Thesis Proposal:

Polarized Subtyping

Zeeshan Lakhani

August 16, 2024

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

Keywords: programming languages, subtyping, semantic-typing, type-systems, call-by-push-value

Abstract

Choices made in programming language and type system design involve tradeoffs between safety, simplicity, extensibility, and usability. Extensible features centered on type structure, like subtyping or mixing evaluation regime (lazy/eager) in functional programming, are often avoided or aborted due to the complexity of efficient type inference or the need for additional syntactic layers. Mainstream languages like TypeScript forsake soundness to capture typing scenarios for *all* of JavaScript and its now lengthy history of existent programs.

In this thesis, we introduce Polarized Subtyping, a flexible structural subtyping system grounded in the call-by-push-value paradigm, separating the language of types into two layers: a positive layer characterized by inductively defined, eagerly evaluated, observable values and a negative layer characterized by coinductively defined, lazily evaluated, possibly infinite computations—with adjoint modalities (or shifts) mediating between them. We extend the underlying call-by-push-value calculus with a decidable equirecursive subtyping variant that is both structural and semantic, forming a higher-order type system combining unfettered recursion, variant and lazy records, consequential property types like intersections, unions, and type difference, and (eventually) parametric polymorphism with subtyping and the interaction with effects at the heart of all these constructs.

Our approach moves beyond the traditional confines of syntactic type soundness by championing a semantic characterization of typing with step-indexing to capture the observation depth of recursive computations, from which we can immediately derive a form of semantic subtyping. This approach offers advantages in understanding complex type system features and maintaining behavioral safety when encountering varying evaluation regimes, nontermination, and computational effects.

Being explicit about values and computations while making subtyping first-class in our system opens up new possibilities for reasoning about the properties of functional programs and how type structure affects subtyping relations and enables compilation optimizations. Furthermore, to make the system practical, we present Polite, a reference implementation that demonstrates the feasibility of our approach. Polite gives us a platform to experiment with various type system features and optimizations in the future.

Acknowledgments

For anything good in here, all the kudos goes to my family (Heejung and Lana), advisor (Frank Pfenning), coworkers, collaborators, and coauthors who have put up with me along the way. For all the bad stuff, you can blame me directly.

I would also like to thank my employer, Oxide Computer, for hiring and not firing me during the course of this work. I am grateful for the flexibility and support they have provided me to pursue my academic interests while working full-time. More shout-outs go out to Brian Ginsburg, Doug Creager, Wode “Nimo” Ni, and Nathan Taylor for reviewing this proposal.

Finally, I would like to thank my thesis committee for their time reviewing this work and for ongoing feedback as we move forward.

Contents

Chapter 1

Introduction

“ *Type structure is a syntactic discipline for maintaining levels of abstraction* ”

— *John Reynolds*, Types, Abstraction, and Parametric Polymorphism [135]

“ *Subtyping, the golden key* ”

— *Aaron Stump*, Iowa Type Theory Commute [149]

Published in 1996, Cardelli’s *Type Systems* [24] begins by stating that “the fundamental purpose of a type system is to prevent the occurrence of type errors during the execution of a program.” This is followed up with the observation that “this [preceding] informal statement motivates the study of type systems, but requires clarification.” Years later, we are still working on clarifying this statement, iterating on capturing more precise and practical type errors effectively while adapting to the perilous interaction and usability concerns of more expressive type constructs surrounding subtyping, property types[1] [61] (e.g., intersections and unions), polymorphism, and effects. As type systems become more expressive, the unified framework of Church’s system of simple types, with multiple paradigms based on the same set of typing rules, continues to fracture [126]. Examples of this include the value restriction for parametric polymorphism in Standard ML [84], a similar restriction for intersection introduction while also discarding the distributivity law for subtyping in call-by-value languages with effects [45], the related (co)value restriction on union types [51, 55, 61, 165, 171], and the critical requirement of proving termination to reconcile laziness and refinement typing [163].

Choices made in programming language and type system design involve tradeoffs between safety, simplicity, extensibility, and usability. Extensible features centered on type structure, like subtyping or mixing evaluation regime (lazy/eager) in functional programming, are often avoided or aborted due to the complexity of efficient type inference or the need for additional syntactic layers. Mainstream languages like TypeScript support a long list of type system features, spanning unions and intersections (over objects) to distributed conditional types, yet forsake soundness unapologetically [156, 157, 158] to capture typing scenarios for *all* of JavaScript and its now

[1]Throughout this proposal, we use the term *property type* to refer to a type that may combine multiple properties (types) of the same expression. This follows from the line of work started by Freeman and Pfenning [67] and continued in Dunfield and Pfenning [61].

lengthy history of existent programs. By abandoning sound interoperation, Typescript and its cohort of languages, such as Hack [164] and MyPy [141], have abandoned one of the original goals of gradual typing [154], which was to provide soundness at the boundary between typed and untyped code. This highlights some of the challenges facing the adoption of sound gradual typing, namely the performance overhead and complexity associated with enforcing soundness [16, 151].

More conventional static type systems for functional programming, such as those in the ML [84] or Haskell [108] family of languages, perform typechecking at compile time and allow the expression of certain program invariants through type abstractions and polymorphism. However, these systems are often too coarse-grained to check many desirable properties, particularly those captured by first-class representations of relations between types, i.e., subtyping. Subtyping is not directly supported in many functional programming languages. When it is, it is often limited to restricted forms of subtyping over objects, such as row polymorphism in OCaml [74] or bounded polymorphism (along with intersections and unions) in Scala/Dotty [22, 81]. Additionally, in Scala, subtyping imposes restrictions on the form of recursive types for soundness reasons [177]. None of these existing systems provides a path forward for integrating richer forms of subtyping within a mixed-evaluation (lazy/eager) setting, nor do they address the interplay between subtyping and typechecking across unbounded recursive types and property types in the presence of effects.

More recent, less mainstream options like Liquid Types (Logically Qualified Data Types), a form of refinement types focused on logical predicates [80], strive for safety without sacrificing soundness, while also avoiding the heavy manual annotations typical of dependently typed systems. Yet, they impose other restrictions [163], affix distinct layers, and rely on external tools such as SMT solvers or embedded proofs [78] to refine type constraints. While liquid typing allows developers to specify and verify a variety of program properties at compile time, it does not reason about recursive computations that may diverge, making those programs not typeable [146]; instead, liquid typing opts to perform termination checking at the same time as regular typechecking [162].

This leads us to the following question:

> Is it possible to increase expressivity, remain sound, and adhere to a simpler typing discipline when capturing complex properties and catching errors without completely losing remnants of usability, modularity, and elegance that we usually expect from a type system?

In this thesis, we introduce Polarized Subtyping, a flexible structural subtyping system grounded in the call-by-push-value paradigm [102, 103, 105], separating the language of types into two layers: a positive layer characterized by inductively defined, eagerly evaluated, observable values and a negative layer characterized by coinductively defined, lazily evaluated, possibly infinite computations—with adjoint modalities [17, 131] (or shifts) mediating between them. We extend the underlying call-by-push-value calculus with decidable equirecursive subtyping [97] that is both structural and semantic, forming a higher-order type system combining unfettered recursion, variant and lazy records, consequential property types like intersections, unions, and type difference, and (eventually) parametric polymorphism with subtyping and the interaction with effects at the heart of all these constructs.

Our approach moves beyond the traditional confines of syntactic type soundness by championing a semantic characterization of typing with step-indexing [7, 8, 10, 52] to capture the observation depth of recursive computations, from which we can immediately derive a form of semantic subtyping [28, 69, 70]. This approach offers advantages in understanding complex type system features and maintaining behavioral safety when encountering varying evaluation regimes, nontermination, and computational effects.

Polarization is not typically an issue of language design, but of type structure [82]. Being explicit about values and computations while making subtyping first-class in our system opens up new possibilities for reasoning about the properties of functional programs and how type structure affects subtyping relations and enables compilation optimizations. Furthermore, to make the system practical, we present Polite [99], a reference implementation that demonstrates the feasibility of our approach. Throughout this proposal, we will showcase some examples that can be run and typechecked directly in the Polite language. Polite gives us a platform to experiment with various type system features and optimizations in the future.

My thesis statement summarizes the above:

Thesis Statement: Polarizing semantic subtyping provides a flexible and effective typechecking discipline for capturing interesting and precise properties of functional programs, particularly in the blending and interaction of advanced types such as intersections, unions, type difference, and polymorphism in order to make more properties expressible. This thesis ties together multiple, sometimes disparate threads related to mixed inductive-coinductive reasoning, property types and effects, equirecursive subtyping, and embedding call-by-name and call-by-value evaluation strategies.

1.1 An Introductory Example

To illustrate an example of the kind of expressivity we are aiming for, highlighting mixed polarities, recursive types, and width and depth subtyping, consider the following program related to one in our inaugural paper [97], which can be written directly in the Polite reference implementation: a stream of natural numbers with a finite amount of *padding* between consecutive numbers.

The intent is for the stream to be lazy and infinite, i.e., no end-of-stream is provided, and to act as a type of arbitrary depth. We do not restrict recursion, so even a well-typed implementation may diverge and fail to produce another number. While the streams here are lazy and infinite, padding must always be finite (eager) 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 is not strictly necessary, as we could just substitute out the definition of pstream.

```
type nat = +{'zero : 1, 'succ : nat}
type pos = +{            'succ : nat}

type pstream = <nat * padding> (* a padded stream *)
```

```
(* [] means thunk, not list! *)
(* finite, positive padding *)
type padding = +{'none : padding, 'some : [pstream]}
type zstream = <nat * +{'some : [zstream]}>

comp compress : [pstream] -> zstream
   = fun x0 => let <x1> = force x0 in
               match x1 { (x2,x3) => <x2, 'some [skip x3]> }
comp skip : padding -> zstream
   = fun x0 => match x0 { 'none x1 => skip x1
                        | 'some x2 => compress x2 }
(* evaluate *)
eval stream0 : pstream
   = <'zero (), 'none 'none 'none 'some [stream0]>
eval stream1 : zstream
   = compress [stream0]
```

Teasing this apart at a high level, nat is a positive value type representing a natural number, where **zero** is the unit type (**1**) and **succ** is a recursive type referencing nat. pos is a positive natural number, and we expect the subtyping relationship of pos $\leq$ nat to hold right off the bat, which is an example of width subtyping augmented with recursive subtyping! The type pstream (meaning *padded stream*) embeds a positive tuple/pair of a nat number and padding, wrapped in angle brackets ($<, >$). This, essentially, maps to a comonad[2], meaning it is embedding a positive value (a "return"), $\uparrow(\mathsf{nat}^+ \otimes \mathsf{padding}^+)$, and is treated as a negative (or lazy) type. padding defines a subtype with zero padding as a variant record made up of **none** and **some** labels, where the former represents no padding and the latter embeds a "thunk" [71], $\downarrow\mathsf{pstream}^-$, in square brackets ($[,]$), containing the monadic[3] **pstream**, which is a negative type contained in a positively typed thunk. Note that *return* and *thunk* embody the adjoint modalities (or shifts, hence the up/down arrows) that make up mixed evaluation, which we will formalize in detail in Chapter 2. zstream also embeds a positive pair, but with the first element being a nat number and the second a variant record type with only a unary label **some**, which we will return to in Section 2.6.1 for our interpretation of isorecursive types into equirecursive ones.

Given these types, we have the subtype relation zstream $\leq$ pstream, which means that we can pass a stream with zero padding into any function expecting one with arbitrary padding. Finally, to compute something useful, there are two mutually recursive functions (compress and skip) that create a stream with zero padding from a stream with arbitrary (but finite) padding, compressing it. In compress, the force operation *forces* the thunk, which is how thunks are evaluated (and eliminated) in our system.

Unlike sized type refinements [2, 6, 144], which can enforce properties on the computation

[2]Dual to monads, *comonads* provide composition over computational contexts, abstracting the notion of values within such a context [160].

[3]A *monad* abstracts the notion of a computation of a value [160], here, captured in a thunk.

of padding, such as finiteness, our type system focuses on the observation of computation steps rather than data constructors. However, we believe that our system is fully compatible with sized types and related frameworks, which are an extension beyond the scope of this thesis.

Conventional type systems struggle to capture the kind of relationship shown in this program, particularly the mixed polarity of the types involved and their connection to evaluation strategy. This example expresses eager (finite) and lazy (infinite *streams!*) evaluation directly in the type system! Despite the syntactic complexity of this example, it does not require a separate layer for refining types, proving properties, or explicitly annotating the subtypes. Later, we will briefly touch on the proposed long-term goal of using this system as an intermediate representation (IR) for optimized compilation and static analysis in Section 4.2.

In Chapter 3, we will see a few more examples of programs written in Polite, but with the additions of intersections, unions, and type difference.

1.2 The Challenge

In addressing *computational effects*, a field without a generally accepted formal definition [14], Filinski offers this challenge:

> The challenge to the semanticist is thus to admit the possibility of effects, while retaining as many as possible of the appealing properties of functional programming [66].

While computational effects are also part of this research, we cast a wider net.

> The challenge addressed in this thesis is to admit the possibility of integrating program evaluation, unbounded recursion, computational effects, and harnessing the expressive power and possible brittleness that comes with advanced type features within a straightforward structural type system, while retaining as many as possible of the appealing properties of functional programming—without introducing a plethora of new layers, rules, and restrictions that make the type system unwieldy or overly complex.

1.3 Contributions

The primary contributions of this work are as follows. Contributions that are well underway are marked in blue text; proposed contributions are marked in red text.

- A simple semantics for types and subtyping in call-by-push-value, separating the language of types into *positive* and *negative* layers. This polarization interprets positive types inductively and negative types coinductively (using mixed induction-coinduction [9, 19, 40]), realized using step indexing, deriving an elegant semantic form of typing and *semantic subtyping*.
- A new decidable system of equirecursive subtyping for call-by-push-value supporting subtyping for variant and lazy records (*width* and *depth* subtyping), property types (intersections,

unions, and type difference), and parametric polymorphism (also called generics).

- A derivation of related systems of subtyping for (a) isorecursive types, (b) call-by-name, and (c) call-by-value translated into our equirecursive call-by-push-value system, all using a structural rather than a nominal interpretation of types.
- An encapsulation of effects in our expressive polarized type system that interacts with subtyping, providing an updated analysis of the value restriction and related phenomena.
- A system of bidirectional typing that captures a straightforward and precise typechecking algorithm, including for intersections, unions, type difference, and polymorphism.
- A reference implementation, Polite, that demonstrates the feasibility of our approach, captures our expressive set of type features, and provides a platform for future experimentation with our system as an underlying intermediate representation.

There are no direct treatments of subtyping recursive types at the intersection of property types and parametric polymorphism in a call-by-push-value extension or applications of a total semantic typing approach in this context with subtyping. As we will show, this is a fruitful setting for this thesis, as the explicit polarization of the language mirrors the mixed reasoning required to analyze the subtyping relations.

1.4 Outline

This thesis proposal is broadly organized as follows. As before, the text marked in blue text refers to the work in progress, while proposed contributions are marked in red text.

Chapter 2. We detail our initial work on Polarized Subtyping, an equirecursive variant of call-by-push-value capturing the possibility of infinite computation by unbounded recursion. This includes a justification of both typing and subtyping through semantic means, a discussion on width and depth subtyping of records, a demonstration of our innovative use of Brotherston and Simpson's system *CLKID*$^\omega$ [21] of circular proofs to provide an elegant and flexible soundness proof for subtyping of *non-property* types, the basics of our bidirectional typing system for algorithmic typechecking, and the result of deriving systems of subtyping for isorecursive types and languages with call-by-name and call-by-value dynamics.

Chapter 3. This chapter focuses on the in-progress extension of our system to include higher-order property types, including intersections, unions, and type difference, and how we tame the complexity inherent in blending them together. We dive into how the introduction of these property types changes our approach to proving soundness of subtyping and what that means for the encapsulation of computational effects within our type system in relation to the value restriction and related phenomena like the linearization of intersection types [75].

Chapter 4. We cover the proposed work on extending our polarized system to include parametric polymorphism, specifically which approaches we will investigate. With polymorphism in tow, we can achieve an expressively complete system for Polarized Subtyping.

Chapter 5. This chapter delves into a discussion of related work central to our underlying theme: how polarization and a semantic representation of typing affect the interaction and definition of subtyping. It includes background on polarized type theory, semantic (sub)typing, recursive types, property types, and refinement types across varying interpretations.

Chapter 6. Finally, we present a timeline for the completion of the ongoing and proposed work as we head to the finish line, culminating in a thesis dissertation.

1.5 Abbreviations

CBN Call-by-name. 9, 18, 20, 21, 33, 42

CBPV Call-by-push-value. 8, 10, 12, 18, 20, 21, 38, 39, 44–46

CBV Call-by-value. 9, 18, 20, 21, 33, 42

IFF If and only if. 20, 42

LSB Least-significant bit. 24

MSB Most-significant bit. 24

OO Object-oriented. 31, 32, 42, 43

Chapter 2

Work to Date

Polarized Subtyping: An Equirecursive Variant of Call-by-push-value

“ *a value* is*, whereas a computation* does ”

— *Paul Blain Levy*, Call-by-push-value: Decomposing call-by-value and call-by-name [105]

Here, we present our established work on Polarized Subtyping, at the onset of this research without property types, serving as the bedrock of this thesis. We necessarily omit a full presentation of the material; most of the omitted details can be found in Lakhani et al. [97].

2.1 Technical Overview

As we have introduced, call-by-push-value [103, 105] is characterized by a separation of types into *positive* τ^+ and *negative* σ^- layers, with shift modalities ($\downarrow, \uparrow$) going back and forth between them. The intuition is that positive types classify *observable values* v while negative types classify *computations* e.

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

In our type system, the usual binary product $\tau \times \sigma$ splits into two: $\tau^+ \otimes \sigma^+$ for eager, directly observable products inhabited by pairs/tuples of values, and $\&\{\ell\colon \sigma_\ell^-\}_{\ell \in L}$ for lazy records[1] with a finite set L of fields we can project out. Binary sums are also generalized to variant record types $\oplus\{\ell\colon \tau_\ell^+\}_{\ell \in L}$[2]. Record representations allow for richer subtyping (see Section 5.1): lazy and

[1]We could have included lazy pairs as well, but opted for lazy records as a sufficient generalization. However, we still demonstrate binary products (tuples) and eager variant records, since pairs are more commonly encountered and provide a familiar context.

[2]We borrow the notation $\oplus$ from linear logic even though no linearity is implied.

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 $\oplus\{\mathbf{false}\colon \mathbf{1}\}$ is a subtype of $\mathsf{bool}^+ = \oplus\{\mathbf{false}\colon \mathbf{1}, \mathbf{true}\colon \mathbf{1}\}$, while $\mathbf{1}$ (the unit type) would not be a subtype of the usual binary $\mathbf{1} + \mathbf{1}$. Neither is $\mathbf{1}$ a subtype of bool^+, which demonstrates the utility of variant record types with one label, such as $\oplus\{\mathbf{false}\colon \mathbf{1}\}$. Similar examples exist for lazy record types. In this way, we recover some of the benefits of refinement types without the syntactic burden of a distinct refinement layer or a separate object system like OCaml with polymorphic variants.

The shift $\downarrow\sigma^-$[3] is inhabited by an unevaluated computation of type σ^- (a *thunk*). Conversely, the shift $\uparrow\tau^+$ includes a value as a trivial computation (a *return*). These appear in Levy's [103] calculus under a different notation, which uses $U\underline{B}$ instead of $\downarrow\sigma^-$ and FA instead of $\uparrow\tau^+$. Referring back to the example in Section 1.1, square brackets ([,]) in the type signature maps to $\downarrow\sigma^-$ and angle brackets (<, >) to $\uparrow\tau^+$.

Function types in this system map arguments of a value type (domain) to results of a computation type (codomain). Historically, this can also translate to modeling the operational semantics of a CK-machine [65], where the computation expression $\tau^+ \to \sigma^-$ pops a value of type τ^+ and proceeds as a computation of type σ^- [104]. In Section 2.6.2, we demonstrate how function types translate to model ML-like (CBV) and Haskell-like (CBN) type expressions—using shifts!

We model recursive types not by explicit constructors $\mu\alpha^+.\,\tau^+$ and $\nu\alpha^-.\,\sigma^-$, typical of isorecursive systems, but by *type names* t^+ and s^- which are defined in a global signature Σ. They may mutually refer to each other. We treat type names as *equirecursive* and require them to be *contractive*, which means that the right-hand side of a type definition cannot itself be a type name. Since we would like to observe the values of positive types directly, the definitions of type names $t^+ = \tau^+$ are *inductive*. This allows inductive reasoning about the 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.

Our use of unique type names plays a crucial role in our subtyping relations and implementation throughout this work. Note that we refer to the term *recursive type* as a catchall for mixed inductive-coinductive types.

2.1.1 The Syntax of Values & Computations

The syntax for values v of a positive type and computations e of a negative type are as follows.

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

Variables x always stand for values and therefore have a positive type. We use j to stand for labels, which name fields of variant records or lazy records, where $j \cdot v$ injects value v into a sum

[3]The downshift represents a suspended computation and the upshift denotes computations producing a value [96].

with an 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.

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. Following Levy, we do not allow names as values because this would add an undesirable notion of computation to values. An alternative would have been explicit fixed point expressions 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 we will model later (see Chapter 3) when using intersection types.

2.1.2 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. By contrast, values do not reduce (i.e. take a step). The dynamics of our extension to CBPV are defined as follows.

$$\frac{}{(\lambda x.\, e)\, v \mapsto [v/x]e} \qquad \frac{e \mapsto e'}{e\, v \mapsto e'\, v} \qquad \frac{}{\textsf{let return } x = \textsf{return } v \textsf{ in } e_2 \mapsto [v/x]e_2}$$

$$\frac{e_1 \mapsto e_1'}{\textsf{let return } x = e_1 \textsf{ in } e_2 \mapsto \textsf{let return } x = e_1' \textsf{ in } e_2} \qquad \frac{(j \in L)}{\{\ell = e_\ell\}_{\ell \in L}.j \mapsto e_j} \qquad \frac{e \mapsto e'}{e.j \mapsto e'.j}$$

$$\frac{}{\textsf{match } \langle v_1, v_2 \rangle\ (\langle x, y \rangle \Rightarrow e) \mapsto [v_1/x][v_2/y]e} \qquad \frac{}{\textsf{match } \langle \rangle\ (\langle \rangle \Rightarrow e) \mapsto e}$$

$$\frac{(j \in L)}{\textsf{case } (j \cdot v)\ (\ell \cdot x_\ell \Rightarrow e_\ell)_{\ell \in L} \mapsto [v/x_j]e_j} \qquad \frac{}{\textsf{force } (\textsf{thunk } e) \mapsto e} \qquad \frac{f : \sigma^- = e \in \Sigma}{f \mapsto e}$$

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

$$\frac{}{\lambda x.\, e \textsf{ terminal}} \qquad \frac{}{\{\ell = e_\ell\}_{\ell \in L} \textsf{ terminal}} \qquad \frac{}{\textsf{return } v \textsf{ terminal}}$$

2.2 Example: A Program Gone Wrong

Our introductory example (Section 1.1) demonstrates a well-typed program, but it is always good to see what happens when things go wrong. Here, we show a decrement function, $\textsf{dec}_0$, for binary numbers in "little-endian" representation (least significant bit first). This example is directly formalized in our types and with our dynamics (e.g., return here maps to our angle bracket $(<, >)$

syntax used within functions in Polite for let expressions, etc.), which fails in our system due to a subtyping error.

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

$$
\begin{aligned}
\mathsf{dec}_0\ &: \mathsf{pos} \rightarrow {\uparrow}\mathsf{std}\\
&= \lambda x.\, \mathsf{match}\ x\ (\,\mathbf{b0}\cdot x' \Rightarrow \mathsf{let\ return}\ y' = \mathsf{dec}_0\, x'\ \mathsf{in\ return}\ \mathbf{b1}\cdot y'\\
&\qquad\qquad\qquad \mid \mathbf{b1}\cdot x' \Rightarrow \mathsf{return}\ \mathbf{b0}\cdot x'\,)
\end{aligned}
$$

The function's input is a *positively-typed* positive binary number, x, and it *returns* a binary number in standard form (no leading zeros), which is negatively typed, hence the up($\uparrow$)-shift. The error here is quite precisely identified by the bidirectional type checker (see Section 2.5). When we inject $\mathbf{b0}\cdot x'$ in the second branch it is not the case that $x' : \mathsf{pos}$ as is required for standard numbers! In fact, $\mathsf{dec}_0\, \mathbf{b1}\cdot\mathbf{e}\cdot\langle\rangle \mapsto^* \mathsf{return}\ \mathbf{b0}\cdot\mathbf{e}\cdot\langle\rangle$ 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 error by discriminating one more level of the input (which could be made slightly more appealing by a compound syntax for nested pattern matching).

$$
\begin{aligned}
\mathsf{dec}\ &: \mathsf{pos} \rightarrow {\uparrow}\mathsf{std}\\
&= \lambda x.\, \mathsf{match}\ x\ (\,\mathbf{b0}\cdot x' \Rightarrow \mathsf{let\ return}\ y' = \mathsf{dec}\, x'\ \mathsf{in\ return}\ \mathbf{b1}\cdot y'\\
&\qquad\qquad\qquad \mid \mathbf{b1}\cdot x' \Rightarrow \mathsf{match}\ x'\ (\,\mathbf{e}\cdot u \Rightarrow \mathsf{return}\ \mathbf{e}\cdot u\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b0}\cdot x'' \Rightarrow \mathsf{return}\ \mathbf{b0}\cdot\mathbf{b0}\cdot x''\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1}\cdot x'' \Rightarrow \mathsf{return}\ \mathbf{b0}\cdot\mathbf{b1}\cdot x''\,)\,)
\end{aligned}
$$

2.3 Semantic (Sub)typing

The notion of semantic typing can be understood as a combination of inductive and coinductive definitions. Values are typed inductively, which yields the correct interpretation of purely positive types such as natural numbers, lists, or trees, which all describe finite data structures. Computations are typed coinductively because they include the possibility of infinite computation by unbounded recursion. Although we assume that we can observe the structure of values, computations e cannot be observed directly, because like streams they may be infinite.

To justify both typing and subtyping by semantic means, we employ *semantic typing* of closed values and computations, written $v \in \tau^+$ and $e \in \sigma^-$ respectively. From this we can, for example, define semantic subtyping for positive types $\tau^+ \subseteq \sigma^+$ as $\forall v. v \in \tau^+ \supset v \in \sigma^+$.

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 that 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 $\tau^+ \to \sigma^-$ and $\&\{\ell\colon \sigma_\ell^-\}_{\ell \in L}$) 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. We define these semantic properties as follows.

Semantic Preservation [97, Section 4] If $e \in \sigma^-$ and $e \mapsto e'$ then $e' \in \sigma^-$.

Semantic Progress [97, Section 4] If $e \in \sigma^-$ then either $e \mapsto e'$ for some e' or e is *terminal* (but not both). Progress here captures the usual slogan that "*well-typed programs do not go wrong*" [111]. 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 $e = \mathsf{return}\ v$ with $v \in \tau^+$.

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

These semantics are similar to their syntactic counterparts, but the fact that we do not rely on any form of syntactic typing is methodologically significant—a point driven home by Dreyer et al. [53] for some time (see Section 5.2 for some pragmatic use cases). For example, if we have a program that does not obey a syntactic typing discipline or is even syntactically well-formed, like the embedding of the untyped λ-calculus (e.g. $(\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$), but behaves correctly according to our semantic typing, our results will apply. In turn, 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) if it terminates.

Although, we will discuss the interplay between Polarized Subtyping in the presence of effects later in this proposal (Section 3.4), we can already see that the semantic typing approach is well suited for reasoning about the behavior of computational effects, to which CBPV was designed to accommodate with its distinction between values and computations [103, Section 2.4]. Through the lens of semantic typing, we can ensure behavioral soundness in the presence of effects.

2.3.1 Capturing Observation Depth of Recursive Computations

There are different frameworks for achieving the mix of inductive and coinductive reasoning methods [3, 15, 34, 35, 40, 86, 94, 100, 116, 119, 132, 144], including approaches such as guarded recursion [115] and sized types, the latter of which we have already touched upon. To simplify our presentation, we have chosen step-indexing [7, 8, 10, 52] to present our semantic definition by turning the coinductive proof into an inductive one, which does not require an additional formal device [11]. 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. The step-index is in fact 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 reduce.

$$\begin{aligned}
v \in_k t &\triangleq v \in_k \tau^+ \text{ for } t = \tau^+ \in \Sigma \\
v \in_k \tau_1^+ \otimes \tau_2^+ &\triangleq v = \langle v_1, v_2 \rangle,\, v_1 \in_k \tau_1^+, \text{ and } v_2 \in_k \tau_2^+ \text{ for some } v_1, v_2 \\
v \in_k \mathbf{1} &\triangleq v = \langle\rangle \\
v \in_k \oplus\{\ell \colon \tau_\ell^+\}_{\ell \in L} &\triangleq v = j \cdot v_j \text{ and } v_j \in_k \tau_j^+ \text{ for some } j \in L \\
v \in_k {\downarrow}\sigma^- &\triangleq v = \mathsf{thunk}\ e \text{ and } e \in_k \sigma^- \text{ for some } e
\end{aligned}$$

$$\begin{aligned}
e \in_0 \sigma^- &\quad \text{always} \\
e \in_{k+1} \sigma^- &\triangleq (e \mapsto e' \text{ and } e' \in_k \sigma^-) \text{ or } (e \ \mathsf{terminal} \text{ and } e \mathrel{\hat{\in}_{k+1}} \sigma^-) \\
e \mathrel{\hat{\in}_{k+1}} s &\triangleq e \mathrel{\hat{\in}_{k+1}} \sigma^- \text{ for } s = \sigma^- \in \Sigma \\
e \mathrel{\hat{\in}_{k+1}} \tau^+ \to \sigma^- &\triangleq e\, v \in_{k+1} \sigma^- \text{ for all } i \leq k \text{ and } v \text{ with } v \in_i \tau^+ \\
e \mathrel{\hat{\in}_{k+1}} \&\{\ell \colon \sigma_\ell^-\}_{\ell \in L} &\triangleq e.j \in_{k+1} \sigma_j^- \text{ for all } j \in L \\
e \mathrel{\hat{\in}_{k+1}} {\uparrow}\tau^+ &\triangleq e = \mathsf{return}\ v \text{ for some } v \in_k \tau^+
\end{aligned}$$

$$\begin{aligned}
v \in \tau^+ &\triangleq v \in_k \tau^+ \text{ for all } k \\
e \in \sigma^- &\triangleq e \in_k \sigma^- \text{ for all } k
\end{aligned}$$

Figure 2.1: Definition of Semantic Typing

The clauses for our definition of semantic typing can be found in figure 2.1. We leverage three main judgments (where all step indices k, i, j range over natural numbers):

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

These judgments are 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'. As we can see, the definition of semantic typing is quite straightforward, and the step-indexing allows us to capture the depth of observation for recursive computations.

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 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.

Determining how step-indexing is affected by dynamics is not necessarily an objective endeavor. There are variations in the literature on how to handle this, including not decreasing the index unless recursion is unrolled [8, 52, 117] to characterize nontermination, or keeping the index constant when analyzing a terminal computation consisting of a *return*. Stripping the return constructor constitutes a form of observation, and therefore decreasing the index seems both appropriate and simplest.

Note that 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, 79]. Therefore, values and computations are semantically well-typed if they are well-typed for *all* step indices.

2.4 Subtyping

We have focused on semantic typing thus far, but we will now prioritize a syntactic definition of subtyping that expresses an algorithm and shows it both sound and complete with respect to the given semantic definition.

To ground this, we start with a definition of 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.

One necessary thorn on the side of subtyping in the presence of recursive types is how to handle empty or value-uninhabited types [95, 106]. Because values are always observable, how do we determine and prove τ^+ is empty in a subtyping relation? What does this mean for negative types?

In this section, we briefly dive into how empty and full types interact with subtyping, highlight our soundness result for subtyping, and demonstrate our novel proof technique for single-suceedent judgments, or *consequents*[4] that proves syntactic versions of typing and subtyping are sound regarding our semantic definitions.

2.4.1 Closing the Circle

In this foundational presentation, we use Brotherston and Simpson's system *CLKID*$^\omega$ of circular proofs [21] to carry out our metatheory. In this system, the succedent of any sequent is either empty or a singleton [18]. Although it is not the usual approach, utilizing proofs for dealing with subtyping is advantageous in this consequent-restricted representation, centered on step-indexing and recursive types, as it has finitely many distinct subderivations, making it finitely presentable for nested types. Later, we will present a more streamlined representation in the Chapter on property types (see Section 3.2), whose rules are not constrained to consequents.

Cyclic reasoning identifies repeated sections of the proof (cycles), and various guardedness conditions are imposed on the proofs to ensure their soundness [20]. Furthermore, cyclic proofs are an alternative to explicit induction; that is, without explicit induction rules, as the induction argument is essentially created implicitly through the discovery of a cyclic proof. This approach succinctly ties together the expansion of our semantic definition (Figure 2.1) with our subtyping

[4]A *consequent* [118] refers to having a single succedent type on the right of a subtyping relation ($\leq$ in subtyping judgments and $\vdash$ in semantic ones).

relation. The next sections will demonstrate examples of our circular proofs for subtyping which are sound with respect to our semantic definitions.

2.4.2 Emptiness (and Fullness)

A straightforward observation is that $\tau^+ \subseteq \sigma^+$ whenever τ^+ is an empty type, regardless of σ^+, because the necessary implication holds vacuously. Suitable for both presentation and implementation, we first put the signature into a normal form that alternates between structural types and type names.

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

As an example, the type $t_0 = \mathbf{1} \otimes t_0$ is empty because we may assume that t_0, where t_0 is a positive type name, is empty while testing $\mathbf{1} \otimes t_0$. Let us build a formal circular derivation for this example, first by bringing the signature $\Sigma = \{u_0 = \mathbf{1},\ \ t_0 = u_0 \otimes t_0\}$ into normal form, and then constructing this proof.

$$\dfrac{t_0 = u_0 \otimes t_0 \qquad \overset{\text{CYCLE}()}{t_0 \ \mathsf{empty}}}{t_0 \ \mathsf{empty}} \ \otimes\text{EMP}_2$$

This derivation follows seamlessly from the rule(s) in Figure 2.2. We construct a circular derivation for t_0 empty and form a valid *cycle* when we encounter a goal t_0 empty as a proper subgoal of t_0 empty.

With our infinitary treatment of negative types, they are *never empty*. Instead, we symmetrically define a computation type σ^- to be *full*, namely that it is inhabited by *every (semantically well-typed) computation*. Derivations may be constructed from the rules in Figure 2.2. 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 that fullness is directly derived from emptiness non-recursively.

The theorems for emptiness and fullness can be found in [98, Appendices D and E].

2.4.3 Syntactic Subtyping

The rules for syntactic subtyping (see Figure 2.3) build a *circular derivation* of $t^+ \leq u^+$ and $s^- \leq r^-$. A circularity arises when a goal $t \leq u$ or $s \leq r$ arises as a subgoal strictly above a goal that is of one of these two forms. For example, let us construct a valid circular proof for an obvious, *positive* relation that holds: $\mathsf{pos} \leq \mathsf{nat}$, a relation exhibiting width and recursive subtyping, previously mentioned in the introductory example (Section 1.1). The types and their signatures are defined as follows.

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

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

$$\frac{s = t_1 \to s_2 \in \Sigma \quad t_1 \text{ empty}}{s \text{ full}}\ {\to}\text{FULL} \qquad \frac{s = \&\{\ \} \in \Sigma}{s \text{ full}}\ \&\text{FULL} \qquad \text{(no rule for } s = {\uparrow} t\text{)}$$

Figure 2.2: Circular Derivation Rules for Emptiness (Positive Types) & Fullness (Negative Types)

$$\begin{aligned} \mathsf{zero}^+ &= \oplus\{\mathbf{zero} : \mathbf{1}\} \\ \mathsf{nat}^+ &= \oplus\{\mathbf{zero} : \mathbf{1}, \mathbf{succ} : \mathsf{nat}\} \\ \mathsf{pos}^+ &= \oplus\{\qquad\qquad \mathbf{succ} : \mathsf{nat}\} \end{aligned}$$

Now, let us look at the circular derivation of the subtyping rule. We annotate each subgoal from the $\oplus$SUB rule with the corresponding label on the record and mark the cycle with an asterisk.

$$\dfrac{\dfrac{}{[\mathbf{zero}]\ \mathbf{1} \leq \mathbf{1}}\ \mathbf{1}\text{SUB} \qquad \dfrac{\text{CYCLE}(*)}{[\mathbf{succ}]\ \mathsf{nat} \leq \mathsf{nat}}}{\dfrac{[\mathbf{succ}]\ \mathsf{nat} \leq \mathsf{nat}\ (*)}{\mathsf{pos} \leq \mathsf{nat}}}$$

The relationship between $t \leq u$ and $v \in_k t \vdash v \in_k u$ interprets the subtyping relation ($t \leq u$) as $t \subseteq u$, that is, every value in t is also a value in u for all k and v.

In the next representation, we show a circular derivation of the subtyping relation in the metatheory using our semantic definitions (see Figure 2.1). This process also concludes with a cycle. The goal is to break down the subtyping relation into a series of subgoals, based on how types are nested, each representing a subtyping relation between smaller types.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{v'' \in_k \mathbf{1} \vdash v'' \in_k \mathbf{1}}\ id}{v'' \in_k \mathbf{1} \vdash \mathbf{zero} \cdot v'' \in_k \mathsf{nat}}\ \in_{\text{R}}}{v' = \mathbf{zero} \cdot v'', v'' \in_k \mathbf{1} \vdash v' \in_k \mathsf{nat}}\ =_{\text{L}} \qquad \dfrac{\dfrac{\dfrac{\text{CYCLE}(*)_{([v''/v])}}{v'' \in_k \mathsf{nat} \vdash v'' \in_k \mathsf{nat}}}{v'' \in_k \mathsf{nat} \vdash \mathbf{succ} \cdot v'' \in_k \mathsf{nat}}\ \in_{\text{R}}}{v' = \mathbf{succ} \cdot v'', v'' \in_k \mathsf{nat} \vdash v' \in_k \mathsf{nat}}\ =_{\text{L}}}{\dfrac{\dfrac{\dfrac{v' \in_k \mathsf{nat} \vdash v' \in_k \mathsf{nat}\ (*)}{v' \in_k \mathsf{nat} \vdash \mathbf{succ} \cdot v' \in_k \mathsf{nat}}\ =_{\text{R}}}{v = \mathbf{succ} \cdot v', v' \in_k \mathsf{nat} \vdash v \in_k \mathsf{nat}}\ =_{\text{L}}}{v \in_k \mathsf{pos} \vdash v \in_k \mathsf{nat}}\ \in_{\text{L}}}\ \oplus_{\text{L}}, \in_{\text{L}}$$

Proof details of the soundness of subtyping theorem can be found in [98, Appendix F]. We

$$\frac{t = t_1 \otimes t_2 \quad u = u_1 \otimes u_2 \quad t_1 \leq u_1 \quad t_2 \leq u_2}{t \leq u} \otimes\text{SUB} \qquad \frac{t = \mathbf{1} \quad u = \mathbf{1}}{t \leq u} \mathbf{1}\text{SUB}$$

$$\frac{t = \oplus\{\ell : t_\ell\}_{\ell \in L} \quad u = \oplus\{k : u_k\}_{k \in K} \quad \forall \ell \in L.\, t_\ell \text{ empty} \vee (\ell \in K \wedge t_\ell \leq u_\ell)}{t \leq u} \oplus\text{SUB}$$

$$\frac{t = {\downarrow}s \quad u = {\downarrow}r \quad s \leq r}{t \leq u} {\downarrow}\text{SUB} \qquad \frac{s = t_1 \to s_2 \quad r = u_1 \to r_2 \quad u_1 \leq t_1 \quad s_2 \leq r_2}{s \leq r} {\to}\text{SUB}$$

$$\frac{s = {\uparrow}t \quad r = {\uparrow}u \quad t \leq u}{s \leq r} {\uparrow}\text{SUB}$$

$$\frac{s = \&\{\ell : s_\ell\}_{\ell \in L} \quad r = \&\{j : r_j\}_{j \in K} \quad \forall j \in K.\, j \in L \wedge s_j \leq r_j}{s \leq r} \&\text{SUB}$$

$$\frac{t \text{ empty} \quad u = \tau^+}{t \leq u} \bot\text{SUB}^+ \qquad \frac{s = {\uparrow}t \quad t \text{ empty} \quad r = \sigma^-}{s \leq r} \bot\text{SUB}^- \qquad \frac{s = \sigma^- \quad r \text{ full}}{s \leq r} \top\text{SUB}$$

Figure 2.3: Circular Derivation Rules for Subtyping

capture the theorem here.

Theorem 1 (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, $s \subseteq r$).*

The proof cases for each subtyping rule follow the same basic pattern: we construct a version of the proof proceeding by a compositional translation of the circular derivation of subtyping into a circular derivation in the intuitionistic metalogic [18]. When the subtyping proof is closed due to a cycle, we close the proof in the metalogic with a corresponding cycle. A valid cycle demonstrates 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.

Taking into account the step-indexing of computations, we separate the cases for $k = 0$ and $k > 0$ with $e \mapsto e'$ because the argument is essentially the same except in the case of $\top$SUB. When $k > 0$ and e terminal we distinguish cases according to the various rules.

Besides soundness, reflexivity and transitivity of syntactic subtyping are two other properties that we prove to ensure that the syntactic subtyping rules are sensible and have no obvious gaps—a common exercise. These proofs can be found in [98, Appendix G]. We can straightforwardly prove that the syntactic subtyping for purely positive types is complete with respect to semantic subtyping. Note that we do not delve into the notion of preciseness in this thesis (for the subset of syntactic subtyping of negative types), considered by Ligatti et al. [106]. It is a property that's highly language-sensitive, dependent on very specific environments for subtyping, as well as non-standard subtyping rules [175].

2.5 Bidirectional Typing

While our previous work presents both a declarative and an algorithmic typing procedure for our calculus, we will focus on the latter: a bidirectional typechecking system that is more practical, user-friendly (less type annotations), and amenable to implementation. Bidirectional typechecking [129] has been a popular choice for algorithmic typing, especially when it comes to subtyping [60], and is decidable for a wide range of rich type systems. It avoids inference issues around subsumption [90] and with our extensive use of type names and variant records. In addition, it has a long history with polarized logics [59, 63]. Moreover, bidirectional typing is quite robust with respect to language extensions whereas various inference procedures are not.

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:

$\Gamma \vdash v \Leftarrow \tau^+$	$\Gamma \vdash v \Rightarrow \tau^+$	In context Γ, value v either checks or synthesizes type τ^+
$\Gamma \vdash e \Leftarrow \sigma^-$	$\Gamma \vdash e \Rightarrow \sigma^-$	In context Γ, expression e either checks or synthesizes type σ^-

The rules for our system, given subtyping, can be found in Figure 2.4, including the rules for positive and negative subsumption. These rules require that the types in annotations be translated into normal form, possibly introducing new (user-invisible) definitions in the signature. 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.

The theorems (with straightforward proofs) for soundness and completeness of our recipe for bidirectional algorithmic typing can be found in [98, Appendix J].

Due to our use of equirecursive types, the implementation of this system can closely follow the structure of the rules in Figures 2.2, 2.3, and 2.4, leveraging a memoization table to construct circular derivations of emptiness, as well as to lazily construct circular derivations of subtyping for positive and negative type names (bottom-up).

2.6 Deriving Related Systems of Subtyping

Levy's CBPV calculus provides a foundational framework that facilitates the translation of evaluation strategies and the incorporation of alternative typing relations, such as isorecursive types, enabling a uniform embedding of both ML-like (CBV) and Haskell-like (CBN) languages within a common core intermediate language.

$$\frac{\Gamma \vdash v_1 \Leftarrow \tau_1^+ \quad \Gamma \vdash v_2 \Leftarrow \tau_2^+}{\Gamma \vdash \langle v_1, v_2 \rangle \Leftarrow \tau_1^+ \otimes \tau_2^+}\ \otimes\mathrm{I} \qquad \frac{\Gamma \vdash v \Rightarrow \tau_1^+ \otimes \tau_2^+ \quad \Gamma, x{:}\tau_1^+, y{:}\tau_2^+ \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \mathsf{match}\ v\ (\langle x, y \rangle \Rightarrow e) \Leftarrow \sigma^-}\ \otimes\mathrm{E}$$

$$\frac{x : \tau^+ \in \Gamma}{\Gamma \vdash x \Rightarrow \tau^+}\ \mathrm{VAR} \qquad \frac{}{\Gamma \vdash \langle\rangle \Leftarrow \mathbf{1}}\ \mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash v \Rightarrow \mathbf{1} \quad \Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \mathsf{match}\ v\ (\langle\rangle \Rightarrow e) \Leftarrow \sigma^-}\ \mathbf{1}\mathrm{E}$$

$$\frac{\Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \mathsf{thunk}\ e \Leftarrow {\downarrow}\sigma^-}\ {\downarrow}\mathrm{I} \qquad \frac{\Gamma \vdash v \Rightarrow {\downarrow}\sigma^-}{\Gamma \vdash \mathsf{force}\ v \Rightarrow \sigma^-}\ {\downarrow}\mathrm{E} \qquad \frac{(j \in L) \quad \Gamma \vdash v \Leftarrow \tau_j^+}{\Gamma \vdash j \cdot v \Leftarrow \oplus\{\ell \colon \tau_\ell^+\}_{\ell \in L}}\ \oplus\mathrm{I}$$

$$\frac{\Gamma \vdash v \Rightarrow \oplus\{\ell \colon \tau_\ell^+\}_{\ell \in L} \quad \forall(\ell \in L) \colon \Gamma, x_\ell{:}\tau_\ell^+ \vdash e_\ell \Leftarrow \sigma^-}{\Gamma \vdash \mathsf{case}\ v\ (\ell \cdot x_\ell \Rightarrow e_\ell)_{\ell \in L} \Leftarrow \sigma^-}\ \oplus\mathrm{E} \qquad \frac{\Gamma, x{:}\tau^+ \vdash e \Leftarrow \sigma^-}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau^+ \to \sigma^-}\ {\to}\mathrm{I}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau^+ \to \sigma^- \quad \Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash e\, v \Rightarrow \sigma^-}\ {\to}\mathrm{E} \qquad \frac{\forall(\ell \in L) \colon \Gamma \vdash e_\ell \Leftarrow \sigma_\ell^-}{\Gamma \vdash \{\ell = e_\ell\}_{\ell \in L} \Leftarrow \&\{\ell \colon \sigma_\ell^-\}_{\ell \in L}}\ \&\mathrm{I}$$

$$\frac{\Gamma \vdash e \Rightarrow \&\{\ell \colon \sigma_\ell^-\}_{\ell \in L} \quad (j \in L)}{\Gamma \vdash e.j \Rightarrow \sigma_j^-}\ \&\mathrm{E}_k \qquad \frac{f : \sigma^- = e \in \Sigma}{\Gamma \vdash f \Rightarrow \sigma^-}\ \mathrm{NAME} \qquad \frac{\Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash \mathsf{return}\ v \Leftarrow {\uparrow}\tau^+}\ {\uparrow}\mathrm{I}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow {\uparrow}\tau^+ \quad \Gamma, x{:}\tau^+ \vdash e_2 \Leftarrow \sigma^-}{\Gamma \vdash \mathsf{let\ return}\ x = e_1\ \mathsf{in}\ e_2 \Leftarrow \sigma^-}\ {\uparrow}\mathrm{E} \qquad \frac{\Gamma \vdash v \Rightarrow \tau^+ \quad \tau^+ \leq \sigma^+}{\Gamma \vdash v \Leftarrow \sigma^+}\ \mathrm{SUB}^+$$

$$\frac{\Gamma \vdash e \Rightarrow \tau^- \quad \tau^- \leq \sigma^-}{\Gamma \vdash e \Leftarrow \sigma^-}\ \mathrm{SUB}^- \qquad \frac{\Gamma \vdash v \Leftarrow \tau^+}{\Gamma \vdash (v : \tau^+) \Rightarrow \tau^+}\ \mathrm{ANNO}^+ \qquad \frac{\Gamma \vdash e \Leftarrow \sigma^-}{\Gamma \vdash (e : \sigma^-) \Rightarrow \sigma^-}\ \mathrm{ANNO}^-$$

Figure 2.4: Bidirectional Typing

2.6.1 Isorecursive Types

Our system uses equirecursive types, which allow for many subtyping relations since there are no term constructors for folding recursive types [174]. Moreover, equirecursive types support the normal form in which constructors are always applied to type names (see Section 2.4.2), simplifying our algorithms, their description, and implementations. Going further, isorecursive types are often treated as an inconvenience, as the explicit folding and unfolding are often abstracted away into other constructs. For example, in functional languages such as Haskell or ML, a flavor of isorecursive types is provided (and hidden) through datatypes [177].

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 standard subtyping rules.

For every recursive type $\mu\alpha^+.\,\tau^+$, we introduce a definition $t^+ = \oplus\{\mathsf{fold}_\mu : [t/\alpha]\tau\}$. Similarly, for every corecursive type $\nu\alpha^-.\,\sigma^-$ we introduce a definition $s^- = \&\{\mathsf{fold}_\nu : [s/\alpha]\sigma\}$. 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 (see [98, Appendix K]) is enabled by our generalization of binary sums and lazy pairs to variant and lazy records, respectively, so we can use them in their unary form. With the addition of this translation, we can model recursive types with explicit constructors.

2.6.2 Call-by-value & Call-by-name Embeddings

Finally, we bring together existing theories and derive systems of subtyping for CBN and CBV from Levy's translations into our equirecursive CBPV variant. Most of the details, including statics and dynamics, circular derivation rules for subtyping, and the soundness and completeness proofs have been elided for brevity, but can be found in [98, Appendix L] (for CBN) and in [98, Appendix M] (for CBV).

Call-by-name. Now consider a CBN language with the following types.

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

Levy [105] presents translations, $(-)^{\boxminus}$, from CBN types and terms to CBPV *negative* types and *expressions*, respectively. An auxiliary translation, $\downarrow(-)^{\boxminus}$, on contexts is also used. Here is an *excerpt* of terms and type translations, leveraging shifts.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\multicolumn{3}{c}{\textit{Types}} & \multicolumn{3}{c}{\textit{Terms}} \\
(\tau \to \sigma)^{\boxminus} &=& {\downarrow}\tau^{\boxminus} \to \sigma^{\boxminus} & (x)^{\boxminus} &=& \mathsf{return}\ x \\
(\tau_1 \otimes \tau_2)^{\boxminus} &=& {\uparrow}({\downarrow}\tau_1^{\boxminus} \otimes {\downarrow}\tau_2^{\boxminus}) & (\lambda x.\, e)^{\boxminus} &=& \lambda x.\, e^{\boxminus} \\
(\mathbf{1})^{\boxminus} &=& {\uparrow}\mathbf{1} & (e_1\, e_2)^{\boxminus} &=& e_1^{\boxminus}\,(\mathsf{thunk}\ e_2^{\boxminus}) \\
(\oplus\{\ell \colon \tau_\ell\}_{\ell \in L})^{\boxminus} &=& {\uparrow}\oplus\{\ell \colon {\downarrow}\tau_\ell^{\boxminus}\}_{\ell \in L} & & & \\
(\&\{\ell \colon \sigma_\ell\}_{\ell \in L})^{\boxminus} &=& \&\{\ell \colon \sigma_\ell^{\boxminus}\}_{\ell \in L} & & &
\end{array}$$

We translate type names t to fresh type names $t^{\boxminus}$. This process involves translating the body of t's definition and inserting additional type names as needed to achieve a normal form that alternates between structural types and type names. Levy [105] has proven that well-typed terms remain well-typed after the translation to CBPV is applied. Since our syntactic typing rules are identical, the theorem carries over into this setting.

We adapt the subtyping system of Gay and Hole [76] to a λ-calculus from the π-calculus, which reverses the direction of subtyping from their classical system and adds empty records. These rules introduce a CBN syntactic subtyping judgment $t \leq u$. To distinguish it from CBPV syntactic subtyping, we use superscript minuses for CBPV type names, with CBN type names not marked. As in CBPV syntactic subtyping, the rules for CBN subtyping build a *circular derivation*. Like before, a circularity arises when a goal $t \leq u$ arises as a proper subgoal of itself.

As demonstrated in our published work [98], the CBN subtyping rules are precisely those for which $t \leq u$ in the CBN language IFF $t^{\boxminus} \leq u^{\boxminus}$ in the CBPV metalanguage. This observation has been proven to be sound and complete in relation to Gay and Hole's CBN subtyping.

In [97, Section 8.1], we discuss the minor differences between CBN subtyping rules and those of CBPV, highlighting an exception that is otherwise analogous to the rules of Gay and Hole [76].

Call-by-value. The procedure for Levy's CBV translation plays out similarly to that of the CBN one, with the same language of terms. An *excerpt* of terms as well as translations from CBV types to CBPV *positive types* and *expressions* is as follows.

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\textit{Types}} \\
(\tau \to \sigma)^{\boxplus} &=& {\downarrow}(\tau^{\boxplus} \to {\uparrow}\sigma^{\boxplus}) \\
(\tau_1 \otimes \tau_2)^{\boxplus} &=& \tau_1^{\boxplus} \otimes \tau_2^{\boxplus} \\
(\mathbf{1})^{\boxplus} &=& \mathbf{1} \\
(\oplus\{\ell\colon \tau_\ell\}_{\ell \in L})^{\boxplus} &=& \oplus\{\ell\colon \tau_\ell^{\boxplus}\}_{\ell \in L} \\
(\&\{\ell\colon \sigma_\ell\}_{\ell \in L})^{\boxplus} &=& {\downarrow}\&\{\ell\colon {\uparrow}\sigma_\ell^{\boxplus}\}_{\ell \in L}
\end{array}
\qquad
\begin{array}{rcl}
\multicolumn{3}{c}{\textit{Terms}} \\
(x)^{\boxplus} &=& \mathsf{return}\ x \\
(f)^{\boxplus} &=& \mathsf{force}\ f \ \mathsf{for}\ f : \tau = e \in \Sigma \\
(\lambda x.\, e)^{\boxplus} &=& \mathsf{return}\ (\mathsf{thunk}\ (\lambda x.\, e^{\boxplus})) \\
(e_1\, e_2)^{\boxplus} &=& \mathsf{let}\ \mathsf{return}\ x = e_2^{\boxplus}\ \mathsf{in} \\
&& \mathsf{let}\ \mathsf{return}\ f = e_1^{\boxplus}\ \mathsf{in} \\
&& \quad (\mathsf{force}\ f)\ x
\end{array}
$$

Similarly to the CBN translation, we also translate type names t to fresh type names $t^{\boxplus}$ and follow suit, as Levy [105]'s theorem also carries over.

We adapt the CBV subtyping system of Ligatti et al. [106] to our setting, which means that we include variants and lazy records with width and depth subtyping and replace isorecursive with equirecursive types. Taking care to distinguish the CBV syntactic subtyping judgment, $t \leq u$, from the CBPV syntactic subtyping, we mark CBPV type names with superscript pluses. The rules for CBV subtyping also build *circular derivations*.

In [97, Section 8.2], we discuss the differences between CBV subtyping rules and those of CBPV, particularly how our rules diverge from those of Ligatti et al. [106] in the handling of "$t \leq u$ if $u = u_1 \to u_2$ and u_1 empty" that generalizes to our subsumption rules involving $\top\textsc{sub}_{\mathrm{V}}^{\to\to}$, $\top\textsc{sub}_{\mathrm{V}}^{\&\to}$, and $\top\textsc{sub}_{\mathrm{V}}^{\to\&}$ (if they had incorporated lazy records).

One notable caveat in this interpretation is that Polarized Subtyping in Levy's CBV translation is incomplete due to the $\downarrow$ shift acting as a *barrier to fullness* [97, Section 8.2], as "$t \leq u$ if $u = {\downarrow}r$ and r full" would be unsound in polarized subtyping. This phenomenon is largely theoretical, affecting only non-applicable functions and empty records in specific comparisons with CBV types, which results from their system's non-standardization. A more nuanced translation could potentially resolve this incompleteness.

Chapter 3

Work in Progress

Now with Unions, Intersections, & Type Difference

“*Union, intersection and recursive types are type constructions which make it possible to formulate very precise types. However, because of type checking difficulties related to the simultaneous use of these constructions, they have only found little use in programming languages. One of the problems is subtyping*”

— *Flemming M. Damm*, Subtyping with union types, intersection types and recursive types [39]

In this chapter, we document a portion our in progress work (i.e., still finalizing results) and background on extending Polarized Subtyping with more advanced property types, specifically intersections, unions, and type difference. We also touch on the challenges involved with the interaction of these type features and effects, where we hope to provide an updated analysis on the value restriction and related phenomena.

3.1 Background: Property Types

3.1.1 Intersection types

Intersection types were introduced in the 1970s and have been extensively studied as a form of type polymorphism [37, 46, 130]. They permit the expression of all possible, explicitly chosen (interesting) types of a program, and can also identify expressions that fail to type-check if they use *different* types to satisfy varying branches in the intersection [37].

Intersections provide a means to attribute multiple types to an expression, either as a binary relationship: $(\tau^+ \wedge \sigma^+)$ or encompassing functions: $(\tau_1^+ \to \sigma_1^-) \wedge (\tau_2^+ \to \sigma_2^-)$. They are an example of a *implicit* typing feature that can describe program behavior only in the language of types, without manifesting itself within the syntactic terms of a program [51].

To illustrate a simple example with intersection types and subtyping in action, we consider the following well-typed program in Polite: incrementing natural numbers and binary numbers.

```
type nat = +{'zero : 1, 'succ : nat}
type bin = +{'e : 1, 'b0: bin, 'b1 : bin}
type std = +{'e : 1, 'b0: pos, 'b1: std}
type pos = +{         'b0: pos, 'b1: std}

(* a fn that increment nats and bins *)
comp inc : nat -> <nat> /\ bin -> <bin>
   = fun x0 => match x0 { 'e u => <'b1 'e u>
                        | 'b0 x1 => <'b1 x1>
                        | 'b1 x2 => let <x3> = inc x2 in <'b0 x3>
                        | 'zero _ => <'succ x0>
                        | 'succ _ => <'succ x0> }
(* evaluate nat(s) *)
eval one : <nat> = <'succ 'zero ()>
eval two : <nat> = let <x> = one in inc x
eval ten : <pos> = <'b1 'b0 'b1 'e ()>
(* evaluate inc with pos (pos is a subtype of nat) *)
eval eleven : <bin> = let <x> = ten in inc x
```

Here, an increment function accepts an intersection of $\mathsf{nat}^+ \to \mathsf{nat}^+ \wedge \mathsf{bin}^+ \to \mathsf{bin}^+$, meaning that it can accept a natural number or a binary number and return the same. The function itself is defined by pattern matching on the input and returning the appropriately typed output. The *eval* expressions demonstrate the use of the increment function with a natural number and a binary number, showing subtyping with the pos^+ binary type.

Although this example is straightforward, the use of intersection types can be quite powerful in more complex scenarios, particularly with width and depth subtyping at our disposal. For instance, consider a program that works with an intersection across different yet precise API request/response record types. Our system, utilizing width subtyping and intersections, can adapt seamlessly to the rich and diverse structures of these APIs without breaking existing callers—a plea made passionately in Hickey's [87] *Maybe Not* presentation on static type systems in functional programming. This approach ensures compatibility and flexibility, allowing robust and maintainable codebases.

Next, we briefly examine a more interesting example demonstrating subtyping with intersection types: the Fibonacci representation of natural numbers.

3.1.2 Example: Fibonacci Sums

Below is an implementation in Polite of the algorithm from Song's [145] *Linear Time Addition of Fibonacci Encodings* for normalizing Fibonacci sums. This algorithm is based on a binary encoding that preserves the property that there are no consecutive 1-bits in the representation. We implement binary form here with explicit labels for "0" and "1" record fields. The algorithm is quite non-trivial and takes advantage of intersections for computing the reverse of the sum at each pass: a downward pass (MSB to LSB) to reverse the result, and an upward pass (LSB to MSB). The normalization function normalize is a composition of the reverse and pass functions, which are defined recursively. The fib63 evaluated represents the 63rd Fibonacci number in the sequence, **11011101**, and the fib63normal evaluated represents the normalized form of that number: **000010001**. Note the essential use of intersection types. We could not express the specifically defined properties of the reverse functions without them.

```
type fib        = +{'0 : fib,       '1 : fib,        'e : 1}
type normal     = +{'0 : normal,    '1 : normal1,    'e : 1}
type normal1    = +{'0 : normal,                     'e : 1}
type buffered   = +{'0 : buffered, '1 : buffered1,  'e : 1}
type buffered1  = +{'0 : buffered, '1 : buffered11, 'e : 1}
type buffered11 = +{'0 : buffered1,                  'e : 1}

comp pass1 : fib -> <buffered>
   = fun x0 => pass1_0 x0

comp pass1_0 : fib -> <buffered1>
   = fun x0 => match x0 { '0 x1 => let <x2> = pass1_0 x1 in
                                  <'0 x2>
                        | '1 x3 => let <x4> = pass1_01 x3 in
                                  <x4>
                        | 'e x5 => <'0 'e ()> }

comp pass1_01 : fib -> <buffered1>
   = fun x0 => match x0 { '0 x1 => let <x2> = pass1_0 x1 in
                                  <'0 '1 x2>
                        | '1 x3 => let <x4> = pass1_0 x3 in
                                  <'1 '0 x4>
                        | 'e x5 => <'0 '1 'e ()> }

comp pass2 : buffered -> <normal>
   = fun x0 => match x0 { '0 x1 => let <x2> = pass2 x1 in
                                  <'0 x2>
                        | '1 x3 => let <x4> = pass2_1 x3 in
                                  <x4>
                        | 'e x5 => <'e ()> }
```

```
comp pass2_1 : buffered1 -> <normal>
   = fun x0 => match x0 { '0 x1 => let <x2> = pass2 x1 in
                                   <'1 '0 x2>
                        | '1 x3 => let <x4> = pass2_11 x3 in
                                   <x4>
                        | 'e x5 => <'1 'e ()> }

comp pass2_11 : buffered11 -> <normal>
   = fun x0 => match x0 { '0 x1 => let <x2> = pass2_1 x1 in
                                   <'0 '0 x2>
                                   | 'e x3 => <'0 '0 '1 'e ()> }

comp rev : fib -> fib -> <fib> /\ buffered -> buffered11 ->
           <buffered> /\ buffered1 -> buffered1 ->
           <buffered> /\ buffered11 -> buffered -> <buffered>
   = fun x0 => fun x1 => match x0 { '0 x2 => rev x2 ('0 x1)
                                  | '1 x3 => rev x3 ('1 x1)
                                  | 'e x4 => <x1> }

comp reverse : fib -> <fib> /\ buffered -> <buffered>
   = fun x0 => rev x0 ('e ())
comp normalize : fib -> <normal>
   = fun x0 => let <x1> = reverse x0 in
               let <x2> = pass1 x1 in
               let <x3> = reverse x2 in
               let <x4> = pass2 x3 in
               <x4>

(* 63 = 11011101 = 1+2+5+8+13+34 (non-canonical) *)
comp fib63 : <fib>
   = <'1 '1 '0 '1 '1 '1 '0 '1 'e ()>

(* fib63normal = 000010001 = 8 + 55 (canonical) *)
eval fib63normal : <normal>
   = let <f63> = fib63 in normalize f63
```

The only property that we cannot check is whether the result always has the same value as the input, but that would require arithmetically indexed types [42, 43], which is beyond the scope of this thesis and involves incorporating additional layers into the type system.

3.1.3 Union Types

Dual to intersection types, union types [107] are another implicit typing feature—independent of any particular expression construct. While the use of intersection types for practical programming has lineage traced to Reynold's Forsythe language [138], unions followed as a dual counterpart in Pierce's dissertation on subtyping [127] and Barbanera, et al.'s type assignment system combining the two [12]. Union types allow for a partitioning of larger types, which is convenient in many contexts where granularity is needed. As an example, in combination with our variant records, the type $\mathsf{bool}^+ = \oplus\{\mathbf{true} : \mathbf{1}, \mathbf{false} : \mathbf{1}\}$ can be partitioned into two unary records: $\mathsf{true}^+ = \oplus\{\mathbf{true} : \mathbf{1}\}$ and $\mathsf{false}^+ = \oplus\{\mathbf{false} : \mathbf{1}\}$.

A straightforward but more interesting example involves a type signature for a function with a codomain (output) that *tiles* the space of natural numbers when combined, but is refuted if one disjunct is not present in it.

```
type nat  = +{'zero : 1, 'succ : nat}
type even = +{'zero : 1, 'succ : odd}
type odd  = +{           'succ : even}

type nat_cover = (even * odd)  \/
                 (even * even) \/
                 (odd  * odd)  \/
                 (odd  * even)
comp ex1 : (nat * nat) -> <nat_cover>
   = fun x => <x>
eval z: <nat> = <'zero ()>
eval ex2 : <nat_cover> = let <x> = z in ex1 (x, x)

type nat_cover_no_oo = (even * odd)  \/
                       (even * even) \/
                       (odd  * even)
fail comp ex3 : (nat * nat) -> <nat_cover_no_oo>
   = fun x => <x>
```

Here, the subtyping relation $\mathsf{nat}^+ \otimes \mathsf{nat}^+ \leq (\mathsf{even} \otimes \mathsf{odd}) \vee (\mathsf{even} \otimes \mathsf{even}) \vee (\mathsf{odd} \otimes \mathsf{odd}) \vee (\mathsf{odd} \otimes \mathsf{even})$ holds (with nat_cover), as it spans the entire nat coverage set of possibilities. However, the subtyping relation $\mathsf{nat}^+ \otimes \mathsf{nat}^+ \not\leq (\mathsf{even} \otimes \mathsf{odd}) \vee (\mathsf{even} \otimes \mathsf{even}) \vee (\mathsf{odd} \otimes \mathsf{even})$ does not hold (with nat_cover_no_oo) when $\mathsf{odd} \otimes \mathsf{odd}$ is removed, which we capture with **fail** in the ex3 expression (**fail** is a keyword for testing expressions where typechecking should fail to hold in Polite).

We are still researching interesting examples that necessitate disjunction. One that comes to mind involves statically checking whether the color invariant of a red-black tree [44, 54, 93] is violated at either the root (the root is red and some child is red), at the left child (the left child is red and one of its children is red), or at the right child.

Another example demonstrating the need for union types relates to nondeterministic tree automata [36], where union types can help define the subtyping relation as the inclusion of recognized languages [165].

Unions are considered an indefinite property type [55, 62], as they can encompass either property τ^- or property σ^- (in the negative case) of a type definition such as $\tau^- \vee \sigma^-$, which, *without restriction*, can lead to nondeterministic semantics where any one of the branches that match can be taken [56, 134]. This is not the case for positive, observable unions. Unions are tricky. Determining the scope of the elimination rules for indefinite types is non-trivial [169]. We discuss these issues in more detail in Section 3.4.

3.1.4 Blending Intersection & Union Types

The interaction between intersection and union types opens up a rich space of possibilities for type systems. However, because of typechecking difficulties related to the simultaneous use of these constructions, a sound foundation has been elusive in popular programming languages [39]. Downen et al. [51] warns:

> From a language designer's perspective, intersection and union types have different metatheoretic properties, and naively extending a language with intersection or union types can easily lead to some undesirable outcomes.

The background and related discussion on some of these undesirable outcomes are covered in Section 5.1. Much of our work in this area is focused on understanding the metatheoretic properties of these types through a polarized lens and their interaction in the presence of effects. Oriented toward the latter goal, we aim to develop a more principled and practical approach to subtyping with the blending of these property types that operates along the lines of pure and/or effectful fragments of the subtyping relation, while identifying a better explanation for their (co)value restrictions (see 3.4.1).

3.1.5 Type Difference

Type difference, or *type subtraction*, is a less common feature in type systems, but has roots in record extensions as restriction and removal operations [25] and is usable for conditional expressions [168]. This notion is related to set difference [28, 70], where the difference of two types $\tau^+ \backslash \sigma^+$ is the type of all values that are in τ^+ but not in σ^+. For example, given the type $\mathsf{nat}^+ \backslash \mathsf{even}^+$ as a function input, we would expect to receive only odd numbers, deriving a signature like $(\mathsf{nat}^+ \backslash \mathsf{even}^+) \to \mathsf{odd}^+$. Note that in Polite, we replace "\" with "–" to avoid conflicts with parsing the syntax.

Next, we present some basic examples of type difference in Polite, testing whether the computation should pass or fail based on the type signature. Most of these are self explanatory, but demonstrate the utility of type difference as a feature in a type system, especially in working with variant record types.

```
type nat  = +{'zero : 1, 'succ : nat}
type even = +{'zero : 1, 'succ : odd}
type odd  = +{            'succ : even}

comp test1 : (nat - odd) -> <even>
   = fun x => <x>
comp test2 : (nat - even) -> <odd>
   = fun x => <x>
comp test3 : even -> <nat>
   = fun x => <x>
(* cannot be even and odd at the same time => bottom *)
comp test4 : (even /\ odd) -> <Bot>
   = fun x => <x>
comp test5 : even -> <nat - odd>
   = fun x => <x>
comp test6 : odd -> <nat - even>
   = fun x => <x>
```

3.2 (Sub)typing

As we did back in Section 2.4, we would like to provide a soundness proof for subtyping now with the addition of these new property types—without the *burden* of effects (at the moment). This undertaking has involved much back and forth due to how these types behave based on polarization. To get started, we will jump right into our types and terms presentation with signatures in normal form, alternating between structural types and type names. Because these property types have no syntactic representation in the language or dynamics, the following summary of our calculus will focus only on the type language.

$$
\begin{aligned}
\tau^+ &::= \ldots \mid t \wedge u \mid t \vee u \mid t \backslash u \mid \top \mid \bot \\
\sigma^- &::= \ldots \mid s \wedge r \mid s \vee r \\
\Gamma &::= \cdot \mid \Gamma, t = \tau^+ \mid \Gamma, s = \sigma^- \\
\Sigma &::= \cdot \mid \Sigma, t = \tau^+ \\
\Delta &::= \cdot \mid \Delta, t = \tau^+
\end{aligned}
$$

Beyond polarity and types, we introduce additional contexts to our language to account for subtyping relations that consist of a multiplicity of type constraints, each representing multiple properties. This shift to subtyping contexts is beneficial because intersections can encapsulate many nested typings. In this setting, the subtyping contexts Γ and Σ are defined as follows:

- Γ is a subtyping context that collects both positive and negative types.
- Σ is a subtyping context that collects *only* positive types.

This characterization of *subtyping contexts* follows from varying lines of work, including Seo and Park [142]. Furthermore, the Δ context is introduced as a mechanism for incrementally building a covering set—i.e., a set of exhaustive patterns—for a context inhabited by product types (see Theorem 2).

After much careful consideration, the current approach views intersections and unions as crossing the aisle, acting as both positive and negative types, while type difference is strictly positive and cannot be constructed in the negative layer. We also do not have to specialize rules for emptiness (and fullness) as we did in Section 2.4.2, as we no longer operate at the level of consequents.

Our updated semantic definition can be found in Figure 3.1, where we have extended it to account for the new property types and contexts. Two key points to note are that step indices for intersections, unions, and type difference (across layers) are not affected, as there are no dynamics associated with them. Secondly, closed values can now be semantically characterized as $v \in_k [\![\bot]\!]$, $v \in_k [\![\top]\!]$, and $v \in_k [\![\Gamma]\!]^\cap$, and $v \in_k [\![\Sigma]\!]^\cup$, where the semantic meaning of $[\![\Gamma]\!]^\cap$ captures the (set) intersection of positive types in Γ and $[\![\Sigma]\!]^\cup$ the (set) union of positive types in Σ. Computations, e, are only semantically characterized contextually with $[\![\Gamma]\!]^\cap$ (for negative types), because they are infinitary and not directly observable.

3.2.1 (In)compatibility

Our system provides a type compatibility check to ensure that a subtyping relation is expressible only if the types τ_1^+, τ_2^+ in $\tau_1^+ \leq \tau_2^+$ have the same top-level constructor (likewise for σ_1^-, σ_2^- in $\sigma_1^- \leq \sigma_2^-$), assuming the types are structural. This check prevents the formation of incompatible, disjoint intersection types, such as $\mathsf{nat}^+ \wedge \mathsf{bool}^+$ for example.

Those with a love for all things intersections would find this compatibility check to be a bit of a downer, as it prevents the formation of the type above, which would have been valid by way of the *merge operator* [57, 89, 136, 138, 139]. It is a property that has been restricted in recent times to impose coherence [89] (or non-ambiguity) in the presence of subtyping. With the addition of unions and type difference, the value of and ambiguity associated with the merge operator is lessened. It is unclear if expressing such disjoint relations is useful in practice.

Given incompatibility, for positive types, for example, we can define a proposition that states that if Γ is *incompat*, then the semantic definition of $[\![\Gamma]\!] = \emptyset$ for positive Γ.

3.2.2 Syntactic Subtyping

When moving from how "we turned the (proof) crank" in Chapter 2 to this in-progress formulation, our direct application of circular proofs did not scale well to the inclusion of property types [121], even when we leave aside the tracking of unbounded nested observations. Adapting the system beyond the consequent judgments in multiple contexts proved problematic and unwieldy (see Figure 3.2 for the well-formed *simple* subtyping judgment of $\mathsf{nat}^+ \leq \mathsf{even}^+ \vee \mathsf{odd}^+$). While our soundness theorem essentially stays the same, we had to rework our subtyping rules to account

$$
\begin{aligned}
&\vdots\\
v \in_k [\![t^+ \wedge u^+]\!] &\triangleq v \in_k [\![t^+]\!] \text{ and } v \in_k [\![u^+]\!]\\
v \in_k [\![t^+ \vee u^+]\!] &\triangleq v \in_k [\![t^+]\!] \text{ or } v \in_k [\![u^+]\!]\\
v \in_k [\![t^+ \backslash u^+]\!] &\triangleq v \in_k [\![t^+]\!] \text{ and } v \notin_k [\![u^+]\!]\\
v \in_k [\![\bot]\!] &\triangleq \mathit{never}\\
v \in_k [\![\top]\!] &\triangleq \mathit{always}\\
v \in_k [\![\Gamma]\!]^\cap &\triangleq (\forall t^+ \in \Gamma).v \in_k [\![t^+]\!]\\
v \in_k [\![\Sigma]\!]^\cup &\triangleq (\exists u^+ \in \Sigma).v \in_k [\![u^+]\!]\\
&\vdots\\
e \in_0 [\![\Gamma]\!]^\cap & \quad \text{always}\\
e \in_{k+1} [\![\Gamma]\!]^\cap &\triangleq (e \mapsto e' \text{ and } e' \in_k [\![\Gamma]\!]) \text{ or } (e \text{ terminal and } e \mathrel{\hat{\in}}_{k+1} [\![\Gamma]\!])\\
e \mathrel{\hat{\in}}_{k+1} [\![s^- \wedge r^-]\!] &\triangleq e \mathrel{\hat{\in}}_{k+1} [\![s^-]\!] \textbf{ and } e \mathrel{\hat{\in}}_{k+1} [\![r^-]\!]\\
e \mathrel{\hat{\in}}_{k+1} [\![s^- \vee r^-]\!] &\triangleq e \mathrel{\hat{\in}}_{k+1} [\![s^-]\!] \textbf{ or } e \mathrel{\hat{\in}}_{k+1} [\![r^-]\!]\\
e \mathrel{\hat{\in}}_{k+1} [\![\Gamma]\!]^\cap &\triangleq (\forall s^- \in \Gamma).e \mathrel{\hat{\in}}_{k+1} [\![s^-]\!]\\
&\vdots
\end{aligned}
$$

Figure 3.1: (Extended) Definition of Semantic Typing

for the new features and proceed with the proof without a formal metalogic. Instead, we opt for an ordinary mathematical inductive argument.

To account for multiple types when typing an expression, we have transformed our subtyping relations into two distinct fragments: (a) a purely positive fragment, $\Gamma^+ \leq \Sigma^+$, with contexts on both the left and right sides of the judgment, and (b) a fragment, $\Gamma^- \leq s^-$, featuring a unary singleton succedent on the right side of the judgment for negative types. The consequent is necessary in the latter case because a negative, infinitary type can never be empty (i.e., inhabit an empty context in this formulation) and must be inhabited by a single type to ensure determinism.

Now, we present our in-progress theorem for the soundness of subtyping in this extended system (without effects). The gist of the proof proceeds over cases (see Figures 3.4 for the subtyping rules, which rely on Figure 3.3 for operations on contexts), by nested induction, first on the step-index k and second on the structure of v / e in Γ.

Theorem 2 (Soundness of Subtyping, Pure/Without Effects)**.**

1. If $\Gamma \leq \Sigma$ *then* $\forall k, v.v \in_k [\![\Gamma]\!]^\cap \supset v \in_k [\![\Sigma]\!]^\cup$

2. If $\Gamma \leq s^-$*, then* $\forall k, e.e \mathrel{\hat{\in}}_{k+1} [\![\Gamma]\!]^\cap \supset e \mathrel{\hat{\in}}_{k+1} [\![s^-]\!]$

In highlighting certain subtyping rules, the specialty cases of $\otimes\textsc{sub}_\varnothing$ and $\oplus\textsc{sub}_\varnothing$ account for empty contexts that could arise when the projection on the coverage set of pairs, $\Gamma.\pi_i$, or on the

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1} \vdash \mathbf{1}}\,id \qquad \overset{\text{CYCLE}(*)}{\mathsf{nat} \vdash \mathsf{odd}, \mathsf{even}}}{\mathsf{nat} \vdash \mathsf{even}, \mathsf{odd}\ (*)}\,\oplus_{\mathrm{L}}, \in_{\mathrm{L}}, =_{\mathrm{L}}, \in_{\mathrm{R}}}{\mathsf{nat} \vdash \mathsf{even} \vee \mathsf{odd}}\,\vee_{\mathrm{R}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v' \in_k \mathbf{1} \vdash v' \in_k \mathbf{1}, \mathbf{zero} \cdot v' \in_k \mathsf{odd}}\,id}{v' \in_k \mathbf{1} \vdash \mathbf{zero} \cdot v' \in_k \mathsf{even}, \mathbf{zero} \cdot v' \in_k \mathsf{odd}}\,\in_{\mathrm{R}}}{v = \mathbf{zero} \cdot v', v' \in_k \mathbf{1} \vdash v \in_k \mathsf{even}, v \in_k \mathsf{odd}}\,=_{\mathrm{L}} \qquad \dfrac{\dfrac{\overset{\text{CYCLE}(*)([v'/v])}{v' \in_k \mathsf{nat} \vdash v' \in_k \mathsf{odd}, v' \in_k \mathsf{even}}}{v' \in_k \mathsf{nat} \vdash \mathbf{succ} \cdot v' \in_k \mathsf{even}, \mathbf{succ} \cdot v' \in_k \mathsf{odd}}\,\in_{\mathrm{R}}}{v = \mathbf{succ} \cdot v', v' \in_k \mathsf{nat} \vdash v \in_k \mathsf{even}, v \in_k \mathsf{odd}}\,=_{\mathrm{L}}}{v \in_k \mathsf{nat} \vdash v \in_k \mathsf{even}, v \in_k \mathsf{odd}\ (*)}\,\oplus_{\mathrm{L}}, \in_{\mathrm{L}}}{v \in_k \mathsf{nat} \vdash v \in_k \mathsf{even} \vee v \in_k \mathsf{odd}}\,\vee_{\mathrm{R}}$$

Figure 3.2: Circular Derivations for $\mathsf{nat}^+ \leq \mathsf{even}^+ \vee \mathsf{odd}^+$

intersection on labels across the set of records (computations defined on contexts in Figure 3.3) returns empty. To prove the soundness for $\oplus$SUB, we define a function split on Σ, which relies on a lemma stating that if $v_1 \in_k [\![\Delta_1]\!]^{\cup}$ or $v_2 \in_k [\![\Delta_2]\!]^{\cup}$ for all $(\Delta_1, \Delta_2) \in \text{split}\ \Sigma$ then $\langle v_1, v_2 \rangle \in [\![\Sigma]\!]^{\cup}$ semantically.

For positive and negative variants of $\wedge$SUB$_L$ and $\vee$SUB$_R$, the *comma* denotes the concatenation of contexts, where with these type judgments, we can destruct in sequence and get both components. Type difference for positive types is covered by $\setminus$SUB$_L^+$ and $\setminus$SUB$_R^+$, where the types are split across contexts depending on the sequent relation.

3.3 Type-level Record Concatenation

We are currently exploring the implications of record concatenation in our type system. Unlike other work that combines record operations and the *merge operator* [85, 176], we are interested in the type-level operations for concatenation, applied to recursive types, that are correct-by-construction. This feature would give our system a way to incorporate a lightweight form of subtyping through inheritance [27]. For instance, consider how we might emulate a simplified queue class or interface that is inherited by another class which extends it with a new method. This is demonstrated in the following OO interface (as typed output), which is a slightly modified version of OCaml's object system:

```
class queue :
  nat list -> object
    val mutable q  : nat list
    method deq     : (nat * queue) option
    method enq     : nat -> queue
  end
class queue_rev :
  nat list -> object
    inherit queue
    ...
    method rev      : queue -> queue
end
```

Note that the base queue class is extended with a reverse operation (a common operation on queues), defined using inheritance directly. Additionally, in a OO system like this, the types do not leverage the possibility of mixed (eager/lazy) evaluation. In our system, implemented in Polite, at the type-level, we can define a new type, queue2 that is the concatenation of the two lazy record types, queue^- and $\mathsf{queue\_rev}^-$, as follows.

```
type queue = &{'enq : nat -> queue,
               'deq : <+{'none : 1, 'some : nat * [queue]}>}
type queue_rev = &{'rev : [queue_rev] -> queue_rev}
type queue2 = queue ++ queue_rev
```

The concatenation operator "++" generates a fresh type for all recursive labels, correct-by-construction, that will point to the type name $\mathsf{queue2}^-$, which expands into the following (at compile time):

$$\begin{aligned}\mathsf{queue2}^- = \ \&\{&\mathbf{enq} : \mathsf{nat} \to \mathsf{queue2},\\ &\mathbf{deq} : \uparrow\oplus\{\mathbf{none} : \mathbf{1}, \mathbf{some} : \mathsf{nat} \otimes \downarrow\mathsf{queue2}\},\\ &\mathbf{rev} : \downarrow\mathsf{queue2} \to \mathsf{queue2}\}\end{aligned}$$

Remember, square brackets ([,]) in the type signature maps to $\downarrow\sigma^-$ (a positive *thunk*) and angle brackets ($<$, $>$) to $\uparrow\tau^+$ (a negative *return*).

Next Steps We plan to finish the implementation of this type-level feature within Polite and finalize the formalism for the concatenation operator.

3.4 In the Presence of Effects

Effects **change** and rule everything around us. Call-by-push-value was designed as a study for the λ-calculus with effects [103, Sec. 2.4], stratifying terms into values (which have no side-effects) and computations (which might). Our contribution to semantic typing [7, 8, 10, 52] is grounded in working with state, exceptions, and unsafe boundaries. Before we discuss rules that may not be sound in the presence of effects, we extend our system with a sound set of bidirectional typing rules in Figure 3.5.

3.4.1 The Value Restriction and Related Phenomena

Figure 3.5 presents the sound extensions (from 2.4) to our polarized-type system. Davies and Pfenning [45] developed the *value restriction* on the introduction rule for intersection types because it is unsound in the presence of mutable references with CBV semantics. They also omit the distributivity rule, $(A \to B) \cap (A \to C) \leq A \to (B \cap C)$, in order to prove progress and preservation. In our calculus, the bidirectional typing variant of $\wedge\text{I}^-$ for the non-pure, CBN layer comes into question. Is this restriction necessary or does it "fall out for free" from polarized subtyping, which already *does not restrict* $\wedge\text{I}^+$ in the positive setting (related to CBV)? Here is the rule in question.

$$\frac{\Gamma \vdash e \Leftarrow \sigma_1^- \quad \Gamma \vdash e \Leftarrow \sigma_2^-}{\Gamma \vdash e \Leftarrow \sigma_1^- \wedge \sigma_2^-}\ \wedge\text{I}^-?$$

We have ported Davies and Pfenning's [45] intersecting mutable references counterexample into our system with polarization and shifts and found that it *also* breaks down accordingly. We show a naive implementation in an ML variant here [49] to relay it back to the literature.

```
(* Counterexample by Rowan Davies and Frank Pfenning *)
(* Suppose we have types nat and pos, where nat is non-negative
   integers and pos is its subtype of positive integers. *)

let x : nat ref /\ pos ref = ref 1
in let () = (x := 0)      (* typechecks: x is a nat ref *)
in let z : pos = !x       (* deref op, typechecks: x is a pos ref *)
in z : pos                (* now we have the positive number zero *)
```

This example is unsound because the variable z cannot be positive if bound to zero, occurring due to the allowance of intersection introduction in the first line of code in the presence of mutable references (and the ability to dereference)! In our system (not ML), is this due to the unsound law of distributivity or the effect itself, generated by the introduction of mutable intersecting references, forcing the value restriction on the rule? Answering this is our imminent follow-up work.

On the other side of the coin, union types also require a (co)value restriction when confronted with effects. This constraint leads to an evaluation context restriction on union elimination in the presence of effects [61] for CBN semantics due to nondeterminism. This restriction also implies that the standard law $(A \to C) \cap (B \to C) \leq (A \cup B) \to C$ is unsound [173].

We now show the questionable bidirectional typing rule(s) here (given evaluation context E). The following union elimination rule was found unsound in the presence of effects since the different occurrences of e could evaluate to a value of type t_1^- or t_2^- nondeterministically.

$$\frac{\Gamma \vdash e \Rightarrow t_1^- \vee t_2^- \quad \Gamma, x : t_1^- \vdash e' \Leftarrow \sigma^- \quad \Gamma, x : t_2^- \vdash e' \Leftarrow \sigma^-}{\Gamma \vdash e'[e/x] \Rightarrow \sigma^-}\ \vee\text{E}^-?$$

This next version of the elimination rule adds the evaluation context restriction, E, discovered by "disillusionment" [173].

$$\frac{\Gamma \vdash e \Rightarrow t_1^- \vee t_2^- \quad \Gamma, x : t_1^- \vdash E[x] \Leftarrow \sigma^- \quad \Gamma, x : t_2^- \vdash E[x] \Leftarrow \sigma^-}{\Gamma \vdash E[e] \Rightarrow \sigma^-} \vee\mathrm{E}^-?$$

Next Steps Given unsound properties in the presence of effects, we have hypothesized that our system returns to the consequent-restricted representation of Chapter 2—moving back to single-succedent judgments rather than contexts Γ, Σ made up of multiple types or succedents—for negative subtyping relations, i.e., $s^- \leq r^-$. This seems to map neatly to the linear-logic-centered, monadic intersection work of Gavazzo et al. [75], as we have to make a singular choice on branches of multiple types. Closing this particular circle is on our immediate agenda.

$$\begin{cases} (\Gamma, t^+).\mathbf{1} & = \Gamma.\mathbf{1}, t^+ \text{ where } t^+ = \mathbf{1} \\ (\cdot).\mathbf{1} & = (\cdot) \end{cases} \qquad \text{(Projection on unit types)}$$

$$\begin{cases} (\Gamma, t_1^+ \otimes t_2^+).\pi_1 & = \Gamma.\pi_1, t_1^+ \\ (\Gamma, t_1^+ \otimes t_2^+).\pi_2 & = \Gamma.\pi_2, t_2^+ \\ (\cdot).\pi_1 & = (\cdot) \\ (\cdot).\pi_2 & = (\cdot) \end{cases} \qquad \text{(Projection on products)}$$

$$\begin{cases} \text{split}((\cdot)) & = \{\langle (\cdot), (\cdot) \rangle\} \\ \text{split}(\Sigma, t_1^+ \otimes t_2^+) & = \{\langle (\Delta_1, t_1^+), \Delta_2 \rangle, \langle \Delta_1, (\Delta_2, t_2^+) \\ & \qquad \mid \langle \Delta_1, \Delta_2 \in \text{split } \Sigma \rangle \rangle\} \end{cases} \qquad \text{(Recursive split on records)}$$

$$\begin{cases} \bigcap(\Gamma, \oplus\{\ell\colon t_\ell^+\}_{\ell \in L}) & = \bigcap \Gamma \cap L \\ \bigcap(\oplus\{\ell\colon t_\ell^+\}_{\ell \in L}) & = L \\ \bigcap(\cdot) & = \top \end{cases} \qquad \text{(Intersection on labels for records)}$$

$$\begin{cases} (\Gamma, t_j).j & = \Gamma.j, t_j \\ (\cdot).j & = (\cdot) \end{cases} \qquad \text{(Projection on records)}$$

$$\begin{cases} (\Gamma, \downarrow s^-).\downarrow & = \Gamma.\downarrow, \downarrow s^- \\ (\cdot).\downarrow & = (\cdot) \end{cases} \qquad \text{(Projection on $\downarrow$-shifts)}$$

$$\begin{cases} (\Gamma, t^+ \to s^-).u^+ & = \Gamma.u^+, s^- \text{ if } u^+ \leq t^+ \\ & = \Gamma.u^+ \text{ if } u^+ \nleq t^+ \\ (\cdot).u^+ & = (\cdot) \end{cases} \qquad \text{(Cond. projection on fns)}$$

$$\begin{cases} (\Gamma, \&\{\ell\colon s_\ell^-\}_{\ell \in L}).j \ (\forall.j \in J) & = \Gamma.j, s_j \text{ if } \ell \in J \\ & = \Gamma.j \ \text{ if } \ell \notin J \\ (\cdot).j & = (\cdot) \end{cases} \qquad \text{(Cond. projection on lazy records)}$$

$$\begin{cases} (\Gamma, \uparrow t^+).\uparrow & = \Gamma.\uparrow, \uparrow t^+ \\ (\cdot).\uparrow & = (\cdot) \end{cases} \qquad \text{(Projection on $\uparrow$-shifts)}$$

Figure 3.3: Computations and Filters on Typing Contexts

$$\frac{t_1^+ \otimes t_2^+ \in \Gamma \quad \Gamma.\pi_i \leq \cdot \quad \forall i \in \{1, 2\}}{\Gamma \leq \cdot} \otimes\textsc{sub}_\varnothing$$

$$\frac{t_1^+ \otimes t_2^+ \in \Gamma \quad \Gamma.\pi_1 \leq \Delta_1 \vee \Gamma.\pi_2 \leq \Delta_2 \quad \forall \langle \Delta_1, \Delta_2 \rangle \in \text{split } \Sigma}{\Gamma \leq \Sigma} \otimes\textsc{sub}$$

$$\frac{t^+ \in \Gamma \quad t^+ = \mathbf{1} \quad u^+ \in \Sigma.\mathbf{1} \quad u^+ = \mathbf{1}}{\Gamma \leq \Sigma} \mathbf{1}\textsc{sub}$$

$$\frac{\oplus\{\ell\colon u_\ell^+\}_{\ell \in L} \in \Gamma \quad \forall j \in (\bigcap \Gamma \cap L) \quad \Gamma.j \leq \cdot}{\Gamma \leq \cdot} \oplus\textsc{sub}_\varnothing$$

$$\frac{\oplus\{\ell\colon u_\ell^+\}_{\ell \in L} \in \Gamma \quad \forall j \in (\bigcap \Gamma \cap L) \subset I \quad \Gamma.j \leq \Sigma.j}{\Gamma \leq \Sigma_I} \oplus\textsc{sub}$$

$$\frac{{\downarrow}s^- \in \Gamma \quad {\downarrow}r^- \in \Sigma.{\downarrow} \quad \Gamma.{\downarrow} \leq r^-}{\Gamma \leq \Sigma} {\downarrow}\textsc{sub} \qquad \frac{\Gamma, t^+, u^+ \leq \Sigma}{\Gamma, t^+ \wedge u^+ \leq \Sigma} \wedge\textsc{sub}_L^+$$

$$\frac{\Gamma \leq \Sigma, t^+ \quad \Gamma \leq \Sigma, u^+}{\Gamma \leq \Sigma, t^+ \wedge u^+} \wedge\textsc{sub}_R^+ \qquad \frac{\Gamma, t^+ \leq \Sigma \quad \Gamma, u^+ \leq \Sigma}{\Gamma, t^+ \vee u^+ \leq \Sigma} \vee\textsc{sub}_L^+$$

$$\frac{\Gamma \leq \Sigma, t^+, u^+}{\Gamma \leq \Sigma, t^+ \vee u^+} \vee\textsc{sub}_R^+ \qquad \frac{\Gamma, t^+ \leq \Sigma, u^+}{\Gamma, t^+ \backslash u^+ \leq \Sigma} \backslash\textsc{sub}_L^+ \qquad \frac{\Gamma \leq \Sigma, t^+ \quad \Gamma, u^+ \leq \Sigma}{\Gamma \leq \Sigma, t^+ \backslash u^+} \backslash\textsc{sub}_R^+$$

$$\frac{t^+ \to s^- \in \Gamma \quad \Gamma.u^+ \leq r^-}{\Gamma \leq u^+ \to r^-} {\to}\textsc{sub} \qquad \frac{\&\{\ell\colon s_\ell^-\}_{\ell \in L} \quad \forall j \in I \quad \Gamma.j \leq s_j}{\Gamma \leq \&\{i\colon s_i^-\}_{i \in I}} \&\textsc{sub}$$

$$\frac{{\uparrow}t^+ \in \Gamma \quad \Gamma.{\uparrow} \leq u^+}{\Gamma \leq {\uparrow}u^+} {\uparrow}\textsc{sub} \qquad \frac{\Gamma, s_1^-, s_2^- \leq r^-}{\Gamma, s_1^- \wedge s_2^- \leq r^-} \wedge\textsc{sub}_L^- \qquad \frac{\Gamma \leq s^- \quad \Gamma \leq r^-}{\Gamma \leq s^- \wedge r^-} \wedge\textsc{sub}_R^-$$

$$\frac{\Gamma, s_1^- \leq r^- \quad \Gamma, s_2^- \leq r^-}{\Gamma, s_1^- \vee s_2^- \leq r^-} \vee\textsc{sub}_L^- \qquad \frac{\Gamma \leq s^-, r^-}{\Gamma \leq s^- \vee r^-} \vee\textsc{sub}_R^-$$

Figure 3.4: (Extended) Rules for Subtyping with Positive and Negative Intersections, Unions, and Postive Type Difference. See Figure 3.3 for compuations and filters on contexts.

$$\frac{\Gamma \vdash v \Leftarrow \tau_1^+ \quad \Gamma \vdash v \Leftarrow \tau_2^+}{\Gamma \vdash v \Leftarrow \tau_1^+ \wedge \tau_2^+} \wedge\mathrm{I}^+ \qquad \frac{\Gamma \vdash v \Rightarrow \tau_1^+ \wedge \tau_2^+}{\Gamma \vdash v \Rightarrow \tau_i^+} \wedge\mathrm{E}_i^+ \qquad \frac{\Gamma \vdash e \Rightarrow \sigma_1^- \wedge \sigma_2^-}{\Gamma \vdash e \Rightarrow \sigma_i^-} \wedge\mathrm{E}_i^-$$

$$\frac{\Gamma \vdash v \Leftarrow \tau_i^+}{\Gamma \vdash v \Leftarrow \tau_1^+ \vee \tau_2^+} \vee\mathrm{I}_i^+ \qquad \frac{\Gamma \vdash v \Rightarrow \tau_1^+ \vee \tau_2^+ \quad \Gamma, x : \tau_1^+ \vdash v' \Leftarrow \tau^+ \quad \Gamma, x : \tau_2 \vdash v' \Leftarrow \tau^+}{\Gamma \vdash [v/x]v' \Rightarrow \tau^+} \vee\mathrm{E}^+$$

$$\frac{\Gamma \vdash e \Leftarrow \sigma_i^-}{\Gamma \vdash e \Leftarrow \sigma_1^- \vee \sigma_2^-} \vee\mathrm{I}_i^-$$

Figure 3.5: (Extended) Sound Bidirectional Typing

Chapter 4

Proposed Work

Towards an Expressively Complete Type System

“*In essence, the semantics of any expression maps related environments into related values*”

—*John C. Reynolds*, Types, Abstraction, and Parametric Polymorphism [135]

Thus far, we have covered the foundational chapter of work (Chapter 2) and its extension, which types (many) more interesting properties of programs in Chapter 3. This chapter will give some background on the proposed path in how we will culminate this thesis, aiming to develop an expressively complete type system for Polarized Subtyping. Such a system would encompass the usual (subtyping-free) functional program systems such as ML or Haskell as special cases. Much of this chapter is “hand-wavy” and speculative, but the output is crucial to the thesis’s success and carries through the novelty of the research.

4.1 Parametric Polymorphism

The notion of parametric polymorphism goes back to the 1967 lecture notes of Christopher Strachey [148], where he proposed a system obtained when a function works uniformly on a range of types that normally exhibit *some common structure* [26]. In modern type systems, parametric polymorphism is fundamental; yet, its integration within a framework that also supports recursive subtyping polymorphism, intersection polymorphism, etc., is not regularly approached. Whereas intersections allow us to express specific properties within composite data types, parametric polymorphism refers to expressions that behave the same at all possible types. Additionally, as is the case with any type expansion that does not alter the syntax and semantics of our terms (in the implicit case for polymorphism [84]), a value restriction arises that forces an evaluation of the type system in the presence of effects.

To fully capture the challenge of this thesis in Section 1.2, we must extend Polarized Subtyping to support parametric polymorphism. The challenge is how to support it while maintaining the properties of the extensions we have added to this polarized CBPV variant thus far. We also need to find examples where programs are not well captured without the combination of mixed-evaluation, property types, and polymorphism.

Even related, recent works like Economou et al. [63] have left the problem of parametric polymorphism open in trying to model a separate refinement system for their variant of CBPV. We discuss two possible paths to achieve this.

4.1.1 Parametric Subtyping

Going beyond current work exploring structural subtyping with row polymorphism [152], one formula for a proposed solution to implement structural parametric polymorphism lies in DeYoung, et al.'s study on *parametric subtyping* [47]. Their work is a significant step toward filling the gap regarding the interplay among recursive types, parametric polymorphism, and structural subtyping. Unlike our backward-search decision procedure (see Sections 2.4.1 and 3.2.2), they implement a saturation-based decision procedure for monomorphic subtyping based on forward inference. However, their investigation only interprets types coninductively, not taking into account uninhabited types, which we find to be a crucial aspect of our approach to handling types in our consequent-restricted representation (Section 2.4.2). Moreover, they do not discuss property types such as intersections and unions. Their work conjectures that their algorithm can be extended to inductive and mixed inductive-coinductive interpretations—making it polarized—but such a decision procedure may cause difficulties when interacting with property types.

4.1.2 Implicit Let-Polymorphism

Another approach to parametric polymorphism is to extend the system with implicit let-polymorphism, as popularized by the influential Hindley-Milner type system [38, 111], which generalizes to type schemes. With our current system's push toward signature(s) in normal form and the heavy use of type names, as well as a typechecking algorithm implemented leveraging memoization, we may be able to avoid the inefficiency issues in the Hindley-Milner algorithm [120]. However, Vytiniotis et al. [166] argue that we should simply abandon *let generalisation* for local lets entirely, thereby reducing the burden of complexity placed on the programmer, removing a feature that can break down in the presence of advanced type features like those we have discussed in this thesis. Nevertheless, some evidence to the contrary is found in Davies [44]'s thesis, suggesting that this path is not closed.

4.2 Polarized Subtyping as an Intermediate Representation

A longer-term goal of this thesis would involve engineering a small prototype that can serve as an intermediate representation (IR) for optimized compilation and/or static analysis [150]. Today, Polite gives us a surface-level language for iterating on and testing type system features and extensions. The bet on CBPV [143] is that it can be used to specialize code based on polarity, which captures the evaluation regime and if the types in the program represent a pure or effectful computation (or one that is mixed). Unlike CPS (continuation-passing style), where evaluation strategies are hard-coded into the program directly, a universal intermediate language built on our vision of Polarized Subtyping is suitable for compiling and optimizing for both strict and lazy functional programs seamlessly. As Downen and Ariola [50] state in their paper, this kind of universal intermediate language has been a long-sought holy grail for compiler writers.

Chapter 5

Related Work and Discussion

“ *Rather than worry about what types are I shall focus on the role of type checking* ”

— *James H. Morris, Jr.*, Types Are Not Sets [112]

We now dive deeper into the research related to how polarization and a semantic representation of typing impact the interaction and definition of subtyping and discuss its implications on this thesis. Our work deeply connects various threads throughout the literature on type systems.

5.1 (Sub)typing

Recursive Types. The groundwork for coinductive interpretations of subtyping equirecursive types has been laid by Amadio and Cardelli [9] and subsequently refined by others [19, 72]. In equirecursive subtyping, recursive types and their unfoldings are considered to be equal. Danielsson and Altenkirch [40] also provided significant inspiration because 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 [32, 41, 77] and, in particular, Gay and Hole's coinductive subtyping algorithm [76], which we used as a template for embedding call-by-name typing into our system (Section 2.6.2).

Our initial treatment of empty—*value-uninhabited*—and full types (Section 2.4.2), as well as our call-by-value translation (Section 2.6.2) builds on Ligatti et al.'s work [106] on precise subtyping with isorecursive types. Our direct interpretation of isorecursive types and translation into an equirecursive setting furthers numerous undertakings relating both formulations [128, 159, 161]. In particular, Abadi and Fiore [1] and more recently Patrigniani et al. [122] 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 inductive-coinductive types like in our presentation.

Finally, Zhou et al. [175] 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 (sub)typing approach can offer a more flexible abstraction for reasoning about expressive type systems while maintaining type safety.

Records. Subtyping and extensible records have a long history with the foundations for typing in object-oriented languages [23, 167], database languages, and their extensions [25]. As a structural type system, our work on Polarized Subtyping naturally supports variant and lazy *records*, not objects. As mentioned in the introduction, row polymorphism in OCaml [74] is used in polymorphic variants, which was proposed as a solution to support union types to Hindley-Milner type systems [74], though in a much more limited capacity, resulting in a segmented type system whose behavior is unintuitive and/or unduly restrictive in some cases [31]. OCaml also supports depth subtyping with objects through coercion if an object's individual methods could safely be coerced. Unfortunately, while the availability of width and depth subtyping in a production-quality language like OCaml is a step in the right direction, the language's object system adds runtime overhead and heavy syntactic complexity. Our work aims to provide a more principled and integrated approach to subtyping with records, avoiding this separation.

Our system introduces a static type-level addition to concatenate variant and lazy records to more easily extend record types with new fields, similar to interfaces in structural OO languages, modeling a lightweight form of inheritance [27]. However, unlike other work exploring record concatenation with record types and subtyping [85, 176], our treatment is not about merging record constructs, but, instead, about constructing a new type with recursive fields pointing to a fresh type name, as demonstrated in Section 3.3.

Intersection and Union Types. Intersections have been a major plot point due to their ability to check more precise and granular properties and their long history of surprising complexity. For instance, in Coppo and Dezani's and Barendregt et al's systems [13, 37], typechecking is undecidable, as a term can be typed IFF its execution terminates [49]. As discussed in Section 3.4.1, Davies and Pfenning [45] developed the *value restriction* on the introduction rule for intersection types due to it being unsound in the presence of mutable references with CBV semantics. The standard subtyping distriutivity law $(A \to B) \cap (A \to C) \leq A \to (B \cap C)$ was also discarded.

Dual to intersection types, the involvement of union types can easily break certain desirable metatheoretical properties of a type system [51], as was already hinted at with the (co)value restriction on union types variably referenced in [51, 55, 61, 165, 171]. This constraint leads to an evaluation context restriction on union elimination in the presence of effects [61] for CBN semantics due to nondeterminism. This restriction also implies that the standard law $(A \to C) \cap (B \to C) \leq (A \cup B) \to C$ is unsound [173].

The blend of intersections and unions can be troublesome and non-trivial [133]! Like others, our goal is to empower *structural* subtyping in type systems with union and intersection types [113]; yet, without having to express unions or intersections at the term level by way of *merge* [57, 89, 136, 138, 139] or *switch* [134] operators that elaborate to sums and products respectively [57, 137], or having to employ *best-match semantics* [134]. With our combination of polarization and semantic typing realized by step-indexing, our system provides a more principled and practical approach to subtyping with intersection and union types in settings both pure and effectful.

The complexity of intersection and union types has been a central theme of this thesis. Much of our original inspiration stemmed from exploring the interplay between intersections and unions in a polarized setting.

Type Difference. As mentioned in Section 3.1.5, type difference, or *type subtraction*, is a less common feature in type systems, but has roots in record extensions as restriction and removal operations [25]. Moreover, OO languages with traits or mixins, such as the Pharo language [153], also include operations to rename and remove methods for resolving conflicts. In the *semantic subtyping* line of work by Frisch and Castagna [28, 70], they have explored type difference in the context of set-difference: $A \backslash B$, with the limitation that set difference does not generalize the record restriction (though we do use the same connective, "$\backslash$", in our formalism). Most recently, the work of Xu, et al. [170] contributes to the canon, formalizing type difference with intersections. However, their work is not polarized and centers on the elaboration of disjoint intersection types and the merge operator, which we avoid.

5.2 Semantic (Sub)typing

Semantic typing goes back to Milner's *semantic soundness theorem* [111], which defined a well-typed program being semantically free of a type violation. As discussed earlier, Dreyer et al. [53] go further by advocating for semantic type soundness in modeling safety with a more liberal interpretation of well-typedness. Whereas syntactic typing specifies a fixed set of syntactic rules from which safe terms can be constructed, semantic typing in our setting 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. Our work to date demonstrates that we can prove our semantic definitions compatible with our syntactic type rules, leaving syntactic type soundness to fall out easily (see Section 2.4.3). With a lens toward more liberal and expressive systems, *step-indexing* has become a prominent approach [7, 8, 10, 52], which we use to observe that a computation in our model *steps* according to our dynamics.

Other examples of semantic typing in practice have been incorporated into trusted libraries for Rust [92] and gradual typing [73, 117]. In the latter case, as long as we can *prove* by any means that the "dynamically typed" part of the program is semantically well-typed (even if not syntactically so), the combination is sound and can be executed without concern, yielding a correctly observable result. Another example is provided by *session types* for message-passing concurrency [88]. While having a syntactic type discipline is important, in a distributed system, processes may be programmed in various languages, some of which may have much weaker guarantees. Demonstrating their semantic soundness ensures the behavioral soundness of the combined system.

Semantic Subtyping. 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 [28, 68, 125]. This line of work exploits 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 [29, 30, 124] and variants [31], recursive types (interpreted coinductively), and union, intersection, and negation connectives [70]. A major theme in this line of work is excising "circularity" [28, 70] by means of an involved bootstraping 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, the interaction between intersections and unions, and the commitment to explicit polarization (including our incorporation of emptiness/fullness). The latter eliminates circularity and links multiple threads defined in this thesis. For us, "types are not sets" as we focus on the role of typechecking [112].

With this combination of semantic (sub)typing, 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 [53], 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 advanced type features.

5.3 Polarized Type Theory & Call-by-push-value

Decomposing Call-by-value & Call-by-name Dynamics. At the core of this research is the call-by-push-value [103, 105] 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 [64] to those reasoning about effects [109] and effect/co-effect tracking [155], and dependent types [123]. New et al.'s [117] 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 it includes no coverage of subtyping.

Although CBPV and polarized type theory generally go hand-in-hand, there are investigations that look at polarization (*focusing*) and algebraic typing and subtyping from alternate perspectives.

Steffen [147] predates Levy's research and presents polarity (a different meaning of the same term) as a kinding system for exploiting monotone and antimonotone operators in subtyping function application. Abel and others [2, 5, 6] built upon this and extended it with *sized types*. 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 inherent connection between types and evaluation strategy has also been studied in the setting of program synthesis [140] and proof theory [114], but these do not share our specific semantic concerns.

Two studies on a global approach to algebraic subtyping [48, 120] define subtyping relationships with generative datatype constructors while discussing polarity (here with a different meaning) and discarding semantic interpretations. The generative nature of datatype constructors in this line of work makes it quite different from ours.

Mixed Inductive-Coinductive View of (Sub)typing. The natural separation of positive and negative layers in CBPV leads to a mixed inductive-coinductive view on (sub)typing. Danielsson and Altenkirch [40] and Jones and Pearce [91] provide definitions for equirecursive subtyping relations in a mixed-setting while using a suspension monad for nonterminating computations, sharing an affinity with *force*/*return* CBPV computations. The former, however, does not try to justify the structural typing rules themselves using semantic typing of values or expressions—only the subtyping rules. Jones and Pearce is 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 [94] manages infinitary subtyping (only for function and recursive types) through 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 [101] mix induction-coinduction in a syntax-directed framework, focusing on circular proof derivations and sized types [6] and managing inductive types coinductively. Cohen and Rowe [35] 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. Chen and Pfenning [33] encode a variant of our work to date (Chapter 2) into a logical framework with higher-order circular terms, showcasing a version of our work used in a different setting. It seems quite plausible that we could use such systems to formalize our study, although we found some merit in using step-indexing and other approaches for induction.

Computational Effects. We have already documented much of the non-triviality of polymorphism, intersections, and unions in the presence of effects, typically resulting in syntactic restrictions and new rules. We aim to dive into this problem area because CBPV was designed as a study for the λ-calculus with effects [103, Section 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 (see 3.4). We also see a tangential tie-in with linear logic and intersection types as described by the monadic type system in Gavazzo et al's recent output [75].

Refinement Types. Another important influence has been the work on *datasort refinement types* [44, 58, 67, 83], 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. Neither does their work add much in the way of effect handling of property types beyond the known value restriction. Note that these types are distinct from those described in Liquid Typing (see [80]). Dunfield [58] offers an approach to extensible refinements but does not tackle type polymorphism, which would be feasible as long as the datatype declarations are not extensible.

Polarization, through the lens of focusing, as an organizing principle for subtyping is present in Zeilberger's thesis [172] and in categorical formulations like Melliès and Zeilberger [110], but address a problem that is fundamentally different in multiple ways, e.g., using "classical" types and continuations, without width and depth subtyping, and treats refinement typing as a secondary layer. 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.

We have already covered the pros and cons with popular Liquid Type implementations [80, 162] that allow refinement with predicates that can mention various attributes. We drilled in on their restriction on recursive predicates of specifications to require termination. Generally, refinement types have drawbacks dealing with the extensibility of types versus the burden placed on the programmer (though still lightweight compared to dependently-type manifestations). The original, datasort-oriented line of work [67] forced programmers to manually specify all refinements, which is cumbersome. Liquid refinement types automatically verify semantic properties of code with a sublanguage for which implication checking is decidable, allowing for the predictable use of external SMT solvers to do the checking; however, such a constraint restricts the specification language and what properties the programmer can enforce and extend the system with. Recent work by Economou, et al. [63] *refines* CBPV with a focalized variant that performs index refinement [169], uses bidirectional checking, and designs a sound, complete, and decidable polarized subtyping relation. However, they still use a refinement layer atop of CBPV.

One can see this thesis as an attempt to free refinement types from some of its restrictions while retaining some of its good properties, all without the complexity of implicit or explicit additional layering.

Chapter 6

Timeline

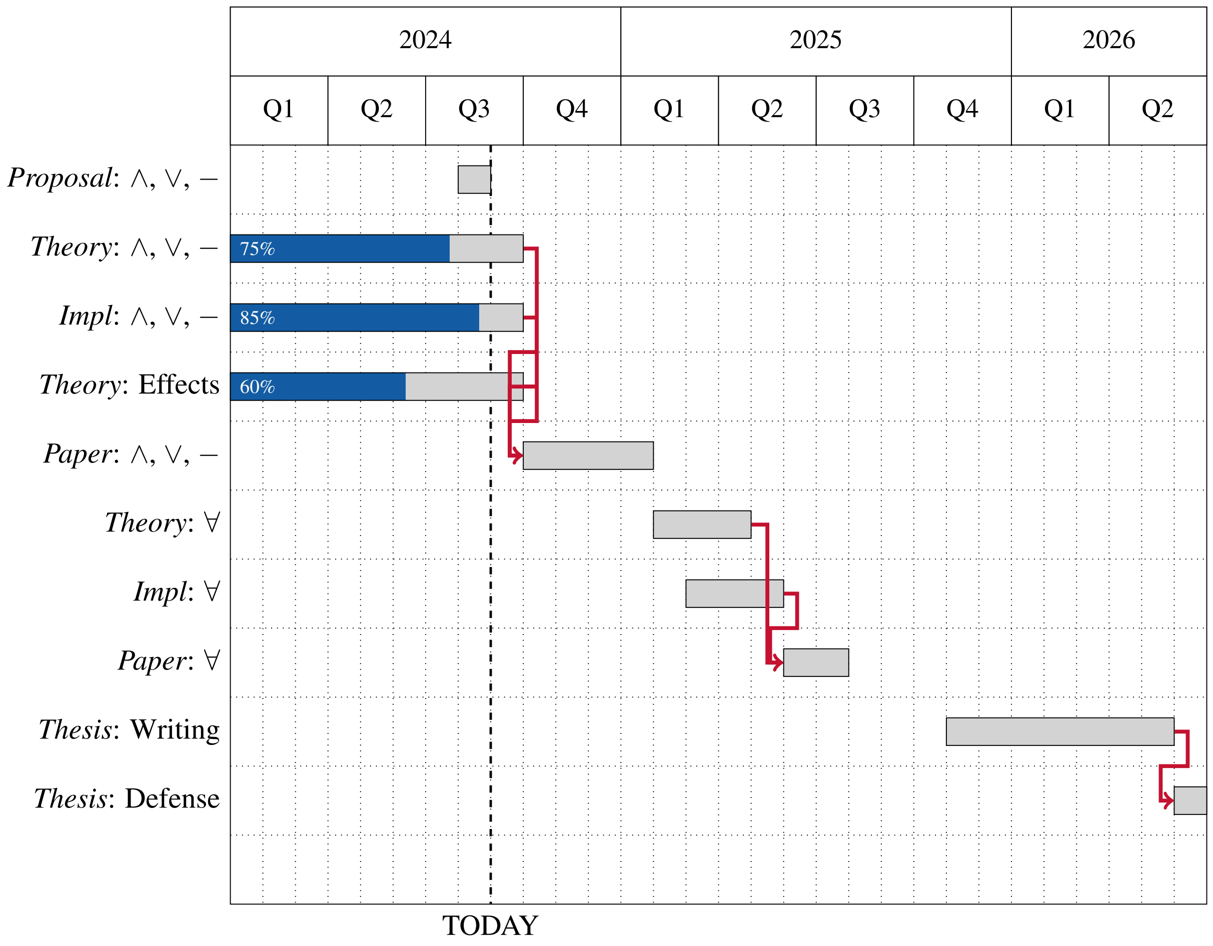

Figure 6.1: A proposed research timeline. **Blue** is the estimated amount of work completed for the task, while **gray** represents the proposed and unfinished work. ∧, ∨, − refer to the in progress directions involving intersections, unions, and type difference; ∀ references the proposed work on parametric polymorphism.

The plan for this road map, post-proposal, is to finish the current work in progress (Chapter 3) and submit it to *ESOP 2025* or, more realistically, to a conference with an early 2025 submission (maybe *ICALP* or elsewhere). We mainly need to tie up some results in our work around effects. In 2025, we plan to engage with the theory and implementation needed to extend the system with parametric polymorphism (Chapter 4) and prepare it for a submission. Finally, the plan in 2026 will be to complete the thesis and defend it. This timeline was prepared with consideration for my full-time work schedule at Oxide Computer and managing a wonderfully curious 5-year-old named Lana along the way!

Bibliography

[1] Abadi, M. and Fiore, M. P. (1996). Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 242–252. IEEE Computer Society.

[2] Abel, A. (2006). 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*, volume 3967 of *Lecture Notes in Computer Science*, pages 381–392. Springer.

[3] Abel, A. (2007). Mixed inductive/coinductive types and strong normalization. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 286–301. Springer.

[4] Abel, A. (2012). Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Miller, D. and Ésik, Z., editors, *Proceedings of the 8th Workshop on Fixed Points in Computer Science*, FICS 2012, pages 1–11. Electronic Proceedings in Theoretical Computer Science 77.

[5] Abel, A. and Pientka, B. (2013). Wellfounded recursion with copatterns: A unified approach to termination and productivity. In Morrisett, G. and Uustalu, T., editors, *International Conference on Functional Programming (ICFP'13)*, pages 185–196, Boston, Massachusetts. ACM.

[6] Abel, A. and Pientka, B. (2016). Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2.

[7] Ahmed, A. J. (2004). *Semantics of types for mutable state*. PhD thesis, Princeton University.

[8] Ahmed, A. J. (2006). Step-indexed syntactic logical relations for recursive and quantified types. In Sestoft, P., editor, *15th European Symposium on Programming (ESOP 2006)*, pages 69–83, Vienna, Austria. Springer LNCS 3924.

[9] Amadio, R. M. and Cardelli, L. (1993). Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631.

[10] Appel, A. W. and McAllester, D. A. (2001). An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems*, 23(5):657–683.

[11] Bahr, P. and Hutton, G. (2022). Monadic compiler calculation (functional pearl). *Proc. ACM Program. Lang.*, 6(ICFP):80–108.

[12] Barbanera, F., Dezaniciancaglini, M., and Deliguoro, U. (1995). Intersection and union types: Syntax and semantics. *Inf. Comput.*, 119(2):202–230.

[13] Barendregt, H., Coppo, M., and Dezani-Ciancaglini, M. (1983). A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940.

[14] Barrett, C. (2023). *On the simply-typed Functional Machine Calculus: Categorical semantics and strong normalisation*. PhD thesis, University of Bath.

[15] Barwise, J. (1989). *The situation in logic*, volume 17 of *CSLI lecture notes series*. CSLI.

[16] Bauman, S., Bolz-Tereick, C. F., Siek, J., and Tobin-Hochstadt, S. (2017). Sound gradual typing: only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):1–24.

[17] Benton, N. and Wadler, P. (2002). Linear logic, monads and the lambda calculus. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 420–431. IEEE Comput. Soc. Press.

[18] Berardi, S. and Tatsuta, M. (2018). Intuitionistic Podelski-Rybalchenko theorem and equivalence between inductive definitions and cyclic proofs. In Cïrstea, C., editor, *Workshop on Coalgebraic Methods in Computer Science (CMCS 2018)*, pages 13–33, Thessaloniki, Greece. Springer LNCS 11202.

[19] Brandt, M. and Henglein, F. (1998). Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338.

[20] Brotherston, J. (2005). Cyclic proofs for first-order logic with inductive definitions. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 78–92. Springer Berlin Heidelberg, Berlin, Heidelberg.

[21] Brotherston, J. and Simpson, A. (2011). Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216.

[22] Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. C. (1989). F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, New York, New York, USA. ACM Press.

[23] Cardelli, L. (1994). Extensible records in a pure calculus of subtyping. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 373–425. MIT Press, Cambridge, MA, USA.

[24] Cardelli, L. (1996). Type systems. *ACM Comput. Surv.*, 28(1):263–264.

[25] Cardelli, L. and Mitchell, J. C. (1991). Operations on records. *Math. Struct. Comput. Sci.*, 1(1):3–48.

[26] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523.

[27] Cartwright, R. (2013). Inheritance is subtyping.

[28] Castagna, G. and Frisch, A. (2005). A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 198–208, New York, NY, USA. Association for

Computing Machinery.

[29] Castagna, G., Nguyen, K., Xu, Z., and Abate, P. (2015). 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*, POPL '15, page 289–302, New York, NY, USA. Association for Computing Machinery.

[30] Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., and Padovani, L. (2014). 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*, POPL '14, page 5–17.

[31] Castagna, G., Petrucciani, T., and Nguyen, K. (2016). Set-theoretic types for polymorphic variants. *SIGPLAN Not.*, 51(9):378–391.

[32] Chen, T.-C., Dezani-Ciancaglini, M., and Yoshida, N. (2014). On the preciseness of subtyping in session types. In *Proceedings of the Conference on Principles and Practice of Declarative Programming (PPDP'14)*, Canterbury, UK. ACM.

[33] Chen, Z. and Pfenning, F. (2023). A logical framework with higher-order rational (circular) terms. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 68–88. Springer Nature Switzerland, Cham.

[34] Cockett, J. R. B. (2001). 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*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 88–127. Elsevier.

[35] Cohen, L. and Rowe, R. N. S. (2020). Integrating induction and coinduction via closure operators and proof cycles. In *10th International Joint Conference on Automated Reasoning (IJCAR 2020)*, pages 375–394, Paris, France. Springer LNCS 12166.

[36] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., and Tommasi, M. (2008). *Tree Automata Techniques and Applications*.

[37] Coppo, M. and Dezani-Ciancaglini, M. (1980). An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Form. Log.*, 21(4).

[38] Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*, New York, New York, USA. ACM Press.

[39] Damm, F. M. (1994). Subtyping with union types, intersection types and recursive types. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 687–706. Springer Berlin Heidelberg, Berlin, Heidelberg.

[40] Danielsson, N. A. and Altenkirch, T. (2010). Subtyping, declaratively. In *10th International Conference on Mathematics of Program Construction (MPC 2010)*, pages 100–118, Québec City, Canada. Springer LNCS 6120.

[41] Das, A., DeYoung, H., Mordido, A., and Pfenning, F. (2021). Nested session types. In Yoshida, N., editor, *30th European Symposium on Programming*, pages 178–206, Luxembourg, Luxembourg. Springer LNCS. Extended version available as arXiv:2010.06482.

[42] Das, A. and Pfenning, F. (2020a). Rast: Resource-aware session types with arithmetic refinements system description.

[43] Das, A. and Pfenning, F. (2020b). Session types with arithmetic refinements and their application to work analysis. *arXiv [cs.PL]*.

[44] Davies, R. (2005). *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University.

[45] Davies, R. and Pfenning, F. (2000). Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 198–208, New York, NY, USA. Association for Computing Machinery.

[46] Della Rocca, S. R. and Venneri, B. (1983). Principal type schemes for an extended type theory. *Theor. Comput. Sci.*, 28(1-2):151–169.

[47] DeYoung, H., Mordido, A., Pfenning, F., and Das, A. (2023). Parametric subtyping for structural parametric polymorphism. *arXiv [cs.PL]*.

[48] Dolan, S. (2017). *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR.

[49] Dolan, S. (2024). Counterexamples in type systems - counterexamples in type systems. https://counterexamples.org/title.html. (Accessed on 07/20/2024).

[50] Downen, P. and Ariola, Z. M. (2018). Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In Ghica, D. and Jung, A., editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[51] Downen, P., Ariola, Z. M., and Ghilezan, S. (2019). The duality of classical intersection and union types. *Fundam. Inform.*, 170(1-3):39–92.

[52] Dreyer, D., Ahmed, A., and Birkedal, L. (2009). 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*, pages 71–80. IEEE Computer Society.

[53] Dreyer, D., Timany, A., Krebbers, R., Birkedal, L., and Jung, R. (2019). What type soundness theorem do you really want to prove?

[54] Dunfield, J. (2007a). Refined typechecking with Stardust. In Stump, A. and Xi, H., editors, *Programming Languages meets Program Verification (PLPV '07)*, Freiburg, Germany.

[55] Dunfield, J. (2007b). *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University. CMU-CS-07-129.

[56] Dunfield, J. (2007c). *A unified system of type refinements*. PhD thesis, Carnegie Mellon University.

[57] Dunfield, J. (2012). Elaborating intersection and union types. *SIGPLAN Not.*, 47(9):17–28.

[58] Dunfield, J. (2017). Extensible datasort refinements. In *Programming Languages and Systems*, pages 476–503. Springer Berlin Heidelberg.

[59] Dunfield, J. and Krishnaswami, N. (2019a). Bidirectional typing. *CoRR*, abs/1908.05839.

[60] Dunfield, J. and Krishnaswami, N. R. (2019b). Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.*, 3(POPL):9:1–9:28.

[61] Dunfield, J. and Pfenning, F. (2003). Type assignment for intersections and unions in call-by-value languages. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 250–266. Springer Berlin Heidelberg, Berlin, Heidelberg.

[62] Dunfield, J. and Pfenning, F. (2004). Tridirectional typechecking. *SIGPLAN Not.*, 39(1):281–292.

[63] Economou, D. J., Krishnaswami, N., and Dunfield, J. (2023). Focusing on refinement typing. *ACM Trans. Program. Lang. Syst.*, 45(4):1–62.

[64] Ehrhard, T. and Tasson, C. (2019). Probabilistic call by push value. *Log. Methods Comput. Sci.*, 15(1).

[65] Felleisen, M. and Friedman, D. P. (1987). Control operators, the secd-machine, and the λ-calculus. In *Formal Description of Programming Concepts*.

[66] Filinski, A. (1996). *Controlling efects*. PhD thesis, Carnegie Mellon University.

[67] Freeman, T. and Pfenning, F. (1991). Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277.

[68] Frisch, A., Castagna, G., and Benzaken, V. (2002). Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 137–146. IEEE Computer Society.

[69] Frisch, A., Castagna, G., and Benzaken, V. (2003). Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Comput. Soc.

[70] Frisch, A., Castagna, G., and Benzaken, V. (2008). Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):1–64.

[71] Führmann, C. (1999). Direct models of the computational lambda-calculus. *Electron. Notes Theor. Comput. Sci.*, 20:245–292.

[72] Gapeyev, V., Levin, M. Y., and Pierce, B. C. (2000). 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*, pages 221–231. ACM.

[73] Garcia, R. and Tanter, É. (2020). Gradual typing as if types mattered. In *Informal Proceedings of the ACM SIGPLAN Workshop on Gradual Typing (WGT20).*

[74] Garrigue, J. (2001). Simple type inference for structural polymorphism. In *APLAS*, pages 329–343.

[75] Gavazzo, F., Treglia, R., and Vanoni, G. (2024). Monadic intersection types, relationally. In *Programming Languages and Systems*, Lecture notes in computer science, pages 22–51. Springer Nature Switzerland, Cham.

[76] Gay, S. J. and Hole, M. (2005). Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225.

[77] Gay, S. J. and Vasconcelos, V. T. (2010). Linear type theory for asynchronous session types.

Journal of Functional Programming, 20(1):19–50.

[78] Ghalayini, J. E. and Krishnaswami, N. (2023). Explicit refinement types. *Proc. ACM Program. Lang.*, 7(ICFP):187–214.

[79] Grädel, E. and Kreutzer, S. (2003). Will deflation lead to depletion? On non-monotone fixed point inductions. In *Symposium on Logic in Computer Science (LICS 2003)*, pages 158–167, Ottawa, Canada. IEEE Computer Society.

[80] Greenberg, M. (2015). A refinement type by any other name — weaselhat. https://www.weaselhat.com/2015/03/16/a-refinement-type-by-any-other-name/. (Accessed on 07/08/2024).

[81] Greenman, B., Muehlboeck, F., and Tate, R. (2014). Getting F-bounded polymorphism into shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA. ACM.

[82] Harper, R. (2012). Polarity in type theory — existential type. https://existentialtype.wordpress.com/2012/08/25/polarity-in-type-theory/#comment-1047. (Accessed on 07/18/2024).

[83] Harper, R. and Duff, W. A. (2015). Type refinements in an open world (extended abstract).

[84] Harper, R., Milner, R., and Tofte, M. (1997). The definition of standard ML (revised). https://mitpress.mit.edu/9780262631327/the-definition-of-standard-ml/.

[85] Harper, R. and Pierce, B. (1991). A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 131–142, New York, NY, USA. Association for Computing Machinery.

[86] Hermida, C. and Jacobs, B. (1998). Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152.

[87] Hickey, R. (2018). Maybe not. https://www.youtube.com/watch?v=YR5WdGrpoug. (Accessed on 07/16/2024).

[88] Hinrichsen, J. K., Louwrink, D., Krebbers, R., and Bengtson, J. (2021). 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*, pages 178–198. ACM.

[89] Huang, X., Zhao, J., and Oliveira, B. C. D. S. (2021). Taming the merge operator. *J. Funct. Prog.*, 31(e28):e28.

[90] Jafery, K. A. and Dunfield, J. (2017). 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*, pages 804–817. ACM.

[91] Jones, T. and Pearce, D. J. (2016). 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*, page 1. ACM.

[92] Jung, R., Jourdan, J., Krebbers, R., and Dreyer, D. (2018). Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34.

[93] Kahrs, S. (2001). Red-black trees with types. *J. Funct. Prog.*, 11(4):425–432.

[94] Komendantsky, V. (2011). 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*, volume 7193 of *Lecture Notes in Computer Science*, pages 17–32. Springer.

[95] Kozen, D., Palsberg, J., and Schwartzbach, M. I. (1993). Efficient recursive subtyping. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, New York, New York, USA. ACM Press.

[96] Krishnaswami, N. (2018). Semantic domain: Polarity and bidirectional typechecking. https://semantic-domain.blogspot.com/2018/08/polarity-and-bidirectional-typechecking.html. (Accessed on 08/10/2024).

[97] Lakhani, Z., Das, A., DeYoung, H., Mordido, A., and Pfenning, F. (2022a). Polarized subtyping. In *Programming Languages and Systems*, pages 431–461. Springer International Publishing.

[98] Lakhani, Z., Das, A., DeYoung, H., Mordido, A., and Pfenning, F. (2022b). Polarized subtyping. *CoRR*, abs/2201.10998.

[99] Lakhani, Z., Das, A., DeYoung, H., Mordido, A., and Pfenning, F. (2022c). Polarized subtyping: Code/artifact.

[100] Lepigre, R. and Raffalli, C. (2016). Subtyping-based type-checking for system F with induction and coinduction. *CoRR*, abs/1604.01990.

[101] Lepigre, R. and Raffalli, C. (2019). Practical subtyping for Curry-style languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41:1 – 58.

[102] Levy, P. B. (1999). Call-by-push-value: A subsuming paradigm. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 228–243. Springer Berlin Heidelberg, Berlin, Heidelberg.

[103] Levy, P. B. (2001). *Call-by-Push-Value*. PhD thesis, University of London.

[104] Levy, P. B. (2003). Adjunction models for call-by-push-value with stacks. *Electron. Notes Theor. Comput. Sci.*, 69:248–271.

[105] Levy, P. B. (2006). Call-by-push-value: Decomposing call-by-value and call-by-name. *High.-order Symb. Comput.*, 19(4):377–414.

[106] Ligatti, J., Blackburn, J., and Nachtigal, M. (2017). On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems*, 39(4):4:1–4:36.

[107] MacQueen, D., Plotkin, G., and Sethi, R. (1986). An ideal model for recursive polymorphic types. *Inf. Contr.*, 71(1-2):95–130.

[108] Marlow, S. et al. (2010). haskell2010.pdf. https://www.haskell.org/definition/haskell2010.pdf. (Accessed on 07/18/2024).

[109] McDermott, D. and Mycroft, A. (2019). 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, volume 11423 of *Lecture Notes in Computer Science*, pages 235–262. Springer.

[110] Melliès, P.-A. and Zeilberger, N. (2015). Functors are type refinement systems. *SIGPLAN Not.*, 50(1):3–16.

[111] Milner, R. (1978). A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375.

[112] Morris, J. H. (1973). Types are not sets. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 120–124, New York, NY, USA. Association for Computing Machinery.

[113] Muehlboeck, F. and Tate, R. (2018). Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA):1–29.

[114] Munch-Maccagnoni, G. (2013). *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)*. PhD thesis, Paris Diderot University, France.

[115] Møgelberg, R. E. and Veltri, N. (2019). Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.*, 3(POPL):1–29.

[116] Nakata, K. and Uustalu, T. (2010). 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*, volume 32 of *EPTCS*, pages 57–75.

[117] New, M. S., Licata, D. R., and Ahmed, A. (2019). Gradual type theory. *Proc. ACM Program. Lang.*, 3(POPL):15:1–15:31.

[118] nLab (2023). consequent in nlab. https://ncatlab.org/nlab/show/consequent. (Accessed on 07/28/2024).

[119] Park, D. M. R. (1979). On the semantics of fair parallelism. In Bjørner, D., editor, *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer.

[120] Parreaux, L. (2020). 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.

[121] Parys, P. (2018). Intersection types for unboundedness problems. In Pagani, M. and Alves, S., editors, *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, DCM/ITRS 2018, Oxford, UK, 8th July 2018*, volume 293 of *EPTCS*, pages 7–27.

[122] Patrignani, M., Martin, E. M., and Devriese, D. (2021). On the semantic expressiveness of recursive types. *Proceedings of the ACM on Programming Languages*, 5:1–29.

[123] Pédrot, P. and Tabareau, N. (2020). The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28.

[124] Petrucciani, T. (2019). *Polymorphic set-theoretic types for functional languages. (Types ensemblistes polymorphes pour les langages fonctionnels)*. PhD thesis, Sorbonne Paris Cité, France.

[125] Petrucciani, T., Castagna, G., Ancona, D., and Zucca, E. (2018). Semantic subtyping for non-strict languages. In *24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal*, volume 130 of *LIPIcs*, pages 4:1–4:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[126] Pfenning, F. (2001). Subtyping and intersection types revisited. *logic. Mathematical Structures in Computer Science*, 11:511–540.

[127] Pierce, B. (1992). *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University.

[128] Pierce, B. (2002). *Types and Programming Languages*. MIT Press.

[129] Pierce, B. C. and Turner, D. N. (1998). Local type inference. In *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[130] Pottinger, G. (1980). A type assignment for the strongly normalizable lambda-terms. In Hindley, J. and Seldin, J., editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press.

[131] Pruiksma, K., Chargin, W., Pfenning, F., and Reed, J. (2018). Adjoint logic.

[132] Raffalli, C. (1994). *L'arithmetique fonctionnelle du second ordre avec points fixes*. PhD thesis, Paris 7. Thèse de doctorat dirigée par Krivine, Jean-Louis Mathématiques. Logique et fondements de l'informatique Paris 7 1994.

[133] Rehman, B. (2023). *A Blend of Intersection Types and Union Types*. PhD thesis, The University of Hong Kong.

[134] Rehman, B., Huang, X., Xie, N., and Oliveira, B. C. d. S. (2022). Union types with disjoint switches.

[135] Reynolds, J. C. (1983). Types, abstraction and parametric polymorphism. *IFIP*, pages 513–523.

[136] Reynolds, J. C. (1988). Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University.

[137] Reynolds, J. C. (1991). The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, pages 675–700. Springer Berlin Heidelberg.

[138] Reynolds, J. C. (1997). Design of the programming language FORSYTHE. In *ALGOL-like Languages, Volume 1*, pages 173–233. Birkhauser Boston Inc., USA.

[139] Rioux, N., Huang, X., Oliveira, B. C. d. S., and Zdancewic, S. (2023). A bowtie for a beast: Overloading, eta expansion, and extensible data types. *Proc. ACM Program. Lang.*, 7(POPL):515–543.

[140] Rioux, N. and Zdancewic, S. (2020). Computation focusing. *Proc. ACM Program. Lang.*,

4(ICFP):95:1–95:27.

[141] ”Rossum, G. V. and Levkivskyi, I. (2014). Pep 483 – the theory of type hints — peps.python.org. https://peps.python.org/pep-0483/. (Accessed on 07/27/2024).

[142] Seo, J. and Park, S. (2013). Judgmental subtyping systems with intersection types and modal types. *Acta Inform.*, 50(7-8):359–380.

[143] Smith, E. (2024). I'm betting on call-by-push-value · thunderseethe's devlog. https://thunderseethe.dev/posts/bet-on-cbpv/#implicit-polarized-system-f. (Accessed on 07/11/2024).

[144] Somayyajula, S. (2024). *Total Correctness Type Refinements for Communicating Processes*. PhD thesis, Carnegie Mellon University.

[145] Song, M. R. (2020). Linear time addition of fibonacci encodings. Master's thesis, Carnegie Mellon University.

[146] Spies, S., Krishnaswami, N., and Dreyer, D. (2021). Transfinite step-indexing for termination. *Proc. ACM Program. Lang.*, 5(POPL):1–29.

[147] Steffen, M. (1999). *Polarized higher-order subtyping*. PhD thesis, University of Erlangen-Nuremberg, Germany.

[148] Strachey, C. (2000). Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13:11–49.

[149] Stump, A. (2023). Iowa type theory commute: Subtyping, the golden key. https://ittc.buzzsprout.com/728558/13186709. (Accessed on 07/16/2024).

[150] Suzanne, H. and Chailloux, E. (2023). A reusable machine-calculus for automated resource analyses. In *Logic-Based Program Synthesis and Transformation*, Lecture notes in computer science, pages 61–79. Springer Nature Switzerland, Cham.

[151] Takikawa, A., Feltey, D., Greenman, B., Max S. New, Vitek, J., and Felleisen, M. (2016). Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA. ACM.

[152] Tang, W., Hillerström, D., McKinna, J., Steuwer, M., Dardha, O., Fu, R., and Lindley, S. (2023). Structural subtyping as parametric polymorphism. *arXiv [cs.PL]*.

[153] Tesone, P., Ducasse, S., Polito, G., Fabresse, L., and Bouraqadi, N. (2020). A new modular implementation for stateful traits. *Sci. Comput. Program.*, 195(102470):102470.

[154] ”Tobin-Hochstadt, S. (2019). Gradual typing from theory to practice — sigplan blog. https://blog.sigplan.org/2019/07/12/gradual-typing-theory-practice/. (Accessed on 07/27/2024).

[155] Torczon, C., Acevedo, E. S., Agrawal, S., Velez-Ginorio, J., and Weirich, S. (2023). Effects and coeffects in call-by-push-value (extended version). *arXiv [cs.PL]*.

[156] Type-checking unsoundness (2016). Type-checking unsoundness: standardize treatment of such issues among typescript team/community? · issue #9825 · microsoft/typescript. https://github.com/Microsoft/TypeScript/issues/9825. (Accessed on 07/18/2024).

[157] TypeScript: Documentation (2024). Typescript: Documentation - type compatibility. https://www.typescriptlang.org/docs/handbook/type-compatibility.html. (Accessed on 07/18/2024).

[158] TypeScript Playground (2024). Typescript playground - soundness. https://

www.typescriptlang.org/play/?strictFunctionTypes=false#example/soundness. (Accessed on 07/18/2024).

[159] Urzyczyn, P. (1995). Positive recursive type assignment. In *Mathematical Foundations of Computer Science 1995*, pages 382–391, Berlin, Heidelberg. Springer Berlin Heidelberg.

[160] Uustalu, T. and Vene, T. (2005). Signals and comonads.

[161] Vanderwaart, J., Dreyer, D., Petersen, L., Crary, K., Harper, R., and Cheng, P. (2003). 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*, pages 98–108. ACM.

[162] Vazou, N., Seidel, E. L., and Jhala, R. (2014a). LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, New York, NY, USA. ACM.

[163] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S. (2014b). Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA. ACM.

[164] "Verlaguet, J. and Menghrajani, A. (2014). Hack: a new programming language for hhvm - engineering at meta. https://engineering.fb.com/2014/03/20/developer-tools/hack-a-new-programming-language-for-hhvm/. (Accessed on 07/27/2024).

[165] Vouillon, J. (2004). Subtyping union types. In *Computer Science Logic*, Lecture notes in computer science, pages 415–429. Springer Berlin Heidelberg, Berlin, Heidelberg.

[166] Vytiniotis, D., Peyton Jones, S., and Schrijvers, T. (2010). Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 39–50, New York, NY, USA. ACM.

[167] Wand, M. (1987). Complete type inference for simple objects. *Proc. - Symp. Log. Comput. Sci.*, pages 37–44.

[168] Widera, M. and Beierle, C. (2001). Function types in complete type inference. In *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01)*, SFP '01, pages 111–122, GBR. Intellect Books.

[169] Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA. ACM.

[170] Xu, H., Huang, X., and Oliveira, B. C. d. S. (2023). Making a type difference: Subtraction on intersection types as generalized record operations. *Proc. ACM Program. Lang.*, 7(POPL):893–920.

[171] Zeilberger, N. (2008). On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3):66–96.

[172] Zeilberger, N. (2009a). *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, USA.

[173] Zeilberger, N. (2009b). Refinement types and computational duality. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 15–26, New York,

NY, USA. ACM.

[174] Zhou, L., Zhou, Y., and Oliveira, B. C. d. S. (2023). Recursive subtyping for all. *Proc. ACM Program. Lang.*, 7(POPL):1396–1425.

[175] Zhou, Y., d. S. Oliveira, B. C., and Zhao, J. (2020). Revisiting iso-recursive subtyping. *Proc. ACM Program. Lang.*, 4(OOPSLA):223:1–223:28.

[176] Zhou, Y., Oliveira, B. C. d. S., and Fan, A. (2022a). A calculus with recursive types, record concatenation and subtyping. In *Programming Languages and Systems*, Lecture notes in computer science, pages 175–195. Springer Nature Switzerland, Cham.

[177] Zhou, Y., Zhao, J., and Oliveira, B. C. D. S. (2022b). Revisiting iso-recursive subtyping. *ACM Trans. Program. Lang. Syst.*, 44(4):1–54.