A Theory of Bidirectional Type Synthesis for Simple Types

ANONYMOUS AUTHOR(S)

There has been much progress in designing bidirectional type systems and associated type synthesis algorithms, but only on a case-by-case basis. This situation is in stark contrast to parsing, for which there have been general theories and widely applicable tools such as parser generators. To remedy the situation, this paper develops a first theory of bidirectional type synthesis. The whole theory works as a verified type-synthesiser generator for syntax-directed simple type systems: From a signature that specifies a simply typed language with a bidirectional type system, the theory produces a type synthesiser that decides whether an input abstract syntax tree has a typing derivation or reports that the input does not have enough type annotations. Within the theory, we formally define soundness, completeness, and mode-correctness, which are sufficient conditions for deriving a correct type synthesiser. We also propose a preprocessing step called 'mode decoration', which helps the user to deal with missing type annotations and streamlines the theory. The entire theory is formulated constructively and has been formalised in the proof assistant Agda with Axiom K.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Syntax**; *Formal software verification*; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: datatype-generic programming, language formalisation, Agda

ACM Reference Format:
Anonymous Author(s). 2023. A Theory of Bidirectional Type Synthesis for Simple Types. *Proc. ACM Program. Lang.* 0, 0, Article 0 (July 2023), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Type inference is an important mechanism for the transition to well-typed programs from untyped abstract syntax trees, which we call *raw terms*. Here 'type inference' refers specifically to algorithms that ascertain the type of any raw term *without type annotations*. However, it was later found that full parametric polymorphism leads to undecidability in type inference, as do dependent types [Dowek 1993; Wells 1999]. In light of these limitations, bidirectional type synthesis emerged as a viable alternative, providing algorithms for deciding the types of raw terms that meet some syntactic criteria and usually contain some type annotations. Dunfield and Krishnaswami [2021] summarised the design principles of bidirectional type synthesis and its wide coverage of languages with simple types, polymorphic types, dependent types, gradual types, among others.

The basic idea of bidirectional type synthesis is that while the problem of type inference is not decidable in general, for certain kinds of terms it is still possible to infer their types (for example, the type of a variable can be looked up in the context); for other kinds of terms, we can switch to the simpler problem of type checking, where the expected type of a term is also given so that there is more information to work with. More formally, every judgement in a bidirectional type system is extended with a *mode:* (1) $\Gamma \vdash t \Rightarrow A$ for *synthesis* and (2) $\Gamma \vdash t \Leftarrow A$ for *checking*. The former indicates that the type A is computed as output, using both the context Γ and the term t as input, while for the latter, all three of Γ, t, and A are given as input. The algorithm of a bidirectional type synthesiser can usually be 'read off' from a well-designed bidirectional type system: as the

2475-1421/2023/7-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

synthesiser traverses a raw term, it switches between synthesis and checking, following the modes assigned to the judgements in the typing rules. As Pierce and Turner [2000] noted, bidirectional type synthesis propagates type information locally within adjacent nodes of a term, and does not require unification as Damas–Milner type inference does. Moreover, introducing long-distance unification constraints would even undermine the essence of locality in bidirectional type synthesis.

Despite sharing the same basic idea, bidirectional typing has only been developed on a case-by-case basis without a theory; this situation is in stark contrast to parsing, which has general theories together with widely applicable techniques and practical tools, notably parser generators.[1] For bidirectional typing, Dunfield and Krishnaswami [2021] could only present informal design principles that the community learned from individual systems, rather than a general theory providing logical specifications and rigorously proven properties for a class of systems. Moreover, unlike the plethora of available parser generators, 'type-synthesiser generators' rarely exist, so each type synthesiser has to be independently built from scratch. To remedy the situation, we initiate a theory of bidirectional typing, which amounts to a verified type-synthesiser generator for syntax-directed simple type systems.

1.1 Background and Related Work

To explain our contributions, first we discuss related work upon which our work is built.

1.1.1 Bidirectional type synthesis in Martin-Löf Type Theory. Our work is partly inspired by Wadler et al.'s [2022] textbook, where programming language concepts are formally introduced using the proof assistant Agda based on Martin-Löf Type Theory, and one particular topic is bidirectional type synthesis for PCF. Their treatment deviates from existing literature: Traditionally, a type synthesis algorithm is presented as *algorithmic system* relations such as $\Gamma \vdash t \Rightarrow A \mapsto t'$, denoting that annotations can be added to t in the surface language to produce t' of type A in the core language. Such an algorithm is then accompanied with *soundness* and *completeness* assertions such that the algorithm correctly synthesises the type of a raw term and every typable term can be synthesised if there are enough annotations. By contrast, Wadler et al. exploit the simultaneously computational and logical nature of Martin-Löf type theory to formulate algorithmic soundness, completeness, and decidability in one go. Recall that the law of excluded middle $P + \neg P$ does not hold as an axiom for every P in Martin-Löf type theory, and we say P is *decidable* if the law holds for P. For example, a proof of *decidable equality*, i.e. $\forall x, y.\ (x = y) + (x \neq y)$, decides whether x and y are equal and accordingly gives an identity proof or a refutation explicitly; such a decidability proof may or may not be possible depending on the domain of x and y, and is non-trivial. Given that all proofs as programs terminate, logical decidability implies *algorithmic decidability*. Further, suppose a proof of the following statement:

> 'For a context Γ and a raw term t, either a typing derivation of $\Gamma \vdash t : A$ exists for some type A or any derivation of $\Gamma \vdash t : A$ for some type A leads to a contradiction'

or rephrased succinctly as

> 'It is *decidable* for any Γ and t whether $\Gamma \vdash t : A$ has a derivation for some A'.

It yields either a typing derivation for the given raw term t or a contradiction proof that such a derivation is impossible where the former case is algorithmic soundness and the latter algorithmic completeness in contrapositive form. Then, both algorithmic soundness and completeness in original form are implied. Our work takes the same approach but expands the scope significantly.

[1] The same could even be said for type checking in general, not just for bidirectional type systems. While there are some efforts to generate type checkers grounded in unification [Gast 2004; Grewe et al. 2015], it should be noted that unification-based approaches are not suited to more complex type theories. Moreover, their algorithms are not proved complete.

1.1.2 Language formalisation frameworks. The vision of formalising the metatheory of every programming language was initiated by the PoplMark challenge [Aydemir et al. 2005]. Earlier than PoplMark, Altenkirch [1993] commented that basic meta-operations and their meta-properties constitute the bulk of formalisation, motivating a number of frameworks [Ahrens et al. 2018, 2022; Allais et al. 2021; Fiore and Szamozvancev 2022; Gheri and Popescu 2020] to define well-scoped/typed terms, substitution, term traversal, and their meta-properties universally (in the sense of universal algebra). One of the core gadgets of these frameworks is the notion of *binding signatures* (coined by Aczel [1978] in line with the term *signature* in universal algebra) for specifying type systems. It is noteworthy that these state-of-the-art frameworks are basically restricted to simple types and cannot specify polymorphic types or dependent types yet, and only Allais et al. discussed meta-operations beyond substitution. Like these frameworks, our work is built on a variant of binding signature for simple types. Unlike these frameworks, we target constructions for a different part of programming language theory, namely the transition from a surface language (raw terms) to a core language (well-typed terms), and our type synthesiser is significantly more complex than those meta-operations previously considered. Another difference is that several of the frameworks [Ahrens et al. 2022; Allais et al. 2021; Fiore and Szamozvancev 2022] focus on intrinsically typed terms, whereas for our theory it is essential to use an extrinsic formulation to make it possible to relate a synthesised typing derivation with an input term. This difference can be easily reconciled by converting extrinsic typing derivations to intrinsically typed terms though.

1.2 Contributions and Plan of This Paper

Instead of focusing solely on a bidirectional type synthesiser like Wadler et al. [2022] (Section 1.1.1), our theory centres around a general specification of type synthesis for an ordinary type system where judgements are not assigned modes, and fills in the gap between language formalisation frameworks based on such type systems (Section 1.1.2) and raw terms produced by parsers. Working similarly to those frameworks, our theory is parametrised by a *bidirectional binding signature*, which extends Aczel's binding signature and describes a language with a bidirectional type system, analogously to a grammar describing a formal language in the theory of parsing. As a first step, we confine our theory to *syntax-directed* systems, meaning that for every raw term construct, the ordinary type system has exactly one typing rule, which is assigned modes in exactly one way in the bidirectional type system.[2] With this assumption, using a single bidirectional binding signature we can specify the raw terms, the ordinary type system, and the bidirectional type system of a language. Then, if the signature is *mode-correct* [Dunfield and Krishnaswami 2021, Section 3.1], we can instantiate a type synthesiser that is formulated similarly to Wadler et al.'s [2022] with respect to the ordinary type system, and performs bidirectional type synthesis internally. Our theory thus works as a verified type-synthesiser generator, analogous to a parser generator.

Within the theory, we formulate essential conditions including *soundness*, *completeness*, and *mode-correctness* that are sufficient to derive a bidirectional type synthesiser and ensure that it implements the general specification, formalising what were only informally outlined by Dunfield and Krishnaswami [2021]. We also pay attention to the problem of whether a raw term has enough type annotations, often overlooked in the literature, by introducing *mode decoration*, which assigns a mode to a raw term with enough annotations or otherwise pinpoints missing annotations. Practically, by running a mode decorator, the user can first make sure that a raw term has enough annotations, and then deal with typing issues without worrying about missing annotations anymore. Theoretically, separating the annotation and typing aspects streamlines our theory, in particular

[2]It is reasonable to assume syntax-directedness (especially as a first step) given that it is a standard idea to re-cast non-syntax-directed typing rules into their syntax-directed form to derive a type synthesiser [Peyton Jones et al. 2007].

$\boxed{V \vdash t}$ Given a list V of variables, t is a raw term with free variables in V

$$\frac{x \in V}{V \vdash x}\,\textsc{Var} \qquad \frac{V \vdash t}{V \vdash (t \mathbin{⨾} A)}\,\textsc{Anno} \qquad \frac{V, x \vdash t}{V \vdash \lambda x.\, t}\,\textsc{Abs} \qquad \frac{V \vdash t \qquad V \vdash u}{V \vdash t\ u}\,\textsc{App}$$

Fig. 1. Raw terms for simply typed λ-calculus

$\boxed{\Gamma \vdash t : A}$ A raw term t has type A under context Γ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\,\textsc{Var} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash (t \mathbin{⨾} A) : A}\,\textsc{Anno} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \supset B}\,\textsc{Abs} \qquad \frac{\Gamma \vdash t : A \supset B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\,\textsc{App}$$

Fig. 2. Typing derivations for simply typed λ-calculus

allowing us to formulate completeness naturally and differently from *annotatability*, which was proposed and conflated with completeness by Dunfield and Krishnaswami [2021]. By combining bidirectional type synthesis, soundness, completeness, and mode decoration, we achieve our main result (Corollary 5.10): a *trichotomy* on raw terms, which is computationally a type synthesiser that checks if a given raw term is sufficiently annotated and if a sufficiently annotated raw term has an ordinary typing derivation, suggesting that type synthesis should not be viewed as a bisection between typable and untypable raw terms but a trichotomy in general.

In summary, we contribute a theory of bidirectional type synthesis that is

(1) *simple yet general*, working for any syntax-directed bidirectional simple type system that can be specified by a bidirectional binding signature;
(2) *constructive* based on Martin-Löf type theory, and compactly formulated in a way that unifies computation and proof, preferring logical decidability over algorithmic soundness, completeness, and decidability; and mode decoration over annotatability.

Our theory was first developed in Agda and then translated to the mathematical vernacular.

This paper is structured as follows. We first present a concrete overview of our theory using simply typed λ-calculus in Section 2, prior to developing a general framework for specifying bidirectional type systems in Section 3. Following this, we discuss mode decoration and related properties in Section 4. The most heavyweight technical contribution lies in Section 5, where we introduce mode-correctness and bidirectional type synthesis. Some examples other than simply typed λ-calculus are given in Section 6. We briefly sketch the design of our formalisation in Section 7, and conclude in Section 8 with potential challenges in further developing our theory.

2 BIDIRECTIONAL TYPE SYNTHESIS FOR SIMPLY TYPED λ-CALCULUS

We start with an overview of our theory by instantiating it to simply typed λ-calculus. Roughly speaking, the problem of type synthesis requires us to take a raw term (i.e. an untyped abstract syntax tree) as input, and produce a typing derivation for the term if possible. To give more precise definitions: The raw terms for simply typed λ-calculus are defined[3] in Figure 1; besides the standard constructs, there is an Anno rule that allows the user to insert type annotations to facilitate type synthesis. Correspondingly, the definition of typing derivations[4] in Figure 2 also includes an Anno

[3]We omit the usual conditions about named representations of variables throughout this paper.
[4]We write '⊃' instead of '→' for the function types of simply typed λ-calculus to avoid confusion with the function types in our type-theoretic meta-language.

$\boxed{\Gamma \vdash t \Rightarrow A}$ A raw term t synthesises a type A under Γ

$\boxed{\Gamma \vdash t \Leftarrow A}$ A raw term t checks against a type A under Γ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\Gamma \vdash t \Leftarrow A}{\Gamma \vdash (t : A) \Rightarrow A}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{\Gamma \vdash t \Rightarrow B \qquad B = A}{\Gamma \vdash t \Leftarrow A}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{\Gamma, x : A \vdash t \Leftarrow B}{\Gamma \vdash \lambda x.\, t \Leftarrow A \supset B}\ \textsc{Abs}^{\Leftarrow} \qquad \frac{\Gamma \vdash t \Rightarrow A \supset B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\ u \Rightarrow B}\ \textsc{App}^{\Rightarrow}$$

Fig. 3. Bidirectional typing derivations for simply typed λ-calculus

rule enforcing that the type of an annotated term does match the annotation. Now we can define what it means to solve the type synthesis problem.

Definition 2.1. Parametrised by an 'excuse' predicate E on raw terms, a *type synthesiser* takes a context Γ and a raw term $|\Gamma| \vdash t$ (where $|\Gamma|$ is the list of variables in Γ) as input, and establishes one of the following outcomes:

(1) there exists a derivation of $\Gamma \vdash t : A$ for some type A,
(2) there does not exist a derivation $\Gamma \vdash t : A$ for any type A, or
(3) E holds for t.

It is crucial to allow the third outcome, without which we would be requiring the type synthesis problem to be decidable, but this requirement would quickly become impossible to meet when the theory is extended to handle more complex types. If a type synthesiser cannot decide whether there is a typing derivation, it is allowed to give an excuse instead of an answer. Acceptable excuses are defined by the predicate E, which describes what is wrong with an input term, for example not having enough type annotations.

Now our goal is to use Definition 2.1 as a specification and implement it using a *bidirectional* type synthesiser, which attempts to produce *bidirectional* typing derivations defined in Figure 3. It is often said that a type synthesis algorithm can be 'read off' from well-designed bidirectional typing rules. Take the $\textsc{App}^{\Rightarrow}$ rule as an example: to synthesise the type of an application $t\ u$, we first synthesise the type of t, which should have the form $A \supset B$, from which we can extract the expected type of u, namely A, and perform checking; then the type of the whole application, namely B, can also be extracted from the type $A \supset B$. Note that the synthesiser is able to figure out the type A for checking u and the type B to be synthesised for $t\ u$ because they have been computed when synthesising the type $A \supset B$ of t. In general, there should be a flow of type information in each rule that allows us to determine unknown types (e.g. types to be checked) from known ones (e.g. types previously synthesised). This is called *mode-correctness*, which we will formally define in Section 5.1.

While it is possible for a bidirectional type synthesiser to do its job in one go, which can be thought of as adding both mode and typing information to a raw term and arriving at a bidirectional typing derivation, it is beneficial to have a preprocessing step which adds only mode information, based on which the synthesiser then continues to add typing information. More precisely, the preprocessing step, which we call *mode decoration*, attempts to produce *mode derivations* as defined in Figure 4, where the rules are exactly the mode part of the bidirectional typing rules (Figure 3).

Definition 2.2. A *mode decorator* decides for any raw term $V \vdash t$ whether $V \vdash t^{\Rightarrow}$.

One (less important) benefit of mode decoration is that it helps to simplify the synthesiser, whose computation can be partly directed by a mode derivation. More importantly, whether there is a

$\boxed{V \vdash t^{\Rightarrow}}$ A raw term t (with free variables in V) is in synthesising mode

$\boxed{V \vdash t^{\Leftarrow}}$ A raw term t (with free variables in V) is in checking mode

$$\frac{x \in V}{V \vdash x^{\Rightarrow}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{V \vdash t^{\Leftarrow}}{V \vdash (t : A)^{\Rightarrow}}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{V \vdash t^{\Rightarrow}}{V \vdash t^{\Leftarrow}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, x \vdash t^{\Leftarrow}}{V \vdash (\lambda x.\, t)^{\Leftarrow}}\ \textsc{Abs}^{\Leftarrow} \qquad \frac{V \vdash t^{\Rightarrow} \qquad V \vdash u^{\Leftarrow}}{V \vdash (t\ u)^{\Rightarrow}}\ \textsc{App}^{\Rightarrow}$$

Fig. 4. Mode derivations for simply typed λ-calculus

mode derivation for a term is actually very useful information to the user, because it corresponds to whether the term has enough type annotations: Observe that the $\textsc{Anno}^{\Rightarrow}$ and $\textsc{Sub}^{\Leftarrow}$ rules allow us to switch between the synthesising and checking modes; the switch from synthesising to checking is free, whereas the opposite direction requires a type annotation. That is, any term in synthesising mode is also in checking mode, but not necessarily vice versa. A type annotation is required wherever a term that can only be in checking mode is required to be in synthesising mode, and a term does not have a mode derivation if and only if type annotations are missing in such places. (We will treat all these more rigorously in Section 4.) For example, an abstraction is strictly in checking mode, but the left sub-term of an application has to be synthesising, so a term of the form $(\lambda x.\, t)\ u$ does not have a mode derivation unless we annotate the abstraction.

Perhaps most importantly, mode derivations enable us to give bidirectional type synthesisers a tight definition: if we restrict the domain of a synthesiser to terms in synthesising mode (i.e. having enough type annotations for performing synthesis), then it is possible for the synthesiser to *decide* whether there is a suitable typing derivation.

Definition 2.3. A *bidirectional type synthesiser* decides for any context Γ and synthesising term $|\Gamma| \vdash t^{\Rightarrow}$ whether $\Gamma \vdash t \Rightarrow A$ for some type A.

Now we can get back to our goal of implementing a type synthesiser (Definition 2.1).

Theorem 2.4. *A type synthesiser that uses 'not in synthesising mode' as its excuse can be constructed from a mode decorator and a bidirectional type synthesiser.*

The construction is straightforward: Run the mode decorator on the input term $|\Gamma| \vdash t$. If there is no synthesising mode derivation, report that t is not in synthesising mode (the third outcome). Otherwise $|\Gamma| \vdash t^{\Rightarrow}$, and we can run the bidirectional type synthesiser. If it finds a derivation of $\Gamma \vdash t \Rightarrow A$ for some type A, return a derivation of $\Gamma \vdash t : A$ (the first outcome), which is possible because the bidirectional typing (Figure 3) is *sound* with respect to the original typing (Figure 2).

Lemma 2.5 (Soundness). *If* $\Gamma \vdash t \Rightarrow A$, *then* $\Gamma \vdash t : A$.

If there is no derivation of $\Gamma \vdash t \Rightarrow A$ for any type A, report that there is no derivation of $\Gamma \vdash t : A$ for any A either (the second outcome), because the bidirectional typing is *complete* with respect to the original typing.

Lemma 2.6 (Completeness). *If* $|\Gamma| \vdash t^{\Rightarrow}$ *and* $\Gamma \vdash t : A$, *then* $\Gamma \vdash t \Rightarrow A$.

We will construct a mode decorator (Section 4.2) and a bidirectional type synthesiser (Section 5) and prove both lemmas (Section 4.1) for all syntax-directed bidirectional simple type systems. To quantify over all such systems, we need their generic definitions, which we formulate next.

$\boxed{\Xi \vdash_\Sigma A}$ A is a type with type variables in Ξ for a signature Σ

$$\frac{X_i \in \Xi}{\Xi \vdash_\Sigma X_i} \qquad \frac{\Xi \vdash_\Sigma A_1 \quad \cdots \quad \Xi \vdash_\Sigma A_n}{\Xi \vdash_\Sigma \mathsf{op}_i(A_1, \ldots, A_n)} \text{ where } n = ar(i)$$

Fig. 5. Simple types

3 SIMPLY TYPED LANGUAGES AND BIDIRECTIONAL TYPE SYSTEMS

This section provides generic definitions of simple types, simply typed languages, and bidirectional type systems, using simply typed λ-calculus in Section 2 as our running example. (These definitions may look dense, especially on first reading. The reader may choose to skim through this section, in particular the figures, and still get some rough ideas from later sections.)

The definitions are formulated in two steps: (1) first we introduce a notion of arity and a notion of signature which includes a set of operation symbols and an assignment of arities to symbols; (2) then, given a signature, we define raw terms and typing derivations by primitive rules such as Var and a rule schema for constructs op_o indexed by an operation symbol o. All these definitions are inductive as usual but include a *rule schema* giving rise to as many rules as there are operation symbols in a signature. As we move from simple types to bidirectional typing, the notion of arity, initially as the number of arguments of an operation, is enriched to incorporate an extension context for variable binding and the mode for the direction of type information flow.

3.1 Signatures and Simple Types

For simple types, the only datum needed for specifying a type construct is its number of arguments:

Definition 3.1. A *signature* Σ for simple types consists of a set[5] I with a decidable equality and an *arity* function $ar\colon I \to \mathbb{N}$. For a signature Σ, a *type* $A : \mathsf{Ty}_\Sigma(\Xi)$ over a variable set Ξ is either

(1) a variable in Ξ or
(2) $\mathsf{op}_i(A_1, \ldots, A_n)$ for some $i : I$ with $ar(i) = n$ and types $A_1, \ldots, A_n$.

Example 3.2. Simply typed λ-calculus includes function types $A \supset B$ and typically a base type b to ensure that the set of all types without type variables is non-empty. The type signature $\Sigma_\supset$ used to define types in simply typed λ-calculus consists of a binary operation fun and a nullary operation b where $ar(\mathsf{fun}) = 2$ and $ar(\mathsf{b}) = 0$. Then, all types in simply typed λ-calculus can be given as $\Sigma_\supset$-types over the empty set with $A \supset B$ introduced as $\mathsf{op}_{\mathsf{fun}}(A, B)$ and b as op_{b}.

Definition 3.3. The *substitution* for a function $\rho\colon \Xi \to \mathsf{Ty}_\Sigma(\Xi')$, denoted by $\rho\colon \mathsf{Sub}_\Sigma(\Xi, \Xi')$, is a map which sends a type $A : \mathsf{Ty}_\Sigma(\Xi)$ to $A\langle\rho\rangle : \mathsf{Ty}_\Sigma(\Xi')$ and is defined as usual.

3.2 Binding Signatures and Simply Typed Languages

A simply typed language specifies (1) a family of sets of raw terms t indexed by a list V of variables (that are currently in scope) where each construct is allowed to bind some variables like Abs and to take multiple arguments like App; (2) a family of sets of typing derivations indexed by a typing context Γ, a raw term t, and a type A. Therefore, to specify a term construct, we enrich the notion of arity with some sort for typing and extension context for variable binding.

Definition 3.4. A *binding arity* with a sort T is an inhabitant of $(T^* \times T)^* \times T$ where T^* is the set of lists over T. In a binding arity $(((\Delta_1, A_1), \ldots, (\Delta_n, A_n)), A)$, every Δ_i and A_i refers to the

[5]Even though our theory is developed in Martin-Löf type theory, the term 'set' is used instead of 'type' to avoid the obvious confusion. Also, as we assume Axiom K, all types are legitimate sets in the sense of homotopy type theory.

$\boxed{V \vdash_{\Sigma,\Omega} t}$ t is a raw term for a simply typed language (Σ, Ω) with free variables in V

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x}\ \textsc{Var} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t}{V \vdash_{\Sigma,\Omega} t \fatsemi A}\ \textsc{Anno}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1 \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n}{V \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)}\ \textsc{Op} \qquad \text{for } o : \Xi \triangleright [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0 \text{ in } \Omega$$

Fig. 6. Raw terms

extension context and the sort of the i-th argument, respectively, and A the target sort. For brevity, a binding arity is denoted by $[\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A$ where $[\Delta_i]$ is omitted if Δ_i is empty.

Example 3.5. Consider the Abs rule (Figure 2). Its binding arity has the sort $\mathsf{Ty}_{\Sigma_\supset}\{A, B\}$ and is $[A]B \to (A \supset B)$. This means that the Abs rule for derivations of $\Gamma \vdash \lambda x.\, t : A \supset B$ contains: (1) a derivation of $\Gamma, x : A \vdash t : B$ as an argument of type B in a typing context extended by a variable x of type A; (2) the type $A \supset B$ for itself. In a similar vein, the arity of the App rule is denoted as $(A \supset B), A \to B$. This specifies that any derivation of $\Gamma \vdash t\ u : B$ accepts derivations $\Gamma \vdash t : A \supset B$ and $\Gamma \vdash u : A$ as its arguments. These arguments have types $A \supset B$ and A respectively and their extension contexts are empty.

Next, akin to a signature, a binding signature Ω consists of a set of operation symbols along with their respective binding arities:

Definition 3.6. For a type signature Σ, a *binding signature* Ω consists of a set O and a function

$$ar(o) : O \to \sum_{\Xi : \mathcal{U}} \left(\mathsf{Ty}_\Sigma(\Xi)^* \times \mathsf{Ty}_\Sigma(\Xi)\right)^* \times \mathsf{Ty}_\Sigma(\Xi).$$

That is, each inhabitant $o : O$ is associated with a set Ξ of type variables and a binding arity $ar(o)$ with the sort $\mathsf{Ty}_\Sigma(\Xi)$, denoted by $o : \Xi \triangleright [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$.

The set Ξ of type variables for each operation, called its *local context*, plays a somewhat important role: To use a rule like Abs in an actual typing derivation, we will need to substitute *concrete types*, i.e. types without any type variables, for variables A, B. In our formulation of substitution (Definition 3.3), we must first identify for which type variables to substitute. As such, this information forms part of the arity of an operation, and typing derivations, defined subsequently, will include functions from Ξ to concrete types specifying how to instantiate typing rules by substitution.

By a *simply typed language* (Σ, Ω), we mean a pair of a type signature Σ and a binding signature Ω. Given a simply typed language, first we define its raw terms.

Definition 3.7. For a simply typed language (Σ, Ω), the family of sets of *raw terms* indexed by a list V of variables consists of (1) variables in V, (2) annotations $t \fatsemi A$ for some raw term t in V and a type A, and (3) a construct $\mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ for some $o : \Xi \triangleright [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ in O, where $\vec{x}_i$'s are lists of variables whose length is equal to the length of Δ_i, and t_i's are raw terms in the variable list $V, \vec{x}_i$. These correspond to rules Var, Anno, and Op in Figure 6 respectively.

Before defining typing derivations, we need a definition of typing contexts.

Definition 3.8. A *typing context* $\Gamma : \mathsf{Cxt}_\Sigma$ is formed by $\cdot$ for the empty context and $\Gamma, x : A$ for an additional variable x with a concrete type $A : \mathsf{Ty}_\Sigma(\emptyset)$. The list of variables in Γ is denoted $|\Gamma|$.

The definition of typing derivations is a bit more involved. We need some information to compare types on the object level during type synthesis and substitute those type variables in a typing

$\boxed{\Gamma \vdash_{\Sigma,\Omega} t : A}$ A raw term t has a concrete type A under Γ for a simply typed language (Σ, Ω)

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\Sigma,\Omega} x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{\Sigma,\Omega} t : A}{\Gamma \vdash_{\Sigma,\Omega} (t \Rightarrow A) : A}\ \textsc{Anno}$$

$$\frac{\rho : \mathsf{Sub}_\Sigma(\Xi, \emptyset) \qquad \Gamma, \vec{x}_1 : \Delta_1\langle\rho\rangle \vdash_{\Sigma,\Omega} t_1 : A_1\langle\rho\rangle \quad \cdots \quad \Gamma, \vec{x}_n : \Delta_n\langle\rho\rangle \vdash_{\Sigma,\Omega} t_n : A_n\langle\rho\rangle}{\Gamma \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A_0\langle\rho\rangle}\ \textsc{Op}$$

$$\text{for } o : \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0 \text{ in } \Omega$$

Fig. 7. Typing derivations

derivation $\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A$ for an operation o in Ω at some point. Here we choose to include a substitution ρ from the local context Ξ to $\emptyset$ as part of its typing derivation explicitly:

Definition 3.9. For a simply typed language (Σ, Ω), the family of sets of *typing derivations* of $\Gamma \vdash t : A$, indexed by a typing context $\Gamma : \mathsf{Cxt}_\Sigma$, a raw term t with free variables in $|\Gamma|$, and a type $A : \mathsf{Ty}_\Sigma(\emptyset)$, consists of

(1) a derivation of $\Gamma \vdash_{\Sigma,\Omega} x : A$ if $x : A$ is in Γ,
(2) a derivation of $\Gamma \vdash_{\Sigma,\Omega} (t \Rightarrow A) : A$ if $\Gamma \vdash_{\Sigma,\Omega} t : A$ has a derivation, and
(3) a derivation of $\Gamma \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A_0\langle\rho\rangle$ for $o : \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ if there is $\rho : \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$ and a derivation of $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash_{\Sigma,\Omega} t_i : A_i\langle\rho\rangle$ for each i,

corresponding to rules Var, Anno, and Op in Figure 7 respectively.

Example 3.10. Raw terms (Figure 1) and typing derivations (Figure 2) for simply typed λ-calculus can be specified by the type signature $\Sigma_\supset$ (Example 3.2) and the binding signature consisting of

$$\mathsf{app} : A, B \rhd (A \supset B), A \to B \quad \text{and} \quad \mathsf{abs} : A, B \rhd [A]B \to (A \supset B).$$

Rules Abs and App in simply typed λ-calculus are subsumed by the Op rule schema, as applications $t\ u$ and abstractions $\lambda x.\, t$ can be introduced uniformly as $\mathrm{op}_{\mathsf{app}}(t, u)$ and $\mathrm{op}_{\mathsf{abs}}(x.t)$, respectively.

3.3 Bidirectional Binding Signatures and Bidirectional Type Systems

For a bidirectional type system, typing judgements appear in two forms: $\Gamma \vdash t \Rightarrow A$ and $\Gamma \vdash t \Leftarrow A$, but these two typing judgements can be considered as a single typing judgement $\Gamma \vdash t :^d A$, additionally indexed by a *mode* $d : \mathsf{Mode}$—which can be either $\Rightarrow$ or $\Leftarrow$. Therefore, to define a bidirectional type system, we enrich the concept of binding arity to *bidirectional binding arity*, which further specifies the mode for each of its arguments and for the conclusion:

Definition 3.11. A *bidirectional binding arity* with a sort T is an inhabitant of

$$(T^* \times T \times \mathsf{Mode})^* \times T \times \mathsf{Mode}.$$

For clarity, a bidirectional binding arity is denoted by $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^d$.

Example 3.12. Consider the $\textsc{Abs}^{\Leftarrow}$ rule (Figure 3) for $\lambda x.\, t$. It has the arity $[A]B^{\Leftarrow} \to (A \supset B)^{\Leftarrow}$, indicating additionally that both $\lambda x.\, t$ and its argument t are checking. Likewise, the $\textsc{App}^{\Rightarrow}$ rule has the arity $(A \supset B)^{\Rightarrow}, A^{\Leftarrow} \to B^{\Rightarrow}$.

Definition 3.13. For a type signature Σ, a *bidirectional binding signature* Ω is a set O with

$$ar : O \to \sum_{\Xi : \mathcal{U}} \left(\mathsf{Ty}_\Sigma(\Xi)^* \times \mathsf{Ty}_\Sigma(\Xi) \times \mathsf{Mode}\right)^* \times \mathsf{Ty}_\Sigma(\Xi) \times \mathsf{Mode}.$$

$\boxed{\Gamma \vdash_{\Sigma,\Omega} t :^{d} A}$ t has a concrete type A in mode d under Γ for a bidirectional type system (Σ, Ω)

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\Sigma,\Omega} x :^{\Rightarrow} A}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}{\Gamma \vdash_{\Sigma,\Omega} (t \,:\!\!\!\in A) :^{\Rightarrow} A}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} B \qquad B = A}{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{\rho \colon \mathrm{Sub}_\Sigma(\Xi, \emptyset) \qquad \Gamma, \vec{x}_1 : \Delta_1\langle\rho\rangle \vdash_{\Sigma,\Omega} t_1 :^{d_1} A_1\langle\rho\rangle \qquad \cdots \qquad \Gamma, \vec{x}_n : \Delta_n\langle\rho\rangle \vdash_{\Sigma,\Omega} t_n :^{d_n} A_n\langle\rho\rangle}{\Gamma \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^{d} A_0\langle\rho\rangle}\ \textsc{Op}$$

$$\text{for } o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d} \text{ in } \Omega$$

Fig. 8. Bidirectional typing derivations

$\boxed{V \vdash_{\Sigma,\Omega} t^{d}}$ t is in mode d with free variables in V for a bidirectional type system (Σ, Ω)

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x^{\Rightarrow}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t^{\Leftarrow}}{V \vdash_{\Sigma,\Omega} (t \,:\!\!\!\in A)^{\Rightarrow}}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{V \vdash_{\Sigma,\Omega} t^{\Rightarrow}}{V \vdash_{\Sigma,\Omega} t^{\Leftarrow}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1{}^{d_1} \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n{}^{d_n}}{V \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^{d}}\ \textsc{Op} \qquad \text{for } o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$$

Fig. 9. Mode derivations

We write $o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$ for an operation o with a variable set Ξ and its bidirectional binding arity with sort $\mathrm{Ty}_\Sigma(\Xi)$. We call it *checking* if d is $\Leftarrow$ or *synthesising* if d is $\Rightarrow$; similarly its i-th argument is checking if d_i is $\Leftarrow$ and synthesising if d_i is $\Rightarrow$. A bidirectional type system (Σ, Ω) refers to a pair of a type signature Σ and a bidirectional binding signature Ω.

Definition 3.14. For a bidirectional type system (Σ, Ω),

- the set of *bidirectional typing derivations* of $\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$, indexed by a typing context Γ, a raw term t under $|\Gamma|$, a mode d, and a type A, is defined in Figure 8 and particularly
$$\Gamma \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^{d} A_0\langle\rho\rangle$$
has a derivation for $o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$ in Ω if there is $\rho \colon \Xi \to \mathrm{Ty}_\Sigma(\emptyset)$ and a derivation of $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash_{\Sigma,\Omega} t_i :^{d_i} A_i\langle\rho\rangle$ for each i;
- the set of *mode derivations* of $V \vdash_{\Sigma,\Omega} t^{d}$, indexed by a list V of variables, a raw term t under V, and a mode d, is defined in Figure 9.

The two judgements $\boxed{\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A}$ and $\boxed{\Gamma \vdash_{\Sigma,\Omega} t \Leftarrow A}$ stand for $\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} A$ and $\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A$, respectively. A typing rule is *checking* if its conclusion mode is $\Leftarrow$ or *synthesising* otherwise.

Every bidirectional binding signature Ω gives rise to a binding signature $|\Omega|$ if we erase modes from Ω, called the *(mode) erasure* of Ω. Hence a bidirectional type system (Σ, Ω) also specifies a simply typed language $(\Sigma, |\Omega|)$, including raw terms and typing derivations.

Example 3.15. Having established generic definitions, we can now specify the simply typed λ-calculus and its bidirectional type system—including raw terms, (bidirectional) typing derivations, and mode derivations—using just a pair of signatures $\Sigma_{\supset}$ (Example 3.2) and $\Omega_{\Lambda}^{\Leftrightarrow}$ which consists of

$$\mathrm{abs} \colon A, B \rhd [A]B^{\Leftarrow} \to (A \supset B)^{\Leftarrow} \quad \text{and} \quad \mathrm{app} \colon A, B \rhd (A \supset B)^{\Rightarrow}, A^{\Leftarrow} \to B^{\Rightarrow}.$$

More importantly, we are able to reason about constructions and properties that hold for any simply typed language with a bidirectional type system once and for all by quantifying over (Σ, Ω).

4 MODE DECORATION AND RELATED PROPERTIES

Our first important construction is mode decoration in Section 4.2, which is in fact generalised to pinpoint any missing type annotations in a given raw term. We also discuss some related properties: By bringing mode derivations into the picture, we are able to give a natural formulation of soundness and completeness of a bidirectional type system with respect to its erasure to an ordinary type system in Section 4.1. We also formulate Dunfield and Krishnaswami's [2021] annotatability and discuss its relationship with our completeness and generalised mode decoration in Section 4.3.

4.1 Soundness and Completeness

Erasure of a bidirectional binding signature removes mode information and keeps everything else intact; this can be straightforwardly extended by induction to remove mode information from a bidirectional typing derivation and arrive at an ordinary typing derivation, which is soundness. We can also remove typing and retain mode information, arriving at a mode derivation instead. Conversely, if we have both mode and typing derivations for the same term, we can combine them and obtain a bidirectional typing derivation, which is completeness. In short, soundness and completeness are no more than the separation and combination of mode and typing information carried by the three kinds of derivations while keeping their basic structure, which is directed by the same raw term. Technically, all these can be summarised in one theorem.

Theorem 4.1. $\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$ *if and only if both* $|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$ *and* $\Gamma \vdash_{\Sigma,|\Omega|} t : A$.

Proof. From left to right: Induction on the bidirectional typing derivation, mapping every rule to its counterpart except when constructing a typing derivation from the Sub$^{\Leftarrow}$ rule, in which case the induction hypothesis $\Gamma \vdash_{\Sigma,|\Omega|} t : B$ suffices due to the premise $B = A$.

From right to left: Induction on the mode derivation. For Var$^{\Rightarrow}$, Anno$^{\Rightarrow}$, and Op, the outermost rule used in the typing derivation must be the corresponding typing rule, so by the induction hypotheses we have bidirectional typing derivations for all the sub-terms, to which we can then apply the corresponding bidirectional typing rule. The Sub$^{\Leftarrow}$ case is similar but slightly simpler: the induction hypothesis directly gives us a derivation of $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A$, to which we apply Sub$^{\Leftarrow}$. □

4.2 Generalised Mode Decoration

The major goal of this section is to construct a mode decorator, which decides for any raw term $V \vdash_{\Sigma,|\Omega|} t$ and mode d whether $V \vdash_{\Sigma,\Omega} t^{d}$ or not. In fact we will do better: If a mode decorator returns a proof that no mode derivation exists, that proof (of a negation) does not provide useful information for the user. It will be more helpful if a decorator can produce an explanation of why no mode derivation exists, and even how to fix the input term to have a mode derivation. We will construct such a *generalised mode decorator* (Theorem 4.4), which can be weakened to an ordinary mode decorator (Corollary 4.6) if the additional explanation is not needed.[6]

Intuitively, a term does not have a mode derivation exactly when there are not enough type annotations, but such negative formulations convey little information. Instead, we can provide more information by pointing out the places in the term that require annotations. For a bidirectional type system, an annotation is required wherever a term is 'strictly' (which we will define shortly) in checking mode but required to be in synthesising mode, in which case there is no rule for switching from checking to synthesising, and thus there is no way to construct a mode derivation.

[6]To simplify the presentation, we use ordinary mode decoration elsewhere in this paper.

$\boxed{V \vdash_{\Sigma,\Omega} t^{d\,g\,s}}$ The raw term t is in mode d, misses some type annotation iff $g = \mathbf{F}$, and is in mode d due to an outermost mode cast iff $s = \mathbf{F}$

$$\frac{x \in V}{V \vdash_{\Sigma,\Omega} x^{\Rightarrow\,\mathbf{T}\,\mathbf{T}}}\ \textsc{Var}^{\Rightarrow} \qquad \frac{\cdot \vdash_{\Sigma} A \qquad V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,s}}{V \vdash_{\Sigma,\Omega} (t : A)^{\Rightarrow\,g\,\mathbf{T}}}\ \textsc{Anno}^{\Rightarrow} \qquad \frac{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,\mathbf{F}\,\mathbf{F}}}\ \textsc{Missing}^{\Rightarrow}$$

$$\frac{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{F}}}\ \textsc{Sub}^{\Leftarrow} \qquad \frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1^{\,d_1\,g_1\,s_1} \quad \cdots \quad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n^{\,d_n\,g_n\,s_n}}{V \vdash_{\Sigma,\Omega} \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^{d\,(\wedge_i g_i)\,\mathbf{T}}}\ \textsc{Op}$$

Fig. 10. Generalised mode derivations

We can, however, consider *generalised mode derivations* defined in Figure 10 that allow the use of an additional Missing$^{\Rightarrow}$ rule for such switching, so that a derivation can always be constructed. Given a generalised mode derivation, if it uses Missing$^{\Rightarrow}$ in some places, then those places are exactly where annotations should be supplied, which is helpful information to the user; if it does not use Missing$^{\Rightarrow}$, then the derivation is *genuine* in the sense that it corresponds directly to an original mode derivation. This can be succinctly formulated as Lemma 4.2 below by encoding genuineness as a boolean g in the generalised mode judgement, which is set to $\mathbf{F}$ only by the Missing$^{\Rightarrow}$ rule. (Ignore the boolean s for now.)

Lemma 4.2. *If* $V \vdash_{\Sigma,\Omega} t^{d\,\mathbf{T}\,s}$, *then* $V \vdash_{\Sigma,\Omega} t^{d}$.

Proof. Induction on the given derivation. The Missing$^{\Rightarrow}$ rule cannot appear because $g = \mathbf{T}$, and the other rules are mapped to their counterparts. □

We also want a lemma that covers the case where $g = \mathbf{F}$.

Lemma 4.3. *If* $V \vdash_{\Sigma,\Omega} t^{d\,\mathbf{F}\,s}$, *then* $V \nvdash_{\Sigma,\Omega} t^{d}$.

This lemma would be wrong if the 'strictness' boolean s was left out of the rules: Having both Sub$^{\Leftarrow}$ and Missing$^{\Rightarrow}$, which we call *mode casts*, it would be possible to switch between the two modes freely, which unfortunately means that we could insert a pair of Sub$^{\Leftarrow}$ and Missing$^{\Rightarrow}$ anywhere, constructing a non-genuine derivation even when there is in fact a genuine one. The 'strictness' boolean s can be thought of as disrupting the formation of such pairs of mode casts: Every rule other than the mode casts sets s to $\mathbf{T}$, meaning that a term is *strictly* in the mode assigned by the rule (i.e. not altered by a mode cast), whereas the mode casts set s to $\mathbf{F}$. Furthermore, the sub-derivation of a mode cast has to be strict, so it is impossible to have consecutive mode casts. Another way to understand the role of s is that it makes the Missing$^{\Rightarrow}$ rule precise: an annotation is truly missing only when a term is *strictly* in checking mode but is required to be in synthesising mode. With strictness coming into play, non-genuine derivations are now 'truly non-genuine'.

Proof of Lemma 4.3. Induction on the generalised mode derivation, in each case analysing an arbitrary mode derivation and showing that it cannot exist. The key case is Missing$^{\Rightarrow}$, where we have a sub-derivation of $V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{T}}$ for some boolean g. We do not have an induction hypothesis and seem to get stuck, because g is not necessarily $\mathbf{F}$. But what matters here is that t is *strictly* in checking mode: if we continue to analyse the sub-derivation, the outermost rule must be Op with $d = \Leftarrow$, implying that t has to be an operation in checking mode. Then a case analysis shows that it is impossible to have a (synthesising) mode derivation of $V \vdash_{\Sigma,\Omega} t^{\Rightarrow}$. □

Now we are ready to construct a generalised mode decorator.

$\boxed{t \sqsupseteq u}$ A raw term t is more annotated than u (for some bidirectional type system (Σ, Ω))

$$\frac{t \sqsupseteq u}{(t : A) \sqsupseteq u}\ \textsc{More} \qquad \frac{}{x \sqsupseteq x} \qquad \frac{t \sqsupseteq u}{(t : A) \sqsupseteq (u : A)} \qquad \frac{t_1 \sqsupseteq u_1 \quad \cdots \quad t_n \sqsupseteq u_n}{\mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \sqsupseteq \mathrm{op}_o(\vec{x}_1.\, u_1; \ldots; \vec{x}_n.\, u_n)}$$

Fig. 11. Annotation ordering between raw terms

Theorem 4.4 (Generalised Mode Preprocessing). *For any raw term $V \vdash_{\Sigma,|\Omega|} t$ and mode d, there is a derivation of $V \vdash_{\Sigma,\Omega} t^{d\,g\,s}$ for some booleans g and s.*

The theorem could be proved directly, but that would mix up two case analyses which respectively inspect the input term t and apply mode casts depending on which mode d is required. Instead, we distill the case analysis on d that deals with mode casts into the following Lemma 4.5, whose antecedent (1) is then established by induction on t in the proof of Theorem 4.4.

Lemma 4.5. *For any raw term $V \vdash_{\Sigma,|\Omega|} t$, if*

$$V \vdash_{\Sigma,\Omega} t^{d'\,g'\,\mathrm{T}} \quad \text{for some mode } d' \text{ and boolean } g' \tag{1}$$

then for any mode d, there is a derivation of $V \vdash_{\Sigma,\Omega} t^{d\,g\,s}$ for some booleans g and s.

Proof. Case analysis on d and d', adding an outermost mode cast to change the given derivation to a different mode if $d \neq d'$. Note that it is permissible to add an outermost mode cast because the antecedent (1) requires the given derivation to be strict. □

Proof of Theorem 4.4. By Lemma 4.5, it suffices to prove statement (1) by induction on t.

- Var is mapped to $\textsc{Var}^{\Rightarrow}$.
- Anno: Let $t = (t' : A)$. By the induction hypothesis and Lemma 4.5, there is a derivation of $V \vdash_{\Sigma,\Omega} t'^{\Leftarrow\,g\,s}$ for some booleans g and s, to which we apply $\textsc{Anno}^{\Rightarrow}$.
- Op: Let $t = \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$. By the induction hypotheses and Lemma 4.5, there is a derivation of $V, \vec{x}_i \vdash_{\Sigma,\Omega} t_i^{\,d_i\,g_i\,s_i}$ for each sub-term t_i; apply Op to all the derivations. □

Finally, having constructed a generalised mode decorator, it is easy to derive an ordinary one.

Corollary 4.6 (Mode Preprocessing). *It is decidable whether $V \vdash_{\Sigma,\Omega} t^d$.*

Proof. By Theorem 4.4, there is a derivation of $V \vdash_{\Sigma,|\Omega|} t^{d\,g\,s}$ for some booleans g and s, and then simply check g and apply either Lemma 4.2 or Lemma 4.3 to obtain $V \vdash_{\Sigma,\Omega} t^d$ or $V \nvdash_{\Sigma,\Omega} t^d$. □

4.3 Annotatability

Dunfield and Krishnaswami [2021, Section 3.2] formulated completeness differently from ours (Lemma 2.6 and Section 4.1) and proposed *annotatability* as a more suitable name. In our theory, we may formulate annotatability as follows.

Proposition 4.7 (Annotatability). *If $\Gamma \vdash_{\Sigma,|\Omega|} t : A$, then there exists t' such that $t' \sqsupseteq t$ and $\Gamma \vdash_{\Sigma,\Omega} t' :^d A$ for some d.*

Defined in Figure 11, the 'annotation ordering' $t' \sqsupseteq t$ means that t' has the same or more annotations than t. In a sense, annotatability is a reasonable form of completeness: if a term of a simply typed language $(\Sigma, |\Omega|)$ is typable in the ordinary type system, it may not be directly typable in the bidirectional type system (Σ, Ω) due to some missing annotations, but will be if those annotations are added correctly. In our theory, Proposition 4.7 can be easily proved with generalised mode decoration.

Proof of Proposition 4.7. By Theorem 4.4, there is a generalised mode derivation of $V \vdash_{\Sigma,|\Omega|} t^{d\,g\,s}$ for some mode d and booleans g and s. Perform induction on this generalised mode derivation to construct a bidirectional typing derivation in the same mode (and also the term t' and a proof of $t' \sqsupseteq t$, which are determined by the derivations and omitted here).

- Var$^{\Rightarrow}$, Anno$^{\Rightarrow}$, and Op: The outermost rule of the given typing derivation must be the corresponding one; apply the corresponding bidirectional typing rule to the induction hypotheses.
- Sub$^{\Leftarrow}$: Apply Sub$^{\Leftarrow}$ to the induction hypothesis.
- Missing$^{\Rightarrow}$: This is the interesting case where we map a Missing$^{\Rightarrow}$ rule to an Anno$^{\Rightarrow}$ rule and add to the term a type annotation, which comes from the given typing derivation. □

On the other hand, when using a bidirectional type synthesiser to implement type synthesis (Theorem 2.4), our completeness is much simpler to use than annotatability, which requires the bidirectional type synthesiser to produce more complex evidence when the synthesis fails. Annotatability also does not help the user to deal with missing annotations like (generalised) mode decoration does: although annotatability seems capable of determining where annotations are missing and even filling them in correctly, its antecedent requires a typing derivation, which is what the user is trying to construct and does not have yet. Therefore we believe that the notion of annotatability is too complex for what it achieves, and our theory offers simpler and more useful alternatives.

5 BIDIRECTIONAL TYPE SYNTHESIS AND CHECKING

This section focuses on defining mode-correctness and deriving bidirectional type synthesis for any mode-correct bidirectional type system (Σ, Ω). We start with Section 5.1 by defining mode-correctness and showing the uniqueness of synthesised types. This uniqueness means that any two synthesised types for the same raw term t under the same context Γ have to be equal. It will be used especially in Section 5.2 for the proof of the decidability of bidirectional type synthesis and checking. Then, we conclude this section with the trichotomy on raw terms in Section 5.3.

5.1 Mode Correctness

As Dunfield and Krishnaswami [2021] outlined, mode-correctness for a bidirectional typing rule means that (1) each 'input' type variable in a premise must be an 'output' variable in 'earlier' premises, or provided by the conclusion if the rule is checking; (2) each 'output' type variable in the conclusion should be some 'output' variable in a premise if the rule is synthesising. Here 'input' variables refer to variables in an extension context and in a checking premise. It is important to note that the order of premises in a bidirectional typing rule also matters, since synthesised type variables are instantiated incrementally during type synthesis.

Consider the rule Abs$^{\Leftarrow}$ (Figure 3) as an example. This rule is mode-correct, as the type variables A and B in its only premise are already provided by its conclusion $A \supset B$. Likewise, the rule App$^{\Rightarrow}$ for an application term $t\ u$ is mode-correct because: (1) the type $A \supset B$ of the first argument t is synthesised, thereby ensuring type variables A and B must be known if successfully synthesised; (2) the type of the second argument u is checked against A, which has been synthesised earlier; (3) as a result, the type of an application $t\ u$ can be synthesised.

Now let us define mode-correctness rigorously. As we have outlined, the condition of mode-correctness for a synthesising rule is different from that of a checking rule, and the argument order also matters. Defining the condition directly for a rule, and thus in our setting for an operation, can be somewhat intricate. Instead, we choose to define the conditions for the argument list—more specifically, triples $\overrightarrow{[\Delta_i]A_i^{d_i}}$ of an extension context Δ_i, a type A_i, and a mode d_i—pertaining to

an operation, for an operation, and subsequently for a signature. We also need some auxiliary definitions for the subset of variables of a type and of an extension context, and the set of variables that have been synthesised:

Definition 5.1. The finite subset[7] of *(free) variables* of a type A is denoted by $fv(A)$. The subset $fv(\Delta)$ of variables in an extension context Δ is defined by $fv(\cdot) = \emptyset$ and $fv(\Delta, A) = fv(\Delta) \cup fv(A)$. For an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$, define its set of *synthesised type variables* inductively by

$$fv^{\Rightarrow}(\cdot) = \emptyset \tag{2}$$

$$fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}, [\Delta_{n+1}]A_{n+1}^{\Leftarrow}) = fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}) \tag{3}$$

$$fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}, [\Delta_{n+1}]A_{n+1}^{\Rightarrow}) = fv(A_{n+1}) \cup fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}). \tag{4}$$

This subset contains type variables of a synthesising argument and they are exactly those type variables that will be synthesised during type synthesis.

Definition 5.2. The *mode-correctness* $\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)$ for an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$ with respect to a subset S of Ξ is defined by

$$\mathsf{MC}_{as}(\cdot) = \top \tag{5}$$

$$\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_n]A_n^{\Leftarrow}\right) = fv(\Delta_n, A_n) \subseteq \left(S \cup fv^{\Rightarrow}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)\right) \wedge \mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right) \tag{6}$$

$$\mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_n]A_n^{\Rightarrow}\right) = fv(\Delta_n) \subseteq \left(S \cup fv^{\Rightarrow}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right)\right) \wedge \mathsf{MC}_{as}\left(\overrightarrow{[\Delta_i]A_i^{d_i}}\right) \tag{7}$$

where (5) means an empty list is always mode-correct.

This definition encapsulates the idea that every 'input' type variable, possibly derived from an extension context Δ_n or a checking argument A_n, must be an 'output' variable from $fv^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ or, if the rule is checking, belong to the set S of 'input' variables in its conclusion. This condition must be met for every tail of the argument list as well to ensure that 'output' variables accessible at each argument position are from preceding arguments only, hence an inductive definition.

Definition 5.3. An operation $o : \Xi \triangleright [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \longrightarrow A_0^{d}$ is *mode-correct* if

(1) either d is $\Leftarrow$, its argument list is mode-correct with respect to $fv(A_0)$, and the union $fv(A_0) \cup fv^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ contains every inhabitant of Ξ;

(2) or d is $\Rightarrow$, its argument list is mode-correct with respect to $\emptyset$, and $fv^{\Rightarrow}(\overrightarrow{[\Delta_i]A_i^{d_i}})$ contains every inhabitant of Ξ and, particularly, $fv(A_0)$.

A bidirectional binding signature Ω is *mode-correct* if its operations are all mode-correct.

For a checking operation, an 'input' variable of an argument could be derived from A_0, as these are known during type checking as an input. Since every inhabitant of Ξ can be located in either A_0 or synthesised variables, we can determine a concrete type for each inhabitant of Ξ during type synthesis. On the other hand, for a synthesising operation, we do not have any known variables at the onset of type synthesis, so the argument list should be mode-correct with respect to $\emptyset$. Also, the set of synthesised variables alone should include every type variable in Ξ and particularly in A_n.

It is easy to check the bidirectional type system $(\Sigma_{\supset}, \Omega_{\Lambda}^{\Leftrightarrow})$ in Example 3.15 for simply typed λ-calculus is mode-correct according to our definition.

[7] There are various definitions for finite subsets of a set within Martin-Löf type theory. However, for our purposes, the choice among these definitions is not a matter of concern.

Remark 5.4. Mode-correctness is fundamentally a condition for bidirectional typing *rules*, not for derivations. Thus, this property cannot be formalised without treating rules as some mathematical object, such as the notion of bidirectional binding signature presented in Section 3. This contrasts with the properties in Section 4, which can still be specified for individual systems even in the absence of a generic definition of bidirectional type systems.

Now, we set out to show the uniqueness of synthesised types for a mode-correct bidirectional type system. For a specific system, its proof is typically a straightforward induction on the typing derivations. However, since mode-correctness is inductively defined on the argument list, our proof proceeds by induction on both the typing derivations and the argument list:

Lemma 5.5 (Uniqueness of Synthesised Types). *In a mode-correct bidirectional type system* (Σ, Ω)*, the synthesised types of any two derivations*

$$\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A \quad \textit{and} \quad \Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$$

for the same term t *must be equal, i.e.* $A = B$.

Proof. We prove the statement by induction on derivations d_1 and d_2 for $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A$ and $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$. Our system is syntax-directed, so d_1 and d_2 must be derived from the same rule:

- Var$^{\Rightarrow}$ follows from that each variable as a raw term refers to the same variable in its context.
- Anno$^{\Rightarrow}$ holds trivially, since the synthesised type A is from the term $t : A$ in question.
- Op: Recall that a derivation of $\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ contains a substitution ρ from the local context Ξ to concrete types. To prove that any two typing derivations has the same synthesised type, it suffices to show that those substitutions ρ_1 and ρ_2 of d_1 and d_2, respectively, agree on variables in $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$ so that $A_0\langle\rho_1\rangle = A_0\langle\rho_2\rangle$. We prove it by induction on the argument list:
  (1) For the empty list, the statement is vacuously true by Equation (2).
  (2) If d_{i+1} is $\Leftarrow$, then the statement holds for Equation (3) by induction hypothesis.
  (3) If d_{i+1} is $\Rightarrow$, then $\Delta_{i+1}\langle\rho_1\rangle = \Delta_{i+1}\langle\rho_2\rangle$ by Equation (7) and induction hypothesis (of the list). Therefore, under the same context $\Gamma, \Delta_{i+1}\langle\rho_1\rangle = \Gamma, \Delta_{i+1}\langle\rho_2\rangle$ the term t_{i+1} must have the same synthesised type $A_{i+1}\langle\rho_1\rangle = A_{i+1}\langle\rho_1\rangle$ by induction hypothesis (of the typing derivation), so ρ_1 and ρ_2 agree on $fv(A_{i+1})$ in addition to $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$, as required for Equation (4) in the definition of $fv^{\Rightarrow}$. □

5.2 Decidability of Bidirectional Type Synthesis and Checking

We have arrived at the main technical contribution of this paper.

Theorem 5.6 (Decidability of Bidirectional Type Synthesis and Checking). *In a mode-correct bidirectional type system* (Σ, Ω)*,*

(1) if $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$*, then it is decidable whether there is* A *such that*

$$\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A;$$

(2) if $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Leftarrow}$*, then it is decidable for any* A *whether*

$$\Gamma \vdash_{\Sigma,\Omega} t \Leftarrow A.$$

The interesting part of the theorem is the case for the Op rule and it is rather complex. Here we shall give its insight first instead of jumping into the details. Recall that a typing derivation for $\mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ contains a substitution $\rho\colon \Xi \to \mathrm{Ty}_\Sigma(\emptyset)$. The goal of type synthesis is exactly to define such a substitution ρ, so we have to start with an 'accumulating' substitution:

a substitution ρ_0 that is partially defined on $fv(A_0)$ if d is $\Leftarrow$ or otherwise nowhere. By mode-correctness, the accumulating substitution ρ_i will be defined on enough synthesised variables so that type synthesis or checking can be performed on t_i with the context $\Gamma, \vec{x}_i : \Delta_i\langle\rho_i\rangle$ based on its mode derivation $|\Gamma|, \vec{x}_i \vdash_{\Sigma,\Omega} t_i^{d_i}$. If we visit a synthesising argument $[\Delta_{i+1}]A^{\Rightarrow}_{i+1}$, then we may extend the domain of ρ_i to include the synthesised variables $fv(A_{i+1})$ if type synthesis is successful and also that the synthesised type can be *unified with* A_{i+1} and thereby *extend* ρ_i to $\bar{\rho}_i = \rho_{i+1}$ with the unifier. Then, if we go through every t_i successfully, we will have a total substitution ρ_n by mode-correctness and a derivation of $\Gamma, \vec{x}_i : \Delta_i \vdash_{\Sigma,\Omega} t_i :^{d_i} A\langle\rho_n\rangle$ for each sub-term t_i.

Remark 5.7. To make the argument above sound, it is necessary to compare types and solve a unification problem. Hence, we assume that the set Ξ of type variables has a decidable equality, thereby ensuring that the set $\mathsf{Ty}_\Sigma(\Xi)$ of types also has a decidable equality.[8]

We need some auxiliary definitions for the notion of extension to state the unification problem:

Definition 5.8. By an *extension* $\sigma \geqslant \rho$ of a partial substitution ρ we mean that the domain $dom(\sigma)$ of σ contains the domain of ρ and $\sigma(x) = \rho(x)$ for every x in $dom(\rho)$. By a *minimal extension* $\bar{\rho}$ of ρ satisfying P we mean an extension $\bar{\rho} \geqslant \rho$ with $P(\bar{\rho})$ such that $\sigma \geqslant \bar{\rho}$ whenever $\sigma \geqslant \rho$ and $P(\sigma)$.

Lemma 5.9. *For any A of $\mathsf{Ty}_\Sigma(\Xi)$, B of $\mathsf{Ty}_\Sigma(\emptyset)$, and a partial substitution $\rho : \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$,*

(1) either there is a minimal extension $\bar{\rho}$ of ρ such that $A\langle\bar{\rho}\rangle = B$,
(2) or there is no extension σ of ρ such that $A\langle\sigma\rangle = B$

This lemma can be inferred from the correctness of first-order unification [McBride 2003a,b], or be proved directly without unification. We are now ready for the decidability proof.

Proof of Theorem 5.6. We prove this statement by induction on the mode derivation $|\Gamma| \vdash_{\Sigma,\Omega} t^d$. The two cases Var$^{\Rightarrow}$ and Anno$^{\Rightarrow}$ are straightforward and have nothing to do with mode-correctness. The case Sub$^{\Leftarrow}$ invokes the uniqueness of synthesised types (Lemma 5.5) to refute the case that $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow B$ but $A \neq B$ for a given type A. The first three cases follow essentially the same reasoning provided by Wadler et al. [2022], but we still present the reasoning here for the sake of completeness. The last case Op is new and has been discussed above. For brevity we omit the subscript (Σ, Ω).

- Var$^{\Rightarrow}$: If $|\Gamma| \vdash t^{\Rightarrow}$, then $(x : A) \in \Gamma$ and thus $\Gamma \vdash x \Rightarrow A$.
- Anno$^{\Rightarrow}$: For $|\Gamma| \vdash (t \,\varsigma\, A)^{\Rightarrow}$, it is decidable whether $\Gamma \vdash t \Leftarrow A$ by induction hypothesis.
  - If $\Gamma \vdash t \Leftarrow A$, then $\Gamma \vdash t \,\varsigma\, A \Rightarrow A$.
  - If $\Gamma \nvdash t \Leftarrow A$ but $\Gamma \vdash t \,\varsigma\, A \Rightarrow$, then by inversion $\Gamma \vdash t \Leftarrow A$, leading to a contradiction.
- Sub$^{\Leftarrow}$: If $|\Gamma| \vdash t^{\Leftarrow}$ by the rule Sub$^{\Leftarrow}$, then $\Gamma \vdash t^{\Rightarrow}$ by inversion. By induction hypothesis, it is decidable whether $\Gamma \vdash t \Rightarrow B$ for some B:
  - If $\Gamma \nvdash t \Rightarrow C$ for any C but $\Gamma \vdash t \Leftarrow A$, then by inversion $\Gamma \vdash t \Rightarrow B$ for some $B = A$, thus a contradiction.
  - If $\Gamma \vdash t \Rightarrow B$ for some B, then by decidable equality on $\mathsf{Ty}_\Sigma(\Xi)$ either $A = B$ or $A \neq B$:
    - if $A = B$ then we have $\Gamma \vdash t \Leftarrow A$;
    - if $A \neq B$ but $\Gamma \vdash t \Leftarrow A$, then by inversion $\Gamma \vdash t \Rightarrow A$. However, by Lemma 5.5, synthesised types A and B must be equal, so we derive a contradiction.
- Op: For a mode derivation of $|\Gamma| \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^d$, we first claim:

[8] To simplify our choice, we could simply confine Ξ to any set within the family of sets $\mathsf{Fin}(n)$ of naturals less than n, given that these sets have a decidable equality and the arity of a type construct is finite. Indeed, in our formalisation, we adopt $\mathsf{Fin}(n)$ as the set of type variables in the definition of Ty_Σ (see Section 7 for details). For the sake of clarity in presentation, though, we keep using named variables and just assume that Ξ has a decidable equality.

Claim. *For an argument list* $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$ *and any partial substitution* ρ *from* Ξ *to* $\emptyset$
(1) either there is a minimal extension $\bar{\rho}$ *of* ρ *such that*

$$dom(\bar{\rho}) \supseteq fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}) \quad \text{and} \quad \Gamma, \vec{x}_i : \Delta_i\langle\bar{\rho}\rangle \vdash t_i : A_i\langle\bar{\rho}\rangle^{d_i} \quad \text{for } i = 1, \ldots, n; \tag{8}$$

(2) or there is no extension σ *of* ρ *such that* (8) *holds.*

Then, we proceed with a case analysis on d in the mode derivation:

- d is $\Rightarrow$: We apply our claim with the partial substitution ρ_0 defined nowhere.
  (1) If there is no $\sigma \geqslant \rho$ such that (8) holds but $\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ for some A, then by inversion we have a substitution $\rho : \mathrm{Sub}_\Sigma(\Xi, \emptyset)$ such that
  $$\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i : A_i\langle\rho\rangle^{d_i}$$
  for every i. Obviously, $\rho \geqslant \rho_0$ and $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i : A_i\langle\rho\rangle^{d_i}$ for every i and it contradicts the assumption that no such an extension exists.
  (2) If there exists a minimal $\bar{\rho} \geqslant \rho_0$ defined on $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$ such that (8) holds, then by mode-correctness $\bar{\rho}$ is total and thus
  $$\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A_0\langle\bar{\rho}\rangle\,.$$
- d is $\Leftarrow$: Let A be a type and apply Lemma 5.9 with ρ_0 defined nowhere.
  (1) If there is no $\sigma \geqslant \rho_0$ such that $A_0\langle\sigma\rangle = A$ but $\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then by inversion there is a substitution ρ such that $A = A_0\langle\rho\rangle$, thus a contradiction.
  (2) If there is a minimal $\bar{\rho} \geqslant \rho_0$ such that $A_0\langle\bar{\rho}\rangle = A$, then apply our claim with $\bar{\rho}$:
    (a) If there is no $\sigma \geqslant \bar{\rho}$ satisfying (8) but $\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then by inversion there is γ such that $A_0\langle\gamma\rangle = A$ and also $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i : A_i\langle\gamma\rangle^{d_i}$ for every i. Given that $\bar{\rho} \geqslant \rho$ is minimal such that $A_0\langle\bar{\rho}\rangle = A$, then γ is an extension of $\bar{\rho}$ but by assumption no such an extension satisfying $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i : A_i\langle\gamma\rangle^{d_i}$ exists, thus a contradiction.
    (b) If there is a minimal $\bar{\bar{\rho}} \geqslant \bar{\rho}$ s.t. (8), then by mode-correctness $\bar{\bar{\rho}}$ is total and
    $$\Gamma \vdash \mathrm{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A_0\langle\bar{\bar{\rho}}\rangle$$
    where $A_0\langle\bar{\bar{\rho}}\rangle = A_0\langle\bar{\rho}\rangle = A$ since $\bar{\bar{\rho}}(x) = \bar{\rho}$ for every x in the domain of $\bar{\rho}$.

Proof of Claim. We prove it by induction on the list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$:
(1) For the empty list, ρ is the minimal extension of ρ itself satisfying (8) trivially.
(2) For $\overrightarrow{[\Delta_i]A_i^{d_i}}, [\Delta_{m+1}]A_{m+1}^{d_{m+1}}$, by induction hypothesis on the list, we have two cases:
  (a) If there is no $\sigma \geqslant \rho$ such that (8) holds for all $1 \leqslant i \leqslant m$ but a minimal $\gamma \geqslant \rho$ such that (8) holds for all $1 \leqslant i \leqslant m+1$, then we clearly have a contradiction.
  (b) There is a minimal $\bar{\rho} \geqslant \rho$ s.t. (8) holds for $1 \leqslant i \leqslant m$. By case analysis on d_{m+1}:
    - d_{m+1} is $\Leftarrow$: By mode-correctness, $\Delta_{m+1}\langle\bar{\rho}\rangle$ and $A_{m+1}\langle\bar{\rho}\rangle$ are defined. By the induction hypothesis $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Leftarrow A_{m+1}\langle\bar{\rho}\rangle$ is decidable. Clearly, if $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Leftarrow A_{m+1}\langle\bar{\rho}\rangle$ then the desired statement is proved; otherwise we can also easily derive a contradiction.
    - d_{m+1} is $\Rightarrow$: By mode-correctness, $\Delta_{m+1}\langle\bar{\rho}\rangle$ is defined. By the induction hypothesis, it is decidable that $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A$ for some A.
      (i) If $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \nvdash t_{m+1} \Rightarrow A$ for any A but there is $\gamma \geqslant \rho$ s.t. (8) holds for $1 \leqslant i \leqslant m+1$, then $\gamma \geqslant \bar{\rho}$. Therefore $\Delta_{m+1}\langle\bar{\rho}\rangle = \Delta_{m+1}\langle\gamma\rangle$ and we derive a contradiction because $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A_{m+1}\langle\gamma\rangle$.
      (ii) If $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A$ for some A, then by Lemma 5.9 we have two cases:

- Suppose no $\sigma \geqslant \bar{\rho}$ such that $A_{m+1}\langle\sigma\rangle = A$ but an extension $\gamma \geqslant \rho$ such that (8) holds for $1 \leqslant i \leqslant m+1$. Then, $\gamma \geqslant \bar{\rho}$ by the minimality of $\bar{\rho}$ and thus $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A_{m+1}\langle\gamma\rangle$. However, by Lemma 5.5, the synthesised type $A_{m+1}\langle\gamma\rangle$ must be unique, so γ is an extension of $\bar{\rho}$ such that $A_{m+1}\langle\gamma\rangle = A$, leading to a contradiction.
- If there is a minimal $\bar{\rho} \geqslant \bar{\rho}$ such that $A_{m+1}\langle\bar{\rho}\rangle = A$, then it is not hard to show that $\bar{\rho}$ is also the minimal extension of ρ such that (8) holds for all $1 \leqslant i \leqslant m+1$.

Therefore, we have proved our claim for any argument list by induction. ■

We now have completed the decidability proof by induction on the mode derivation $|\Gamma| \vdash_{\Sigma,\Omega} t^d$. □

The formal counterpart of the above proof in Agda functions as two top-level programs for type checking and synthesis. These programs either compute the typing derivation or provide a proof of contradiction. Each case analysis simply branches depending on the outcomes of bidirectional type synthesis and checking for each sub-term, as well as the unification process. If a contradiction proof is not of interest for implementation, these programs can be simplified by disregarding the cases that yield such contradiction proofs. Alternatively, we could consider generalising typing derivations instead, like our generalised mode derivations (Figure 10). This could accommodate ill-typed cases and reformulate contradiction proofs positively to deliver more informative error messages. This would assist programmers in resolving issues with ill-typed terms, rather than simply returning a blatant 'no'.

5.3 Trichotomy on Raw Terms by Type Synthesis

Combining the bidirectional type synthesiser with the mode decorator, soundness, and completeness from Section 4, we get a type synthesiser parameterised by (Σ, Ω), generalising Theorem 2.4.

Corollary 5.10 (Trichotomy on Raw Terms). *For any mode-correct bidirectional type system (Σ, Ω), exactly one of the following holds:*

(1) ($|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$ and) $\Gamma \vdash_{\Sigma,\Omega} t : A$ for some type A.
(2) ($|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$ but) $\Gamma \nvdash_{\Sigma,\Omega} t : A$ for any type A.
(3) $|\Gamma| \nvdash_{\Sigma,\Omega} t^{\Rightarrow}$.

Proof. Combine Theorem 5.6 with Theorem 4.1 and Corollary 4.6. □

6 EXAMPLES

Our theory works for any language that can be specified by a bidirectional binding signature. To showcase its general applicability, we present two additional examples:

(1) PCF with products and let bindings, and
(2) spine calculus, a simply typed λ-calculus with applications in spine form.

The first is a standard exercise in implementing a type synthesiser and the second is an example of a system with infinitely many operations.

6.1 PCF with Products and Let Bindings

Although we have developed a theory for presenting a bidirectional type system and a condition for the decidability of bidirectional type synthesis, designing a bidirectional type system, or its signature, remains a creative activity.[9] While the focus of this paper is not to design bidirectional

[9]This has also been the case for parsing: while there are grammars and parser generators, one still has to design a grammar to parse or disambiguate an ambiguous grammar.

type systems, we adopt the example of a bidirectional PCF from Wadler et al. [2022], which extends bidirectional simply typed λ-calculus in Section 2 by adding the following rules:

$$\frac{}{\Gamma \vdash \mathsf{z} \Leftarrow \mathsf{nat}} \qquad \frac{\Gamma \vdash t \Leftarrow \mathsf{nat}}{\Gamma \vdash \mathsf{s}(t) \Leftarrow \mathsf{nat}} \qquad \frac{\Gamma \vdash t \Rightarrow \mathsf{nat} \qquad \Gamma \vdash t_0 \Leftarrow A \qquad \Gamma, x : \mathsf{nat} \vdash t_1 \Leftarrow A}{\Gamma \vdash \mathtt{ifz}(t_0; x.t_1)(t) \Leftarrow A}$$

$$\frac{\Gamma \vdash t \Leftarrow A \qquad \Gamma \vdash u \Leftarrow B}{\Gamma \vdash (t, u) \Leftarrow A \times B} \qquad \frac{\Gamma \vdash t \Rightarrow A \times B}{\Gamma \vdash \mathtt{proj}_1(t) \Rightarrow A} \qquad \frac{\Gamma \vdash u \Rightarrow A \times B}{\Gamma \vdash \mathtt{proj}_2(u) \Rightarrow A}$$

$$\frac{\Gamma, x : A \vdash t \Leftarrow A}{\Gamma \vdash \mu x.\, x \Leftarrow A} \qquad \frac{\Gamma \vdash t \Rightarrow A \qquad \Gamma, x : A \vdash u \Leftarrow B}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u \Leftarrow B}$$

In this language, we introduce additional type constructs apart from $\supset$. To do so, we extend $\Sigma_{\supset}$ (as shown in Example 3.2) with the operations nat and times such that

$$ar(\mathsf{nat}) = 0 \quad \text{and} \quad ar(\mathsf{times}) = 2$$

enabling the introduction of nat and $A \times B$ as $\mathsf{op}_{\mathsf{nat}}$ and $\mathsf{op}_{\mathsf{times}}(A, B)$, respectively. Following this, we also extend the bidirectional binding signature $\Omega^{\Leftrightarrow}_{\Lambda}$ with an operation for each rule above:

$\mathsf{z} : \cdot \triangleright \cdot \longrightarrow \mathsf{nat}^{\Leftarrow}$ $\quad \mathsf{s} : \cdot \triangleright \mathsf{nat}^{\Leftarrow} \longrightarrow \mathsf{nat}^{\Leftarrow}$ $\quad \mathsf{ifz} : A \triangleright \mathsf{nat}^{\Rightarrow}, A^{\Leftarrow}, A^{\Leftarrow} \longrightarrow A^{\Leftarrow}$

$\mathsf{pair} : A, B \triangleright A^{\Leftarrow}, B^{\Leftarrow} \longrightarrow A \times B^{\Leftarrow}$ $\quad \mathsf{proj}_1 : A, B \triangleright A \times B^{\Rightarrow} \longrightarrow A^{\Rightarrow}$ $\quad \mathsf{proj}_2 : A, B \triangleright A \times B^{\Rightarrow} \longrightarrow B^{\Rightarrow}$

$\mathsf{mu} : A \triangleright [A]A^{\Leftarrow} \longrightarrow A^{\Leftarrow}$ $\quad \mathsf{let} : A, B \triangleright A^{\Rightarrow}, [A]B^{\Leftarrow} \longrightarrow B^{\Leftarrow}$

With these operations, a straightforward check shows that every rule is mode-correct, therefore this is a mode-correct bidirectional type system. As a result, we can instantiate the type synthesiser presented in Section 5 for this system, eliminating the need for re-implementation.

6.2 Spine Calculus

A spine application, denoted as $t\ u_1\ \ldots\ u_n$, is a variant of application that consists of a head term t and an indeterminate number of arguments $u_1\ u_2\ \ldots\ u_n$. This arrangement allows direct access to the head term, making it practical in various applications. For instance, Agda's core language employs this form of application, as does its reflected syntax used for metaprogramming.

At first glance, accommodating this form of application may seem impossible, given that the number of arguments a construct can accept has to be fixed in a signature. Nonetheless, there is no constraint on the total number of operation symbols a signature can have, allowing us to establish corresponding constructs for each number n of arguments, thereby exhibiting the necessity of having an arbitrary set (with decidable equality) for operation symbols.

To illustrate this, let us consider the following rule:

$$\frac{\Gamma \vdash t \Rightarrow \prod_{i=1}^{n} A_i \supset B \qquad \Gamma \vdash u_1 \Leftarrow A_1 \qquad \ldots \qquad \Gamma \vdash u_n \Leftarrow A_n}{\Gamma \vdash t\ u_1\ \ldots\ u_n \Rightarrow B}$$

The type signature $\Sigma_{\supset}$ can be extended with fun_n where $ar(\mathsf{fun}_n) = n$ for each $n : \mathbb{N}$. Similarly, the bidirectional binding signature $\Omega^{\Leftrightarrow}_{\Lambda}$ can be extended with

$$\mathsf{app}_n : A_1, \ldots, A_n, B \triangleright \mathsf{op}_{\mathsf{fun}_n}(A_1, \ldots, A_n) \supset B^{\Rightarrow}, A_1^{\Leftarrow}, \ldots, A_n^{\Leftarrow} \longrightarrow B$$

also for each $n : \mathbb{N}$. Then each spine application $t\ u_1\ \ldots\ u_n$ can be introduced as $\mathsf{op}_{\mathsf{app}_n}(t; u_1; \ldots; u_n)$.

7 FORMALISATION

As we have mentioned in Section 1, our theory was initially developed with Agda. While the translation to the natural language is reasonably straightforward, understanding the formalisation itself could pose some difficulty. If the reader is comfortable with the informal presentation and assured by the existence of its formalisation, they may feel free to skip this section.

Revisiting language formalisation frameworks. Unlike prior frameworks [Ahrens et al. 2022; Allais et al. 2021; Fiore and Szamozvancev 2022] that have primarily focused on meta-properties centred around substitution for intrinsically-typed terms, our theory of bidirectional type synthesis does not require term substitution but structural induction for extrinsically-typed terms. The formal definitions of extrinsic typing are more complex than their intrinsic counterparts. Take our formal definition of typing derivations $\Gamma \vdash_{\Sigma,\Omega} t : A$ as an example. Its intrinsic definition is just one family of sets of typed terms indexed by A and Γ, but its extrinsic counterpart is a generic family of sets indexed by additionally a generic raw term t, involving constructions of two different layers.

Category-theoretic analysis of well-typed terms. The difference between intrinsic typing and extrinsic typing could be partly manifested by Fiore et al. [1999]'s theory of abstract syntax and variable binding which forms the foundation of Fiore and Szamozvancev [2022]'s framework and inspires other referenced frameworks. Let us take a closer look of their idea. The set of (untyped) abstract syntax trees for a language can be understood as (1) a family of sets Tm_Γ of well-scoped terms under a context Γ with (2) variable renaming for a function $\sigma \colon \Gamma \to \Delta$ between variables acting as a functorial map from Tm_Γ to Tm_Δ, i.e. a presheaf $\mathsf{Tm} \colon \mathbb{F} \to \mathsf{Set}$, with (3) an *initial* algebra $[\mathsf{v}, \mathsf{op}]$ on Tm given by the variable rule as a map v from the presheaf $V \colon \mathbb{F} \hookrightarrow \mathsf{Set}$ of variables to Tm and other constructs as $\mathsf{op} \colon \Sigma\mathsf{Tm} \to \mathsf{Tm}$ where $\mathbb{F}$ is the category of contexts and the functor $\Sigma \colon \mathsf{Set}^{\mathbb{F}} \to \mathsf{Set}^{\mathbb{F}}$ encodes the binding arities of constructs. The initiality amounts to structural recursion on terms, namely *term traversal*. Substitution for Tm is modelled as a monoid in the category of presheaves. To put it succinctly, it is the free Σ-monoid over the presheaf V of variables.

Such an initial algebra can be constructed by the Initial Algebra Theorem [Trnková 1975].

Type-theoretic construction of well-typed terms. Fortunately, in type theory, constructing the initial algebra of well-typed terms boils down to defining an inductive type with a few constructors that align primitive rules (such as Var) and a rule schema; term traversal can be defined by pattern matching as usual for the inductive type [Fiore and Szamozvancev 2022].

Lack of a theory for extrinsic typing. While the theory of Fiore et al. [1999] and others provide significant insights, they do not consider extrinsic typing. We find inspiration in the interpretation of structural induction as algebras for an endofunctor on the category of predicates over a base category, as put forward by Hermida and Jacobs [1998]. From this perspective, we interpret extrinsic typing as a signature 'functor' on the category of predicates consisting of a $\mathbb{N}$-indexed family X of sets and another family Y of sets over X; constructing typing derivations as its initial algebra over well-scoped terms. Our formal construction appears novel and we believe that a theoretic understanding of our construction and its connection to intrinsic typing as a bridge between raw terms and well-typed terms deserve some attention. However, carrying out a category-theoretic analysis requires expertise in category theory, which goes beyond the scope of this paper.

Therefore, in this section, we will primarily provide an overview of our design and offer intuitive explanations. We will discuss the construction of simple types in Section 7.1, raw terms in Section 7.2, and bidirectional typing rules in Section 7.3. Since the formal proofs largely reflect their informal presentation, we will limit our illustration to the 'if' part of Theorem 4.1 as an example in Section 7.4.

```
record SigD : Set₁ where
  field
    Op    : Set
    decEq : DecEq Op
    ar    : Op → ℕ
```

```
⟦_⟧ : SigD → Set → Set
⟦ D ⟧ X = Σ[ i ∈ D .Op ] X ^ (D .ar i)

data Ty (Ξ : ℕ) : Set where
  `_ : Fin Ξ      → Ty Ξ
  op : ⟦ D ⟧ (Ty Ξ) → Ty Ξ
```

Fig. 12. Simple types in Agda

```
record ArgD (Ξ : ℕ) : Set
  where
  field
    cxt  : Cxt Ξ
    type : Ty Ξ
    mode : Mode
ArgsD = List ∘ ArgD
```

```
record OpD : Set where
  constructor ι
  field
    {tvars} : ℕ
    mode    : Mode
    type    : TExp tvars
    args    : ArgsD tvars
```

```
record SigD : Set₁ where
  constructor sigd
  field
    Op : Set
    ar : Op → OpD
```

Fig. 13. Bidirectional binding signature in Agda

7.1 Defining Simple Types

We start with the formal definition of signatures and simple types (Figure 12), which parallels its informal counterpart (Definition 3.1), with the exception of ⟦_⟧. To prevent circular references during the inductive definition of simple types, we initially define ⟦ D ⟧ on an arbitrary Set instead of Ty. Thus, types specified by a signature can be defined inductively where each inhabitant op (i , ts) for the Op rule is a pair of an operation symbol and a list of length n.

From a categorical view, ⟦ D ⟧ is a functor from Set to Set, mapping X to an Op-indexed coproduct $\sum_{i \in \mathsf{Op}} X^{\mathsf{ar}(i)}$ of products of X. The type Ty is the free ⟦ D ⟧-algebra over the type Fin Ξ or the initial algebra for the functor Fin Ξ + ⟦ D ⟧.

7.2 Defining Raw Terms

As shown in Figure 13, we establish ArgD, OpD, and SigD to represent arguments $[\Delta]A^d$, bidirectional binding operations $o : \Xi \triangleright [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^d$, and bidirectional binding signatures, respectively, in line with their informal definitions. By removing mode from these definitions, we retrieve the definitions of binding arguments, arities, and signatures.

To construct well-scoped raw terms indexed by a list V of free variables, we introduce another signature functor mapping Fam to Fam, where Fam is the family of sets defined as $\mathbb{N} \to$ Set, indexed by a natural number indicating the number of free variables. This functor for raw term construction resembles the one for defining simple types. However, we define four distinct functors instead of directly defining one . These are ⟦ Δ ⟧^{a} for an extension context Δ, ⟦ Ds ⟧^{as} for an argument list Ds, ⟦ ι Ds _ ⟧^{c} for an operation (ι Ds A), and ⟦ D ⟧^{c} for a binding signature D. Notably, the functor ⟦ Δ ⟧^{a} extends the context by mapping the family of sets X n to X (length Δ + n).

The inductive type of raw terms, indexed by a list of free variables, is defined as shown in Figure 14. This mirrors our informal definition (Figure 6), where A ∋ t corresponds to the raw term $t \circledcirc A$. These definitions align closely with those found in other referenced frameworks.

```
⟦_⟧ᵃ : TExps Ξ → Fam → Fam
⟦ Δ ⟧ᵃ X n =
  X (length Δ + n)

⟦_⟧ᵃˢ : ArgsD Ξ → Fam → Fam
⟦ [] ⟧ᵃˢ _ _ = ⊤
⟦ (Δ ⊢ A) :: Ds ⟧ᵃˢ X n =
  ⟦ Δ ⟧ᵃ X n × ⟦ Ds ⟧ᵃˢ X n

⟦_⟧ᶜ : OpD → Fam → Fam
⟦ ι Ds _ ⟧ᶜ X = ⟦ Ds ⟧ᵃˢ X

⟦_⟧ : SigD → Fam → Fam
⟦ D ⟧ X n =
  Σ[ i ∈ D .Op ] ⟦ D .ar i ⟧ᶜ X n

data Raw : ℕ → Set where
  `_ : Fin n → Raw n
  _∋_ : Ty → Raw n → Raw n
  op : ⟦ D ⟧ Raw n → Raw n
```

Fig. 14. Signature functor for raw terms and raw terms in Agda

7.3 Defining Extrinsic Bidirectional Typing Derivations

Formally defining extrinsic derivations is much trickier. This requires defining a family of sets, indexed by context, type, mode, and notably, a raw term, involving another signature endofunctor alone with the functor used for constructing raw terms.

We begin with the family of sets on which the functor acts. A typing judgement $\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$ can be viewed as a predicate over raw terms with four indices: a context, a $\mathbb{N}$-indexed family of sets, a mode, and a type. The type of this predicate can be generalised to a family of sets over the $\mathbb{N}$-indexed family of sets, as follows:

```
Fam : R.Fam → Set _
Fam X = (Γ : Cxt 0) → X (length Γ) → Mode → Ty → Set
```

Observe that the second index X depends on the length of the first index Γ. This is because a typing derivation for a raw term $V \vdash_{\Sigma,|\Omega|} t$ requires the same number of variables as in V.

In a similar vein, we define four functors for the construction of bidirectional typing derivations, albeit in a top-down manner. An endofunctor on predicates must act as a *lifting* of the functor on its domain, illustrated by the signature endofunctor ⟦_⟧ for a bidirectional binding signature:

```
⟦_⟧ : (D : SigD) (X : R.Fam) (Y : Fam X) → Fam (R.⟦ erase D ⟧ X)
⟦ D ⟧ X Y Γ (o , ts) d A = ⟦ D .ar o ⟧ᶜ X Y Γ ts d A
```

This functor maps a predicate Y over X into a predicate ⟦ D ⟧ X Y over R.⟦ erase D ⟧ X. Here, erase D represents the mode erasure of D, and the prefix R in R.⟦_⟧ indicates the signature endofunctor used in raw term construction.

The functor ⟦ D .ar o ⟧ᶜ is defined for an operation $o : \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$:

```
⟦_⟧ᶜ : (D : OpD) (X : R.Fam ℓ) (Y : Fam ℓ′ X)
     → Fam ℓ′ (R.⟦ eraseᶜ D ⟧ᶜ X)
⟦ ι {Ξ} d A₀ D ⟧ᶜ X Y Γ xs d′ A =
  d ≡ d′ × Σ[ p ∈ TSub Ξ 0 ] A₀ ⟨ p ⟩ ≡ A × ⟦ D ⟧ᵃˢ X Y σ Γ xs
```

A set within the family is only inhabited if the indices correspond with the mode and the type as specified by the operation. Therefore, in defining the set ⟦ ι {Ξ} d A_0 D ⟧ᶜ X Y Γ xs d′ A, it takes identity proofs for d ≡ d′ and $A_0 \langle \sigma \rangle \equiv A$, with A_0 instantiated by a substitution σ from Ξ to $\emptyset$. This ensures that the set is inhabited only if the indices align with what the arity (D .ar o) specifies.

The signature endofunctor for an argument list Ds is defined in a manner that it maps a predicate to a family of sets indexed by a context and a family of product sets, considering that each premise in a typing derivation shares a common context Γ and a term from a list of subterms.

```
⟦_⟧ᵃˢ : (D : ArgsD Ξ) (X : R.Fam) (Y : Fam X)
      → TSub Ξ 0 → (Γ : Cxt 0) → R.⟦ eraseᵃˢ D ⟧ᵃˢ X (length Γ) → Set
⟦ [] ⟧ᵃˢ _ _ _ _ _ = ⊤
```

```
mutual
  soundness : Γ ⊢ r [ d ] A → Γ ⊢ r ꞉ A
  soundness (var i eq)  = var i eq
  soundness (A ∋ t)     = A ∋ soundness t
  soundness (t ↑ refl)  = soundness t
  soundness (op ts)     =
    op (soundness^c (BD .rules _) ts)

  soundness^c
    : (D : OpD)
    → ⟦ D ⟧^c Raw _⊢_[_]_ Γ rs d A
    → T.⟦ erase^c D ⟧^c Raw _⊢_꞉_ Γ rs A
  soundness^c (ι _ _ Ds) (_ , σ , σ-eq , ts) =
    σ , σ-eq , soundness^as Ds ts

  soundness^as
    : (Ds : ArgsD Ξ)
    → ⟦ Ds ⟧^as          Raw _⊢_[_]_ σ Γ rs
    → T.⟦ erase^as Ds ⟧^as Raw _⊢_꞉_  σ Γ rs
  soundness^as []       _                    = tt
  soundness^as ((Δ ⊢[ _ ] _) :: Ds) (t , ts) =
    soundness^a Δ t , soundness^as Ds ts

  soundness^a
    : (Δ : TExps Ξ)
    → ⟦ Δ ⟧^a   Raw (λ Γ′ r′ → Γ′ ⊢ r′ [ d ] A) σ Γ r
    → T.⟦ Δ ⟧^a Raw (λ Γ′ r′ → Γ′ ⊢ r′ ꞉ A)    σ Γ r
  soundness^a []       t = soundness t
  soundness^a (_ :: Δ) t = soundness^a Δ t
```

Fig. 15. The soundness proof in Agda

```
⟦ Δ ⊢[ d ] A :: Ds ⟧^as X Y σ Γ (x , xs) =
  ⟦ Δ ⟧^a X (λ Γ′ x′ → Y Γ′ x′ d (A ⟨ σ ⟩))) σ Γ x × ⟦ Ds ⟧^as X Y σ Γ xs
```

The signature endofunctor $\llbracket\ \Delta\ \rrbracket^a$ for an extension context Δ maps a predicate Y over a $\mathbb{N}$-indexed family X of sets into a predicate ($\llbracket\ \Delta\ \rrbracket^a$ X Y) whose context is extended by the types in Δ.

```
⟦_⟧^a : (Δ : Cxt Ξ) (X : R.Fam) (Y : (Γ : Cxt 0) → X (length Γ) → Set)
      → TSub Ξ 0 → (Γ : Cxt 0) → R.⟦ Δ ⟧^a X (length Γ) → Set
⟦ []     ⟧^a X Y σ Γ t = Y Γ t
⟦ A :: Δ ⟧^a X Y σ Γ t = ⟦ Δ ⟧^a X Y σ ((A ⟨ σ ⟩) :: Γ) t
```

This completes our definition of the signature functor for constructing extrinsic typing derivations.

At last, we can define the type of bidirectional typing derivations for a signature D, similarly to how we have defined simple types and raw terms. This involves the following four constructors, corresponding to rules Var$^{\Rightarrow}$, Anno$^{\Rightarrow}$, Sub$^{\Leftarrow}$, and Op in Figure 7.

```
data _⊢_[_]_ : Fam Raw where
  var : (i : A ∈ Γ)
      → L.index i ≡ j
      → Γ ⊢ (` j) ⇒ A
  _∋_ : (A : Ty)
      → Γ ⊢ r      ⇐ A
      → Γ ⊢ (A ∋ r) ⇒ A
  _↑_ : Γ ⊢ r ⇒ B
      → A ≡ B
      → Γ ⊢ r ⇐ A
  op  : ⟦ D ⟧ Raw _⊢_[_]_ Γ rs d A
      → Γ ⊢ op rs [ d ] A
```

7.4 A Formal Proof Example: Soundness

Induction proofs in our formalisation typically consist of three mutual definitions—one for the inductive type of derivations, one for a list of arguments, and one for an extension context.

As an illustration, consider the soundness proof in Figure 15, i.e. the 'if' part of Theorem 4.1. Each constructor of bidirectional typing derivations is mapped to a corresponding constructor of typing derivations. For the Sub$^{\Rightarrow}$ case, the induction hypothesis soundness t is invoked directly.

8 FUTURE WORK

More characteristics of bidirectional typing. We have formalised some of the most important concepts discussed by Dunfield and Krishnaswami [2021]. There are other concepts that may be formalisable with some more effort, for example Pfenning's recipe for bidirectionalising typing rules. We believe that our theory lays the foundation for further formalising these concepts. There are some other concepts that may be more difficult to pin down, notably 'annotation character', which is roughly about how easy it is for the user to write annotated programs. These may continue to require creativity from the designer of type systems.

Beyond syntax-directed type systems. As a first theory of bidirectional type synthesis, our work focuses on syntax-directed type systems. To explore more general settings, an extreme possibility is to independently specify an ordinary type system and a bidirectional one over the same raw terms and then investigate possible relationships between the two systems, but a more manageable and practical setting suitable for a next step is probably one that only mildly generalises ours: for each raw term construct, there can be several ordinary typing rules, and each typing rule can be refined to several bidirectional typing rules with different mode assignments. Even for this mildly generalised setting, there is already a lot more work to do: Both the mode decorator and the bidirectional type synthesiser will have to backtrack, and become significantly more complex. For soundness and completeness, it should still be possible to treat them as the separation and combination of mode and type information, but the completeness direction will pose a problem—for every node of a raw term, a mode derivation chooses a mode assignment while a typing derivation chooses a typing rule, but there may not be a bidirectional typing rule for this particular combination. There should be ways to fix this problem while retaining the mode decoration phase—for example, one idea is to make the mode decorator produce all possible mode derivations, and refine completeness to say that any typing derivation can be combined with one of these mode derivations into a bidirectional typing derivation. All these considerations are rather too complex for a first theory, however.

Beyond simple types. We leave the problem of presenting bidirectional typing for more general and advanced language features as future work (such as those related to polymorphic types [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007; Pierce and Turner 2000; Xie and Oliveira 2018]). While algebraic approaches to polymorphic types have been developed [Fiore and Hamana 2013; Hamana 2011], these approaches do not take subtyping into account. Subtyping is essential for formulating important concepts such as *principal types* in type synthesis. On the other hand, in the realm of dependent types, Cartmell's [1986] generalised algebraic theories can handle a wide variety of dependent type theories. Bezem et al. [2021] investigate the notion of presentation (extending the notion of signature) in the context of generalised algebraic theories. Nonetheless, type synthesis for dependent types requires normalisation or some form of conversion to check type equality. Normalisation in its generic form still remains out of reach, and advances in this topic are only recently discussed in the doctoral thesis by Valliappan [2023]. Before we can embark on a richer theory of bidirectional type synthesis, language formalisation frameworks for these more advanced language features have to be developed.

REFERENCES

Peter Aczel. 1978. A general Church–Rosser theorem. (1978). http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf Unpublished note.

Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. 2018. From signatures to monads in UniMath. *Journal of Automated Reasoning* 63, 2 (2018), 1–34. https://doi.org/10.1007/s10817-018-9474-4

Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. 2022. Implementing a category-theoretic framework for typed abstract syntax. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*

(Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 307–323. https://doi.org/10.1145/3497775.3503678

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *Journal of Functional Programming* 31, 1996 (Oct. 2021), e22. https://doi.org/10.1017/S0956796820000076

Thorsten Altenkirch. 1993. A formalization of the strong normalization proof for System F in LEGO. In *Typed Lambda Calculi and Applications. TLCA 1993 (Lecture Notes in Computer Science, Vol. 664)*, Marc Bezem and Jan Friso Groote (Eds.). Springer, Berlin, Heidelberg, 13–28. https://doi.org/10.1007/BFb0037095

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science, Vol. 3603)*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65. https://doi.org/10.1007/11541868_4

Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Hötzel Escardó. 2021. On generalized algebraic theories and categories with families. *Mathematical Structures in Computer Science* 31, 9 (Oct. 2021), 1–18. https://doi.org/10.1017/S0960129521000268

John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32 (1986), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

Gilles Dowek. 1993. The undecidability of typability in the Lambda-Pi-calculus. In *Typed Lambda Calculi and Applications. TLCA 1993 (Lecture Notes in Computer Science, Vol. 664)*, Marc Bezem and Jan Friso Groote (Eds.). Springer, Berlin, Heidelberg, 139–145. https://doi.org/10.1007/BFb0037103

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional typing. *Comput. Surveys* 54, 5 (May 2021), 98:1–98:38. https://doi.org/10.1145/3450952

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582

Marcelo Fiore and Makoto Hamana. 2013. Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational Logic. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, New Orleans, LA, USA, 520–529. https://doi.org/10.1109/LICS.2013.59

Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–29. https://doi.org/10.1145/3498715

Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract syntax and variable binding. In *Proceedings. 14th Symposium on Logic in Computer Science*. IEEE, Trento, Italy, 193–202. https://doi.org/10.1109/LICS.1999.782615

Holger Gast. 2004. *A generator for type checkers*. Ph. D. Dissertation. Universität Tübingen. http://hdl.handle.net/10900/48845

Lorenzo Gheri and Andrei Popescu. 2020. A formalized general theory of syntax with bindings: Extended version. *Journal of Automated Reasoning* 64, 4 (April 2020), 641–675. https://doi.org/10.1007/s10817-019-09522-2

Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. 2015. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 137–150. https://doi.org/10.1145/2814228.2814239

Makoto Hamana. 2011. Polymorphic abstract syntax via Grothendieck construction. In *Foundations of Software Science and Computational Structures. FoSSaCS 2011*, Martin Hofmann (Ed.). Lecture Notes in Computer Science, Vol. 6604. Springer, Berlin, Heidelberg, 381–395. https://doi.org/10.1007/978-3-642-19805-2_26

Claudio Hermida and Bart Jacobs. 1998. Structural induction and coinduction in a fibrational setting. *Information and Computation* 145, 2 (1998), 107–152. https://doi.org/10.1006/inco.1998.2725

Conor McBride. 2003a. First-order unification by structural recursion. *Journal of Functional Programming* 13, 6 (2003), 1061–1075. https://doi.org/10.1017/S0956796803004957

Conor McBride. 2003b. First-order unification by structural recursion: Correctness proof. (2003). http://www.strictlypositive.org/foubsr-website/ Supplement to the paper 'First-Order Unification by Structural Recursion'.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (jan 2000), 1–44. https://doi.org/10.1145/345099.345100

Adámek Jiří Koubek Václav Reiterman Jan Trnková, Věra. 1975. Free algebras, input processes and free monads. *Commentationes Mathematicae Universitatis Carolinae* 016, 2 (1975), 339–351. http://eudml.org/doc/16691

Nachiappan Valliappan. 2023. *Modular Normalization with Types*. Ph. D. Dissertation. Department of Computer Science & Engineering, Chalmers University of Technology, Gothenburg, Sweden. https://research.chalmers.se/en/publication/

535726

Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/22.08/

Joe B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (June 1999), 111–156. https://doi.org/10.1016/S0168-0072(98)00047-5

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let arguments go first. In *Programming Languages and Systems. ESOP 2018 (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 272–299. https://doi.org/10.1007/978-3-319-89884-1_10