v \Leftarrow \uparrow \tau^+} \uparrow \text{I} \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow \uparrow \tau^+ \quad \Gamma, x:\tau^+ \vdash e_2 \Leftarrow \sigma^-}{\Gamma \vdash \text{let return } x = e_1 \text{ in } e_2 \Leftarrow \sigma^-} \uparrow \text{E} \quad \frac{\Gamma \vdash v \Rightarrow \tau^+ \quad \tau^+ \leq \sigma^+}{\Gamma \vdash v \Leftarrow \sigma^+} \text{SUB}^+ \\
\\
\frac{\Gamma \vdash e \Rightarrow \tau^- \quad \tau^- \leq \sigma^-}{\Gamma \vdash e \Leftarrow \sigma^-} \text{SUB}^- \quad \frac{\Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash (v : \tau^+) \Rightarrow \tau^+} \text{ANNO}^+ \quad \frac{\Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash (e : \sigma^-) \Rightarrow \sigma^-} \text{ANNO}^-
\end{array}$$

Fig. 4. Bidirectional Typing

well as the approach's deep integration with polarized logics [29, Section 8.3]. Moreover, bidirectional typing is quite robust with respect to language extensions where various inference procedures are not.

Bidirectional typechecking [68] has been a popular choice for presenting algorithmic typing, especially when concerned with subtyping [30], and is decidable for a wide range of rich type systems. This approach splits each of the typing judgments, $\Gamma \vdash v : \tau^+$ and $\Gamma \vdash e : \sigma^-$, into *checking* (\Leftarrow) and *synthesis* (\Rightarrow) judgments for values and expressions, respectively: $\Gamma \vdash v \Leftarrow \tau^+$, $\Gamma \vdash v \Rightarrow \tau^+$ and $\Gamma \vdash e \Leftarrow \sigma^-$, $\Gamma \vdash e \Rightarrow \sigma^-$.

We follow the recipe laid out by [25, 32]: introduction rules check and elimination rules synthesize. More precisely, the *principal judgment*, premise or conclusion, has the connective being introduced by checking or eliminated by synthesis.

We introduce two new forms of syntactic values ($v : \tau^+$) and computations ($e : \sigma^-$) which exist purely for typechecking purposes and are erased before evaluation. This is not actually used on any of our examples because definitions in the signature already require annotations.

Applying the recipe, we can easily convert our declarative rules into bidirectional ones, as laid out in Section 5. The only rules we add to the system are ANNO^+ and ANNO^- , which allow us to prove completeness. All the examples in Section 2.2 check with these rules and only require type annotations at the top level of the declarations in the signature.

Due to our use of equirecursive types, the implementation of this system can closely follow the structure of the rules in Figures 2, 3, and 4. First, as mentioned in Section 4.1, we convert the signature into a normal form that alternates structural types and type names. Then, we determine all the empty type names using a memoization table for t^+ **empty** to easily construct circular derivations of emptiness (bottom-up) using the rules in Figure 2. If constructing such a derivation fails then t^+ is nonempty. Fullness is derived from emptiness non-recursively. From there, we build a memoization table for $t^+ \leq u^+$ and $s^- \leq r^-$, for positive and negative type names, so we can construct circular derivations of subtyping between names (also bottom-up). This happens lazily, only computing $t^+ \leq u^+$ or $s^- \leq r^-$ if typechecking requires this information.

Bidirectional typing, given subtyping, follows the rules in Figure 4, including the rules for positive and negative subsumption, but it requires that the types in annotations are also translated to normal form, possibly introducing new (user-invisible) definitions in the signature.

The theorems (with straightforward proofs) for soundness and completeness of bidirectional typechecking can be found in [49, Appendix J, Thms. 12 and 13].

7 Interpretation of Isorecursive Types

Our system uses equirecursive types, which allow many subtyping relations since there are no term constructors for folding recursive types. Moreover, equirecursive types support the normal form where constructors are always applied to type names (see Section 4.1), simplifying our algorithms, their description and implementations. Most importantly, perhaps, equirecursive types are more general because we can directly interpret isorecursive types, which are embodied by *fold* and *unfold* operators, into our equirecursive setting and apply our results.

We give a short sketch here; details can be found in [49, Appendix K]. For every recursive type $\mu\alpha^+.\tau^+$ we introduce a definition $t^+ = \oplus\{\text{fold}_\mu : [t/\alpha]\tau\}$. Similarly, for every corecursive type $\nu\alpha^-\sigma^-$ we introduce a definition $s^- = \&\{\text{fold}_\nu : [s/\alpha]\sigma\}$. Now, the labels fold_μ and fold_ν tagging the sole choice of a unary variant or lazy record, respectively, play exactly the role that the *fold* constructor plays for recursive types. This entirely straightforward translation is enabled by our generalization of the binary sum and lazy pairs to variant and lazy records, respectively, so we can use them in their unary form.

8 Call-by-Name and Call-by-Value

More familiar than call-by-push-value (CBPV) are the lazy, call-by-name (CBN) and eager, call-by-value (CBV) operational semantics that underlie the Haskell

and ML families of functional programming languages. Levy [54] has shown that both CBN and CBV exist as fragments of CBPV, exhibiting translations from CBN and CBV types and terms into the CBPV language. In this section, we derive systems of subtyping for CBN and CBV from these translations into ours and prove them sound and complete. We discover that they are minor variants of existing systems for CBN [39] and CBV [55] subtyping.

Because polarized subtyping is able to connect Levy's translations with existing systems for CBN and CBV subtyping, it serves as further evidence that those prior translations and our subtyping rules are, in some sense, canonical. Moreover, it is yet one more piece of evidence that CBPV is an effective synthesis of evaluation orders in which to study the theory of functional programming.

8.1 Call-by-name

Consider a CBN language with the following types. The language of terms and the standard statics and dynamics can be found in [49, Appendix L].

$$\tau, \sigma ::= \tau \rightarrow \sigma \mid \tau_1 \otimes \tau_2 \mid \mathbf{1} \mid \oplus\{\ell: \tau_\ell\}_{\ell \in L} \mid \&\{\ell: \tau_\ell\}_{\ell \in L}$$

In this section, we will focus on function types $\tau \rightarrow \sigma$ and variant record types $\oplus\{\ell: \tau_\ell\}_{\ell \in L}$ and their corresponding terms.

Levy [54] presents translations, $(-)^{\square}$, from CBN types and terms to CBPV *negative* types and *expressions*, respectively. An auxiliary translation, $\downarrow(-)^{\square}$, on contexts is also used. Here, we elide the translation of terms other than variables and the terms for function and variant record types; the full translation on terms can be found in [54].

<i>Types</i>	<i>Terms</i>
$(\tau \rightarrow \sigma)^{\square} = \downarrow\tau^{\square} \rightarrow \sigma^{\square}$	$(x)^{\square} = \text{return } x$
$(\tau_1 \otimes \tau_2)^{\square} = \uparrow(\downarrow\tau_1^{\square} \otimes \downarrow\tau_2^{\square})$	$(\lambda x. e)^{\square} = \lambda x. e^{\square}$
$(\mathbf{1})^{\square} = \uparrow\mathbf{1}$	$(e_1 e_2)^{\square} = e_1^{\square} (\text{thunk } e_2^{\square})$
$(\oplus\{\ell: \tau_\ell\}_{\ell \in L})^{\square} = \uparrow\oplus\{\ell: \downarrow\tau_\ell^{\square}\}_{\ell \in L}$	
$(\&\{\ell: \sigma_\ell\}_{\ell \in L})^{\square} = \&\{\ell: \sigma_\ell^{\square}\}_{\ell \in L}$	

We also translate type names t to fresh type names t^{\square} , translating the body of t 's definition and inserting additional type names as required for the normal form that alternates between structural types and type names. Levy [54] proves that well-typed terms are well-typed after the translation to CBPV is applied. Our syntactic typing rules are the same, so the theorem carries over to our setting.

We adapt the subtyping system of Gay and Hole [39] to a λ -calculus from the π -calculus, which reverses the direction of subtyping from their classical system and adds empty records, obtaining the CBN syntactic subtyping rules shown in Figure 5.

These rules introduce a CBN syntactic subtyping judgment $t \leq u$. To distinguish it from CBPV syntactic subtyping, we will take care in this section to

always include superscript pluses and minuses for CBPV type names, with CBN type names being unmarked. As for CBPV syntactic subtyping, the rules for CBN subtyping shown in Figure 5 build a *circular derivation*. Just as before, a circularity arises when a goal $t \leq u$ arises as a proper subgoal of itself.

$$\begin{array}{c}
\frac{t = t_1 \rightarrow t_2 \quad u = u_1 \rightarrow u_2 \quad u_1 \leq t_1 \quad t_2 \leq u_2}{t \leq u} \rightarrow_{\text{SUB}_N} \\
\frac{t = t_1 \otimes t_2 \quad u = u_1 \otimes u_2 \quad t_1 \leq u_1 \quad t_2 \leq u_2}{t \leq u} \otimes_{\text{SUB}_N} \qquad \frac{t = \mathbf{1} \quad u = \mathbf{1}}{t \leq u} \mathbf{1}_{\text{SUB}_N} \\
\frac{t = \oplus\{\ell: t_\ell\}_{\ell \in L} \quad u = \oplus\{j: u_j\}_{j \in J} \quad (L \subseteq J) \quad \forall(\ell \in L): t_\ell \leq u_\ell}{t \leq u} \oplus_{\text{SUB}_N} \\
\frac{t = \&\{\ell: t_\ell\}_{\ell \in L} \quad u = \&\{j: u_j\}_{j \in J} \quad (L \supseteq J) \quad \forall(j \in J): t_j \leq u_j}{t \leq u} \&_{\text{SUB}_N} \\
\frac{t = \oplus\{\}}{t \leq u} \perp_{\text{SUB}_N} \qquad \frac{t = \tau \quad u \text{ full}}{t \leq u} \top_{\text{SUB}_N} \qquad \frac{t = \&\{\}}{t \text{ full}} \&_{\text{FULL}_N}
\end{array}$$

Fig. 5. Circular Derivation Rules for Call-by-Name Subtyping

These rules are exact analogues of those of Gay and Hole [39], with one exception. The three rules involving empty variants and records, namely \perp_{SUB_N} , \top_{SUB_N} , and $\&_{\text{FULL}_N}$, have no analogues in [39] only because their language did not include the corresponding empty internal and external choice types.

As we will prove below, the CBN subtyping rules in Figure 5 are exactly those for which $t \leq u$ in the CBN language if and only if $t^\square \leq u^\square$ in the CBPV metalanguage. We thereby show that our polarized subtyping on the image of Levy’s CBN translation is sound and complete with respect to Gay and Hole’s CBN subtyping.

Before proceeding to those proofs, it is worth pointing out that many of these CBN subtyping rules exactly follow CBPV, with a few notable differences. First, the \oplus_{SUB_N} rule does not permit empty branches that do not occur in the supertype. This is because the \downarrow shifts that appear in $(\oplus\{\ell: \tau_\ell\}_{\ell \in L})^\square$ prevent each branch from being empty—there is no emptiness rule for \downarrow shifts in the CBPV subtyping. Second, for this CBN language, only types $t = \&\{\}$ are full. In particular, a CBN function type $t = t_1 \rightarrow t_2$ is never full, even though a CBPV function type $s^- = t_1^+ \rightarrow s_2^-$ is full if the argument type t_1^+ is empty. This stems from the \downarrow shift that appears in the argument type in $(\tau \rightarrow \sigma)^\square = \downarrow\tau^\square \rightarrow \sigma^\square$. Third, the reader may be surprised by the omission of an emptiness judgment for CBN types. The \perp_{SUB_N} rule mentions the CBN type $t = \oplus\{\}$, which looks like it ought to be an empty type—the CBPV type $t_0^+ = \oplus\{\}$ is empty, after all.

Yes, but the CBN translation of $t = \oplus\{\}$ is in fact the negative type $t^\boxplus = \uparrow\oplus\{\}$, and negative types are never empty. Nevertheless, $t^\boxplus = \uparrow\oplus\{\} \leq u^\boxplus$ in this case.

Now we prove that polarized subtyping on the image of Levy's CBN embedding, $(-)^{\boxplus}$, is sound and complete with respect to the CBN subtyping rules of Figure 5. The proofs can be found in [49, Appendix L].

Theorem 7 (Soundness of Polarized Subtyping, Call-by-Name).

1. If t^\boxplus full, then t full.
2. If $t^\boxplus \leq u^\boxplus$, then $t \leq u$.

Theorem 8 (Completeness of Polarized Subtyping, Call-by-Name).

1. If t full, then t^\boxplus full.
2. If $t \leq u$, then $t^\boxplus \leq u^\boxplus$.

8.2 Call-by-Value

We can play through a similar procedure for Levy's CBV translation. Consider a CBV language with the following types. The language of terms, typing rules, and standard dynamics can be found in [49, Appendix M].

$$\tau, \sigma ::= \tau \rightarrow \sigma \mid \tau_1 \otimes \tau_2 \mid \mathbf{1} \mid \oplus\{\ell: \tau_\ell\}_{\ell \in L} \mid \&\{\ell: \sigma_\ell\}_{\ell \in L}$$

The translations that Levy [54] presents from CBV types and terms to CBPV *positive types* and *expressions* are as follows. We only present the translation of variables, function abstractions, and function applications; the full translation on terms can be found in [54].

<i>Types</i>	<i>Terms</i>
$(\tau \rightarrow \sigma)^\boxplus = \downarrow(\tau^\boxplus \rightarrow \uparrow\sigma^\boxplus)$	$(x)^\boxplus = \text{return } x$
$(\tau_1 \otimes \tau_2)^\boxplus = \tau_1^\boxplus \otimes \tau_2^\boxplus$	$(f)^\boxplus = \text{force } f \text{ for } f: \tau = e \in \Sigma$
$(\mathbf{1})^\boxplus = \mathbf{1}$	$(\lambda x. e)^\boxplus = \text{return } (\text{thunk } (\lambda x. e^\boxplus))$
$(\oplus\{\ell: \tau_\ell\}_{\ell \in L})^\boxplus = \oplus\{\ell: \tau_\ell^\boxplus\}_{\ell \in L}$	$(e_1 e_2)^\boxplus = \text{let return } x = e_2^\boxplus \text{ in}$
$(\&\{\ell: \sigma_\ell\}_{\ell \in L})^\boxplus = \downarrow\&\{\ell: \uparrow\sigma_\ell^\boxplus\}_{\ell \in L}$	$\text{let return } f = e_1^\boxplus \text{ in}$
	$(\text{force } f) x$

We also translate type names t to fresh type names t^\boxplus , translating the body of t 's definition and inserting additional type names as required for the normal form that alternates between structural types and type names.

Levy proves that well-typed terms translate to well-typed expressions. Because our syntactic typing rules are the same as his, his theorem carries over.

We adapt the CBV subtyping system of Ligatti et al. [55] to our setting, which means that we include variants and lazy records with width and depth subtyping and replace isorecursive with equirecursive types. We obtain the syntactic subtyping rules shown in Figure 6. Once again, we will take care to

$$\begin{array}{c}
\frac{t = t_1 \rightarrow t_2 \quad u = u_1 \rightarrow u_2 \quad u_1 \leq t_1 \quad t_2 \leq u_2}{t \leq u} \rightarrow_{\text{SUB}_V} \\
\\
\frac{t = t_1 \otimes t_2 \quad u = u_1 \otimes u_2 \quad t_1 \leq u_1 \quad t_2 \leq u_2}{t \leq u} \otimes_{\text{SUB}_V} \qquad \frac{t = \mathbf{1} \quad u = \mathbf{1}}{t \leq u} \mathbf{1}_{\text{SUB}_V} \\
\\
\frac{t = \oplus\{\ell: t_\ell\}_{\ell \in L} \quad u = \oplus\{j: u_j\}_{j \in J} \quad \forall(\ell \in L \setminus J): t_\ell \text{ empty} \quad \forall(\ell \in L \cap J): t_\ell \leq u_\ell}{t \leq u} \oplus_{\text{SUB}_V} \\
\\
\frac{t = \&\{\ell: t_\ell\}_{\ell \in L} \quad u = \&\{j: u_j\}_{j \in J} \quad (L \supseteq J) \quad \forall(j \in J): t_j \leq u_j}{t \leq u} \&_{\text{SUB}_V} \\
\\
\frac{t \text{ empty} \quad u = \sigma}{t \leq u} \perp_{\text{SUB}_V} \qquad \frac{t = t_1 \rightarrow t_2 \quad u = u_1 \rightarrow u_2 \quad u_1 \text{ empty}}{t \leq u} \top_{\text{SUB}_V}^{\rightarrow \rightarrow} \\
\\
\frac{t = \&\{\ell: t_\ell\}_{\ell \in L} \quad u = u_1 \rightarrow u_2 \quad u_1 \text{ empty}}{t \leq u} \top_{\text{SUB}_V}^{\& \rightarrow} \qquad \frac{t = t_1 \rightarrow t_2 \quad u = \&\{\}}{t \leq u} \top_{\text{SUB}_V}^{\rightarrow \&} \\
\\
\frac{t = t_1 \otimes t_2 \quad t_i \text{ empty}}{t \text{ empty}} \otimes_{\text{EMP}_V} \qquad \frac{t = \oplus\{\ell: t_\ell\}_{\ell \in L} \quad \forall(\ell \in L): t_\ell \text{ empty}}{t \text{ empty}} \oplus_{\text{EMP}_V} \\
\\
\text{(no emptiness rules for } \mathbf{1}, \rightarrow, \text{ and } \&\text{)}
\end{array}$$

Fig. 6. Circular Derivation Rules for Call-by-Value Subtyping

distinguish the CBV syntactic subtyping judgment, $t \leq u$, from CBPV syntactic subtyping by marking CBPV type names with pluses and minuses. The rules shown in Figure 6 build *circular derivations*.

These rules match those of Ligatti et al., with one minor exception that we will detail below. As we will prove, these rules are exactly those for which $t \leq u$ in the CBV language if and only if $t^{\boxplus} \leq u^{\boxplus}$ in the CBPV metalanguage.

Before proceeding to the proofs, a few remarks about these rules. First, unlike the CBN \oplus_{SUB_N} rule, the \oplus_{SUB_V} rule here includes the possibility that some components of a variant record type may be empty. More generally, the differences between CBN and CBV subtyping arise from the differences in emptiness and fullness between the two calculi. Emptiness and fullness are quite sensitive to the eager/lazy distinction between the two evaluation strategies. Because this distinction manifests in almost every layer of a complex type, the two subtyping systems diverge more than one might expect.

Second, besides the adaptations mentioned above, the rules of Figure 6 diverge from those of Ligatti et al. in only one way. Ligatti et al. [55] have the rule “ $t \leq u$ if $u = u_1 \rightarrow u_2$ and $u_1 \text{ empty}$ ” that generalizes the $\top_{\text{SUB}_V}^{\rightarrow \rightarrow}$, $\top_{\text{SUB}_V}^{\& \rightarrow}$, and $\top_{\text{SUB}_V}^{\rightarrow \&}$ rules of Figure 6 (assuming that Ligatti et al. would also have “ $t \leq u$ if $u = \&\{\}$ ” if they had included lazy records in their language).

Somewhat unexpectedly, polarized subtyping on the image of Levy’s CBV translation would be incomplete with respect to this more general rule. This is because the \downarrow shift inserted by Levy’s translation acts as a barrier to fullness: “ $t \leq u$ if $u = \downarrow r$ and r full” would be unsound in polarized subtyping. For example, Ligatti et al. have $\mathbf{1} \leq \mathbf{0} \rightarrow \mathbf{1}$ for an empty type $\mathbf{0}$, but we do not have $\mathbf{1}^{\boxplus} = \mathbf{1} \leq \downarrow(\mathbf{0} \rightarrow \uparrow\mathbf{1}) = (\mathbf{0} \rightarrow \mathbf{1})^{\boxplus}$ because the unit value $\langle \rangle$ does not have type $\downarrow(\mathbf{0} \rightarrow \uparrow\mathbf{1})$. This phenomenon is primarily of theoretical interest since it is confined to functions that can never be applied to any arguments and empty records (and only when they are compared against CBV types $t_1 \otimes t_2$, $\mathbf{1}$, and $\oplus\{\ell: t_\ell\}_{\ell \in L}$). Nevertheless, we conjecture a more differentiated translation of types and terms could restore completeness.

These observations notwithstanding, we can prove that the CBV subtyping rules of Figure 6 are sound and complete with respect to the subtyping rules for CBPV under Levy’s translation. The proofs can be found in [49, Appendix M].

Theorem 9 (Soundness of Polarized Subtyping, Call-by-Value).

1. If t^{\boxplus} empty, then t empty.
2. If $t^{\boxplus} \leq u^{\boxplus}$, then $t \leq u$.

Theorem 10 (Completeness of Polarized Subtyping, Call-by-Value).

1. If t empty, then t^{\boxplus} empty.
2. If $t \leq u$, then $t^{\boxplus} \leq u^{\boxplus}$.

9 Related Work and Discussion

We now dive deeper into research related to our underlying theme on how polarization affects the interaction and definition of subtyping with recursive types across varying interpretations.

Subtyping Recursive Types. The groundwork for coinductive interpretations of subtyping equirecursive types has been laid by Amadio and Cardelli [9], subsequently refined by others [13, 37]. Danielsson and Altenkirch [22] also provided significant inspiration since they formally clarify that subtyping recursive types relies on a mixed induction/coinduction. In using an equirecursive presentation within different calculi, our work has been influenced by its predominant use in session types [19, 23, 40] and, in particular, Gay and Hole’s coinductive subtyping algorithm [39], which we take as a template for call-by-name typing.

Another important influence has been the work on *refinement types* [24, 34] which are also recursive but exist within predefined universes of generative types. As such, subtyping relations are *simpler* in their interactions, but face many of the same issues such as emptiness checking. One can see this paper as an attempt to free refinement types from some of its restrictions while retaining some of its good properties. The key ingredients are (1) explicitly separating values from computations via polarization, (2) the introduction of variant and lazy records

and their width and depth subtyping rules (owing much to [70]), and (3) simple bidirectional typechecking. What is still missing is the use of *intersections* and *unions* that allow subtyping to propagate more richly to higher-order types [31].

Our treatment of empty—*value-uninhabited*—and full types in Section 4.1, as well as our call-by-value interpretation in Section 8.2 builds on Ligatti et al.’s work [55] on precise subtyping with isorecursive types.

Our direct interpretation of isorecursive types and translation into an equirecursive setting furthers numerous works either comparing or relating both formulations [67, 73, 74]. In particular, Abadi and Fiore [1] and more recently Patrigniani et al. [63] prove that terms in one equirecursive setting can be typed in the other (and vice versa) with varying approaches. The former treats type equality inductively and is focused on syntactic considerations. The latter treats type equality coinductively and analyzes types semantically. Neither of these handle subtyping or mixed coinductive/inductive types like in our study.

Finally, Zhou et al. [76] serves as a helpful overview paper on subtyping recursive types at large and discusses how Ligatti et al.’s complete set of rules requires very specific environments for subtyping, as well as non-standard subtyping rules. This observation demonstrates why our semantic typing/subtyping approach can offer a more flexible abstraction for reasoning about expressive type systems while maintaining type safety.

Semantic Typing and Subtyping. Semantic typing goes back to Milner’s *semantic soundness theorem* [57], which defined a well-typed program being semantically free of a type violation. Whereas syntactic typing specifies a fixed set of syntactic rules that safe terms can be constructed from, semantic typing here combines two requirements: positive types circumscribe observable values, *exposing their structure*, and computations of negative types are only required to *behave* in a safe way. As we demonstrate throughout section 5, we can prove our semantic definitions compatible with our syntactic type rules, leaving syntactic type soundness to fall out easily (Theorem 6).

Milner’s initial model didn’t scale well to richer types, like recursive types. With a lens toward more expressive systems, *step indexing* has become a prominent approach [7, 8, 10, 27], which we use to observe that a computation in our model *steps* according to our dynamics.

As with syntactic/semantic typing, syntactic subtyping is the more typical approach in modeling subtyping relations over its semantic counterpart. Nonetheless, in what’s operated almost parallel to the research on semantic types, research on semantic subtyping has also made strides [35, 15, 66]. Mainly, these exploit semantic subtyping for developing type systems based on set-theoretic subtyping relations and properties, particularly in the context of handling richer types, including polymorphic functions [17, 16, 65] and variants [18], recursive types (interpreted coinductively), and union, intersection, and negation connectives [36]. A major theme in this line of work is excising “circularity” [15, 36] by means of an involved bootstrapping technique, as issues arise when the denotation of a type is defined simply as the set of values having that type.

We depart from this line of research in the treatment of functions (defined computationally rather than set-theoretically), recursive types (equirecursive setting; inductive for the positive layer and coinductive for the negative layer), both variant and lazy record types, and the commitment to explicit polarization (including our incorporation of emptiness/fullness). The latter of which eliminates circularity and ties together multiple threads defined in this study.

With this combination of semantic typing and subtyping, our work provides a metatheory for a more interesting set of typed expressions while also providing a stronger and more flexible basis for type soundness [28], as semantic typing can reason about syntactically ill-typed expressions as long as those expressions are semantically well-typed. This combination scales well to our polarized, mixed setting and focus on subtyping in the presence of recursive types.

Polarized Type Theory and Call-by-Push-Value. At the core of this work has been the call-by-push-value [53, 54] (CBPV) calculus with its notions of values, computations, and the shifts between them. Beyond Levy’s work, this subsuming paradigm has formed the foundation of much recent research, ranging from probabilistic domains [33] to those reasoning about effects [56] and dependent types [64]. New et al.’s [60] gradual typing extension to the calculus shares similarities with our use of step indexing, but its relations (binary rather than unary), dynamics, and step-counting are treated differently, and its goals are very different as well, including no coverage on subtyping.

To our knowledge, there are no direct treatments of subtyping recursive types in a CBPV system or applying a full semantic typing approach in this context with subtyping. It is, as we’ve shown, a fruitful setting for our investigation since the explicit polarization of the language mirrors the mixed reasoning required to analyze the subtyping.

Though CBPV and polarized type theory typically go hand-in-hand, there are investigations that look at polarization (*focusing*) and algebraic typing and subtyping from alternate perspectives. Steffen [72] predates Levy’s research and presents polarity as a kinding system for exploiting monotone and antimonotone operators in subtyping function application. Abel [2] built upon this and extended it with sized types. The inherent connection between types and evaluation strategy has also been studied in the setting of program synthesis [71] and proof theory [58], but these do not share our specific semantic concerns.

Polarization as an organizing principle for subtyping is present in Zeilberger’s thesis [75], but addresses a problem that is fundamentally different in multiple ways, e.g. using “classical” types and continuations, and no width and depth subtyping. The biggest difference, however, is that its setting considers refinement types, while we do not have a refinement relation and show that some of the advantages of refinement types can be achieved without the additional layer.

Two studies on a global approach to algebraic subtyping [26, 62] define subtyping relationships with generative datatype constructors while discussing polarity (here with a different meaning) and discarding semantic interpretations. However, the generative nature of datatype constructors in this work makes its quite different from ours.

Mixed Coinductive/Inductive Reasoning for Recursive Types. The natural separation of positive and negative layers in CBPV led us through the literature on mixed coinductive/inductive definitions for recursive types. Related to our work in this paper, Danielsson and Altenkirch [22] and Jones and Pearce [46] provide definitions for equirecursive subtyping relations in a mixed setting while using a suspension monad for non-terminating computations, which shares an affinity with force/return CBPV computations. Danielsson and Altenkirch, however, do not try to justify the structural typing rules themselves via semantic typing of values or expressions—only the subtyping rules. Jones and Pearce are closer to our approach since they also use a semantic interpretation of types for expressions. While not polarized, they do consider inductive/coinductive types separately, but do not lift them to cover function types, instead studying other constructs such as unions.

Komendantsky [48] manages infinitary subtyping (for only function and recursive types) via a semantic encoding by folding an inductive relation into a coinductive one. We work in the opposite direction, turning the coinductive portion into an inductive one by step indexing. Lepigre and Raffali [52] mix induction and coinduction in a syntax-directed framework, focusing on circular proof derivations and sized types [6]; also managing inductive types coinductively. Cohen and Rowe [21] provide a proposal for circular reasoning in a mixed setting, but the focus is on a transitive closure logic built around least and greatest fixed point operators. It seems quite plausible that we could use such systems to formalize our investigation, although we found some merit in using step-indexing and Brotherston and Simpson’s circular proof system for induction [14].

10 Conclusion

We introduced a rich system of subtyping for an equirecursive variant of call-by-push-value and proved its soundness via semantic means. We also provided a bidirectional type checking algorithm and illustrated its expressiveness through several different kinds of examples. We showed the fundamental nature of the results by deriving systems of subtyping for isorecursive types and languages with call-by-name and call-by-value dynamics. The limitations of the present systems lie primarily in the lack of intersection and union types and parametric polymorphism which are the subject of ongoing work.

Acknowledgements. We wish to express our gratitude to the anonymous reviewers of this paper for their comments. Support for this research was provided by the NSF under Grant No. 1718276 and by FCT through the CMU Portugal Program, the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), and the project SafeSessions (PTDC/CCI-COM/6453/2020).

References

1. Abadi, M., Fiore, M.P.: Syntactic considerations on recursive types. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science. pp. 242–252. IEEE Computer Society (1996), <https://doi.org/10.1109/LICS.1996.561324>
2. Abel, A.: Polarized subtyping for sized types. In: Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3967, pp. 381–392. Springer (2006). https://doi.org/10.1007/11753728_39
3. Abel, A.: Mixed inductive/coinductive types and strong normalization. In: Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4807, pp. 286–301. Springer (2007). https://doi.org/10.1007/978-3-540-76637-7_19
4. Abel, A.: Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In: Miller, D., Ésik, Z. (eds.) Proceedings of the 8th Workshop on Fixed Points in Computer Science. pp. 1–11. FICS 2012, Electronic Proceedings in Theoretical Computer Science 77 (2012). <https://doi.org/10.4204/EPTCS.77.1>
5. Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Morrisett, G., Uustalu, T. (eds.) International Conference on Functional Programming (ICFP’13). pp. 185–196. ACM, Boston, Massachusetts (Sep 2013), <https://doi.org/10.1145/2500365.2500591>
6. Abel, A., Pientka, B.: Well-founded recursion with copatterns and sized types. Journal of Functional Programming **26**, e2 (2016), <https://doi.org/10.1017/S0956796816000022>
7. Ahmed, A.J.: Semantics of Types for Mutable State. Ph.D. thesis, Princeton University (2004), <http://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf>, aAI3136691
8. Ahmed, A.J.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) 15th European Symposium on Programming (ESOP 2006). pp. 69–83. Springer LNCS 3924, Vienna, Austria (Mar 2006). https://doi.org/10.1007/11693024_6
9. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems **15**(4), 575–631 (1993), <https://doi.org/10.1145/155183.155231>
10. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. Transactions on Programming Languages and Systems **23**(5), 657–683 (2001), <https://doi.org/10.1145/504709.504712>
11. Barwise, J.: The situation in logic, CSLI lecture notes series, vol. 17. CSLI (1989)
12. Berardi, S., Tatsuta, M.: Intuitionistic Podelski-Rybalchenko theorem and equivalence between inductive definitions and cyclic proofs. In: Ciirstea, C. (ed.) Workshop on Coalgebraic Methods in Computer Science (CMCS 2018). pp. 13–33. Springer LNCS 11202, Thessaloniki, Greece (Apr 2018), https://doi.org/10.1007/978-3-030-00389-0_3
13. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundamenta Informaticae **33**(4), 309–338 (1998), <https://doi.org/10.3233/FI-1998-33401>
14. Brotherston, J., Simpson, A.: Sequent calculi for induction and infinite descent. Journal of Logic and Computation **21**(6), 1177–1216 (2011), <https://doi.org/10.1093/logcom/exq052>

15. Castagna, G., Frisch, A.: A gentle introduction to semantic subtyping. In: Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal. pp. 198–199. ACM (2005), <https://doi.org/10.1145/1069774.1069793>
16. Castagna, G., Nguyen, K., Xu, Z., Abate, P.: Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 289–302. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2676726.2676991>
17. Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 5–17. POPL '14 (2014). <https://doi.org/10.1145/2535838.2535840>
18. Castagna, G., Petrucciani, T., Nguyen, K.: Set-theoretic types for polymorphic variants. Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (2016), <https://doi.org/10.1145/3022670.2951928>
19. Chen, T.C., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: Proceedings of the Conference on Principles and Practice of Declarative Programming (PPDP'14). ACM, Canterbury, UK (Sep 2014), <https://doi.org/10.1145/2643135.2643138>
20. Cockett, J.R.B.: Deforestation, program transformation, and cut-elimination. In: Coalgebraic Methods in Computer Science, CMCS 2001, a Satellite Event of ETAPS 2001, Genova, Italy, April 6-7, 2001. Electronic Notes in Theoretical Computer Science, vol. 44, pp. 88–127. Elsevier (2001), [https://doi.org/10.1016/S1571-0661\(04\)80904-6](https://doi.org/10.1016/S1571-0661(04)80904-6)
21. Cohen, L., Rowe, R.N.S.: Integrating induction and coinduction via closure operators and proof cycles. In: 10th International Joint Conference on Automated Reasoning (IJCAR 2020). pp. 375–394. Springer LNCS 12166, Paris, France (Jul 2020), https://doi.org/10.1007/978-3-030-51074-9_21
22. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In: 10th International Conference on Mathematics of Program Construction (MPC 2010). pp. 100–118. Springer LNCS 6120, Québec City, Canada (Jun 2010), https://doi.org/10.1007/978-3-642-13321-3_8
23. Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Nested session types. In: Yoshida, N. (ed.) 30th European Symposium on Programming. pp. 178–206. Springer LNCS, Luxembourg, Luxembourg (Mar 2021), <http://www.cs.cmu.edu/~fp/papers/esop21.pdf>, extended version available as arXiv:2010.06482
24. Davies, R.: Practical Refinement-Types Checking. Ph.D. thesis, Carnegie Mellon University (May 2005), <https://www.cs.cmu.edu/~rwh/students/davies.pdf>, available as Technical Report CMU-CS-05-110
25. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Wadler, P. (ed.) Proceedings of the Fifth International Conference on Functional Programming (ICFP'00). pp. 198–208. ACM Press, Montreal, Canada (Sep 2000), <https://doi.org/10.1145/351240.351259>
26. Dolan, S.: Algebraic Subtyping: Distinguished Dissertation 2017. BCS, Swindon, GBR (2017), <https://www.cs.tufts.edu/~nr/cs257/archive/stephen-dolan/thesis.pdf>
27. Dreyer, D., Ahmed, A., Birkedal, L.: Logical step-indexed logical relations. In: Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science,

- LICS 2009, 11-14 August 2009, Los Angeles, CA, USA. pp. 71–80. IEEE Computer Society (2009), <https://doi.org/10.1109/LICS.2009.34>
28. Dreyer, D., Timany, A., Krebbers, R., Birkedal, L., Jung, R.: What type soundness theorem do you really want to prove? (Oct 2019), <https://blog.sigplan.org/2019/10/17/what-type-soundness-theorem-do-you-really-want-to-prove>
 29. Dunfield, J., Krishnaswami, N.: Bidirectional typing. CoRR **abs/1908.05839** (2019), <http://arxiv.org/abs/1908.05839>
 30. Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. Proc. ACM Program. Lang. **3**(POPL), 9:1–9:28 (2019). <https://doi.org/10.1145/3290322>
 31. Dunfield, J., Pfenning, F.: Type assignment for intersections and unions in call-by-value languages. In: Gordon, A. (ed.) Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03). pp. 250–266. Springer-Verlag LNCS 2620, Warsaw, Poland (Apr 2003), https://doi.org/10.1007/3-540-36576-1_16
 32. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: X.Leroy (ed.) Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04). pp. 281–292. ACM Press, Venice, Italy (Jan 2004), <https://doi.org/10.1145/964001.964025>, extended version available as Technical Report CMU-CS-04-117, March 2004
 33. Ehrhard, T., Tasson, C.: Probabilistic call by push value. Log. Methods Comput. Sci. **15**(1) (2019), [https://doi.org/10.23638/LMCS-15\(1:3\)2019](https://doi.org/10.23638/LMCS-15(1:3)2019)
 34. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation. pp. 268–277. ACM Press, Toronto, Ontario (Jun 1991), <https://doi.org/10.1145/113445.113468>
 35. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 137–146. IEEE Computer Society (2002), <https://doi.org/10.1109/LICS.2002.1029823>
 36. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**, 19:1–19:64 (2008), <https://dl.acm.org/doi/10.1145/1391289.1391293>
 37. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed: functional pearl. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000. pp. 221–231. ACM (2000), <https://doi.org/10.1145/351240.351261>
 38. Garcia, R., Tanter, É.: Gradual typing as if types mattered. In: Informal Proceedings of the ACM SIGPLAN Workshop on Gradual Typing (WGT20) (2020), <https://wgt20.irif.fr/wgt20-final28-acmpaginated.pdf>
 39. Gay, S.J., Hole, M.: Subtyping for session types in the π -calculus. Acta Informatica **42**(2–3), 191–225 (2005), <https://doi.org/10.1007/s00236-005-0177-z>
 40. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. Journal of Functional Programming **20**(1), 19–50 (Jan 2010), <https://doi.org/10.1017/S0956796809990268>
 41. Grädel, E., Kreutzer, S.: Will deflation lead to depletion? On non-monotone fixed point inductions. In: Symposium on Logic in Computer Science (LICS 2003). pp. 158–167. IEEE Computer Society, Ottawa, Canada (Jun 2003), <https://doi.org/10.1109/LICS.2003.1210055>
 42. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, second edn. (Apr 2016)

43. Hermida, C., Jacobs, B.: Structural induction and coinduction in a fibrational setting. *Inf. Comput.* **145**(2), 107–152 (1998), <https://doi.org/10.1006/inco.1998.2725>
44. Hinrichsen, J.K., Louwrik, D., Krebbers, R., Bengtson, J.: Machine-checked semantic session typing. In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. pp. 178–198. ACM (2021). <https://doi.org/10.1145/3437992.3439914>
45. Jafery, K.A., Dunfield, J.: Sums of uncertainty: refinements go gradual. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. pp. 804–817. ACM (2017). <https://doi.org/10.1145/3009837.3009865>
46. Jones, T., Pearce, D.J.: A mechanical soundness proof for subtyping over recursive types. In: *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2016, Rome, Italy, July 17-22, 2016*. p. 1. ACM (2016). <https://doi.org/10.1145/2955811.2955812>
47. Jung, R., Jourdan, J., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* **2**(POPL), 66:1–66:34 (2018). <https://doi.org/10.1145/3158154>
48. Komendantsky, V.: Subtyping by folding an inductive relation into a coinductive one. In: *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 7193, pp. 17–32. Springer (2011), https://doi.org/10.1007/978-3-642-32037-8_2
49. Lakhani, Z., Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Polarized subtyping. *CoRR abs/2201.10998v1* (2022), <https://arxiv.org/abs/2201.10998v1>, extended version.
50. Lakhani, Z., Das, A., DeYoung, H., Mordido, A., Pfenning, F.: Polarized subtyping: Code/artifact (jan 2022). <https://doi.org/10.5281/zenodo.5913940>
51. Lepigre, R., Raffalli, C.: Subtyping-based type-checking for system F with induction and coinduction. *CoRR abs/1604.01990* (2016), <http://arxiv.org/abs/1604.01990>
52. Lepigre, R., Raffalli, C.: Practical subtyping for Curry-style languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **41**, 1 – 58 (2019), <https://doi.org/10.1145/3285955>
53. Levy, P.B.: Call-by-Push-Value. Ph.D. thesis, University of London (2001), <http://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf>
54. Levy, P.B.: Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* **19**(4), 377–414 (2006), <https://doi.org/10.1007/s10990-006-0480-6>
55. Ligatti, J., Blackburn, J., Nachtigal, M.: On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems* **39**(4), 4:1–4:36 (Mar 2017), <https://doi.org/10.1145/2994596>
56. McDermott, D., Mycroft, A.: Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11423, pp. 235–262. Springer (2019), https://doi.org/10.1007/978-3-030-17184-1_9
57. Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17**, 348–375 (Aug 1978), [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)

58. Munch-Maccagnoni, G.: Syntax and Models of a non-Associative Composition of Programs and Proofs. (Syntaxe et modèles d'une composition non-associative des programmes et des preuves). Ph.D. thesis, Paris Diderot University, France (2013), <https://tel.archives-ouvertes.fr/tel-00918642>
59. Nakata, K., Uustalu, T.: Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: an exercise in mixed induction-coinduction. In: Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010. EPTCS, vol. 32, pp. 57–75 (2010), <https://doi.org/10.4204/EPTCS.32.5>
60. New, M.S., Licata, D.R., Ahmed, A.: Gradual type theory. Proc. ACM Program. Lang. **3**(POPL), 15:1–15:31 (2019), <https://doi.org/10.1145/3290328>
61. Park, D.M.R.: On the semantics of fair parallelism. In: Bjørner, D. (ed.) Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings. Lecture Notes in Computer Science, vol. 86, pp. 504–526. Springer (1979), https://doi.org/10.1007/3-540-10007-5_47
62. Parreaux, L.: The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). Proc. ACM Program. Lang. **4**(ICFP), 124:1–124:28 (2020), <https://doi.org/10.1145/3409006>
63. Patrignani, M., Martin, E.M., Devriese, D.: On the semantic expressiveness of recursive types. Proceedings of the ACM on Programming Languages **5**, 1–29 (2021), <https://doi.org/10.1145/3434302>
64. Pédrot, P., Tabareau, N.: The fire triangle: how to mix substitution, dependent elimination, and effects. Proc. ACM Program. Lang. **4**(POPL), 58:1–58:28 (2020), <https://doi.org/10.1145/3371126>
65. Petrucciani, T.: Polymorphic set-theoretic types for functional languages. (Types ensemblistes polymorphes pour les langages fonctionnels). Ph.D. thesis, Sorbonne Paris Cité, France (2019), <https://tel.archives-ouvertes.fr/tel-02119930>
66. Petrucciani, T., Castagna, G., Ancona, D., Zucca, E.: Semantic subtyping for non-strict languages. In: 24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal. LIPIcs, vol. 130, pp. 4:1–4:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). <https://doi.org/10.4230/LIPIcs.TYPES.2018.4>, <https://arxiv.org/abs/1810.05555>
67. Pierce, B.: Types and Programming Languages. MIT Press (2002)
68. Pierce, B.C., Turner, D.N.: Local type inference. In: Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98) (1998), <https://doi.org/10.1145/268946.268967>, full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44
69. Raffalli, C.: L'arithmétique fonctionnelle du second ordre avec points fixes. Ph.D. thesis, Paris 7 (1994), <http://www.theses.fr/1994PA077080>, thèse de doctorat dirigée par Krivine, Jean-Louis Mathématiques. Logique et fondements de l'informatique Paris 7 1994
70. Reynolds, J.C.: Design of the programming language Forsythe. Tech. Rep. CMU-CS-96-146, Carnegie Mellon University (Jun 1996)
71. Rioux, N., Zdancewic, S.: Computation focusing. Proc. ACM Program. Lang. **4**(ICFP), 95:1–95:27 (2020). <https://doi.org/10.1145/3408977>
72. Steffen, M.: Polarized higher-order subtyping. Ph.D. thesis, University of Erlangen-Nuremberg, Germany (1999), <http://d-nb.info/958020493>
73. Urzyczyn, P.: Positive recursive type assignment. In: Mathematical Foundations of Computer Science 1995. pp. 382–391. Springer Berlin Heidelberg, Berlin, Heidelberg (1995), https://doi.org/10.1007/3-540-60246-1_144

74. Vanderwaart, J., Dreyer, D., Petersen, L., Crary, K., Harper, R., Cheng, P.: Typed compilation of recursive datatypes. In: Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003. pp. 98–108. ACM (2003), <https://doi.org/10.1145/604174.604187>
75. Zeilberger, N.: The Logical Basis of Evaluation Order and Pattern-Matching. Ph.D. thesis, Carnegie Mellon University, USA (2009), <http://noamz.org/thesis.pdf>
76. Zhou, Y., d. S. Oliveira, B.C., Zhao, J.: Revisiting iso-recursive subtyping. Proc. ACM Program. Lang. 4(OOPSLA), 223:1–223:28 (2020), <https://doi.org/10.1145/3428291>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

