Bidirectional Typing

JANA DUNFIELD, Queen's University, Canada
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom

Bidirectional typing combines two modes of typing: type checking, which checks that a program satisfies a known type, and type synthesis, which determines a type from the program. Using checking enables bidirectional typing to support features for which inference is undecidable; using synthesis enables bidirectional typing to avoid the large annotation burden of explicitly typed languages. In addition, bidirectional typing improves error locality. We highlight the design principles that underlie bidirectional type systems, survey the development of bidirectional typing from the prehistoric period before Pierce and Turner's local type inference to the present day, and provide guidance for future investigations.

CCS Concepts: • **Software and its engineering → Data types and structures**; • **Theory of computation → Type theory**

Additional Key Words and Phrases: Type checking, type inference

ACM Reference format:
Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (May 2021), 38 pages.
https://doi.org/10.1145/3450952

1 INTRODUCTION

Type systems serve many purposes. They allow programming languages to reject nonsensical programs. They allow programmers to express their intent and to use a type checker to verify that their programs are consistent with that intent. Type systems can also be used to automatically insert implicit operations, and even to guide program synthesis.

Automated deduction and logic programming give us a useful lens through which to view type systems: *modes* [Warren 1977]. When we implement a typing judgment, say $\Gamma \vdash e : A$, is each of the meta-variables (Γ, e, A) an input, or an output? If the typing context Γ, the term e, and the type A are inputs, then we are implementing type checking. If the type A is an output, then we are implementing type inference. (If only e is input, then we are implementing *typing inference* [Jim 1995; Wells 2002]; if e is output, then we are implementing program synthesis.) The status of each meta-variable—input or output—is its *mode*.

As a general rule, outputs make life more difficult. In complexity theory, it is often relatively easy to check that a given solution is valid, but finding (synthesizing) a solution may be complex or even

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through Discovery Grant No. RGPIN–2018–04352.
Authors' addresses: J. Dunfield, Queen's University, School of Computing, Goodwin Hall 557, Kingston, ON, K7L 3N6, Canada; email: jd169@queensu.ca; N. Krishnaswami, University of Cambridge, Computer Laboratory, William Gates Building, Cambridge, CB3 0FD, United Kingdom; email: nk480@cl.cam.ac.uk.

0360-0300/2021/05-ART98 $15.00
https://doi.org/10.1145/3450952

undecidable. This general rule holds for type systems: synthesizing types may be convenient for the programmer, but computationally intractable.

To go beyond the specific feature set of traditional Damas–Milner typing, it might seem necessary to abandon synthesis.[1] Instead, however, we can *combine* synthesis with checking. In this approach, *bidirectional typing*, language designers are not forced to choose between a rich set of typing features and a reasonable volume of type annotations: implementations of bidirectional type systems alternate between treating the type as input, and treating the type as output.

The practice of bidirectional typing has, at times, exceeded its foundations: the first commonly cited paper on bidirectional typing appeared in 1997 but mentioned that the idea was known as "folklore" (see Section 10). Over the next few years, several bidirectional systems appeared, but the principles used to design them were not always made clear. Some work did present underlying design principles—but within the setting of some particular type system with other features of interest, rather than focusing on bidirectional typing as such. For example, Dunfield and Pfenning [2004] gave a broadly applicable design recipe for bidirectional typing, but their focus was on an idea—typing rules that decompose evaluation contexts—that has been applied more narrowly.

Our survey has two main goals: (1) to collect and clearly explain the design principles of bidirectional typing, to the extent they have been discovered; and (2) to provide an organized summary of past research related to bidirectional typing.

We begin by describing a tiny bidirectional type system (Section 2). Section 3 presents some design criteria for bidirectional type systems. Section 4 describes a modified version of the recipe of Dunfield and Pfenning [2004] and relates it to our design criteria. Section 5 discusses work on combining (implicit) polymorphism with bidirectional typing, and Section 6 surveys other variations on bidirectional typing. Sections 7 and 8 give an account of connections between bidirectional typing and topics such as proof theory, focusing and polarity, and call-by-push-value. Section 9 cites other work that uses bidirectional typing. We conclude with historical notes (Section 10), a summary of notation (Section 11), and some possible directions for future work (Section 12).

2 BIDIRECTIONAL SIMPLY TYPED LAMBDA CALCULUS

To develop our first bidirectional type system, we start with a (non-bidirectional) **simply typed lambda calculus (STLC)**. This calculus is not the smallest possible one: we include a term $()$ of type unit, to elucidate the process of bidirectionalization. We also include a "type equality" rule.

Our non-bidirectional STLC (Figure 1, left side) has six rules deriving the judgment $\Gamma \vdash e : A$: a variable rule, a type equality rule, a rule for type annotations, an introduction rule for unit, an introduction rule for $\rightarrow$, and an elimination rule for $\rightarrow$. These rules do not immediately lead to an algorithm, because in $\rightarrow$I, the type A_1 of x must be guessed.

These rules are standard except for the type equality rule TypeEq, which says that if e has type A and A equals B, then e has type B. Such a rule, with a nontrivial form of equality, is found in many non-bidirectional systems; we include it because its bidirectional version is necessary.

Given these six STLC typing rules, we produce each bidirectional rule in turn (treating the type equality rule last). Some of our design choices will become clear only in the light of the "recipe" in Section 4, but the recipe would not be clear without seeing a bidirectional type system first.

The two directions of bidirectional typing are defined by two judgment forms: *checking* $\Gamma \vdash e \Leftarrow A$, where the context of typing assumptions Γ, expression e, and type A are given (inputs), and *synthesis* $\Gamma \vdash e \Rightarrow A$, where Γ and e are inputs but the type A is output (synthesized).

[1] We choose to say *synthesis* instead of *inference*. This is less consistent with one established usage, "type inference," but more consistent with another, "program synthesis."

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \mid () \\ \text{Types} & A, B, C & ::= & \mathsf{unit} \mid A \to A \\ \text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

$\boxed{\Gamma \vdash e : A}$ Under context Γ, expression e has type A

$\boxed{\begin{array}{l}\Gamma \vdash e \Leftarrow A \\ \Gamma \vdash e \Rightarrow A\end{array}}$ $\begin{array}{l}\text{Under } \Gamma\text{, expression } e \text{ checks against type } A \\ \text{Under } \Gamma\text{, expression } e \text{ synthesizes type } A\end{array}$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \mathsf{Var} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \mathsf{Var}\Rightarrow$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A}\ \mathsf{Anno} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}\ \mathsf{Anno}\Rightarrow$$

$$\frac{}{\Gamma \vdash () : \mathsf{unit}}\ \mathsf{unitI} \qquad \frac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}}\ \mathsf{unitI}\Leftarrow$$

$$\frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash (\lambda x.\, e) : A_1 \to A_2}\ \to\mathsf{I} \qquad \frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash (\lambda x.\, e) \Leftarrow A_1 \to A_2}\ \to\mathsf{I}\Leftarrow$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \to\mathsf{E} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \to\mathsf{E}\Rightarrow$$

$$\frac{\Gamma \vdash e : A \qquad A = B}{\Gamma \vdash e : B}\ \mathsf{TypeEq} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \mathsf{Sub}\Leftarrow$$

Fig. 1. A simply typed λ-calculus ($:$ judgment) and a bidirectional version ($\Rightarrow$ and $\Leftarrow$ judgments).

In this section, we produce exactly one bidirectional rule for each type assignment rule, but this design decision is not universal (e.g., the sum elimination rules in Section 4.1).

(1) The variable rule Var has no typing premise, so our only decision is whether the conclusion should synthesize A or check against A. The information that x has type A is in Γ, so we synthesize A. In a checking version of the rule, the type would be an input and would therefore have to be known *from the enclosing term*, a very strong restriction: even $f\ x$ would require a type annotation.

(2) From the annotation rule Anno, we produce Anno$\Rightarrow$, which synthesizes its conclusion: We have the type A in $(e : A)$, so we do not need A to be given as input. In the premise, we check e against A; synthesizing A would prevent the rule from typing a non-synthesizing e, which would defeat the purpose of the annotation.

(3) Unit introduction unitI *checks*. At this point in the article, we prioritize internal consistency: checking $()$ is analogous to the introduction rule for $\to$ (discussed next), and with the introduction rule for products (Section 4.1).

(4) Arrow introduction $\to$I checks. This decision is better motivated: To synthesize $A_1 \to A_2$ for $\lambda x.\, e$, we would have to synthesize a type for the body e. That raises two issues. First, by requiring that the body synthesize, we would need a second rule to handle λs whose body checks but does not synthesize. Second, if we are synthesizing $A_1 \to A_2$, then *we do not know* A_1 *yet*, which prevents us from building the context $\Gamma, x : A_1$. (Placeholder mechanisms, which allow building $\Gamma, x : \hat{\alpha}$ and "solving" the existential type variable $\hat{\alpha}$ later, are described in Section 5.)

Since the conclusion is checking, we know A_2, so we might as well check in the premise.

(5) For arrow elimination →E, the *principal judgment* is the premise $\Gamma \vdash e_1 : A \to B$, because that premise contains the connective being eliminated. We make that judgment synthesize; this choice is the one suggested by our "recipe," and happens to work nicely: If e_1 synthesizes $A \to B$, then we have A and can check the argument (so the rule will work even when e_2 cannot synthesize), and we have B, so we can synthesize B in the conclusion. (It is possible to have the premise typing e_1 be a checking judgment. In that case, the argument e_2 *must* synthesize, because we need to know what A is to check e_1 against $A \to B$. Similarly, the conclusion must be checking, because we need to know B; see Section 6.5.1.)

(6) Finally, we come to the type equality rule TypeEq. Where the type assignment premise $\Gamma \vdash e : A$ and conclusion $\Gamma \vdash e : B$ are identical (since A is exactly equal to B), the duplication of these identical premises enables us to give them different directions in the bidirectional system. Either (1) the conclusion should synthesize (and the premise check), or (2) the conclusion should check (and the premise synthesize). (If we made the premise and conclusion have the same direction, then the rule would not add anything to the system.)

Option (1) cannot be implemented: If the conclusion synthesizes B, then B is *not* an input; we do not know B, which means we also do not know A for checking.
Option (2) works: If we want to check e against a known B in the conclusion, and e synthesizes a type A, then we verify that $A = B$.

Neither Sub⇐ nor Anno⇒ is tied to an operational feature (as, for instance, →E⇒ is tied to functions); Anno⇒ is tied to a syntactic form, but (supposing a type erasure semantics) not to any operational feature. Moreover, Sub⇐ and Anno⇒ have a certain symmetry: Sub⇐ moves from a checking conclusion to a synthesizing premise, while Anno⇒ moves from a synthesizing conclusion to a checking premise.

3 ELEMENTS OF BIDIRECTIONAL TYPING

From a rules-crafting perspective, bidirectionality adds a degree of design freedom to every judgment (premises and conclusion) in a rule: Should a particular premise or conclusion synthesize a type or check against a known type? Covering all possibilities with rules that have every combination of synthesis and checking judgments would lead to an excessive number of rules. For example, the following rules are superficially valid bidirectional versions of the standard →-elimination rule.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 1 \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 2 \qquad \frac{\Gamma \vdash e_1 \Leftarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 3 \qquad \frac{\Gamma \vdash e_1 \Leftarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 4$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 5 \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 6 \qquad \frac{\Gamma \vdash e_1 \Leftarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 7 \qquad \frac{\Gamma \vdash e_1 \Leftarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 8$$

What criteria should guide the designer in crafting a manageable set of rules with good practical (and theoretical) properties?

3.1 First Criterion: Mode-correctness

The first criterion comes from logic programming [Warren 1977]. We want to avoid having to guess types: in an ideal world, whenever we synthesize a type, the type should come from known information—rather than, say, enumerating all possible types. A rule is *mode-correct* if there is a strategy for recursively deriving the premises such that two conditions hold:

(1) The premises are mode-correct: for each premise, every input meta-variable is known (from the inputs to the rule's conclusion and the outputs of earlier premises).
(2) The conclusion is mode-correct: if all premises have been derived, the outputs of the conclusion are known.

Our last rule, in which every judgment is checking ($\Leftarrow$), is *not* mode-correct: In the first premise $\Gamma \vdash e_1 \Leftarrow A \to B$, the context Γ and term e_1 are known from the inputs Γ and $e_1\ e_2$ in the conclusion $\Gamma \vdash e_1\ e_2 \Leftarrow B$. However, the type $A \to B$ cannot be constructed, because A is not known. For the same reason, the second premise $\Gamma \vdash e_2 \Leftarrow A$ is not mode-correct. (The conclusion is mode-correct, because all the meta-variables are inputs.)

Only four of the above eight rules are mode-correct:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 1 \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ 2 \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 6 \qquad \frac{\Gamma \vdash e_1 \Leftarrow A \to B \quad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ 7$$

The mode-correctness of the fourth rule seems questionable: If we insist on recursively deriving the first premise $\Gamma \vdash e_1 \Leftarrow A \to B$ before the second premise $\Gamma \vdash e_2 \Rightarrow A$, then it is not mode-correct, but if we swap the premises and view the rule as

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \vdash e_1 \Leftarrow A \to B}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ ,$$

then it is mode-correct: The subterm e_2 synthesizes A, and B is an input in the conclusion, so $A \to B$ is known and e_1 can be checked against it.

This seems disturbing: The order of premises should not affect the meaning of the rule, in the sense that a rule determines a set of possible derivations. But mode-correctness is not about meaning in that sense; rather, it is about a particular strategy for applying the rule. For any one strategy, we can say whether the strategy is mode-correct; then we can say that a rule is mode-correct if there exists some strategy that is mode-correct. The set of strategies is the set of permutations of the premises. So the rule above has two strategies, one for each of the two permutations of its premises; since one of the strategies is mode-correct, the rule is mode-correct.

A bidirectional type system that is *not* mode-correct cannot be directly implemented, defeating the goal that the derivability of a typing judgment should be decidable. Thus, mode-correctness is necessary. However, mode-correctness alone does not always lead to a practical algorithm: if more than one rule is potentially applicable, a direct implementation requires backtracking. When the inputs (the context, the term, and—if in the checking direction—the type) match the conclusion of only one rule, the system is syntax-directed: we can "read off" an implementation from the rules.

3.2 Second Criterion: Completeness (Annotatability)

The empty set of rules satisfies the first criterion: every rule is mode-correct, because there are no rules.

Our second criterion rejects the empty system. A bidirectional system is *complete with respect to a type assignment system* if every use of a rule in the type assignment system can be "matched" by some rule in the bidirectional system. This matching is approximate, because applying the bidirectional rule might require that we change the term—generally, by adding type annotations (sometimes called ascriptions).

For example, forgetting to include an $\to$-elimination rule in a bidirectional type system would make the system incomplete: it would reject all function applications. In general, completeness is easy to achieve, provided we begin with a type assignment system and "bidirectionalize" each rule.

Because the move from a type assignment derivation to a bidirectional derivation may require adding annotations, the related theorem is sometimes called annotatability instead of completeness. A separate criterion considers the quantity and quality of the required annotations (Section 3.4).

Requiring that every type connective have at least one introduction rule and at least one elimination rule would be too strict: the empty type $\bot$ should not have an introduction rule, and the top type $\top$ and the unit type do not require elimination rules. (We might choose to include an elimination rule for $\top$, but our criteria for bidirectional systems should not force this choice.)

3.3 Third Criterion: Size

Our third criterion refers to the number of typing rules. This is not always a reliable measure; by playing games with notation, one can inflate or deflate the number of typing rules. But the measure can be used effectively when comparing two systems in an inclusion relationship: a system of two rules R1 and R2 is clearly smaller than a system that has R1, R2, and a third rule R3. Smaller systems tend to be easier to work with: for example, in a meta-theoretic proof that considers cases of the rule concluding a given derivation, each new rule leads to an additional proof case.

3.4 Fourth Criterion: Annotation Character

A bidirectional system that required an annotation on *every* subterm would satisfy completeness. To rule out such a system, we need another criterion. We call it *annotation character*, an umbrella term for attributes that are sometimes in conflict:

(i) Annotations should be *lightweight*: they should constitute a small portion of the program text.
(ii) Annotations should be *predictable*: programmers should be able to easily determine whether a subterm needs an annotation. That is, there should be a clear *annotation discipline*.
(iii) Annotations should be *stable*: a small change to a program should have a small effect on annotation requirements.
(iv) Annotations should be *legible*: the form of annotation should be easy to understand.

Attribute (i) is the easiest to measure, but that does not make it the most important.

Attribute (ii) is harder to measure, because it depends on the definition of "easily" (alternatively, on the definition of "clear"). In the absence of empirical studies comparing bidirectional type systems with different annotation disciplines, we can form some hypotheses:

(1) A discipline that needs only *local* information is preferable to one that needs global information. That is, we want to know whether a subterm needs annotation from the "neighbourhood" of that subterm, not by looking at the whole program. The Pfenning recipe (Section 4) leads to an annotation discipline in which the syntactic forms of the subterm (e.g., that it is a pair) and the subterm's immediate context (its parent in the syntax tree) suffice to determine whether the subterm needs an annotation. (The subterm alone is not enough: a subterm that cannot synthesize does not require an annotation if it is being checked, and whether the subterm is checked depends on its position in the program.)

    In an annotation discipline that needs only local information, a change to one part of a program cannot affect the need for annotations in distant parts of the program. Hence, such a discipline is *stable* (attribute (iii)).
(2) A discipline that requires all *non-obvious* type annotations, and no *obvious* type annotations, is preferable. Unfortunately, it is not easy to agree on which type annotations are obvious.

(3) A discipline that needs obvious type annotations in certain situations is acceptable, if those situations are rare. For example, we might tolerate annotations on every `while` loop in SML, because SML programmers rarely use `while`.

These hypotheses can be found, in a different form, in earlier work. The term *local type inference* [Pierce and Turner 2000] implies that bidirectional typing should focus on (or even be restricted to) local information, suggesting that annotation disciplines should not need global information. Hypotheses (2) and (3) correspond to this explanation, from the same paper:

> The job of a partial type inference algorithm should be to eliminate especially those type annotations that are both *common* and *silly*—i.e., those that can be neither justified on the basis of their value as checked documentation nor ignored, because they are rare. [Pierce and Turner 2000, Section 1.1; emphasis in original]

4 A BIDIRECTIONAL RECIPE

Fortunately, we have a methodical approach that produces bidirectional systems that satisfy many of the above criteria.

We call this approach[2] the *Pfenning recipe* [Dunfield and Pfenning 2004]. It yields a set of bidirectional typing rules that is mode-correct (our first criterion), annotatable (our second criterion), small (our third criterion) and whose annotation discipline is moderately *lightweight* and highly *predictable* (attributes of our fourth criterion). Some disadvantages of the recipe—particularly in terms of a lightweight annotation discipline—can be mitigated, in exchange for a larger set of rules. Thus, the foremost virtue of the recipe is that it gives a *starting point* for a practical system: it tells you what the ground floor of the building (type system) should look like, allowing a taller building if desired. Even if the recipe is not followed completely, some of its steps are useful in designing bidirectional systems; we use this opportunity to explain those steps. Another virtue of the recipe is that it guarantees a subformula property, which can be of practical as well as theoretical interest; see Section 7.1.

Nonetheless, this recipe is not the only way to design a bidirectional type system. This fact is demonstrated by the wide variety of systems that do not exactly follow the recipe, and by the existence of systems that diverge radically from it (Section 6.5).

The most interesting component of the recipe is what it says to do for introduction and elimination rules (Section 4.1). We then explain how to construct the rules for annotations, variables, and subsumption (Sections 4.2–4.4). Section 4.5 examines whether the recipe meets the criteria, and Section 4.6 considers principal typing properties in the presence of stationary rules.

4.1 Introduction and Elimination Rules

This part of the recipe pertains to each rule that is an introduction rule (introducing a type connective that occurs in the conclusion) or an elimination rule (eliminating a type connective that occurs in a premise).

Step 1: Find the principal judgment. The *principal connective* of an introduction (resp. elimination) rule is the connective that is being introduced (resp. eliminated). The *principal judgment* of a rule is the judgment containing the principal connective; in an introduction rule, the principal judgment

[2]Our presentation of the recipe is intended as a more detailed explanation of the original, with one exception. The exception is that, instead of (implicitly) advocating that a `case` expression (sum elimination) have a single rule with a checking conclusion, our version of the recipe allows for *two* rules: one checking, one synthesizing.

is usually[3] the conclusion, and in an elimination rule, the principal judgment is usually the first premise.

Step 2: Bidirectionalize the principal judgment. This is the magic ingredient! If the rule is an *introduction* rule, then make the principal judgment *checking*. If the rule is an *elimination* rule, then make the principal judgment *synthesizing*.

Step 3: Bidirectionalize the other judgments. The direction of the principal judgment provides guidance for the other directions. The first question is, in what order should we bidirectionalize the other judgments? If the principal judgment is the conclusion (often true for introduction rules), then the only judgments left are the premises, but if the principal judgment is a premise (probably the first premise), we have a choice. The better choice seems to be to bidirectionalize the premises, then the conclusion: this maximizes the chance of having enough information to synthesize the type of the conclusion.

The second question is, which directions should we choose? First, they should be mode-correct. Second, we are guided by our criteria of annotatability and annotation character: the bidirectional system should be complete with respect to "ground truth" (roughly, the given type assignment system) without needing too many annotations.

Therefore, we should *utilize known information*: If we already know the type of a judgment, then the judgment should be checking. Thus, if the conclusion checks against a type A and a premise also has type A, then the premise should be checking. Similarly, if an earlier premise synthesizes B, then a later premise having type B should be checking. Choosing to synthesize would ignore known information, restricting the system's power for no reason.

For example, to get a bidirectional version of a product introduction rule,

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \times\text{I},$$

we (1) identify the conclusion, which contains the connective $\times$, as the principal judgment (since it contains the connective $\times$); (2) make the conclusion a checking judgment; and (3) observe that, since A_1 and A_2 are known, the premises can be checking judgments (utilizing known information):

$$\frac{\Gamma \vdash e_1 \Leftarrow A_1 \qquad \Gamma \vdash e_2 \Leftarrow A_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Leftarrow A_1 \times A_2} \text{Chk}\times\text{I}.$$

Step 3, continued. Unfortunately, what should count as "ground truth" is not always clear. It may not be exactly the set of type assignment rules. In our experience, most type assignment rules can be taken as ground truth and transformed by the recipe with satisfactory results, but certain type assignment rules must be viewed more critically. Consider a standard sum-elimination rule, on a pattern-matching term $\mathsf{case}(e,\ \mathsf{inj}_1\, x_1.\, e_1,\ \mathsf{inj}_2\, x_2.\, e_2)$ where x_1 is bound in e_1 and x_2 is bound in

[3] In an introduction rule in which the introduced connective is only available in a lexically scoped subterm, the principal connective is added to the context in a premise. In such a rule, the premise with the extended context—not the conclusion—is the principal judgment. An example is a rule typing a product introduction for a "let-pair" construct: the third premise is the principal judgment, because it contains the introduced connective $\times$:

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2 \qquad \Gamma, x : A_1 \times A_2 \vdash e : B}{\Gamma \vdash \mathsf{let}\ x = \langle e_1, e_2 \rangle\ \mathsf{in}\ e : B}.$$

e_2:

$$\frac{\Gamma \vdash e : (A_1 + A_2) \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash e_1 : B \\ \Gamma, x_2 : A_2 \vdash e_2 : B\end{array}}{\Gamma \vdash \mathsf{case}(e,\ \mathsf{inj}_1\, x_1.\, e_1,\ \mathsf{inj}_2\, x_2.\, e_2) : B}\ {+}\mathrm{Elim}$$

If we erase the terms from this rule and rewrite + as $\vee$, then we get a logical or-elimination rule:

$$\frac{\Gamma \vdash (A_1 \vee A_2) \qquad \begin{array}{c}\Gamma, A_1 \vdash B \\ \Gamma, A_2 \vdash B\end{array}}{\Gamma \vdash B}\ \vee\mathrm{Elim}$$

This rule is an *instance* of a fundamental logical principle: reasoning by cases. In a mathematical proof, reasoning by cases works the same regardless of the goal we want to prove: whether we want to conclude "formula B is true" or "kind κ is well-formed" or "real number x is negative" or "machine H halts," our proof can consider the two cases (A_1 is true; A_2 is true) of the disjunctive formula $A_1 \vee A_2$. So the ground principle from which $\vee$Elim is instantiated allows for a conclusion $\mathcal{J}$ that has *any* form:

$$\frac{\Gamma \vdash (A_1 \vee A_2) \qquad \begin{array}{c}\Gamma, A_1 \vdash \mathcal{J} \\ \Gamma, A_2 \vdash \mathcal{J}\end{array}}{\Gamma \vdash \mathcal{J}}\ \vee\mathrm{Elim\text{-}general}$$

Instantiating $\mathcal{J}$ to "formula B is true" results in $\vee$Elim, while instantiating $\mathcal{J}$ to "machine H halts" would result in

$$\frac{\Gamma \vdash (A_1 \vee A_2) \qquad \begin{array}{c}\Gamma, A_1 \vdash H \text{ halts} \\ \Gamma, A_2 \vdash H \text{ halts}\end{array}}{\Gamma \vdash H \text{ halts}}\ \vee\mathrm{Elim\text{-}halting\text{-}goal}$$

If we consider $\vee$Elim-general to be the basis of +Elim, then we see that $\vee$Elim-general should give rise to *two* bidirectional rules, because a bidirectional system has two judgment forms:

$$\frac{\Gamma \vdash e \Rightarrow (A_1 + A_2) \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash e_1 \Leftarrow B \\ \Gamma, x_2 : A_2 \vdash e_2 \Leftarrow B\end{array}}{\Gamma \vdash \mathsf{case}(e,\ \mathsf{inj}_1\, x_1.\, e_1,\ \mathsf{inj}_2\, x_2.\, e_2) \Leftarrow B} \qquad \frac{\Gamma \vdash e \Rightarrow (A_1 + A_2) \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash e_1 \Rightarrow B \\ \Gamma, x_2 : A_2 \vdash e_2 \Rightarrow B\end{array}}{\Gamma \vdash \mathsf{case}(e,\ \mathsf{inj}_1\, x_1.\, e_1,\ \mathsf{inj}_2\, x_2.\, e_2) \Rightarrow B}$$

The original recipe [Dunfield and Pfenning 2004] resulted in only one typing rule for case, the checking rule; if you can only have one rule, the checking rule is more general and nicely matches the checking premise of $\rightarrow$Intro. Note that whether we have one rule or two, we are still following Steps 1 and 2 of the recipe: in both versions of the rule, the principal judgment synthesizes. The complication is that making the principal judgment synthesize does not determine the directions of the conclusion and the other premises.

Similar considerations arise in typing a let-expression, with one additional wrinkle: a let-expression is not an elimination form. Thus, we seem to have no guidance at all, beyond mode-correctness. Whether the conclusion checks or synthesizes, we do not yet know the type A of the let-bound expression, so the first premise must synthesize (similar to a case expression):

$$\frac{\Gamma \vdash e \Rightarrow A \qquad \Gamma, x : A \vdash e' : B}{\Gamma \vdash \mathsf{let}\, x = e\, \mathsf{in}\, e' : B}\ .$$

Now we have the same choice as in case elimination: If a let-expression represents a general reasoning principle, then we may want two rules, one where the second premise and conclusion are

checking and one where they synthesize. If we prioritize a small number of rules, then the checking rule is more general than the synthesis rule alone. (It is actually possible to design a system where e is checked; we explore that territory in Section 6.5.)

Dunfield and Pfenning [2004] claimed that annotations were needed only on redexes, but that claim was false. The claim holds for elimination forms that do not bind variables, but fails with elimination forms that *do* bind variables, like case expressions. If we, instead, have both rules for case expressions, using a variable-binding elimination to derive the principal judgment of a variable-binding elimination does not incur an annotation (so, unlike in the original recipe, $\mathsf{case}(\mathsf{case}(x, \cdots), \cdots)$ can be typed if all arms of the inner case can synthesize). However, some combinations of binding and non-binding elimination do incur annotations despite having no redexes.[4] We discuss the details in Section 4.5.4.

4.2 Annotation

Assume we have, in the given type assignment system, the following annotation rule (if we had no concern for ease of implementation, then we might not need such a rule):

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A}.$$

This is not an introduction or elimination rule, so it has no principal connective and thus no principal judgment. However, the conclusion feels closer to being the principal judgment, because—while the type A is not tied to any particular connective—the type must match the type appearing in the term $(e : A)$. In contrast, the premise $\Gamma \vdash e : A$ imposes no constraints at all.

Thus, we will start by bidirectionalizing the conclusion. Again, the rule is neither an introduction nor an elimination so we cannot pick a direction based on that. Instead, we follow the principle in Step 3 (above): we should try to use all available knowledge. We know the type A, because it appears in the term $(e : A)$, so synthesizing A in the conclusion utilizes this knowledge.

Now we turn to the premise, $\Gamma \vdash e : A$. Since A is known, we should utilize it by checking e against A, resulting in the rule

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}.$$

It does not really matter whether we start with the conclusion or the premise. If we start with the premise, we notice that A is known from $(e : A)$ and make the premise checking; then we notice that A is known in the conclusion.

4.3 Variables

A typing rule for variables is neither an introduction nor elimination rule. Instead, it corresponds to a fundamental principle of deduction: the use of an assumption. Instead of interpreting the assumption $x : A$ to mean that x has (is assigned) type A, we interpret $x : A$ as $x \Rightarrow A$: We assume that x synthesizes A, and so the variable rule is synthesizing:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \mathsf{Var}.$$

It might be more clear for the form of assumptions in Γ to be $x \Rightarrow A$ rather than $x : A$, making clear that this rule is simply the use of an assumption, but the form $x : A$ is standard.

[4] Admittedly, the truth of the claim depends on the meaning of "redex"; if we include various commuting conversions in our notion of reduction, then the claim becomes accurate.

Alternatively, we can view the direction of Var as being determined by what the recipe does for →I, producing →I⇐ (Figure 1): Since λ is checked, its argument type is known and can be added to Γ. Assumptions $x \Leftarrow A$ arise in a reversed bidirectional system (Section 6.5).

4.4 Change of Direction (Subsumption)

To see why this part of the recipe is needed, consider the following type assignment derivation.

$$\frac{}{x : A \vdash x : A}\,.$$

Our bidirectional Var rule can synthesize a type for x, but cannot derive a checking judgment. So, we cannot synthesize a type for f applied to x, even though both their types are available in the typing context:

$$\dfrac{\dfrac{(f : A \to B) \in (f : A \to B,\ x : A)}{f : A \to B,\ x : A \vdash f \Rightarrow A \to B}\ \text{Var} \qquad f : A \to B,\ x : A \not\vdash x \Leftarrow A}{f : A \to B,\ x : A \not\vdash f\ x \Rightarrow B}\ \text{Syn}{\to}\text{Elim}$$

We know, from the type of f, that x needs to have type A. The var rule can tell us that x also synthesizes that type. So, we need a rule that verifies that a fact (x synthesizes type A) is consistent with a requirement (x needs to check against type A). One version of that rule would be

$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}\ \text{changedir-0}$$

However, the recipe prefers an equivalent rule:

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \text{changedir-1}$$

Since $A = B$ is only equality, rule changedir-1 has exactly the same power as changedir-0. Rules changedir-0 and changedir-1 can be implemented in exactly the same way, but the structure of changedir-1 is closer to that implementation: first, make a recursive call to synthesize A; second, check that A is equal to the given type B. (In logic programming terminology, in changedir-1 the premise $\Gamma \vdash e \Rightarrow A$ has *output freeness*: the output A is unconstrained, with the constraint imposed later by $A = B$. In changedir-0, the output A is constrained to be exactly the type from the conclusion.)

The more significant advantage of changedir-1 is that it can be easily extended to support subtyping. To turn changedir-1 into a subsumption rule, we only have to replace "=" with a subtyping relation "<:":

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \text{Sub}$$

(In a sense, changedir-1 is already a subsumption rule: equality is reflexive and transitive, so it is a sensible—if extremely limited—subtyping relation. If we choose equality as the definition of <:, then the rule Sub is exactly the rule changedir-1.)

Since B is an input, and A is an output of the first premise, both A and B are known: the subtyping judgment can be implemented with both types as input, with no need to "guess" the subtype or supertype.

Subtyping can lead to concerns not addressed solely by the presence of Sub; we discuss these concerns in Section 4.6.

The rule Sub is not syntax-directed: its subject may be any form of expression. Because its premise is synthesizing, however, we have some guidance about when to apply it: when there is a rule whose conclusion can synthesize a type for that form of expression. In some bidirectional type systems, such as Davies and Pfenning [2000], no expression form has a rule with a synthesizing conclusion and a rule with a checked conclusion. In such systems, we can classify the expression forms themselves as checked or synthesizing, and use Sub exactly when checking a synthesizing form. If we have, say, two rules for case expressions, then the situation is more complex. It is generally best to apply Sub as late as possible (that is, toward the leaves of the derivation), because that preserves the information provided by the type being checked against.

4.5 Assessing the Recipe

When we design a bidirectional system according to the recipe, which criteria are satisfied?

4.5.1 First Criterion: Mode-correctness. All rules are mode-correct:

- The variable, annotation and subsumption rules are "pre-cooked" and it is straightforward to verify they are mode-correct.
- In the introduction and elimination rules, the judgment directions are chosen to be mode-correct.

4.5.2 Second Criterion: Annotatability. We write $e' \sqsupseteq e$ to mean that e' is a "more annotated" version of e. For example, $(x : A) \sqsupseteq x$ and $x \sqsupseteq x$. Annotatability says that, if $\Gamma \vdash e : A$ (in the type assignment system), then (1) there exists $e' \sqsupseteq e$ such that $\Gamma \vdash e' \Leftarrow A$, and (2) there exists $e'' \sqsupseteq e$ such that $\Gamma \vdash e'' \Rightarrow A$. We prove this by induction on the type assignment derivation of $\Gamma \vdash e : A$, considering cases of the type assignment rule concluding that derivation.

Each type assignment rule has a single corresponding bidirectional rule. If the conclusion of that bidirectional rule is *checking*, then proving part (1) is completely straightforward: Applying the induction hypothesis to the derivation of each premise (of the type assignment rule) yields a set of annotated subterms; combining these annotated subterms gives us our e', which is typed by the bidirectional rule. This approach also works if we are proving part (2) and the conclusion of the bidirectional rule is *synthesizing*.

Going "into the wind"—proving part (1) with a synthesis rule, or part (2) with a checking rule—needs only a little more work:

- If the conclusion of the bidirectional rule corresponding to the type assignment rule is *synthesis* and we want to prove part (1), then we can show part (2) as above to derive
$$\Gamma \vdash e' \Rightarrow A.$$
Now, we want to find e'' such that $\Gamma \vdash e'' \Leftarrow A$. Assuming subtyping is reflexive (a condition satisfied even by weak subtyping relations, including equality), we can derive $A <: A$ and use subsumption, giving $\Gamma \vdash e' \Leftarrow A$. In this case, e' and e'' are the same.
- If the conclusion of the bidirectional rule corresponding to the type assignment rule is *checking* and we want to prove part (2), then we can show part (1):
$$\Gamma \vdash e'' \Leftarrow A.$$
Now we want to find e' such that $\Gamma \vdash e' \Rightarrow A$. We cannot reuse e'', because $\Gamma \vdash e'' \Leftarrow A$ was derived using a checking rule; since the recipe produces only one corresponding bidirectional rule, we have no rule that can derive $\Gamma \vdash e'' \Rightarrow A$. We must add an annotation:
$$e' = (e'' : A).$$
The last step is to use our annotation rule, deriving $\Gamma \vdash (e' : A) \Rightarrow A$.

4.5.3 Third Criterion: Size. For each type assignment rule, the recipe produces exactly one rule. (If we take the view that a type assignment rule like ∨Elim represents a set of rules, as we can with ∨Elim-general in Section 4.1, then we produce one bidirectional rule for each rule in the set.) Producing less than one rule is unacceptable, because it would be incomplete with respect to the type assignment system.

If the original type assignment system did not have a subsumption rule, then the recipe adds one, but this is unavoidable (see the example in Section 4.4). (An alternative would be to duplicate rules and vary the direction of their premises, but for most type systems, that would lead to more than one additional rule.)

Similarly, the annotation rule is needed to enable a term whose rule has a checking conclusion to be used in a synthesizing position. For example, we cannot type the term $(\lambda x.\, e)e'$, because the first premise of Syn→Elim is synthesizing and the conclusion of Chk→Intro is checking. We need the annotation rule to allow us to type the annotated term $((\lambda x.\, e) : A \to B)e'$.

Thus, it is not only impossible to remove a single rule, but there is no alternative approach that can, in general, produce a smaller set of rules.

4.5.4 Fourth Criterion: Annotation Character. Our notion of annotation character (Section 3.4) posits that annotations should be (i) lightweight, (ii)–(iii) predictable and stable (with a clear *annotation discipline*), and (iv) legible.

We also argued that a good annotation discipline should require only local information. On this point, the recipe does well: an annotation is required on a subterm *if and only if* an introduction form meets an elimination form.

To see why, let us consider how the recipe treats introduction and elimination forms. Introduction rules type introduction forms, like λ; elimination rules type elimination forms, like function application. Following the recipe, the principal judgment in an elimination form is synthesizing, so eliminating a variable never requires an annotation. For example, $f\ x$ needs no annotation, because f synthesizes. Nor does $(f\ x)\ y$ need an annotation, because $f\ x$ synthesizes. Variable-binding elimination forms, like case, can also be nested without annotation: the type of the outer case is propagated to the inner case. For example, the type int is propagated from the conclusion to the inner case:

$$\frac{\begin{array}{l}\Gamma \vdash y \Rightarrow (\mathsf{bool} + \mathsf{bool}) + \mathsf{int} \\ \Gamma, x_1 : (\mathsf{bool} + \mathsf{bool}) \vdash \mathsf{case}(x_1,\ \mathsf{inj}_1\, x_{11}.\, 0,\ \mathsf{inj}_2\, x_{22}.\, 1) \Leftarrow \mathsf{int} \\ \Gamma, x_2 : \mathsf{int} \vdash x_2 \Leftarrow \mathsf{int}\end{array}}{\Gamma \vdash \Big(\mathsf{case}(y,\ \mathsf{inj}_1\, x_1.\, \mathsf{case}(x_1,\ \mathsf{inj}_1\, x_{11}.\, 0,\ \mathsf{inj}_2\, x_{22}.\, 1),\ \mathsf{inj}_2\, x_2.\, x_2)\Big) \Leftarrow \mathsf{int}}\ {+}\mathsf{Elim}$$

However, we need an annotation at the boundary between introduction and elimination: in $(\lambda x.\, e_1)e_2$, the introduction form λ meets the elimination form $(\cdots)e_2$. Since the first premise of Syn→Elim is synthesizing, and the conclusion of Chk→Intro is checking, an annotation is needed around $(\lambda x.\, e_1)$.

Similarly, in $\mathsf{case}(\mathsf{inj}_1\, e,\ \cdots)$, the introduction form inj meets the elimination case, so $\mathsf{inj}_1\, e$ needs an annotation.

In those two examples, we introduced and *immediately* eliminated the same type (→ in the first example and + in the second). An introduction that is not immediate also requires an annotation:

$$\Big(\mathsf{case}(y,\ \mathsf{inj}_1\, x_1.\, (\lambda z_1.\, e_1),\ \mathsf{inj}_2\, x_2.\, (\lambda z_2.\, e_2))\Big)z.$$

Because the case expression appears as the function part of the application $(\cdots)z$, it needs to synthesize a type (so we can derive the first premise of Syn→Elim). But the case arms $\lambda z_1.\, e_1$ and $\lambda z_2.\, e_2$, being introduction forms, do not synthesize. Therefore, we need a type annotation around

the case, or—more verbosely—two annotations, one around each λ.[5] Note that if we push the application to z into each arm, we get a term where eliminations immediately follow introductions (and, therefore, need annotations):

$$\begin{aligned}&\mathsf{case}(y, \mathrm{inj}_1\, x_1.\, (\lambda z_1.\, e_1)z,\\ &\qquad\quad \mathrm{inj}_2\, x_2.\, (\lambda z_2.\, e_2)z)\end{aligned}$$

4.6 Subtyping and Principal Types

In this section, we examine subtyping in the bidirectional recipe in more depth. We focus on intersection types, which have a nontrivial but relatively economical subtyping relation, allowing us to illustrate the issues in a relatively economical way. Intersection types are a form of polymorphism that, unlike parametric polymorphism (Section 5), does not involve quantifiers or variable instantiation. If a value has type $A_1 \wedge A_2$, then it has type A_1 and type A_2; if a value has type $\forall\alpha.\, A$, then it has the type $[B/\alpha]A$ for any type B. An intersection type can be seen as a polymorphic type ranging over only two possibilities; the polymorphic type $\forall\alpha.\, A$ can be seen as an infinite intersection. As usual, we assume that subtyping is reflexive and transitive.

A typing rule is *stationary* [Leivant 1986, p. 55] if the subject of the premise(s) is the same as the subject of the conclusion—in contrast to (perhaps more familiar) rules where each premise types a proper subterm of the subject of the conclusion. In the stationary rules we consider, the subject is *any* expression e. Thus, these rules are not syntax-directed; if the conclusion of a rule types e, the rule is potentially applicable at every step of type checking.

Our rule Sub is stationary, as are the following rules for intersection types $A \wedge B$ and (implicit) parametric polymorphism $\forall\alpha.\, A$: the premises type the same e as the conclusion:

$$\frac{\Gamma \vdash e : A_1 \qquad \Gamma \vdash e : A_2}{\Gamma \vdash e : A_1 \wedge A_2}\ \wedge\text{Intro} \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\ \wedge\text{Elim1} \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\ \wedge\text{Elim2}$$

$$\frac{\Gamma, \alpha\ \mathsf{type} \vdash e : A}{\Gamma \vdash e : \forall\alpha.\, A} \qquad \frac{\Gamma \vdash e : \forall\alpha.\, A \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash e : [B/\alpha]A}$$

With typing rules like these, it makes sense for subtyping to allow $A_1 \wedge A_2 <: A_1$: the presence of $\wedge$Elim1 means that every term having type $A_1 \wedge A_2$ also has type A_1. Similarly, $A_1 \wedge A_2 <: A_2$, because of $\wedge$Elim2. Observe that both subtyping and Gentzen's rule notation are forms of implication $\supset$: by treating types as propositions, $A <: B$ becomes $A \supset B$; a rule is read as *premises* $\supset$ *conclusion*. So, we can translate $\wedge$Elim1:

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\ \wedge\text{Elim1}\ \text{ becomes } A_1 \wedge A_2 \supset A_1 \text{ becomes } \frac{}{A_1 \wedge A_2 <: A_1}.$$

The rule $\wedge$Elim2 can be treated similarly.

However, the rule $\wedge$Intro cannot be translated in this way: the two premises mean that the rule cannot be read as "$\cdots$ implies $\cdots$," but only as "$\cdots$ and $\cdots$ together imply $\cdots$." A subtyping judgment $A <: B$ can be read as the sequent $A \vdash B$, but it is a limited sequent calculus: in addition to allowing only one succedent B (which is a common restriction in sequent calculi), subtyping allows only one antecedent A. The subtyping rule we would like to construct would need *two*

[5] In the original recipe, the only option would be an annotation around the case. Since the original recipe had only one rule for case, which had a checking conclusion, annotating the individual arms would have no effect: Typing could not "pass through" the case to notice the annotations.

antecedents, A_1 and A_2, which do not fit:

$$\frac{\Gamma \vdash e : A_1 \qquad \Gamma \vdash e : A_2}{\Gamma \vdash e : A_1 \wedge A_2}\ \wedge\text{Intro}\ \text{ becomes } A_1 \textit{ and } A_2 \supset A_1 \wedge A_2 \text{ becomes } \frac{}{A_1, A_2 <: A_1 \wedge A_2}.$$

The subtyping relation induced by "translating" *only* the stationary typing rules is weaker (smaller) than we might desire: it yields *shallow* subtyping. For example, to derive the following through subsumption, we would need $(A \to (B_1 \wedge B_2)) <: (A \to B_2)$, because the type of g does not literally match $A \to B_2$. But the rules for $\to$ are not stationary, letting us forget to add a subtyping rule for $\to$: in the subderivation typing g, we need g to have type $A \to B_2$ but it has only the type $A \to (B_1 \wedge B_2)$:

$$\frac{\ldots \vdash f : (A \to B_2) \to C \qquad \cdots \not\vdash g : A \to B_2}{f : (A \to B_2) \to C,\ g : (A \to (B_1 \wedge B_2)) \not\vdash f\ g : C}\ \to\text{Elim}$$

Note that if we η-expand g to $\lambda x.\, g\ x$, the term can be typed with only shallow subtyping:

$$\dfrac{\ldots \vdash f : (A \to B_2) \to C \qquad \dfrac{\dfrac{\dfrac{\ldots, x : A \vdash g : A \to (B_1 \wedge B_2) \qquad \ldots, x : A \vdash x : A}{\ldots, x : A \vdash g\ x : B_1 \wedge B_2}\ \to\text{Elim}}{\ldots, x : A \vdash g\ x : B_2}\ \wedge\text{Elim2}}{\ldots \vdash (\lambda x.\, g\ x) : A \to B_2}\ \to\text{Intro}}{f : (A \to B_2) \to C,\ g : (A \to (B_1 \wedge B_2)) \vdash f\ (\lambda x.\, g\ x) : C}\ \to\text{Elim}$$

The technique of η-expanding to simulate deep subtyping, e.g., for intersection and union types [Dunfield 2014], is (as far as we know) due to Barendregt et al. [1983]; they showed that putting a $\beta\eta$-expansion rule in a type system made subsumption admissible (see their Lemma 4.2).

Whether we choose a subtyping relation that is shallow or deep, we can "optimize" a type assignment system by dropping stationary rules that are encompassed by subsumption. For example, $\wedge$Elim1 and $\wedge$Elim2 are admissible by using Sub with the appropriate subtyping rules. This engineering optimization is not uniform: Since $\wedge$Intro cannot be translated, we end up with a type system that has an introduction rule for $\wedge$, but buries the elimination rules inside subtyping.

Fortunately (if we dislike non-uniform optimizations), in the bidirectional version of intersection typing, the bidirectional versions of $\wedge$Elim1 and $\wedge$Elim2 are *not* admissible:

$$\frac{\Gamma \vdash e \Rightarrow A_1 \wedge A_2}{\Gamma \vdash e \Rightarrow A_1}\ \wedge\text{Elim1} \qquad \frac{\Gamma \vdash e \Rightarrow A_1 \wedge A_2}{\Gamma \vdash e \Rightarrow A_2}\ \wedge\text{Elim2}$$

These rules have a synthesizing conclusion, which means that Sub cannot simulate them.

It is worth noting, however, that these rules are mode-correct, but not syntax-directed. With stationary synthesis rules, a term can synthesize many possible types—in this example, e can synthesize any of $A_1 \wedge A_2$, A_1, and A_2. This is not inherently a bad thing—after all, the whole point of an intersection type discipline is to allow ascribing many types to the same term. However, managing this nondeterminism requires some care both in the design of the type system, and its implementation.

4.6.1 Principal Synthesis. A type assignment system has *principal types* if there always exists a "best type"—a type that represents all possible types. In systems with subtyping, the best type is the smallest type, so the principal typing property says that if e is well-typed, there is a principal type A such that e has type A and $A <: B$, for all types B of e:

Definition 1 (Principal Types). A type assignment system has *principal types* if, for all terms e well-typed under Γ, there exists a type A such that $\Gamma \vdash e : A$ and for all B such that $\Gamma \vdash e : B$, we have $A <: B$.

A type inference system has *principal inference* if it implements a type assignment system that has principal types (Definition 1) and always infers the principal type. The definition says that if we can infer A, then A is principal.

Definition 2 (Principal Inference). A type inference system $\Gamma \vdash e$ infer A has *principal inference* if, whenever $\Gamma \vdash e$ infer A, for all B such that $\Gamma \vdash e : B$, we have $A <: B$.

That is, every type B that can be assigned to e is a supertype of the inferred type A. (In Damas–Hindley–Milner inference [Hindley 1969; Damas and Milner 1982], this property is stated for type schemes; an inferred type scheme for e is principal when it can be instantiated to every (monomorphic) type of e.)

We can adapt Definition 2 to bidirectional type systems by using checking, rather than type assignment, to define "all possible types."

Definition 3 (Principal Synthesis). A bidirectional type system has *principal synthesis* if, given $\Gamma \vdash e \Rightarrow A$, for all B such that $\Gamma \vdash e \Leftarrow B$, we have $A <: B$.

Principal synthesis is sometimes easy: If a bidirectional system has uniqueness of both synthesis and checking, that is, if $\Gamma \vdash e \Rightarrow A$ and $\Gamma \vdash e \Rightarrow B$, then $A = B$ (and, respectively, for checking), then $A <: B$, because $A = B$. (In a system in which synthesis always produces the same type A_1, and checking always works against only a single type A_2 for a given term, it had better be the case that $A_1 = A_2$!)

For many sophisticated type systems, principal synthesis either does not hold or requires some extra design work. In common formulations of intersection types, such as the one in Section 4.6, synthesis and checking are not unique. For example, if we synthesize $x : (A_1 \wedge A_2) \vdash x \Rightarrow B$, then the point of the intersection type is to allow either the behaviour A_1 or A_2, so it must be possible to derive both,

$$x : (A_1 \wedge A_2) \vdash x \Rightarrow A_1 \text{ and } x : (A_1 \wedge A_2) \vdash x \Rightarrow A_2,$$

as well as $x : (A_1 \wedge A_2) \vdash x \Rightarrow A_1 \wedge A_2$. Saying that synthesis should only produce $A_1 \wedge A_2$ is not compatible with the recipe: If x is a function, then the rule $\rightarrow$Elim needs to synthesize a function type; we are not checking x against a known type, so we cannot rely on subtyping (which has only a checking conclusion) to eliminate the intersection.

Non-uniqueness means that a straightforward implementation of the rules must do backtracking search, trying all three types A_1, A_2 and $A_1 \wedge A_2$, even when the choice is irrelevant. Some backtracking is difficult to avoid with intersection types, but naively trying all three choices in all circumstances is excessive.

To address this, the first author's implementation of a bidirectional intersection and union type system split the synthesis judgment into two: one judgment that "maintains principality," and one that "wants an ordinary type." The "maintains principality" judgment lacked rules like $\wedge$Elim1 and $\wedge$Elim2; the "ordinary type" judgment included such rules. The choice of synthesis judgment depended on the rule. The premise of a let rule, synthesizing a type for the let-bound expression, used the "maintains principality" judgment to ensure that the variable typing added in the body of the let was principal. So, for example, if the let-bound expression was simply a variable x of type $A_1 \wedge A_2$, the typing in the body would also have $A_1 \wedge A_2$, with no backtracking between choices of $\wedge$Elim1 and $\wedge$Elim2. However, the premise of $\rightarrow$Elim used the "wants an ordinary type" judgment, because we may need to apply rules like $\wedge$Elim1 to expose the $\rightarrow$ connective. See Dunfield [2007, Section 6.7.1 on pp. 186–187].

Davies [2005, Section 2.10.2] includes a bidirectional typing (*sort checking*) system with a principal synthesis property. It appears that Davies asserts the property (page 41) without formally stating or proving it, but from our reading of the rules on page 42, it holds as follows: Principal synthesis is achieved through an auxiliary judgment that, when applying a function of type $(R_1 \to S_1) \wedge \ldots \wedge (R_n \to S_n)$, gathers all the components $R_i \to S_i$ such that the function argument checks against R_i, and synthesizes the intersection of all such S_i. (Davies [2005, Section 3.10] also discusses a principal sorts property in a non-bidirectional type inference setting, but this is less relevant to our survey.)

5 POLYMORPHISM

Damas–Milner type inference [Damas and Milner 1982] allows only prefix polymorphism: the quantifiers in a type must appear on the outside, allowing $\forall\alpha.\ (\forall\beta.\ \alpha \to \beta \to \alpha)$ but not $\forall\alpha.\ \alpha \to (\forall\beta.\ \beta \to \alpha)$ and $\forall\beta.\ (\forall\alpha.\ \alpha \to \alpha) \to \beta \to \beta$. This restriction is called *prefix* or *prenex* polymorphism. In their terminology, types contain no quantifiers at all; only type *schemes* can have quantifiers (on the outside). Polymorphism can be introduced only on `let` expressions.

If programs must indicate both where and how to introduce and eliminate polymorphism, then polymorphism is *fully explicit*. Explicit *introduction* of polymorphism can readily cope with less restrictive forms of polymorphism, such as *higher-rank* polymorphism, which allows quantifiers to be nested anywhere in a type (including to the left of arrows, as in the type $\forall\beta.\ (\forall\alpha.\ \alpha \to \alpha) \to \beta \to \beta$ mentioned above), and even *impredicative* polymorphism, which allows quantifiers to be instantiated with polymorphic types.

Adding fully explicit polymorphism to a bidirectional system is straightforward: Since the term is an input, both the introduction and elimination rules can use the information in the term. We start with System F rules for explicit polymorphism:

$$\frac{\Gamma, \alpha \vdash e : A}{\Gamma \vdash (\Lambda\alpha.\ e) : \forall\alpha.\ A} \qquad \frac{\Gamma \vdash e : \forall\alpha.\ A}{\Gamma \vdash e\,[B] : [B/\alpha]A}.$$

Following the recipe, we easily obtain bidirectional rules:

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A}{\Gamma \vdash (\Lambda\alpha.\ e) \Leftarrow \forall\alpha.\ A}\ \forall\mathsf{I}\Leftarrow \text{ (explicit)} \qquad \frac{\Gamma \vdash e \Rightarrow \forall\alpha.\ A}{\Gamma \vdash e\,[B] \Rightarrow [B/\alpha]A}\ \forall\mathsf{E}\Rightarrow \text{ (explicit)}.$$

Unfortunately, fully explicit polymorphism is often considered unusable, mostly because of the explicit eliminations: it is burdensome to say how to instantiate every quantifier. We can easily create an implicit version of $\forall\mathsf{I}\Leftarrow$, but the implicit version of $\forall\mathsf{E}\Rightarrow$ must guess the instantiation B:

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A}{\Gamma \vdash e \Leftarrow \forall\alpha.\ A}\ \forall\mathsf{I}\Leftarrow \text{ (implicit)} \qquad \frac{\Gamma \vdash e \Rightarrow \forall\alpha.\ A}{\Gamma \vdash e \Rightarrow [B/\alpha]A}\ \forall\mathsf{E}\Rightarrow \text{ (implicit)}.$$

These are stationary rules, so we can try to translate them into subtyping rules, as we translated the stationary rules for intersection types (Section 4.6). This does not quite work with the introduction rule: the assumption α does not fit into the conclusion, similar to how $\wedge$Intro led to a rule with two subtypes in the conclusion. However, we can translate the elimination rule, leading to a rule that captures the idea that an "at least as polymorphic as" relation is a form of subtyping:

$$\frac{}{\forall\alpha.\ A <: [B/\alpha]A}.$$

But this rule, like the implicit $\forall\mathsf{E}\Rightarrow$, requires guessing B. Restricting instantiation to monotypes (types containing no quantifiers) does not solve the problem: We still have to guess the monotype.

This *instantiation problem* has been tackled from several directions, which we examine in turn.

5.1 Local Type Inference

The first widely known paper on bidirectional typing [Pierce and Turner 2000] considered the problem in a setting with subtyping, and answered it by local constraint solving (hence the title *Local Type Inference*) around function applications. Subtyping leads to considering the upper and lower bounds of a type, rather than equations on types. Pierce and Turner restricted instantiation to prefix polymorphism, though their source language allowed impredicative polymorphism if the programmer explicitly instantiates the quantifier.

Their bidirectional rules [Pierce and Turner 2000, p. 18] look quite different from what our recipe might produce in their setting. Their function type allows multiple arguments and is combined with polymorphic quantification. To infer as much as possible, they have several function application rules. These rules handle both explicit and implicit instantiation, and include several combinations of judgment directions: S-App synthesizes a type for the function and checks the arguments (similar to our recipe), but S-App-InfSpec synthesizes types for the arguments as well. C-App-InfSpec is similar to S-App-InfSpec, but has a checking conclusion; the type checked against is used to guide instantiation of polymorphic arguments.

Two aspects of their rules prefigure later work on bidirectional typing. First, considering multiple arguments simultaneously is related to spine form (Section 8.3). Second, synthesizing arguments is related to Xie and Oliveira [2018], discussed in Section 6.5.1.

Hosoya and Pierce [1999] discuss the annotation burden of local type inference for several example programs.

One can argue that *all* type systems have subtyping, where some systems have only trivial subtyping (A is a subtype of B iff $A = B$). A more moderate perspective is that "most" type systems have subtyping: Even in prefix polymorphism, types that are "more polymorphic" can be considered subtypes. By the substitution principle of Liskov and Wing [1994], $\forall\alpha.\ \alpha \to \alpha$ should be a subtype of $\mathsf{unit} \to \mathsf{unit}$: any program context that expects an identity function on unit—of type $\mathsf{unit} \to \mathsf{unit}$—should be satisfied by a polymorphic identity function of type $\forall\alpha.\ \alpha \to \alpha$. (In many systems, including Damas–Milner, types cannot contain quantifiers—only type schemes can—but the perspective could be adapted to subtyping on type schemes, and is conceptually useful in any case.) In systems with higher-rank polymorphism, the perspective that polymorphism is a form of subtyping is salient: Since quantifiers can appear to the left of arrows, we may want to pass a "more polymorphic" argument to a function that expects something less polymorphic.

5.2 "Complete and Easy" Polymorphism

In this subsection, we explain the key elements of our technique [Dunfield and Krishnaswami 2013], discuss some typing rules, and describe its history in more detail.

5.2.1 Greedy Instantiation. The key idea taken from Cardelli [1993] was that, when eliminating a polymorphic type, we can treat the first plausible solution as *the* solution. For example, if we are calling a function of type $\forall\alpha.\alpha \to \alpha \to \alpha$ (assuming parametricity, such a function can only return one of its arguments) and pass as the first argument something of type Cat, then we instantiate α to Cat. This works perfectly well when the second argument has the same type, or when the second argument is of a subtype of Cat (e.g., Tabby), but fails when the second argument is of a larger type. If the first argument has type Tabby but the second argument has type Cat, then the second argument will fail to check against Tabby, since not all cats are tabbies.

In its original setting, this "greedy" method's vulnerability to argument order was rather unfortunate. In a setting of predicative higher-rank polymorphism without other forms of subtyping, however, it can work nicely. The "tabby-first problem" cannot arise, because the only way a type can become strictly smaller is by being strictly more polymorphic, and if the first argument is

polymorphic, then we would be instantiating α with a polymorphic type, which would violate predicativity.

5.2.2 Systems and Judgments. The paper focused on two bidirectional type systems: a *declarative* system, whose ∀-elimination rule "guesses" types, and an *algorithmic* system, which instead uses greedy instantiation.

Our declarative system followed a looser version of the Pfenning recipe: in addition to the rules produced by the recipe, the declarative system included synthesizing introduction rules for unit and →. A subsumption rule, DeclSub, used a declarative subtyping relation ≤ whose "∀-left" rule—the rule concluding $(\forall\alpha.\, A) \leq B$—guessed a monotype τ to use in the premise $[\tau/\alpha]A \leq B$.

We also incorporated an *application judgment*, written $\Psi \vdash e \bullet A \Rrightarrow C$ meaning that under the declarative context Ψ, if a function *of type* A is applied to an argument e, the entire application will have result type C. The rules for this judgment eliminate $\forall\alpha.\, A$ by substituting an unsolved type variable $\hat{\alpha}$ for α.

5.2.3 Ordered Typing Contexts. Rather than passing along a "bag of constraints," we can store the (solved and unsolved) type variables (written $\hat{\alpha}$, $\hat{\beta}$, etc.) in an *ordered* context. Issues of circularity and scope still need care, but the way to handling them is clarified: If $\hat{\alpha}$ appears to the left of $\hat{\beta}$ and we need to constrain them to be equal, then we must solve $\hat{\beta}$ to $\hat{\alpha}$, not the other way around.

In our algorithmic system, the three typing judgments—checking, synthesis and application—included an *output context* Δ. For example, if the *input* context $\Gamma = (\hat{\alpha}, x : \hat{\alpha})$, meaning that $\hat{\alpha}$ is an unsolved existential variable and x has type $\hat{\alpha}$, checking x against unit will solve $\hat{\alpha}$:

$$\hat{\alpha}, x : \hat{\alpha} \vdash x \Leftarrow \mathsf{unit} \dashv \hat{\alpha} = \mathsf{unit}, x : \hat{\alpha}.$$

More generally, in a derivation of $\Gamma \vdash \cdots \dashv \Delta$, the output context Δ *gains information*: any solutions present in Γ are also present in Δ, but unsolved $\hat{\alpha}$ in Γ may gain solutions in Δ.

We made this idea of information gain precise by defining *context extension*: whenever a judgment $\Gamma \vdash \cdots \dashv \Delta$ is derivable, the output context Δ is an extension of Γ, written $\Gamma \longrightarrow \Delta$. As in the $x \Leftarrow \mathsf{unit}$ example, information about existential type variables may increase in Δ; also, new existential vis either part ofariables (unsolved or solved) may appear in Δ. However, the "ordinary" program variable typings $x : A$ must not change.

Our system never adds wrong information that would lead to backtracking: once a solution to a type variable is found, a complete type derivation (if one exists) will have the same solution. If we have a *declarative* type derivation under $[\Omega]\Gamma$ (the declarative context produced by applying all the solutions in Ω) and an algorithmic type derivation $\Gamma \vdash \cdots \dashv \Delta$, where $\Gamma \longrightarrow \Omega$, then we must also have $\Delta \longrightarrow \Omega$. That is, the derivation leading to the output context Δ cannot have solutions that differ from Ω, even if those solutions were not present in the input context Γ.

5.2.4 Contexts as Substitutions. In our 2013 paper, we allowed contexts to be used as substitutions: if Δ contains $\hat{\alpha} = \mathsf{unit}$, then $[\Delta]\hat{\alpha} = \mathsf{unit}$. This usage pervaded the system. For instance, the subsumption rule applies Θ, the output context of the first premise, to the inputs in the second premise:

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \mathrm{Sub}$$

Such applications—found in all our rules with more than one premise—guarantee that whenever the input types in a judgment do not contain existential variables already solved in the input context, the output types do not contain existential variables that are solved in the *output* context. That is, all types are "solved" as much as possible. While this property made the rules a little more complicated, it seemed to make the system easier to work with.

5.2.5 Historical Notes and Other Approaches. The first author combined the two key elements, greedy instantiation and ordered contexts, in a workshop paper [Dunfield 2009]; the idea of using ordered contexts is due to Brigitte Pientka. Unfortunately, key proofs in the paper were severely flawed. Despite these flaws, the second author liked the key ideas and wanted to use them in a type system with higher kinds and inverse types. We have not built that system yet, but the "preliminary" step of shoring up the workshop paper became its own paper [Dunfield and Krishnaswami 2013]. Gundry et al. [2010] offer a reformulation of Algorithm W that is based on information increase over ordered contexts.

We know of several languages that have used our approach or a variant of it: Discus,[6] PureScript,[7] and Hackett.[8] Xie et al. [2018] extended the approach and its metatheory; their work is discussed in Section 9.2.

The first implementation of higher-rank polymorphism to see widespread use was in GHC Haskell, and was documented in Peyton Jones et al. [2007]. The techniques introduced in this paper were quite similar to ours [Dunfield and Krishnaswami 2013], but as usual we did not understand the closeness of the relationship until after we reinvented our own version. They specified their algorithm in the Hindley–Milner style: in their specification, variables are automatically instantiated when used and re-generalized as needed, via generalization and specialization rules that are not syntax-directed. Their algorithm eagerly instantiates quantifiers, re-generalizing let-bound expressions; it uses subtyping (specialization) only at bidirectional mode switches. Our algorithm is lazy, never instantiating a quantifier unless it has to. Eisenberg et al. [2016] extended the algorithm of Peyton Jones et al. [2007] with explicit quantifier instantiations. Implementing this approach required lazy instantiation so that the algorithm could delay instantiation until user applications had been processed.

Zhao et al. [2019] give a new machine-checked formalization of type inference, using our declarative specification [Dunfield and Krishnaswami 2013]. However, they give a new algorithm for easier machine verification, in a "worklist" style: Type inference is broken into subproblems just as with other bidirectional systems, but instead of writing the type inference algorithm as a simple recursion on the structure of the syntax, the problems are pushed onto a stack as the syntax is decomposed. This technique avoids the need for an output context—as type checking refines the values of the existential variables, this is automatically propagated to all the remaining subproblems. By associating contexts with judgments on the worklist, rather than threading a context through the derivation, they avoid the need for our "scope markers." The larger tradeoffs are not entirely clear, but we can say that our algorithm solves problems in a more predictable order, while their approach is potentially more powerful: it enables the algorithm to defer solving problems until more information is available.

5.3 Extensions to Polymorphism

Peyton Jones et al. [2007], our 2013 system, and Zhao et al. [2019] support predicative polymorphism, where quantifiers can be instantiated only with monotypes. Inferring instantiations for impredicative polymorphism is undecidable in general. Building on the approach of Peyton Jones et al. [2007], Serrano et al. [2020] support *guarded* impredicativity, where the quantifier occurs inside a type constructor.

Another extension to polymorphism is **generalized algebraic datatypes (GADTs)**, in which datatypes are not uniformly polymorphic: the type arguments can depend on the constructor

[6] http://blog.discus-lang.org/2017/10/the-disciplined-disciple-compiler-v051.html.
[7] http://www.purescript.org/.
[8] https://github.com/lexi-lambda/hackett.

[Xi et al. 2003]. Pottier and Régis-Gianas [2006] use bidirectional typing to produce GADT-related annotations, which provide the missing information for non-bidirectional type inference. OutsideIn [Vytiniotis et al. 2011] and Dunfield and Krishnaswami [2019] also use bidirectional typing for GADTs. Both use unification to propagate equality information. Our 2019 system has a declarative specification, though our proofs that our algorithm is sound and complete are quite involved. OutsideIn is more liberal about when it can unify two types without an annotation, which makes it more powerful than our system. However, formulating a specification for OutsideIn remains an open problem.

6 VARIATIONS ON BIDIRECTIONAL TYPING

Bidirectional typing explicitly distinguishes inputs and outputs. In this section, we consider systems that make this distinction within types (Section 6.1), within sorts in logic programming (Section 6.2), and within type connectives (Section 6.3). We also discuss systems that use one judgment form with two types, one input and one output (Section 6.4). Finally, we describe a "backwards" version of bidirectional typing with connections to linear type theory (Section 6.5).

When discussing work that uses different notation from ours, e.g., $\downarrow$ and $\uparrow$ instead of $\Leftarrow$ and $\Rightarrow$, we replace the original notation with ours. See Section 11.

6.1 Mixed-direction Types

Instead of distinguishing checking from synthesis at the judgment level, Odersky et al. [2001] make a distinction in type syntax: (1) *inherited* types ${}^{\vee}A$ serve the purpose of the checking judgment, and (2) *synthesized* types ${}_{\wedge}A$ serve the purpose of the synthesis judgment. In their system, *general types* combine inherited and synthesized types. For example, in ${}^{\vee}({}_{\wedge}\mathsf{int} \to {}^{\vee}\mathsf{bool})$ the outermost $\vee$ denotes that the connective $\to$ is inherited (in our terminology, checked), the $\wedge$ that precedes int denotes that the domain of the function is synthesized, and the $\vee$ that precedes bool denotes that the range of the function is inherited (checked). Their subtyping judgment does not synthesize nontrivial supertypes: ${}^{\vee}\mathsf{int} <: {}_{\wedge}\top$ is *not* derivable, but ${}^{\vee}\mathsf{int} <: {}_{\wedge}\mathsf{int}$ is derivable (the supertype is trivial, being equal to the subtype). When the supertype is inherited (checked), as in ${}^{\vee}\mathsf{int} <: {}^{\vee}\top$, subtyping for Odersky et al. corresponds to the bidirectional subsumption rule.

Another approach to pushing these distinctions into the type syntax can be found in the work on *boxy types* [Vytiniotis et al. 2006]. The type syntax of boxy types is simpler than in the work of Odersky et al. [2001], since it does not permit arbitrary interleaving of checked and synthesized type components of a type: inferred types occur in boxes, and boxes are not allowed to nest. In addition, the treatment of variables does not follow the basic bidirectional recipe; instead, variables are *checked*, which is more similar to the backwards approach to bidirectional typing we discuss in Section 6.5.

6.2 Directional Logic Programming

One lesson that can already be drawn is that the flow of information through the typing judgments is a key choice in the design of bidirectional systems, and that it is often desirable to go beyond the simple view of modes as either inputs or outputs: for example, Odersky et al. [2001] track whether each part of a type is an input or output. So describing bidirectional typing algorithms can require a more subtle notion of mode.

Reddy [1993] adapts ideas from classical linear logic to characterize *directional* logic programs. Directional logic programming subsumes moded logic programming. For example, a ternary predicate p in regular multi-sorted predicate logic might be given the type:

$$p : \mathtt{List(Int)} \times \mathtt{Int} \times \mathtt{Bool} \to \mathtt{prop}.$$

This says that a proposition of the form $p(X, Y, Z)$ has three arguments, with X a list of integers, Y an integer, and Z a Boolean. With an ordinary mode system, each of these three arguments must be classified entirely as an input or output.

However, in directional logic programming, modes become part of the structure of the sorts, which permits giving sorts like

$$p : \texttt{List}(\texttt{Int}^{\perp}) \otimes \texttt{Int} \otimes \texttt{Bool}^{\perp} \to \text{prop}.$$

Now, in a predicate occurrence of the form $p(X, Y, Z)$, the argument Y is an input integer, and Z is an output Boolean, with the an output of sort τ marked as $\tau^{\perp}$. The argument X has the sort $\texttt{List}(\texttt{Int}^{\perp})$, meaning that the list is structurally an input (so its length is known) but its elements are outputs: the type `Int` denotes an integer that is an input, while $\texttt{Int}^{\perp}$ denotes an integer that is an output.

The notation $A^{\perp}$ corresponds to negation, and the classical nature of the sort structure arises from the fact that outputting an output $A^{\perp\perp}$ is the same as an input A—i.e., the user must supply a box that will be filled with an A.

This more fine-grained structure lets us give sort declarations that capture the fact that (for example) the boxes in boxy types are outputs but the rest of the type is an input.

6.3 Mode Annotations

Davies [2005, pp. 242–243] describes *mode annotations* that would allow programmers to declare which functions should be typed using synthesis (instead of checking) and whether the entire application should be checking (instead of synthesizing). As far as we know, mode annotations were never implemented.

Davies motivated this annotation form for polymorphic functions like "higher-order case," which takes as arguments an instance of a datatype (`bits`) and a series of functions corresponding to case arms. That is, the "first-class" case expression

$$\mathsf{case}(e_{\text{bits}},\ e_{\text{bnil-case}} \mid x.\, e_{\text{b0-case}} \mid y.\, e_{\text{b1-case}})$$

is written $\texttt{bcase}\ e_{\text{bits}}\ (\lambda().\, e_{\text{bnil-case}})\ (\lambda x.\, e_{\text{b0-case}})\ (\lambda y.\, e_{\text{b1-case}})$.

We can declare the type and *mode* of the function `bcase`:

```
val bcase : bits → (unit → α) → (bits → α) → (bits → α) → α,
mode bcase : inf → chk → chk → chk → chk.
```

The second line is a mode annotation. It says that the first argument of `bcase` should synthesize (be *inf*erred), the remaining three arguments should be checked, and the entire application should be checked (the last *chk*). Synthesizing the type of the first argument corresponds to the synthesizing principal judgment of the elimination rule +Elim; checking the other arguments corresponds to the checking premises of +Elim; checking the entire application corresponds to the checking conclusion of +Elim.

One can view mode annotations as instructions for transforming a type $A \to B$ into a "decorated" type—something like Odersky et al. [2001], but where the connective itself is decorated. The recipe's elimination rule →Elim would correspond to a type ${}^{\Rightarrow}\!\to^{\Leftarrow}_{\Rightarrow}$, matching the scheme ${}^{pr1}\!\to^{pr2}_{conc}$ where *pr1* is the first premise, *pr2* is the second premise and *conc* is the conclusion. Not all such decorations would be mode correct.

6.4 Simultaneous Input and Output

Another way to blend input and output was developed by Pottier and Régis-Gianas [2006] for type inference for GADTs [Xi et al. 2003]. The overall structure of their system is unusual (their

bidirectional algorithm produces *shapes*, which are then given to a non-bidirectional type inference algorithm); for brevity, we explain the idea in a more ordinary setting involving types, not shapes. The basic idea is to combine synthesis and checking judgments into a single judgment with two types: $\Gamma \vdash e \Leftarrow A \Rightarrow B$ means "check e against A, synthesizing B," where B is a subtype of A. The system thus operates in both checking and synthesis modes simultaneously.

A similar idea is used in the program synthesis work of Polikarpova et al. [2016], *round-trip type checking*: combines a checking judgment $\Gamma \vdash e \Leftarrow A$ with a *type strengthening* judgment $\Gamma \vdash e \Leftarrow A \Rightarrow B$.

For example, the strengthening rule for variables [Polikarpova et al. 2016, Figure 4] checks against a given type A, but utilizes Γ's refinement type $\{b \mid \psi\}$ (base type b such that the constraint ψ holds) to produce a strengthened type $\{b \mid \nu = x\}$:

$$\frac{\Gamma(x) = \{b \mid \psi\} \qquad \Gamma \vdash \{b \mid \psi\} <: A}{\Gamma \vdash x \Leftarrow A \Rightarrow \{b \mid \nu = x\}} \textsc{VarSc}.$$

Synthesis becomes a special case of strengthening: $\Gamma \vdash e \Rightarrow B$ can be written $\Gamma \vdash e \Leftarrow \mathsf{top} \Rightarrow B$. Since every type is a subtype of top, the synthesized type B is stronger than the goal type (top).

6.5 Backwards Bidirectional Typing

In the basic Pfenning recipe, the principal judgment in an introduction rule is checked, and the principal judgment in an elimination rule synthesizes. However, Zeilberger [2015] observed that in a multiplicative linear type theory, bidirectional typing works precisely as well if you did it backwards, changing all occurrences of synthesis to checking, and vice versa. Zeilberger's observation was made in the context of a theorem relating graph theory and type theory, but it is a sufficiently striking result that it is worth spelling out in its own right. We will not precisely replicate his system, but we will discuss our divergences when relating it to other bidirectional type systems.

First, we give the syntax of multiplicative linear logic.

$$\begin{array}{lll} \text{Types} & A ::= 1 \mid A \otimes B \mid A \multimap B, \\ \text{Terms} & e ::= x \mid \lambda x.\, e \mid e\, e' \\ & \quad \mid () \mid \mathsf{let}\, () = e\, \mathsf{in}\, e' \\ & \quad \mid \langle e, e' \rangle \mid \mathsf{let}\, \langle x, y \rangle = e\, \mathsf{in}\, e', \\ \text{Contexts} & \Gamma ::= \cdot \mid \Gamma, x \Leftarrow A. \end{array}$$

The types of MLL are the unit type 1, the tensor product $A \otimes B$, and the linear function space $A \multimap B$. Unit and tensor are introduced by $()$ and $\langle e, e' \rangle$, and are eliminated by pattern matching. Functions are introduced by $\lambda x.\, e$ and eliminated using applications $e\, e'$.

Contexts are a bit unusual—they pair together variables and their types as usual, but instead of treating a variable as a placeholder for a *synthesizing* term, we treat variables as placeholders for *checking* terms. This will have substantial implications for the mode discipline of the algorithm, but we will defer discussion of this point until the whole system is presented.

Now, we give the typing rules, starting with those for the unit type:

$$\frac{}{\cdot \vdash () \Rightarrow 1} \qquad \frac{\Delta \vdash e' \Rightarrow A \qquad \Gamma \vdash e \Leftarrow 1}{\Gamma, \Delta \vdash \mathsf{let}\, () = e\, \mathsf{in}\, e' \Rightarrow A}.$$

The introduction rule says that in an empty context, the unit value $()$ *synthesizes* the type 1. The pattern-matching style elimination $\mathsf{let}\, () = e\, \mathsf{in}\, e'$ first synthesizes a type A for the body e', and then checks that the scrutinee e has the unit type 1.

Thus, we synthesize a type for the continuation first, before checking the type of the data we are eliminating; this is the exact reverse of the Pfenning recipe. For the unit type, this is a mere

curiosity, but it gets more interesting with the tensor product type $A_1 \otimes A_2$:

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Delta \vdash e_2 \Rightarrow A_2}{\Gamma, \Delta \vdash \langle e_1, e_2 \rangle \Rightarrow A_1 \otimes A_2} \qquad \frac{\Gamma, x_1 \Leftarrow A_1, x_2 \Leftarrow A_2 \vdash e' \Rightarrow C \qquad \Delta \vdash e \Leftarrow A_1 \otimes A_2}{\Gamma, \Delta \vdash \mathsf{let}\, \langle x_1, x_2 \rangle = e \,\mathsf{in}\, e' \Rightarrow C}\ .$$

The synthesis rule for pairs remains intuitive, though it reverses the direction given by the Pfenning recipe: for a pair $\langle e_1, e_2 \rangle$, we first synthesize A_1 for e_1 and A_2 for e_2, then conclude that the pair has type $A_1 \otimes A_2$.

However, the elimination rule typing $\mathsf{let}\, \langle x_1, x_2 \rangle = e \,\mathsf{in}\, e'$ is startling. First, it synthesizes the type C for the continuation e'; we learn from having typed e' that x_1 and x_2 need to have types A_1 and A_2, respectively. This gives us the information we need to check e against $A_1 \otimes A_2$. The linear function type $A_1 \multimap A_2$ has a similar character:

$$\frac{\Gamma, x \Leftarrow A \vdash e \Rightarrow B}{\Gamma \vdash \lambda x.\, e \Rightarrow A \multimap B} \qquad \frac{\Gamma \vdash e' \Rightarrow A \qquad \Delta \vdash e \Leftarrow A \multimap B}{\Gamma, \Delta \vdash e\, e' \Leftarrow B}\ .$$

Here, to synthesize a type for the introduction form $\lambda x.\, e$, we synthesize B for the body e, and then look up what type A the argument x needs to have in order for the body e to be well typed. To check that an application $e\, e'$ has the type B, we synthesize A for the argument e', and then check the function e against $A \multimap B$.

The rule for product elimination suggests a reversed rule for let-expressions, which allows us to defer checking the bound expression e until we know what type it needs to have:

$$\frac{\Gamma, x \Leftarrow A \vdash e' \Leftarrow B \qquad \Gamma \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{let}\, x = e \,\mathsf{in}\, e' \Leftarrow B}\ .$$

Again, the checking/synthesis modes are reversed from most bidirectional type systems. We can see how this reversal plays out for the following variables:

$$\frac{}{x \Leftarrow A \vdash x \Leftarrow A} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ .$$

Here, when we check that the variable x has type A, the context must be such that it requires x to have the type A. However, the switch between checking and synthesis is standard.

Relative to most bidirectional systems, the information flow in the variable rule (as well as for pattern matching for pairs and lambda-abstraction for functions) is strange. Usually, the context would give the type of each variable. However, in this case the context *is told* the type of each variable. This system of rules is still well-moded in the logic programming sense, but the moding is more exotic than simple inputs or outputs. Within a given context, the variables are inputs, but their types are outputs. Following Reddy [1993], the moding of checking and synthesis might be given as

$$\texttt{mode check : (List (Var} \otimes \texttt{Type}^{\perp}\texttt{)} \otimes \texttt{Term} \otimes \texttt{Type)} \to \mathsf{prop},$$
$$\texttt{mode synth : (List (Var} \otimes \texttt{Type}^{\perp}\texttt{)} \otimes \texttt{Term} \otimes \texttt{Type}^{\perp}\texttt{)} \to \mathsf{prop}.$$

This mode declaration says that for both checking and synthesis, the spine of the context and the variable names are inputs, but the ascribed type for each variable is an output. Similarly, the term is an input in both judgments, but the type is an input in `check` but an output in `synth`.

We can relatively easily prove a substitution theorem for the backwards system:

Theorem 1. *(Backwards Substitution) If* $\Delta \vdash e \Leftarrow A$, *then*

(1) If $\Gamma, x \Leftarrow A, \Theta \vdash e' \Leftarrow C$, *then* $\Gamma, \Delta, \Theta \vdash [e/x]e' \Leftarrow C$,
(2) If $\Gamma, x \Leftarrow A, \Theta \vdash e' \Rightarrow C$, *then* $\Gamma, \Delta, \Theta \vdash [e/x]e' \Rightarrow C$.

Unfortunately, we do not presently know how to nicely characterize the set of terms that is typable under this discipline, unlike the characterization for the Pfenning recipe that annotation-free terms are the β-normal terms.

6.5.1 Applications of Backwards Bidirectional Typing. When designing a bidirectional type system, what approach should we use—the Pfenning recipe, its pure reversal (as presented above), or something else? The most popular answer seems to be "something else": Many practical systems synthesize types for literals (unit, Booleans, etc.) and for pairs, which—being introduction forms—can only be checked under the strict Pfenning recipe. However, a number of papers have used reversed rules in more subtle ways. Which approach to take depends on the choice of tradeoffs: for example, we can require fewer annotations if we complicate the flow of type information.

Drawing inspiration from relevance logic, which requires variables to be used at least once (as opposed to the exactly-once constraint of linear logic), Chlipala et al. [2005] require a let-bound variable to be used at least once. This allows them to reverse the typing rule for let-expressions, reducing the annotation burden of the basic Pfenning recipe: no annotation is needed on the let-bound expression. Their typing contexts contain checking variables, whose moding is similar to the variables in our backwards bidirectional system. Such a variable must occur at least once in a checking position; that occurrence determines the type of the variable, which can be treated as synthesizing in all other occurrences.

Xie and Oliveira [2018] present another bidirectional type system for polymorphism. Their rule for function application is very similar to the backwards rule presented here, with the idea that backwards typing means that applications like $(\lambda x.\, e)\ e'$ do not need a type annotation at the redex. This requires fewer annotations on let-bindings, as in the work of Chlipala et al. [2005], but with support for polymorphism.

Zeilberger [2015] (and its follow-up work Zeilberger [2018]) did not use any type annotations at all. Instead, his bidirectional system was used to deduce a type scheme for the linear lambda terms, in the style of ML type inference, to find a simple proof of the fact every linearly typed term has a most general type, and moreover that the structure of its β-normal, η-long form is determined by this type scheme.

Intersection types can reconcile multiple occurrences of the same variable at different types; it appears that the type inference algorithm of Dolan [2016] can be viewed as calculating intersections via a computable lattice operation on types.

7 PROOF THEORY, NORMAL FORMS, AND TYPE ANNOTATIONS

Logic and deduction are connected to bidirectional typing. One connection is that the Pfenning recipe leads to systems with a subformula property, which is remarkably useful in some settings. Another connection is to the idea of verifications—normal deductions that verify (check) propositions—and *uses* that are extracted (synthesized) from assumptions.

7.1 Subformula Property

In cut-free sequent calculi, every formula (proposition) that appears in a derivation is a subformula of some formula in the conclusion. For example, in the following sequent calculus derivation, the formulas $(P \wedge Q) \wedge R$ and $P \wedge Q$ are subformulas (subterms) of the conclusion's assumption $(P \wedge Q) \wedge R$.

$$\dfrac{\dfrac{\dfrac{}{(P \wedge Q) \wedge R \vdash (P \wedge Q) \wedge R}}{(P \wedge Q) \wedge R \vdash P \wedge Q}}{(P \wedge Q) \wedge R \vdash Q}$$

Through the Curry–Howard correspondence, a property of formulas becomes a property of types, but the property is still called the *subformula* property, to avoid confusion with "subtype."

A consequence of the subformula property is that if a connective appears in a formula, such as the $\wedge$ in $P \wedge Q$ in the middle step, the connective must appear in the conclusion. This consequence is useful in a number of type systems, because it ensures that problematic type connectives appear only with the programmer's permission. Bidirectional type systems based on the recipe in Section 4 only synthesize types that are (subformulas of) annotations: to eliminate an $\rightarrow$, the function subterm must synthesize by reason of being a variable, an annotation, or an elimination form. The type of a variable flows from an annotation on the binding form (e.g., a λ inside an annotation) or from the synthesizing premise of an elimination rule (e.g., the premise typing the scrutinee of a case).

Consider, for example, intersection and union types. Efficient type-checking for intersections and unions is difficult [Reynolds 1996; Dunfield 2007]; intersections and unions that come "out of nowhere" in the middle of a derivation—without being requested via a type annotation—would aggravate this difficulty. Another example is found in type systems that encode multiple evaluation strategies: if a programmer generally prefers call-by-value, but occasionally wants to use call-by-name, the subformula property implies that call-by-name connectives appear only when requested [Dunfield 2015]. Risky connectives abound in gradual type systems: *unknown* or *uncertain* types should appear only with the programmer's permission, because they permit more dynamic failures than other type connectives do [Jafery and Dunfield 2017].

In type inference, all of typing is in a single judgment. Without a checking judgment, there is no goal type; to increase typing power, one must put more and more "cleverness" into inference. Certain kinds of cleverness destroy the subformula property: automatic generalization, for example, creates "for all" connectives out of nowhere.[9] If we relax the recipe by including synthesis rules for `()`, integer literals, and similar constructs, then we break the subformula property: `()` synthesizes unit when the programmer never wrote unit. However, a weaker—and still interesting—version of the property may hold, since every type appearing in a derivation is either a subformula of a type in the conclusion *or* the "obvious" type of a literal constant. That is, we can view `()` as a request for the type unit. Note that if we think of `()` and integer literals as constants whose type is given in a primordial context, so that instead of $\Gamma \vdash e : A$ we have

$$() : \mathsf{unit},\ 0 : \mathsf{int},\ -1 : \mathsf{int},\ 1 : \mathsf{int},\ -2 : \mathsf{int},\ \ldots,\ \Gamma \vdash e : A,$$

then the full subformula property holds. In effect, the author of the primordial context (the language designer) has requested that `()` and all the integer literals be permitted. Other conveniences, such as synthesizing the types of monomorphic functions, can also be justified (with a little more work; one must think of λ as a sort of polymorphic constant).

In bidirectional systems, the goal in the checking judgment can steer typing and avoid a measure of cleverness. Thus, while the subformula property is not enjoyed by every imaginable bidirectional type system, bidirectionality seems to make the property easier to achieve.

7.2 Verifications and Uses

We briefly discuss verifications and uses, which are analogues of checking and synthesis in the setting of deductive systems.

[9] It can be argued that automatic generalization is acceptable, because "for all" is less problematic. One might still want a weaker version of the subformula property, saying that every type is either a subformula or related by generalization (and instantiation).

In the linear simply typed lambda calculus of Cervesato and Pfenning [2003], typing is presented using two judgments: a *pre-canonical* judgment that "validates precisely the well-typed terms… in η-long form" and a *pre-atomic* judgment that "handles intermediate stages of their [the terms'] construction." As suggested by the word "validates," the pre-canonical judgment corresponds to checking, and the pre-atomic judgment corresponds to synthesis.

Their notation differs from most of the early papers on bidirectional typing: they write pre-canonical judgments as $M \Uparrow a$ and pre-atomic judgments as $M \downarrow a$, which is almost exactly the reverse of (for example) DML [Xi and Pfenning 1999], which used $\uparrow$ for synthesis and $\downarrow$ for checking. (Both notations are reasonable: computer scientists usually write trees with the root at the top, so Xi's arrows match the flow of type information through a syntax tree; Gentzen put the root of a derivation tree at the bottom, so Cervesato and Pfenning's arrows match the flow of type information through the typing derivation.)

This division into validation (checking) and handling intermediate values (synthesis) persists, with different terminology, in Frank Pfenning's teaching on verifications and uses: A *verification* of a proposition checks that it is true; a *use* of an assumption decomposes the assumed proposition. We are not aware of a published paper describing verifications and uses, but the idea appears in many of Pfenning's lecture notes. The earliest seems to be Pfenning [2004, p. 29], with similar notation to Cervesato and Pfenning [2003]:

$A \Uparrow$ Proposition A has a normal deduction, and
$A \downarrow$ Proposition A is extracted from a hypothesis.

Later lecture notes introduce the terminology of *verifications* and *uses*, writing $\uparrow$ and $\downarrow$, respectively [Pfenning 2009, 2017]. Verification is related [Pfenning 2017, p. 2] to "intercalation" in proof search [Sieg and Byrnes 1998].

8 FOCUSING, POLARIZED TYPE THEORY, AND BIDIRECTIONAL TYPE SYSTEMS

A widespread folklore belief among researchers is that bidirectional typing arises from *polarized* formulations of logic. This belief is natural, helpful, and (surprisingly) wrong.

8.1 Bidirectional Typing and the Initial Cartesian Closed Category

The naturalness of the connection can be seen from Figure 2, which gives a bidirectional type system that precisely characterizes β-normal, η-long terms. The only necessary changes from Figure 1 were:

- The annotation rule was removed. Since annotations are only required at β-redexes, the omission of this rule forces all typable terms to be β-normal.
- The mode-switch rule from synthesis to checking is restricted to allow mode switches only at base type. This makes it impossible to partially apply a function in the checking mode: if $f : b \to b \to b$ and $x : b$, then $f\ x\ x$ checks but $f\ x$ does not. If a partial application is desired, then it must be η-expanded to $\lambda y.\, f\ x\ y$.

Together, these two restrictions ensure that only β-normal, η-long terms typecheck. Moreover, this characterization is easy to extend to products:

$$\frac{}{\Gamma \vdash () \Leftarrow 1}\ \text{(no unit elim.)} \qquad \frac{\Gamma \vdash e_1 \Leftarrow A_1 \qquad \Gamma \vdash e_2 \Leftarrow A_2}{\Gamma \vdash (e_1, e_2) \Leftarrow A_1 \times A_2} \qquad \frac{\Gamma \vdash e \Rightarrow A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i(e) \Rightarrow A_i}\ .$$

This type system now characterizes normal forms in the STLC with units and products. Recall that the lambda calculus with units, pairs, and functions is a syntax for the initial cartesian closed category, when terms are quotiented by the $\beta\eta$ theory for each type [Lambek 1985].

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \\ \text{Types} & A, B, C & ::= & b \mid A \to A \\ \text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

$\boxed{\begin{array}{l}\Gamma \vdash e \Leftarrow A \\ \Gamma \vdash e \Rightarrow A\end{array}}$ Under Γ, expression e checks against type A
Under Γ, expression e synthesizes type A

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \mathsf{Var}\Rightarrow \qquad \frac{\Gamma \vdash e \Rightarrow b \qquad b = B}{\Gamma \vdash e \Leftarrow B}\ \mathsf{Sub}\Leftarrow$$

$$\frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash (\lambda x.\, e) \Leftarrow A_1 \to A_2}\ \to\mathsf{I}\Leftarrow \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \to\mathsf{E}\Rightarrow$$

Fig. 2. A bidirectional type system characterizing β-normal, η-long normal forms.

Since this bidirectional system requires β-normal, η-long terms, we can see it as a calculus that presents the initial model of Cartesian closed categories *without* any quotienting. Morphisms are well-typed terms, and two morphisms are equal when they are α-equivalent.

All that remains is to show that identities and composition are definable. In the ordinary presentation of the initial CCC, a morphism is a term with a free variable, and composition is substitution. In the bidirectional system, however, a morphism $A \to B$ is a checking term $x : A \vdash e \Leftarrow B$, and substituting a checking term for a variable does not preserve the β-normal, η-long property. However, if we use *hereditary substitution* [Pfenning and Davies 2001; Watkins et al. 2003; Nanevski et al. 2008]—a definition of substitution that inspects the structure of the term being substituted and "re-normalizes" as it goes—then substitution preserves β-normal, η-long terms.

This means that the bidirectional type system constitutes a term model for the initial CCC:

(1) The objects of the term are the types of the programming language.
(2) Morphisms $X \to Y$ are terms $x : X \vdash e \Leftarrow Y$.
(3) The identity morphism $id : X \to X$ is the η-expansion of the single free variable.
(4) Composition of morphisms is given by hereditary substitution.

The usual presentation of the term model requires quotienting terms by $\beta\eta$-equivalence, but the term model built from the bidirectional system has the property that equality of morphisms is just α-equivalence. By interpreting a non-normal term into this category, any two $\beta\eta$-equal terms will have the same denotation.

8.2 Adding Problems with Sums

This construction is so beautiful that it is essentially irresistible to add sums to the language. Unfortunately, doing so introduces numerous difficulties. These are most simply illustrated by using the basic bidirectional recipe of Dunfield and Pfenning [2004], which yields an introduction and elimination rule for sum types as follows:

$$\frac{\Gamma \vdash e \Leftarrow A_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{inj}_i\, e \Leftarrow A_1 + A_2}\ ,$$

$$\frac{\Gamma \vdash e \Rightarrow A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash e_1 \Leftarrow C \qquad \Gamma, x_2 : A_2 \vdash e_2 \Leftarrow C}{\Gamma \vdash \mathsf{case}(e,\ \mathsf{inj}_1\, x_1.\, e_1,\ \mathsf{inj}_2\, x_2.\, e_2) \Leftarrow C}\ .$$

These rules say that both the injection and case rules have a checking conclusion, but that the scrutinee e in the case must synthesize a sum type. As we noted in Section 4.1, this imposes some restrictions on which terms are typable. For example, because the rule for case has a checking conclusion, we cannot use a case in function position without a type annotation:

$$a : ((b \to A) + (b \to A)), x : b \not\vdash \mathsf{case}(a,\ \mathsf{inj}_1\, f.\, f,\ \mathsf{inj}_2\, g.\, g)\ x \Leftarrow A.$$

Instead of applying an argument to a case expression of function type, we must push the arguments into the branches:

$$a : ((b \to A) + (b \to A)), x : b \vdash \mathsf{case}(a,\ \mathsf{inj}_1\, f.\, f\ x,\ \mathsf{inj}_2\, g.\, g\ x) \Leftarrow A.$$

If we intend to type only normal forms, then this seems desirable: These rules are prohibiting certain term forms that correspond to commuting conversions of allowed terms. The need for commuting conversions has never been popular with logicians: witness Girard's lament [Girard 1989, p. 79] that "one tends to think that natural deduction should be modified to correct such atrocities." However, the simple bidirectional system does not completely eliminate the need for commuting conversions. For example, consider the term $f : b \to b, x : b + b \vdash f\ \left(\mathsf{case}(x,\ \mathsf{inj}_1\, y.\, y,\ \mathsf{inj}_2\, z.\, z)\right) \Rightarrow b$. This term is equivalent to the previous one by a commuting conversion, but both are still typable.

Note that allowing the case form to synthesize a type (as in Section 4.1) allows *more* terms to typecheck, which is the opposite of what we want (in this section, anyway). In practice, it can be difficult to support an unannotated case form, which synthesizes its type. Concretely, if the arm e_1 synthesizes the type C_1 and the arm e_2 synthesizes the type C_2, we have to check that they are equal. However, in the general case (e.g., in dependent type theory) equality is *relative to the context*, and the context is different in each branch (with $\Gamma, x_1 : A_1$ in one branch and $\Gamma, x_2 : A_2$ in the other). This is why dependent type theories like Coq end up requiring case expressions to be annotated with a return type: this resolves the problem by having the programmer solve it herself.

8.3 A Polarized Type Theory

At this point, we can make the following pair of observations:

(1) The simple bidirectional system for the STLC with products has the property that two terms are $\beta\eta$-equal if and only if they are the same: it fully characterizes $\beta\eta$-equality.
(2) Adding sum types to the bidirectional system breaks this property: two terms equivalent up to (some) commuting conversions may both be typable.

To restore this property, two approaches come to mind. The first approach is to find even more restrictive notions of normal form, which prohibit the commuting conversions. We will not pursue this direction in this article, but see Scherer [2017] and Ilik [2017] for examples of this approach.

The second approach is to find type theories in which the commuting conversions *no longer preserve equality*. By adding (abstract) effects to the language, terms that used to be equivalent can now be distinguished, ensuring that term equality once again coincides with semantic equality. This is the key idea embodied in what is variously called *polarized type theory*, *focalization*, or *call-by-push-value* [Levy 2001].

In Figure 3, we give a polarized type theory resembling those of Simmons [2014] and Espírito Santo [2017]. Our main change is to adjust the proof term assignment to look more familiar to functional programmers.

The key idea in polarized type theory is to divide types into two categories: *positive* types P (sums, strict products, and suspended computations) and *negative* types N (basically, functions). Positive types are eliminated by pattern matching, and negative types are eliminated by supplying arguments. Negative types can be embedded into positive types using the "downshift" type

$$
\begin{array}{lrcl}
\text{Positive types} & P, Q & ::= & \mathsf{unit} \mid P \times Q \mid P + Q \mid {\downarrow} N \\
\text{Negative types} & N, M & ::= & P \to N \mid {\uparrow} P \\
\text{Values} & v & ::= & u \mid () \mid (v, v) \mid \mathrm{inj}_i\, v \mid \{t\} \\
\text{Spines} & s & ::= & \cdot \mid v\ s \\
\text{Terms} & t & ::= & \mathsf{return}\ v \mid \lambda \overrightarrow{p_i \to t_i} \mid \mathsf{match}\ x \cdot s\ \mathsf{of}\ [\overrightarrow{p_i \to t_i}] \\
\text{Patterns} & p & ::= & u \mid () \mid (p, p') \mid \mathrm{inj}_i\, p \mid \{x\} \\
\text{Contexts} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : N \mid \Gamma, u : P
\end{array}
$$

Typing values $\Gamma \vdash v \Leftarrow P$ Typing terms $\Gamma \rhd t \Leftarrow N$
Typing spines $\Gamma \vdash s : N \gg M$ Typing patterns $p : P \rightsquigarrow \Delta$

Fig. 3. Syntax and judgment forms of polarized type theory.

${\downarrow} N$ (representing suspended computations); positive types can be embedded into negative types using the "upshift" ${\uparrow} P$ (denoting computations producing P's). Non-strict products, eliminated by projections, would be negative types. Both call-by-value and call-by-name can be encoded into polarized type theory, similar to call-by-push-value [Levy 2001].

The semantics of call-by-push-value offer insight into the design of this calculus: positive types correspond to objects of a category of values (such as sets and functions), and negative types correspond to objects of a category of computations (objects are algebras of a signature for the computations, and morphisms are algebra homomorphisms). Upshift and downshift form an adjunction between values and computations, and the monads familiar to functional programmers arise via the composite: $T(P) \triangleq {\downarrow}{\uparrow} P$.

While this calculus arises from meditation upon invariants of proof theory, its syntax is much closer to practical functional programming languages than the pure typed lambda calculus, including features like clausal definitions and pattern matching. But the price we pay is a proliferation of judgments. We usually introduce separate categories of *values* (that introduce positive types) and *spines* (argument lists for calling functions), as well as *terms* (that put values and spines together in computations, and introduce negative types) and *patterns* (that eliminate positive types).

Contexts have two kinds of variables, $x : N$ for negative variables and $u : P$ for positive variables.

8.3.1 Typing Values. The judgment $\Gamma \vdash v \Leftarrow P$ checks the type of positive values:

$$
\frac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}} \qquad \frac{\Gamma \vdash v \Leftarrow P \qquad \Gamma \vdash v' \Leftarrow Q}{\Gamma \vdash (v, v') \Leftarrow P \times Q} \qquad \frac{\Gamma \vdash v \Leftarrow P_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathrm{inj}_i\, v \Leftarrow P_1 + P_2}
$$

$$
\frac{\Gamma \rhd t \Leftarrow N}{\Gamma \vdash \{t\} \Leftarrow {\downarrow} N} \qquad \frac{(u : Q) \in \Gamma \qquad P \equiv Q}{\Gamma \vdash u \Leftarrow P}\ .
$$

The rules for units, pairs and sums are unchanged from the simple bidirectional recipe. The rule for downshift says that if a term t checks against a negative type N, then the thunk $\{t\}$ will check against the downshifted type ${\downarrow} N$. Finally, a variable u checks against a type P if the context says that u has a type Q equal to P. (With subtyping, we would instead check that Q is a subtype of P.)

8.3.2 Typing Spines. Before we give the typing rules for all terms, we will give the rules deriving the spine judgment $\Gamma \vdash s : N \gg M$, read "if spine s is applied to a head of type N, it will produce a result of type M." The type N is an algorithmic input, and the type M is an output:

$$
\frac{}{\Gamma \vdash \cdot : N \gg N} \qquad \frac{\Gamma \vdash v \Leftarrow P \qquad \Gamma \vdash s : N \gg M}{\Gamma \vdash v\ s : P \to N \gg M}\ .
$$

The first rule says that an empty argument list does nothing to the type of the head: the result is the same as the input. The second rule says that a non-empty argument list $v\ s$ sends the function type $P \to N$ to M, if v is a value of argument type P and s is an argument list sending N to M.

8.3.3 Typing Terms. With values and spines in hand, we can talk about terms, in the term typing judgment $\Gamma \rhd t \Leftarrow N$, which checks that a term t has the type N.

$$\frac{\Gamma \vdash v \Leftarrow P}{\Gamma \rhd \mathsf{return}\ v \Leftarrow \uparrow P} \qquad \frac{\begin{array}{c}\text{for all } i < n.\\ p_i : P \rightsquigarrow \Delta_i\\ \Gamma, \Delta_i \rhd t_i \Leftarrow N\end{array}}{\Gamma \rhd \lambda \overrightarrow{p_i \to t_i}^{i<n} \Leftarrow P \to N} \qquad \frac{\begin{array}{cc}(x : M) \in \Gamma & \text{for all } i < n.\\ \Gamma \vdash s : M \gg \uparrow Q & p_i : Q \rightsquigarrow \Delta_i\\ & \Gamma, \Delta_i \rhd t_i \Leftarrow \uparrow P\end{array}}{\Gamma \rhd \mathsf{match}\ x \cdot s\ \mathsf{of}\ [\overrightarrow{p_i \to t_i}^{i<n}] \Leftarrow \uparrow P}.$$

The rule for return v says that we embed a value v of type P into the upshift type $\uparrow P$ by immediately returning it. Lambda abstractions are pattern-style—instead of a single binder $\lambda x.\, t$, we give a list of patterns and branches $\lambda \overrightarrow{p_i \to t_i}$ to check at type $P \to N$. As a result, we need a judgment $p_i : P \rightsquigarrow \Delta_i$ giving the types of the bindings Δ_i of the pattern p_i. Then, we check each branch t_i against the result type N in a context extended by Δ_i.

We face similar issues in the match expression match $x \cdot s$ of $[\overrightarrow{p_i \to t_i}]$. We find the variable x in the context, apply it to arguments s to get a result of type $\uparrow Q$, and then pattern-match on Q. (In typical bidirectional systems, a synthesis judgment would type $x \cdot s$; in this polarized system of normal forms, the synthesis judgment is absorbed into the rule for match.) We check that the spine s sends M to the type $\uparrow Q$, produce variables Δ_i for patterns p_i at type Q, and check each t_i against $\uparrow P$. Restricting the type at which we can match forces us to η-expand terms of function type.

Both lambdas and application/pattern-matching use the judgment $p : P \rightsquigarrow \Delta$ to find the types of the bindings. The rules for these are straightforward:

$$\frac{}{() : \mathsf{unit} \rightsquigarrow \cdot} \qquad \frac{p_1 : P_1 \rightsquigarrow \Delta_1 \qquad p_2 : P_2 \rightsquigarrow \Delta_2}{(p_1, p_2) : (P_1 \times P_2) \rightsquigarrow \Delta_1, \Delta_2} \qquad \frac{p : P_i \rightsquigarrow \Delta \qquad i \in \{1, 2\}}{\mathsf{inj}_i\, p : (P_1 + P_2) \rightsquigarrow \Delta}$$

$$\frac{}{\{x\} : \downarrow N \rightsquigarrow x : N} \qquad \frac{}{u : P \rightsquigarrow u : P}.$$

Units yield no variables at type unit, pair patterns (p_1, p_2) return the variables of each component, injections $\mathsf{inj}_i\, p$ return the variables of the sub-pattern p, and thunk patterns $\{x\}$ at type $\downarrow N$ return that variable x at type N. Here, we omit a judgment to check whether a set of patterns is complete; see Krishnaswami [2009] for a polarization-based approach. Value bindings $u : P$ bind a value of type P to a variable u, which allows values to be pattern-matched without being fully decomposed. Hence, our calculus is weakly focused in the sense of Pfenning and Simmons [2009].

8.4 Discussion

Bidirectional typing integrates very nicely into a perspective based on polarized logic. Indeed, the "application judgment" in Dunfield and Krishnaswami [2013] can be seen as a special case of the spine judgment, and even systems not designed from an explicitly bidirectional perspective, such as Serrano et al. [2018], have found it beneficial to work with entire argument lists. In our view, this is because the spine judgment is well-moded, making it easy to manage the flow of information through the argument list.

This close fit is not limited to argument lists, but also extends to other features that go beyond kernel calculi, such as pattern matching. Krishnaswami [2009] shows how ML-style pattern matching arises as the proof terms of a focused calculus, and indeed that paper's type system is

bidirectional. It only covered simple types, but the approach scales well. Our bidirectional type system for generalized algebraic data types [Dunfield and Krishnaswami 2019] goes much further, including both universal and existential quantification, GADTs, and pattern matching. Nevertheless, it is built upon essentially the same idea of applying bidirectional typing to a focused type theory.

However, despite the fact that the standard recipe of bidirectional typing fits beautifully with focused logics, we should not lose sight of the fact that the essence of bidirectional typing is the management of information flow. Consequently, these techniques apply more broadly than polarized calculi, as fundamental as they may be. In Section 6, we saw a number of systems with a different mode structure, such as the mixed-direction types of Odersky et al. [2001], the strict type inference of Chlipala et al. [2005], and the backwards bidirectional system of Zeilberger [2015]. All of these reject the basic bidirectional recipe, but are undeniably bidirectional.

Thus, we would advise designers of new bidirectional systems to seek inspiration from polarized type theory, but not to restrict themselves to it.

9 OTHER APPLICATIONS OF BIDIRECTIONAL TYPING

9.1 Dependent Types, Refinement Types, and Intersection Types

The DML system [Xi and Pfenning 1999; Xi 1998] used bidirectional typing, because type inference for index refinements (a form of refinement type) is undecidable. DML followed a "relaxed" version of the Pfenning recipe that allowed some rules that are not strictly necessary, such as an introduction rule that synthesizes a type for (e_1, e_2) if e_1 and e_2 synthesize.

The first datasort refinement system [Freeman and Pfenning 1991] used a form of type inference similar to abstract interpretation; the later systems SML-CIDRE [Davies and Pfenning 2000; Davies 2005] and Stardust [Dunfield 2007] used bidirectional typing. SML-CIDRE eschewed type inferencewe in favour of bidirectional typing: type inference finds *all* behaviours, not only the *intended* behaviours. The type annotations in bidirectional typing, especially when following the Pfenning recipe as SML-CIDRE did, force programmers to write down the behaviours they intend. In Stardust, bidirectional typing was also motivated by the undecidability of inference for index refinements.

In contextual modal type theory [Nanevski et al. 2008], typing is bidirectional to make type checking decidable in the presence of dependent types. That theory is the main foundation for Beluga, which is bidirectional for the same reason. The original core of Beluga [Pientka 2008; Pientka and Dunfield 2008] follows the Pfenning recipe, but the full system [Pientka 2013] extends the recipe, supporting both checking and synthesis for spines (lists of function arguments).

Bidirectional type checking was folklore among programming language developers since the 1980s, and was known and used by developers of proof assistants since the 1990s: Coquand [1996] presents a type-checking algorithm for a small dependent type system (dependent products, plus let-expressions), in which "type-checking" $(\ldots M \Rightarrow v)$ and "type inference" $(\ldots M \mapsto v)$ are defined "inductively and simultaneously." Coquand's version of a subsumption rule [Coquand 1996, p. 173, third part of definition] says that M checks against v if M synthesizes w and w is convertible to v. Moreover, by removing the parts related to dependent typing, we see that Coquand's rule for λ-application (the fifth part of his definition) is essentially our standard rule: to synthesize a type for $M_1\, M_2$, synthesize a type for M_1 and check M_2 against the domain of that type. Strikingly, the Gofer code in the paper has functions `checkExp` and `inferExp` that have the expected type signatures; for example, `checkExp` returns a Boolean.

Scherer and Abel [2012] give a bidirectional algorithm for judgmental equality in dependent type theory. They give a non-bidirectional specification of a dependent type system. Then they

give a bidirectional algorithmic system, which defines where to do equality tests and leads to a nice algorithm for how to do judgmental equality. They prove soundness, completeness, and decidability of their algorithm using two intricate logical relations; the combination of soundness and completeness leads to decidability. Judgmental equality is bidirectional: comparison of neutral terms synthesizes, and comparison of normal terms checks. This cleverly exploits the fact that off-diagonal cases (e.g., atomic terms compared against a normal terms) can be omitted, as an atomic term t can only equal another term t' if t' reduces to another term with the same head variable as t. Bidirectional formulations of judgment equality are not original to Scherer and Abel [2012]; for example, Harper and Pfenning [2005] use a similar idea for LF. We focus on Scherer and Abel [2012], because their techniques can scale up to universes and large eliminations.

McBride [2016] advocates a bidirectional system as the specification of a dependent type system. Other bidirectional dependent type systems include PiSigma [Altenkirch et al. 2010], intended as a small core system for dependently typed languages, and Zombie [Sjöberg and Weirich 2015]. As with subtyping, conversion checking is not syntax-directed and is guided by bidirectionality. Taking the bidirectional system as the specification simplifies the metatheory in some ways. However, proving that an algorithmic conversion relation is an equivalence relation, congruent for all the syntactic forms, still seems to require a sophisticated argument.

Intersection types, originally formulated in undecidable type assignment systems, have motivated the use of bidirectional typing in several systems, including refinement intersection types [Dunfield and Pfenning 2004] and unrestricted intersection types [Dunfield 2014; Oliveira et al. 2016] with polymorphism [Alpuim et al. 2017]. Some of these systems also include union types.

9.2 Gradual Typing

Gradual typestate [Wolff et al. 2011] uses bidirectional typing to structure the flow of information about access permissions, specified in annotations. Their language, descended from Featherweight Java, is imperative in flavour; its expression forms are not easy to classify as introductions or eliminations, making it hard to apply the Pfenning recipe. Our discussion of reasoning by cases (step 3 in Section 4.1) carries over to their typing rules for `let`, which allow either (1) the body of the `let` to synthesize, and hence the entire `let`, or (2) the body to be checked, based on a type against which the entire `let` is to be checked. (Our judgment form $\Gamma \vdash \cdots \dashv \Delta$ from Section 5 looks similar to the gradual typestate judgment $\Delta \vdash \cdots \dashv \Delta'$; moreover, in both settings, the left-hand context is called the input context and the right-hand context is called the output context. However, for gradual typestate, the output context describes the *state* after running the subject expression, so the output context can carry different information from the input context.)

Gradual sum types [Jafery and Dunfield 2017] are formulated in a functional style, so the Pfenning recipe works. The subformula property ensures that uncertain types—connectives that relax the guarantees of static typing—appear only when the programmer asks for them. In their subsumption rule (ChkCSub), the subtyping judgment is replaced by *directed consistency*, a relation that contains subtyping but also allows shifts between more precise (less uncertain, more static) and less precise (more uncertain, less static) types.

Xie et al. [2018] develop a gradual type system with consistent subtyping (related to directed consistency) and higher-rank polymorphism. Their bidirectional system closely follows Dunfield and Krishnaswami [2013], discussed in Section 5.2; this approach leads to a subformula property that, as in Jafery and Dunfield [2017], ensures that the unknown type appears only by programmer request.

9.3 Other Work

Çiçek et al. [2019] define a relational type system where each judgment has two subject terms (expressions): $\Gamma \vdash e_1 \backsim e_2 : \tau$ relates the terms e_1 and e_2 at type τ. Their bidirectionalization follows the Pfenning recipe in its original form, for example, their rule **alg-r-if** for if expressions has a checking conclusion.

10 HISTORICAL NOTES

Pierce and Turner's paper "Local Type Inference"—which appeared as a technical report (1997), at POPL (1998) and in TOPLAS (2000)—is the earliest frequently cited paper on bidirectional typing, but Pierce noted that "John Reynolds first acquainted us [BCP] with the idea of bidirectional typechecking around 1988."

That year also saw the first version of the report on Forsythe [Reynolds 1988], where Reynolds noted that intersection types would require some type information in source programs. The second version of the report [Reynolds 1996, Appendix C] describes an algorithm that combines "bottom-up" and "top-down" type checking, but the precise connection to bidirectional typing is not clear to us.

Lee and Yi [1998] present Algorithm $\mathcal{M}$, a type inference algorithm used in some versions of Caml Light (the predecessor of OCaml). In contrast to Algorithm W [Milner 1978], if we reformulate Algorithm $\mathcal{M}$ as a set of typing rules, then they are all checking rules: There is an input type ρ to check against. In some situations—for example, when typing a let-bound expression—the input type is a fresh unification variable, but the input type often carries information. They show that $\mathcal{M}$ can find type errors earlier than W, which is consistent with the idea that bidirectional checking provides better error messages.

Dunfield and Pfenning [2004] has two authors, but the recipe was invented by Frank Pfenning, so we call it the Pfenning recipe.

11 SUMMARY OF BIDIRECTIONAL TYPING NOTATION

Table 1 summarizes some of the symbols that have been used to denote checking and synthesis. Until about 2008, most authors used vertical arrows ($\downarrow$ for checking and $\uparrow$ for synthesis), though Pierce and Turner [2000] used $\overleftarrow{\in}$ for checking and $\overrightarrow{\in}$ for synthesis. The arrows were meant to represent information flow, but vertical arrows are unclear, because syntax trees and derivation trees put the root at opposite ends: does $e \uparrow A$ mean that the type flows from a leaf of a syntax tree (at the bottom, away from the root), or from the conclusion of a derivation tree?

Horizontal arrows avoid this confusion: nearly all authors write the subject term to the left of the type in a judgment, so $e \Rightarrow A$ means that the type is flowing from the term and $e \Leftarrow A$ means that the type is flowing "into" the term.

The $\ni$/$\in$ notation [McBride 2016] has the advantage that information always flows left to right.

12 CONCLUSION

Bidirectional typing is usually straightforward to implement. However, while the bidirectional approach allows us to prove soundness, completeness and decidability of state-of-the-art typing algorithms, the proofs are often extremely involved. Moreover, the proofs are about type systems that focus on a few features of research interest. Since realistic programming languages combine many typing features, doing the metatheory for a full-scale type system seems intractable using current techniques. A key challenge for future research is to discover techniques to simplify the proofs of soundness, completeness, and decidability, enabling researchers to prove these key properties for complete programming languages.

Table 1. Historical and Recent Notation

	Checks against	Synthesizes
Coquand [1996]	$\Rightarrow$	$\mapsto$
Pierce and Turner [2000]	$\overleftarrow{\in}$	$\overrightarrow{\in}$
Xi and Pfenning [1999]; Davies and Pfenning [2000]; Dunfield and Pfenning [2004]; Polikarpova et al. [2016]; Çiçek et al. [2019]	$\downarrow$	$\uparrow$
Chlipala et al. [2005]; Pottier and Régis-Gianas [2006]; Dunfield [2009]	$\Downarrow$	$\Uparrow$
Peyton Jones et al. [2007]	$\vdash_{\Downarrow}$	$\vdash_{\Uparrow}$
Davies [2005]	$\overleftarrow{\in}$	$\overrightarrow{\in}$
Nanevski et al. [2008]; Pientka [2008], Pientka and Dunfield [2008]; Wolff et al. [2011]; Dunfield [2012, 2014, 2015]; Dunfield and Krishnaswami [2013]; Oliveira et al. [2016]; Jafery and Dunfield [2017]; Xie and Oliveira [2018]; Xie et al. [2018]	$\Leftarrow$	$\Rightarrow$
McBride [2016]	$A \ni e$	$e \in A$
Lindley et al. [2017]	$e : A$	$e \Rightarrow A$

Another future research direction is that polarity, focusing and call-by-push-value seem to fit nicely with bidirectional typing, but are not the same as bidirectional typing. We would like to understand the actual relationship between these ideas, making it easier to integrate advances in proof theory into language design.

Designers of bidirectional type systems often employ information flow that goes beyond checking (where the entire type is input) and synthesis (where the entire type is output). We would like to have a better understanding of how such information flow interacts with core metatheoretic properties, such as substitution principles, to arrive at broadly applicable design principles for type system design.

ACKNOWLEDGMENTS

We thank Michael Arntzenius, Dimitrios J. Economou, Ronald Garcia, Dionna Glaze, and Max New for discussions about the ideas in this article and feedback on earlier versions. We also thank the anonymous reviewers for their thorough reading and suggestions.

REFERENCES

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *Proceedings of the European Symposium on Programming*. Springer, 1–28.

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and N. Oury. 2010. ΠΣ: Dependent types without the sugar. In *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS'10)*. Springer, 40–55.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment. *J. Symbol. Logic* 48, 4 (1983), 931–940.

Luca Cardelli. 1993. *An Implementation of $F_{<:}$*. Research report 97. DEC/Compaq Systems Research Center.

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'19)*. ACM Press, 533–547.

Iliano Cervesato and Frank Pfenning. 2003. A linear spine calculus. *J. Logic Comput.* 13, 5 (2003), 639–688.

Adam Chlipala, Leaf Petersen, and Robert Harper. 2005. Strict bidirectional type checking. In *Proceedings of the Workshop on Types in Language Design and Implementation (TLDI '05)*. ACM Press, 71–78.

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Sci. Comput. Program.* 26, 1–3 (1996), 167–177.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'82)*. ACM, 207–212.

Rowan Davies. 2005. *Practical Refinement-Type Checking*. Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-05-110.

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the International Conference on Functional Programming (ICFP'00)*. ACM Press, 198–208.

Stephen Dolan. 2016. *Algebraic Subtyping*. Ph.D. Dissertation. University of Cambridge.

Jana Dunfield. 2007. *A Unified System of Type Refinements*. Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-07-129.

Jana Dunfield. 2009. Greedy bidirectional polymorphism. In *Proceedings of the Machine Learning Workshop (ML'09)*. ACM Press, 15–26. Retrieved from http://research.cs.queensu.ca/~jana/papers/poly/.

Jana Dunfield. 2012. Elaborating intersection and union types. In *Proceedings of the International Conference on Functional Programming (ICFP'12)*. ACM Press, 17–28.

Jana Dunfield. 2014. Elaborating intersection and union types. *J. Functional Programming* 24, 2–3 (2014), 133–165.

Jana Dunfield. 2015. Elaborating evaluation-order polymorphism. In *Proceedings of the International Conference on Functional Programming (ICFP'15)*. ACM Press, 256–268. arXiv:1504.07680 [cs.PL].

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the International Conference on Functional Programming (ICFP'13)*. ACM Press, 429–442. arXiv:1306.6032 [cs.PL].

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proc. ACM Program. Lang.* 3, Article 9 (Jan. 2019), 28 pages. arXiv:1601.05106 [cs.PL].

Jana Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'04)*. ACM Press, 281–292.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible type application. In *Proceedings of the European Symposium on Programming*, Vol. 9632. Springer, 229–254.

José Espírito Santo. 2017. The polarized λ-calculus. *Electronic Notes Theor. Comput. Sci.* 332 (2017), 149–168.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Programming Language Design and Implementation*. ACM Press, 268–277.

Jean-Yves Girard. 1989. *Proofs and Types*. Cambridge University Press.

Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In *Proceedings of the Conference on Mathematically Structured Functional Programming (MSFP'10)*.

Robert Harper and Frank Pfenning. 2005. On equivalence and canonical forms in the LF type theory. *Trans. Comput. Logic* 6 (2005), 61–101. Issue 1.

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

Haruo Hosoya and Benjamin C. Pierce. 1999. *How Good is Local Type Inference?* Technical Report MS-CIS-99-17. University of Pennsylvania.

Danko Ilik. 2017. The exp-log normal form of types: Decomposing extensional equality and representing terms compactly. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. ACM Press, 387–399.

Khurram A. Jafery and Jana Dunfield. 2017. Sums of uncertainty: Refinements go gradual. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. ACM Press, 804–817.

Trevor Jim. 1995. *What are Principal Typings and What are They Good For?* Technical memorandum MIT/LCS/TM-532. MIT.

Neelakantan R. Krishnaswami. 2009. Focusing on pattern matching. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'09)*. ACM Press, 366–378. https://doi.org/10.1145/1480881.1480927

Joachim Lambek. 1985. Cartesian closed categories and typed lambda-calculi. In *Proceedings of the 13th Spring School of the LITP on Combinators and Functional Programming Languages (LNCS, Vol. 242)*, Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet (Eds.). Springer, 136–175. https://doi.org/10.1007/3-540-17184-3_44

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a Folklore let-polymorphic type inference algorithm. *ACM Trans. Prog. Lang. Sys.* 20, 4 (July 1998), 707–723.

Daniel Leivant. 1986. Typing and computational properties of lambda expressions. *Theor. Comput. Sci.* 44 (1986), 51–68.

Paul Blain Levy. 2001. *Call-By-Push-Value*. Ph.D. Dissertation. Queen Mary and Westfield College, University of London.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. ACM Press, 500–514.

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Sys.* 16, 6 (Nov. 1994), 1811–1841.

Conor McBride. 2016. I got plenty o' nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, S. Lindley, C. McBride, P. Trinder, and D. Sannella (Eds.). Springer, 207–233.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages.

Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored local type inference. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 41–53.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the International Conference on Functional Programming (ICFP'16)*. ACM Press, 364–377.

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

Frank Pfenning. 2004. Sequent Calculus. Lecture notes for 15–317: Constructive Logic, Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/fp/courses/atp/handouts/ch3-seqcalc.pdf.

Frank Pfenning. 2009. Lecture Notes on Harmony. Lecture notes for 15–317: Constructive Logic, Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/~fp/courses/15317-f09/lectures/03-harmony.pdf.

Frank Pfenning. 2017. Lecture Notes on Verifications. Lecture notes for 15–317: Constructive Logic, Carnegie Mellon University. Retrieved from https://www.cs.cmu.edu/~crary/317-f18/lectures/05-intercalation.pdf.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Math. Struct. Comput. Sci.* 11, 4 (2001), 511–540.

Frank Pfenning and Robert J. Simmons. 2009. Substructural operational semantics as ordered logic programming. In *Proceedings of the Symposium on Logic in Computer Science (LICS'09)*. IEEE, 101–110.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, 371–382.

Brigitte Pientka. 2013. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. *J. Function. Program.* 23, 1 (2013), 1–37. https://doi.org/10.1017/S0956796812000408

Brigitte Pientka and Jana Dunfield. 2008. Programming with proofs and explicit contexts. In *Proceedings of the Conference on Principles and Practice of Declarative Programming (PPDP'08)*. ACM Press, 163–173.

Benjamin C. Pierce and David N. Turner. 1997. *Local type inference*. Technical Report CSCI #493. Indiana University.

Benjamin C. Pierce and David N. Turner. 1998. Local type inference. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 252–265. Full version in *ACM Trans. Prog. Lang. Sys.*, 22(1):1–44, 2000.

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

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation*. ACM Press, 522–538.

François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'06)*. ACM Press, 232–244.

Uday S. Reddy. 1993. A typed foundation for directional logic programming. In *Extensions of Logic Programming*, E. Lamma and P. Mello (Eds.). Springer, 282–318.

John C. Reynolds. 1988. *Preliminary Design of the Programming Language Forsythe*. Technical Report CMU-CS-88-159. Carnegie Mellon University. http://doi.library.cmu.edu/10.1184/OCLC/18612825

John C. Reynolds. 1996. *Design of the Programming Language Forsythe*. Technical Report CMU-CS-96-146. Carnegie Mellon University.

Gabriel Scherer. 2017. Deciding equivalence with sums and the empty type. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. ACM Press, 374–386.

Gabriel Scherer and Andreas Abel. 2012. On irrelevance and algorithmic equality in predicative type theory. *Logic. Methods Comput. Sci.* 8 (2012), 1–29.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, Article 89 (2020), 29 pages. https://doi.org/10.1145/3408971

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM Press, 783–796. https://doi.org/10.1145/3192366.3192389

Wilfried Sieg and John Byrnes. 1998. Normal natural deduction proofs (in classical logic). *Studia Logica* 60, 1 (1998), 67–106.

Robert J. Simmons. 2014. Structural focalization. *ACM Trans. Comput. Logic* 15, 3, Article 21 (Sept. 2014), 33 pages.

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to congruence. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'15)*. ACM Press, 369–382.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Program.* 21, 4–5 (2011), 333–412.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: Inference for higher-rank types and impredicativity. In *Proceedings of the International Conference on Functional Programming (ICFP'06)*. ACM Press, 251–262. https://doi.org/10.1145/1159803.1159838

David H. D. Warren. 1977. *Applied Logic—Its use and implementation as a programming tool*. Ph.D. Dissertation. University of Edinburgh.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2003. *A concurrent logical framework I: Judgments and properties*. Technical Report CMU-CS-02-101. Carnegie Mellon University.

J. B. Wells. 2002. The essence of principal typings. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Springer, 913–925.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer, 459–483.

Hongwei Xi. 1998. *Dependent Types in Practical Programming*. Ph.D. Dissertation. Carnegie Mellon University.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, 224–235.

Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'99)*. ACM Press, 214–227.

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent subtyping for all. In *Proceedings of the European Symposium on Programming*. Springer, 3–30.

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let arguments go first. In *Proceedings of the European Symposium on Programming*. Springer, 272–299.

Noam Zeilberger. 2015. Balanced polymorphism and linear lambda calculus. In *Proceedings of the International Conference on Types for Proofs and Programs (TYPES'15)*.

Noam Zeilberger. 2018. A theory of linear typings as flows on 3-valent graphs. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*. ACM Press, 919–928. https://doi.org/10.1145/3209108.3209121

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.* 3, Article 112 (July 2019), 29 pages. https://doi.org/10.1145/3341716

Received August 2019; revised February 2021; accepted February 2021