A formal treatment of bidirectional typing*

Liang-Ting Chen[0000−0002−3250−1331] and Hsiang-Shang Ko[0000−0002−2439−1048]

Institute of Information Science, Academia Sinica, Taiwan

Abstract There has been much progress in designing bidirectional type systems and associated type synthesis algorithms, but mainly on a case-by-case basis. To remedy the situation, this paper develops a *general* and *formal* theory of bidirectional typing and as a by-product of our formalism provides a *verified* generator of *proof-relevant* type synthesisers for simply typed languages: for every signature that specifies a mode-correct bidirectionally typed language, there exists a proof-relevant type synthesiser that for an input abstract syntax tree constructs a typing derivation if any, gives its refutation if not, or reports that the input does not have enough type annotations. Sufficient conditions for deriving a type synthesiser such as soundness, completeness, and mode-correctness, are studied universally for all signatures. We propose a preprocessing step called *mode decoration*, which helps the user to deal with missing type annotations in a given abstract syntax tree. The entire development is formalised in Agda and can be further integrated with other language-formalisation frameworks.

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, full parametric polymorphism leads to undecidability in type inference, as do dependent types [9, 30]. 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 type annotations. In their survey paper, Dunfield and Krishnaswami summarised [10] the design principles of bidirectional type synthesis and its wide coverage of languages with simple types, polymorphic types, dependent types, gradual types, among others.

While 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. Bidirectional type synthesis combines type *inference* on this subset of terms with type *checking* (based on a given type) on the rest. Formally, every judgement in a bidirectional type system is extended with a *mode:* (i) $\Gamma \vdash t \Rightarrow A$ for *synthesis* and (ii) $\Gamma \vdash t \Leftarrow A$ for *checking*. The former

* The work is supported by the National Science and Technology Council of Taiwan under grant NSTC 112-2221-E-001-003-MY3.

indicates that the type A is an 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 be 'read off' from a well-designed bidirectional type system: as the synthesiser traverses a raw term, it switches between synthesis and checking, following the modes assigned to the judgements in the typing rules.

Despite sharing the same basic idea, bidirectional typing has been mostly developed on a case-by-case basis. Dunfield and Krishnaswami present informal design principles learned from individual bidirectional type systems, but in addition to crafting special techniques for individual systems, we should start to consolidate concepts common to a class of bidirectional type systems into a general and formal theory that gives mathematically precise definitions and proves theorems for the class of systems once and for all. In this paper, we develop such a theory of bidirectional typing with the proof assistant Agda.

Proof-relevant type synthesis Our work adopts a proof-relevant approach to (bidirectional) type synthesis, as illustrated by Wadler et al. [29] for PCF. The proof-relevant formulation deviates from the usual one: traditionally, a type synthesis algorithm is presented as *algorithmic rules*, for example in the form $\Gamma \vdash t \Rightarrow A \mapsto t'$, which denotes that t in the surface language can be transformed to a well-typed term t' of type A in the core language [24]. Such an algorithm is accompanied by soundness and completeness assertions that the algorithm correctly synthesises the type of a raw term and every typable term can be synthesised. By contrast, the proof-relevant approach exploits the simultaneously computational and logical nature of Martin-Löf type theory and formulates 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 constructively, and we say that P is logically *decidable* if the law holds for P. Since Martin-Löf type theory is logical and computational, a decidability proof is a proof-relevant decision procedure that computes a yes-or-no answer with a proof of P or its refutation, so logical decidability is algorithmic decidability. More specifically, consider the statement of the type inference problem

> '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',

which can be rephrased more succinctly as

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

A proof of this statement would also be a program that produces either a typing derivation for the given raw term t or a negation proof that such a derivation is impossible. The first case is algorithmic soundness, while the second case is algorithmic completeness in contrapositive form (implying the original form due to the decidability). Hence, proving the statement is the same as constructing a verified proof-relevant type inference algorithm, which returns not only an answer

but also its justification. This is an economic way to bridge the gap between theory and practice, where proofs double as verified programs, in contrast to separately exhibiting a theory and an implementation that are loosely related.

Annotations in the type synthesis problem As mentioned in the beginning, with bidirectional typing we avoid the generally undecidable problem of type inference, and instead solve the simpler problem about the typability of 'sufficiently annotated' raw terms, which we call the type synthesis problem to distinguish it from type inference. Annotations therefore play an important role even in the definition of the problem solved by bidirectional typing, but have not received enough attention. In our theory, we define *mode derivations* so that we can explicitly take annotations into account and formulate the type synthesis problem with sufficiently annotated raw terms. Accordingly, a preprocessing step called *mode decoration* is proposed to help the user to work with annotations.

The type synthesis problem is not just about deciding whether a raw term is typable—there is a third possibility that the term does not have sufficient annotations. Therefore, to solve the type synthesis problem, before attempting to decide typability (using a bidirectional type synthesiser), we should first decide whether the raw term has sufficient annotations, which corresponds to whether the term has a mode derivation. Our theory gives a (proof-relevant) *mode decorator*, which constructs a mode derivation for a raw term or provides information that refutes the existence of any mode derivation and helps the user to pinpoint missing annotations. Then a bidirectional type synthesiser is required to decide the typability of only mode-decorated raw terms instead of all raw terms. Soundness and completeness of bidirectional typing is reformulated as a one-to-one correspondence between bidirectional typing derivations and pairs of a typing derivation and a mode derivation for the same raw term; this reformulation is more useful than annotatability [10, Section 3.2] which is typically formulated in the literature of bidirectional typing.

Mode-correctness and general definitions of languages The most essential characteristics of bidirectional typing is *mode-correctness*, since an algorithm can often be 'read off' from the definition of a bidirectionally typed language if mode-correct. As illustrated by Dunfield and Krishnaswami [10], it seems that the implications of mode-correctness have only been addressed informally so far and mode-correctness is not yet formally defined as a *property of languages*.

In order to make the notion of mode-correctness precise, we first give a general definition of bidirectional simple type systems, called *bidirectional binding signature*, extending the typed version of Aczel's binding signature [1] with modes. A general definition of typed languages allows us to define mode-correctness and to investigate its consequences rigorously, including the uniqueness of synthesised types and the decidability of bidirectional type synthesis for all mode-correct signatures. The proof of the latter theorem amounts to a generator of proof-relevant bidirectional type synthesisers (analogous to a parser generator working for unambiguous or disambiguated grammars).

To make our exposition accessible, the theory in this paper focuses on simply typed languages with a syntax-directed bidirectional type system, so that the decidability of bidirectional type synthesis can be established without any other technical assumptions. It should be possible to extend the theory to deal with more expressive types and assumptions other than mode-correctness. For instance, we briefly discuss how the theory can handle polymorphically typed languages such as System F, System $\mathsf{F}_{<:}$, and those systems using implicit type applications with additional assumptions in Section 7.

Contributions and plan of this paper In short, we develop a general and formal theory of bidirectional type synthesis for simply typed languages, including

1. general definitions for bidirectional type systems and mode-correctness;
2. mode derivations for explicitly dealing with annotations in the theory, and mode decoration for helping the user to work with annotations in practice;
3. rigorously proven consequences of mode-correctness—the uniqueness of synthesised types and the decidability of bidirectional type synthesis, which amounts to
4. a fully verified generator of proof-relevant type-synthesisers.

Our theory is fully formally developed [8] with Agda but translated to the mathematical vernacular for presentation in this paper. The formal theory doubles as a verified implementation, with no gap in between.

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 main 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 discuss further developments in Section 7 and include a demonstration in Appendix A for the use of our Agda formalisation as programs.

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[1] in Figure 1; besides the standard constructs, there is an Anno rule that allows the user to insert type annotations to facilitate type synthesis.

[1] We omit the usual conditions about named representations of variables throughout this paper.

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

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

Figure 1. Raw terms for simply typed λ-calculus

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

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

$$\frac{\Gamma \vdash t : A \supset B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ \textsc{App}$$

Figure 2. Typing derivations for simply typed λ-calculus

Correspondingly, the definition of typing derivations[2] in Figure 2 also includes an Anno 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

[2] We write '$\supset$' instead of '$\to$' for the function types of the 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

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

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

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

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.

$\boxed{V \vdash t^{\Rightarrow}}$ 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

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

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

Figure 4. Mode derivations for simply typed λ-calculus

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 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 using '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); if there is no derivation of $\Gamma \vdash t \Rightarrow A$ for any type A, then there is no derivation of $\Gamma \vdash t : A$ for any A either (the second outcome), because the bidirectional typing is *complete*:

Theorem 2.5 (Soundness and Completeness). *$\Gamma \vdash t \Rightarrow A$ if and only if $|\Gamma| \vdash t^{\Rightarrow}$ and $\Gamma \vdash t : A$.*

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

3 Bidirectionally simply typed languages

This section provides general definitions of simple types, simply typed languages, and bidirectional type systems, and uses the 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: (i) first we introduce a notion of arity and a notion of signature which includes a set[3] of operation symbols and an assignment of arities to symbols; (ii) then, given a signature, we define raw terms and typing derivations inductively by primitive rules such as VAR and a rule schema for constructs op_o indexed by an operation symbol o. 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 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. Function types $A \supset B$ and typically a base type b are included in simply typed λ-calculus and can be specified by the type signature $\Sigma_\supset$ consisting of operations fun and 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.

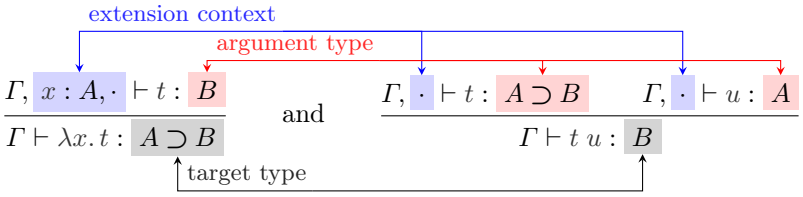
3.2 Binding signatures and simply typed languages

A simply typed language specifies (i) 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; (ii) 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 set of types for typing and extension context for variable binding.

[3] Even though our theory is developed in Martin-Löf type theory, the term 'set' is used instead of 'type' to avoid the obvious confusion. Indeed, as we assume Axiom K, all types are legitimately sets in the sense of homotopy type theory [28, Definition 3.1.1].

Definition 3.4 ([13, p. 322]). A *binding arity* with a set T of types 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 *extension context* and the *type of the i-th argument*, respectively, and A the *target type*. For brevity, it is denoted by $[\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A$ where $[\Delta_i]$ is omitted if empty.

Example 3.5. Observe that Abs and App rules in Figure 2 can be read as



if the empty extension context $\cdot$ is added verbosely, so they can be specified by binding arities $[A]B \to (A \supset B)$ and $(A \supset B), A \to B$ respectively.

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 is a set O with a function

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

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

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

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 (i) variables in V, (ii) annotations $t : A$ for some raw term t in V and a type A, and (iii) a construct $\mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ for some $o \colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ 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 5 respectively.

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

Definition 3.8. A *typing context* $\Gamma \colon \mathsf{Cxt}_\Sigma$ 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 $|\Gamma|$.

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

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

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1 \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)}\ \textsc{Op}$$

for $o \colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ in Ω

Figure 5. Raw terms

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 derivation $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A$ 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 $\Gamma \vdash_{\Sigma,\Omega} (t \fatsemi A) : A$ if $\Gamma \vdash_{\Sigma,\Omega} t : A$ has a derivation, and
3. a derivation of $\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) : A_0\langle\rho\rangle$ for an operation $o \colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ if there are $\rho \colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$ 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 6 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 $\mathsf{app} \colon A, B \rhd (A \supset B), A \to B$ and $\mathsf{abs} \colon A, B \rhd [A]B \to (A \supset B)$. 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 $\mathsf{op}_{\mathsf{app}}(t, u)$ and $\mathsf{op}_{\mathsf{abs}}(x.t)$, 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 set T of types is an inhabitant of

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

For clarity, an 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

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

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 $\mathsf{Ty}_\Sigma(\Xi)$ as types. 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 7 and particularly

  $$\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^d A_0\langle\rho\rangle$$

  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 $\rho \colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$ 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 8.

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 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 $\Omega^{\Leftrightarrow}_{\Lambda}$ which consists of

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

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).

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

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

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

for $o \colon \Xi \rhd [\Delta_1]A_1, \ldots, [\Delta_n]A_n \to A_0$ in Ω

Figure 6. Typing derivations

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

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

$$\frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} B \qquad B = A}{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}\ \textsc{Sub}^{\Leftarrow}$$

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

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

Figure 7. Bidirectional typing derivations

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

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

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1{}^{d_1} \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n{}^{d_n}}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^{d}}\ \textsc{Op}$$

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

Figure 8. Mode derivations

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 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 formulate annotatability [10] 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 modes and keeps everything else intact; this can be straightforwardly extended by induction to remove modes from a bidirectional typing derivation and arrive at an ordinary typing derivation, which is soundness. Instead we can remove typing and retain modes, arriving at a mode derivation. Conversely, if we have both mode and typing derivations for the same term, we can combine them to 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, directed by the same raw term. All these can be summarised in one theorem (proved by induction).

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

4.2 Generalised mode decoration

The 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 shall do better: if a mode decorator returns a proof that no mode derivation exists, that negation proof does not give useful information for the user. It will be 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.[4]

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. We can, however, consider *generalised mode derivations* (Figure 9) that allow the use of an additional MISSING$^{\Rightarrow}$ rule for such

[4] For the sake of simplicity, we use ordinary mode decoration elsewhere in this paper.

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

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

$$\frac{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,\mathbf{F}\,\mathbf{F}}}\ \textsc{Missing}^{\Rightarrow} \qquad \frac{V \vdash_{\Sigma,\Omega} t^{\Rightarrow\,g\,\mathbf{T}}}{V \vdash_{\Sigma,\Omega} t^{\Leftarrow\,g\,\mathbf{F}}}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{V, \vec{x}_1 \vdash_{\Sigma,\Omega} t_1{}^{d_1\;g_1\;s_1} \qquad \cdots \qquad V, \vec{x}_n \vdash_{\Sigma,\Omega} t_n{}^{d_n\;g_n\;s_n}}{V \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\,t_1; \ldots; \vec{x}_n.\,t_n)^{\,d\;(\bigwedge_i g_i)\;\mathbf{T}}}\ \textsc{Op}$$

Figure 9. Generalised mode derivations

switching, so that a derivation can always be constructed. Given a generalised mode derivation, if it uses $\textsc{Missing}^{\Rightarrow}$ in some places, then those places are exactly where annotations should be supplied; if it does not use $\textsc{Missing}^{\Rightarrow}$, 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 $\textsc{Missing}^{\Rightarrow}$ rule. (Ignore the boolean s for now.)

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

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 $\textsc{Sub}^{\Leftarrow}$ and $\textsc{Missing}^{\Rightarrow}$, 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 $\textsc{Sub}^{\Leftarrow}$ and $\textsc{Missing}^{\Rightarrow}$ 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 $\textsc{Missing}^{\Rightarrow}$ 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'.

Now we are ready to construct a generalised mode decorator.

Theorem 4.4 (Generalised mode decoration). *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* 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*

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

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

With a generalised mode decorator, it is now easy to derive an ordinary one.

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

4.3 Annotatability

Dunfield and Krishnaswami [10, Section 3.2] formulated completeness differently from our Theorem 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 10, 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 straightforwardly proved by induction on the derivation produced by generalised mode decoration (Theorem 4.4) to construct a bidirectional typing derivation in the same mode. The interesting case is MISSING$^{\Rightarrow}$, which is mapped to ANNO$^{\Rightarrow}$, adding to the term a type annotation that comes from the given typing derivation.

$\boxed{t \sqsupseteq u}$ A raw term t is more annotated than u

$$\frac{t \sqsupseteq u}{(t : A) \sqsupseteq u}\ \textsc{More} \qquad \frac{}{x \sqsupseteq x}\ \textsc{Var} \qquad \frac{t \sqsupseteq u}{(t : A) \sqsupseteq (u : A)}\ \textsc{Anno}$$

$$\frac{t_1 \sqsupseteq u_1 \quad \cdots \quad t_n \sqsupseteq u_n}{\mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \sqsupseteq \mathsf{op}_o(\vec{x}_1.\, u_1; \ldots; \vec{x}_n.\, u_n)}\ \textsc{Op}$$

Figure 10. Annotation ordering between raw terms

On the other hand, when using a bidirectional type synthesiser to implement a type synthesiser, for example in Theorem 2.4, if the bidirectional type synthesiser concludes that there does not exist a bidirectional typing derivation, we use the contrapositive form of completeness to establish that such an ordinary typing derivation does not exist either. Now, annotatability is a kind of completeness because (roughly speaking) it turns an ordinary typing derivation bidirectional. Therefore it is conceivable that we could use annotatability in place of completeness in the proof of Theorem 2.4. However, in the contrapositive form of annotatability, the antecedent is 'there does not exist t' that is more annotated than t and has a bidirectional typing derivation', which is more complex than the bidirectional type synthesiser would have to produce. Annotatability also does not help the user to deal with missing annotations: 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 our theory offers simpler and more useful alternatives than the notion of annotatability.

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 [10] outlined, mode-correctness for a bidirectional typing rule means that (i) 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; (ii) 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 $\textsc{Abs}^{\Leftarrow}$ (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 $\textsc{App}^{\Rightarrow}$ for an application term $t\ u$ is mode-correct because: (i) 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; (ii) the type of the second argument u is checked against A, which has been synthesised earlier; (iii) 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 $[\Delta_i]A_i^{d_i}$ 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 subsets[5] of *(free) variables* of a type A and of variables in an extension context Δ are denoted by $fv(A)$ and $fv(\Delta)$ respectively. For an argument list $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$ the set of type variables $A_i^{d_i}$ with d_i being $\Rightarrow$ is denoted by $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$, i.e. $fv^{\Rightarrow}$ gives the set of 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 a predicate defined by

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

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

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

where $\mathsf{MC}_{as}(\cdot) = \top$ means that 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 also be met for every tail of the argument list to ensure that 'output' variables accessible at each argument are from preceding arguments only, hence an inductive definition.

Definition 5.3. An operation $o\colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$ 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.

[5] There are various definitions for finite subsets of a set within type theory, but for our purposes the choice among these definitions is not a matter of concern.

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.

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

It is easy to check the bidirectional type system $(\Sigma_{\supset}, \Omega^{\Leftrightarrow}_{\Lambda})$ for the simply typed λ-calculus is mode-correct by definition or by the following lemma:

Lemma 5.5. *For any operation $o\colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d}$, it is decidable whether o is mode-correct.*

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 5.6 (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 $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ 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.

2. If d_{i+1} is $\Leftarrow$, then the statement holds by induction hypothesis.
3. If d_{i+1} is $\Rightarrow$, then $\Delta_{i+1}\langle\rho_1\rangle = \Delta_{i+1}\langle\rho_2\rangle$ 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 $fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n})$.

□

5.2 Decidability of bidirectional type synthesis and checking

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

Theorem 5.7. *In a mode-correct bidirectional type system* (Σ, Ω)*,*

1. *if* $|\Gamma| \vdash_{\Sigma,\Omega} t^{\Rightarrow}$*, then it is decidable whether* $\Gamma \vdash_{\Sigma,\Omega} t \Rightarrow A$ *for some* 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 $\mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)$ contains a substitution $\rho\colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$. 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 $[\Delta_{i+1}]A_{i+1}^{\Rightarrow}$, 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 5.8. 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.[6]

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

[6] 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 Appendix A for details). For the sake of clarity in presentation, though, we keep using named variables and just assume that Ξ has a decidable equality.

Definition 5.9. 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 5.10. *For any A of $\mathsf{Ty}_\Sigma(\Xi)$, B of $\mathsf{Ty}_\Sigma(\emptyset)$, and a partial substitution $\rho\colon \Xi \to \mathsf{Ty}_\Sigma(\emptyset)$,*

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 [21, 22], or be proved directly without unification. We are now ready for Theorem 5.7:

Proof (of Theorem 5.7). 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 independent of mode-correctness. The case SUB$^{\Leftarrow}$ invokes the uniqueness of synthesised types 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. [29], so we only detail the last case OP which is new and has been discussed informally above. For brevity we omit the subscript (Σ, Ω).

For a mode derivation of $|\Gamma| \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n)^d$, we first claim:

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

$$dom(\bar{\rho}) \supseteq fv^{\Rightarrow}([\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}) \text{ and } \Gamma, \vec{x}_i : \Delta_i\langle\bar{\rho}\rangle \vdash t_i\colon A_i\langle\bar{\rho}\rangle^{d_i} \quad (2)$$

   for $i = 1, \ldots, n$
2. or there is no extension σ of ρ such that (2) 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 (2) holds but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A$ for some A, then by inversion we have $\rho\colon \mathsf{Sub}_\Sigma(\Xi, \emptyset)$ such that

$$\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i\colon A_i\langle\rho\rangle^{d_i}$$

      for every i. Obviously, $\rho \geqslant \rho_0$ and $\Gamma, \vec{x}_i : \Delta_i\langle\rho\rangle \vdash t_i\colon A_i\langle\rho\rangle^{d_i}$ 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 (2) holds, then by mode-correctness $\bar{\rho}$ is total and thus

$$\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Rightarrow A_0\langle\bar{\rho}\rangle\,.$$

– d is $\Leftarrow$: Let A be a type and apply Lemma 5.10 with ρ_0 defined nowhere.

1. If there is no $\sigma \geqslant \rho_0$ s.t. $A_0\langle\sigma\rangle = A$ but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then inversion gives us a substitution ρ s.t. $A = A_0\langle\rho\rangle$—a contradiction.
2. If there is a minimal $\bar{\rho} \geqslant \rho_0$ s.t. $A_0\langle\bar{\rho}\rangle = A$, then apply our claim with $\bar{\rho}$:
   (a) If there is no $\sigma \geqslant \bar{\rho}$ satisfying (2) but $\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A$, then by inversion there is γ s.t. $A_0\langle\gamma\rangle = A$ and, for every i, $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i \colon A_i\langle\gamma\rangle^{d_i}$. Given that $\bar{\rho} \geqslant \rho$ is minimal s.t. $A_0\langle\bar{\rho}\rangle = A$, then γ is an extension of $\bar{\rho}$ but by assumption no such an extension satisfying $\Gamma, \vec{x}_i : \Delta_i\langle\gamma\rangle \vdash t_i \colon A_i\langle\gamma\rangle^{d_i}$ exists, thus a contradiction.
   (b) If there is a minimal $\bar{\bar{\rho}} \geqslant \bar{\rho}$ s.t. (2), then by mode-correctness $\bar{\bar{\rho}}$ is total and
   $$\Gamma \vdash \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) \Leftarrow A_0\langle\bar{\bar{\rho}}\rangle$$
   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 $dom(\bar{\rho})$.

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 (2) 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 $\sigma \geqslant \rho$ s.t. (2) holds for all $1 \leqslant i \leqslant m$ but a minimal $\gamma \geqslant \rho$ such that (2) holds for all $1 \leqslant i \leqslant m+1$, then we have a contradiction.
   (b) There is a minimal $\bar{\rho} \geqslant \rho$ s.t. (2) 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 ind. hyp. $\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Leftarrow A_{m+1}\langle\bar{\rho}\rangle$ 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 easily derive a contradiction.
   - d_{m+1} is $\Rightarrow$: By mode-correctness, $\Delta_{m+1}\langle\bar{\rho}\rangle$ is defined. By the ind. hyp., '$\Gamma, \vec{x}_{m+1} : \Delta_{m+1}\langle\bar{\rho}\rangle \vdash t_{m+1} \Rightarrow A$ for some A' is decidable:
     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. (2) 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 Lemma 5.10 gives following two cases:
     - Suppose no $\sigma \geqslant \bar{\rho}$ s.t. $A_{m+1}\langle\sigma\rangle = A$ but an extension $\gamma \geqslant \rho$ s.t. (2) 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 5.6, the synthesised type $A_{m+1}\langle\gamma\rangle$ must be unique, so γ is an extension of $\bar{\rho}$ s.t. $A_{m+1}\langle\gamma\rangle = A$, i.e. a contradiction.
     - If there is a minimal $\bar{\bar{\rho}} \geqslant \bar{\rho}$ such that $A_{m+1}\langle\bar{\bar{\rho}}\rangle = A$, then it is not hard to show that $\bar{\bar{\rho}}$ is also the minimal extension of ρ such that (2) holds for all $1 \leqslant i \leqslant m+1$.

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

We have completed the proof by induction on the 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 provide either the typing derivation or its negation proof. Each case analysis branches depending on the outcomes of bidirectional type synthesis and checking for each sub-term, as well as the unification process. If a negation proof is not of interest in practice, these programs can be simplified by discarding the cases that yield negation proofs. Alternatively, we could consider generalising typing derivations instead, like our generalised mode derivations (Figure 9), to reformulate negation proofs positively to deliver more informative error messages. This would assist programmers in resolving issues with ill-typed terms, rather than 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 derive a type synthesiser parameterised by (Σ, Ω), generalising Theorem 2.4.

Corollary 5.11 (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, *or*
3. $|\Gamma| \nvdash_{\Sigma,\Omega} t^{\Rightarrow}$.

6 Examples

To exhibit the applicability of our approach, we discuss two more examples: one has infinitely many operations and the other includes many more constructs than simply typed λ-calculus, exhibiting the practical side of a general treatment.

6.1 Spine application

A spine application $t\ u_1\ \ldots\ u_n$ is a form 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, and has been used by Agda's core language.

At first glance, accommodating this form of application may seem impossible, given that the number of arguments for a construct is finite and has to be fixed. Nonetheless, the total number of operation symbols in a signature need not be finite, allowing us to establish a corresponding construct for each number n of arguments, i.e. viewing the following rule

$$\frac{\Gamma \vdash t \Rightarrow A_1 \supset (A_2 \supset (\cdots \supset (A_n \supset B)\ldots)) \qquad \Gamma \vdash u_1 \Leftarrow A_1 \qquad \cdots \qquad \Gamma \vdash u_n \Leftarrow A_n}{\Gamma \vdash t\ u_1\ \ldots\ u_n \Rightarrow B}$$

as a rule schema parametrised by n, so the signature $\Omega^{\Leftrightarrow}_{\Lambda}$ can be extended with

$$\mathsf{app}_n \colon A_1, \ldots, A_n, B \rhd A_1 \supset (A_2 \supset (\cdots \supset (A_n \supset B)\ldots))^{\Rightarrow}, A_1^{\Leftarrow}, \ldots, A_n^{\Leftarrow} \to B$$

Each application $t\ u_1\ \ldots\ u_n$ can be introduced as $\mathsf{op}_{\mathsf{app}_n}(t; u_1; \ldots; u_n)$, thereby exhibiting the necessity of having an arbitrary set for operation symbols.

6.2 Computational calculi

Implementing a stand-alone type synthesiser for a simply typed language is typically a straightforward task. However, the code size increases proportionally to the number of type constructs and of arguments associated with each term construct. When dealing with a fixed number n of type constructs, for each synthesising construct, there are two cases for a checking argument but $n+1$ cases for each synthesising argument—the successful synthesis of the expected type, an instance where it fails, or $n-1$ cases where the expected type does not match; similarly for a checking construct—making the task tedious. Thus, having a generator is helpful and can significantly reduce the effort for implementation.

For illustrative purposes, consider a computational calculus [23] with additional constructs listed in Table B.1. The extended language has 'only' 15 constructs, far fewer than a realistic programming language would have, but there are nearly 100 possible cases to consider in bidirectional type synthesis.

On the other hand, similar to a parser generator, a type-synthesiser generator only needs a specification (Σ, Ω) from the user to produce a corresponding synthesiser. Such a specification can be derived by extending $\Sigma_{\supset}$ with additional type constructs with nat, prod, sum, and T such that

$$ar(\mathsf{nat}) = 0 \qquad ar(\mathsf{T}) = 1 \qquad ar(\mathsf{prod}) = ar(\mathsf{sum}) = 2.$$

Types $\mathtt{nat}$, $A \times B$, $A + B$, and $T(A)$ are given as $\mathsf{op}_{\mathsf{nat}}$, $\mathsf{op}_{\mathsf{prod}}(A, B)$, $\mathsf{op}_{\mathsf{sum}}(A, B)$, and $\mathsf{op}_{\mathsf{T}}(A)$ respectively. The signature $\Omega^{\Leftrightarrow}_{A}$ is then extended with operations listed in Table B.1. Mode-correctness can be derived by Lemma 5.5 and its type synthesiser by Corollary 5.11 with the specification (Σ, Ω) directly.[7]

7 Discussion

We believe that our formal treatment lays a foundation for further investigation, as the essential aspects of bidirectional typing have been studied rigorously. While our current development is based on simply typed languages to highlight the core ideas, it is evident that many concepts and aspects remain untouched. In this section we discuss related work and potential directions for future research.

Language formalisation frameworks The study of presenting logics universally at least date back to universal algebra and model theory where structures are studied for certain notions of arities and signatures. In programming language theory, Aczel's binding signature [1] is an example which is used to prove a general confluence theorem. Many general definitions and frameworks for defining logics and type theories have been proposed, and we classify them into two groups by where signatures reside: the meta level or the object level of a meta-language:

1. Harper et al.'s logical framework LF [17] and its family of variants [5, 11, 18] are *extensions* of Martin-Löf type theory where signatures are on the *meta level* and naturally capable of specifying dependent type theories;

[7] For a demonstration in Agda, see Appendix A.

2. general dependent type theories [6, 7, 19, 27], categorical semantics [4, 12–16, 25, 26] (which includes the syntactic model as a special case), and frameworks for substructural systems [25, 26, 31] are developed *within* a meta-theory (set theory or type theory) where signatures are on the *object level* and their expressiveness varies depending on their target languages.

LF is naturally expressive but each extension requires a different LF and thus a different implementation to check formal LF proofs. Formalising LF and its variants is at least as hard as formalising a dependent type theory, so they are mostly implemented separately from their theory and unverified.

For the second group, theories developed in set theory can often be reformulated in type theory and thus manageable for formalisation in a type-theoretic proof assistant. Such examples include frameworks developed by Ahrens et al. [2], Allais et al. [3], Fiore and Szamozvancev [14] in Agda, which are relatively correct to their meta-language, although their expressiveness is limited to simply typed theories. Our work belongs to the second group, as we aim for a formalism in a type theory to minimise the gap between theory and implementation.

Beyond simple types Bidirectional type synthesis plays a crucial role in handling more complex types than simple types, and we sketch how our theory can be extended to treat a broader class of languages. First, we need a general definition of languages, discussed above, in question (Sections 3.1 and 3.2). Then, this definition can be augmented with modes (Section 3.3) and the definition of mode-correctness (Definition 5.3) can be adapted by adding the mode information into signatures. Soundness and completeness (Theorem 4.1) should still hold, as they amount to the separation and combination of mode and typing information for a given raw term (in a syntax-directed formulation). Mode decoration (Section 4.2) involves annotating a raw term with modes and marking missing annotations, and should also work. As for the decidability of bidirectional type synthesis, we discuss two cases involving polymorphic types and dependent types below.

Polymorphic types In the case of languages like System F and others that permit type-level variable binding, we can start with the notion of polymorphic signature, as introduced by Hamana [16]—(i) each type construct in a signature is specified by a binding arity with only one type $*$, and (ii) a term construct can employ a pair of extension contexts for term variables and type variables.

Extending general definitions for bidirectional typing and mode derivations from Hamana's work is straightforward. For example, the universal type $\forall\alpha.\, A$ and type abstraction in System F can be specified as operations:

$$\mathsf{all} : * \triangleright [*]* \to * \quad \text{and} \quad \mathsf{tabs} : [*]A \triangleright \langle * \rangle\, A^{\Leftarrow} \to \mathsf{op}_{\mathsf{all}}(\alpha.A)^{\Leftarrow}$$

The decidability of bidirectional type synthesis (Theorem 5.7) should also carry over, as no equations are imposed on types and no guessing (for type application) is required. Adding subtyping $A <: B$ to languages can be done by replacing type equality with a subtyping relation $<:$ and type equality check with subtyping

check, so polymorphically typed languages with subtyping such as System $\mathsf{F}_{<:}$ can be specified. The main idea of bidirectional typing does not change, so it should be possible to extend the formal theory without further assumptions too.

However, explicit type application in System F and System $\mathsf{F}_{<:}$ is impractical but its implicit version results in a *stationary rule* [20]:

$$\frac{\Gamma \vdash t \Rightarrow \forall \alpha.A}{\Gamma \vdash t \Rightarrow A[B/\alpha]}$$

which is not syntax-directed. By translating the above rule to subtyping, we have the *instantiation problem* which amounts to guessing B in $\forall \alpha.A <: A[B/\alpha]$. A theory that accommodates various solutions to the problem is left as future work.

Dependent types In the context of dependent type theory, type synthesis entails a term equality check for type equality. Achieving decidable bidirectional type synthesis depends on the decidability of term equality, which, in turn, relies on the ability to specify computation rules within a language specification.

Extending the logical framework Dedukti [5], Felicissimo's recent CompLF [11] is capable of defining dependent type theories with computational rules and enables generic bidirectional type synthesis. They propose a notion of LF signature with modes (called *moded signatures* op. cit.), a definition of mode-correctness (called *well-typed* op. cit.), and an algorithm of generic bidirectional type synthesis which is decidable, sound, and complete with respect to mode-decorated terms, provided that the specified language is mode-correct, and the set of computational rules is *well-behaved*, i.e. type-preserving, confluent, and strongly normalising. However, the last assumption is challenging to establish even for specific languages and lacks a general understanding.

Beyond syntax-directedness To explore general settings, a possibility is to independently specify an ordinary and a bidirectional type system over the same raw terms and then investigate relationships between the two systems.

However, a practical setting suitable for a next step is probably this: 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 and different orders of premises. For this setting, both the mode decorator and the bidirectional type synthesiser will have to backtrack. 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. It should be possible 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 mode derivations into a bidirectional typing derivation.

We could instead focus on simpler cases first where each construct has exactly one typing rule, but each typing rule can be refined to more than one bidirectional typing rule. In such cases, the mode decorator would need to find all mode derivations, but the type synthesiser should still work in a syntax-directed manner on each mode derivation as input. Completeness could still take the simple form presented in this paper too.

Towards a richer formal theory There are more techniques in bidirectional typing that could be formally studied in general, with one notable example being the Pfenning recipe for bidirectionalising typing rules [10, Section 4]. There are also concepts that may be hard to fully formalise, for example 'annotation character' [10, Section 3.4], which is roughly about how easy it is for the user to write annotated programs, but it would be interesting to explore to what extent such concepts can be formalised.

Acknowledgements. We thank Kuen-Bang Hou (Favonia) and anonymous reviewers for their comments and suggestions.

Bibliography

[1] Aczel, P.: A general Church–Rosser theorem (1978), URL http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf, unpublished note

[2] Ahrens, B., Matthes, R., Mörtberg, A.: Implementing a category-theoretic framework for typed abstract syntax. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 307–323, CPP 2022, Association for Computing Machinery, New York, NY, USA (2022), https://doi.org/10.1145/3497775.3503678

[3] Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. Journal of Functional Programming **31**(1996), e22 (Oct 2021), ISSN 0956-7968, https://doi.org/10.1017/S0956796820000076

[4] Arkor, N., Fiore, M.: Algebraic models of simple type theories: A polynomial approach. In: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 88–101, LICS '20, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450371049, https://doi.org/10.1145/3373718.3394771, URL https://doi.org/10.1145/3373718.3394771

[5] Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory (2023), URL https://arxiv.org/abs/2311.07185

[6] Bauer, A., Haselwarter, P.G., Lumsdaine, P.L.: A general definition of dependent type theories (2020), https://doi.org/10.48550/arXiv.2009.05539

[7] Bauer, A., Komel, A.P.: An extensible equality checking algorithm for dependent type theories. Logical Methods in Computer Science **18**(1) (Jan 2022), https://doi.org/10.46298/lmcs-18(1:17)2022

[8] Chen, L.T., Ko, H.S.: A formal treatment of bidirectional typing (artefact) (2024), https://doi.org/10.5281/zenodo.10458840

[9] Dowek, G.: The undecidability of typability in the lambda-pi-calculus. In: Bezem, M., Groote, J.F. (eds.) Typed Lambda Calculi and Applications. TLCA 1993, Lecture Notes in Computer Science, vol. 664, pp. 139–145, Springer, Berlin, Heidelberg (1993), ISBN 978-3-540-47586-6, https://doi.org/10.1007/BFb0037103

[10] Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Computing Surveys **54**(5), 98:1–98:38 (May 2021), ISSN 0360-0300, https://doi.org/10.1145/3450952

[11] Felicissimo, T.: Generic bidirectional typing for dependent type theories (2023), https://doi.org/10.48550/arXiv.2307.08523

[12] Fiore, M., Hamana, M.: Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In: 2013 28th Annual

ACM/IEEE Symposium on Logic in Computer Science, pp. 520–529, IEEE, New Orleans, LA, USA (2013), ISBN 978-1-4799-0413-6, ISSN 10436871, https://doi.org/10.1109/LICS.2013.59

[13] Fiore, M., Hur, C.K.: Second-order equational logic (extended abstract). In: Dawar, A., Veith, H. (eds.) Computer Science Logic. CSL 2010, Lecture Notes in Computer Science, vol. 6247, pp. 320–335, Springer Berlin Heidelberg, Berlin, Heidelberg (2010), ISBN 978-3-642-15205-4, https://doi.org/10.1007/978-3-642-15205-4_26

[14] Fiore, M., Szamozvancev, D.: Formal metatheory of second-order abstract syntax. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (Jan 2022), ISSN 2475-1421, https://doi.org/10.1145/3498715

[15] Fiore, M.P., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: Proceedings. 14th Symposium on Logic in Computer Science, pp. 193–202, IEEE, Trento, Italy (1999), ISBN 0-7695-0158-3, https://doi.org/10.1109/LICS.1999.782615, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=782615

[16] Hamana, M.: Polymorphic abstract syntax via Grothendieck construction. In: Hofmann, M. (ed.) Foundations of Software Science and Computational Structures. FoSSaCS 2011, Lecture Notes in Computer Science, vol. 6604, pp. 381–395, Springer, Berlin, Heidelberg (2011), https://doi.org/10.1007/978-3-642-19805-2_26

[17] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of ACM **40**(1), 143–184 (jan 1993), ISSN 0004-5411, https://doi.org/10.1145/138027.138060

[18] Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of Functional Programming **17**(4-5), 613–673 (2007), https://doi.org/10.1017/S0956796807006430

[19] Haselwarter, P.G., Bauer, A.: Finitary type theories with and without contexts (2021), URL https://arxiv.org/abs/2112.00539

[20] Leivant, D.: Typing and computational properties of lambda expressions. Theoretical Computer Science **44**, 51–68 (1986), ISSN 0304-3975, https://doi.org/10.1016/0304-3975(86)90109-X

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

[22] McBride, C.: First-order unification by structural recursion: Correctness proof (2003), URL http://www.strictlypositive.org/foubsr-website/, supplement to the paper 'First-Order Unification by Structural Recursion'

[23] Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the 4th Symposium on Logic in Computer Science (LICS'89), pp. 14–23, IEEE Computer Society Press, Los Alamitos, Calif. (1989), https://doi.org/10.1109/LICS.1989.39155

[24] Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems **22**(1), 1–44 (jan 2000), ISSN 0164-0925, https://doi.org/10.1145/345099.345100

[25] Tanaka, M., Power, A.J.: Pseudo-distributive laws and axiomatics for variable binding. Higher-Order and Symbolic Computation **19**(2-3), 305–337 (Sep 2006), ISSN 1388-3690, https://doi.org/10.1007/s10990-006-8750-x, URL http://link.springer.com/10.1007/s10990-006-8750-x
[26] Tanaka, M., Power, J.: A unified category-theoretic semantics for binding signatures in substructural logic. Journal of Logic and Computation **16**(1), 5–25 (02 2006), ISSN 0955-792X, https://doi.org/10.1093/logcom/exi070, URL https://doi.org/10.1093/logcom/exi070
[27] Uemura, T.: Abstract and Concrete Type Theories. Ph.D. thesis, University of Amsterdam (July 2021), URL https://hdl.handle.net/11245.1/41ff0b60-64d4-4003-8182-c244a9afab3b
[28] Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
[29] Wadler, P., Kokke, W., Siek, J.G.: Programming Language Foundations in Agda (Aug 2022), URL https://plfa.inf.ed.ac.uk/22.08/
[30] Wells, J.B.: Typability and type checking in System F are equivalent and undecidable. Annals of Pure and Applied Logic **98**(1-3), 111–156 (Jun 1999), ISSN 01680072, https://doi.org/10.1016/S0168-0072(98)00047-5
[31] Wood, J., Atkey, R.: A framework for substructural type systems. In: Sergey, I. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 13240, pp. 376–402, Springer International Publishing, Cham (2022), ISBN 978-3-030-99336-8, https://doi.org/10.1007/978-3-030-99336-8_14

A Demonstration

As our theory is developed with Agda constructively, the formal counterparts of our development can be used as programs directly. We sketch their use with our running example, the simply typed λ-calculus, by (i) specifying the language $(\Sigma_{\supset}, \Omega^{\Leftrightarrow})$, (ii) showing it mode-correct, and then (iii) instantiating its type synthesiser.

Specifying a language Signatures $\Lambda_t\mathsf{D}$ and $\Lambda{\Leftrightarrow}\mathsf{D}$ are defined for $\Sigma_{\supset}$ and $\Omega^{\Leftrightarrow}$:

```
data ΛtOp : Set where base imp : ΛtOp

ΛtD : S.SigD
ΛtD = sigd ΛtOp λ where base → 0; imp → 2

open import Syntax.BiTyped.Signature ΛtD

data ΛOp : Set where `app `abs : ΛOp

Λ⇔D : SigD
Λ⇔D .Op = ΛOp
Λ⇔D .ar = λ where
```

```
`app → 2 ▷ ρ[        [] ⊢[ Chk ] ` 1 ] ρ[ [] ⊢[ Syn ] ` 1 ⊃ ` 0 ] [] ⇒ ` 0
`abs → 2 ▷ ρ[ ` 1 :: [] ⊢[ Chk ] ` 0 ]                          [] ⇐ ` 1 ⊃ ` 0
```

where (i) S.SigD is the type of type signatures, (ii) SigD is the type of bidirectional binding signatures, (iii) 2 indicates the number of type variable variables in an operation, (iv) ` i is the i-th type variable, and the definitions of decidable equality for Λ_tOp and ΛOp are omitted above.

Proving mode-correctness To prove that the specified language $(\Sigma_\supset, \Omega^\Leftrightarrow)$ is mode-correct, we simply invoke Lemma 5.5 for each construct:

```
open import Theory.ModeCorrectness.Decidability Λt⇔D

mcΛ⇔D : ModeCorrect Λ⇔D
mcΛ⇔D `app = from-yes (ModeCorrect^c? (Λ⇔D .ar `app))
mcΛ⇔D `abs = from-yes (ModeCorrect^c? (Λ⇔D .ar `abs))
```

where from-yes extracts the positive witness from an inhabitant of the Dec type.

Instantiating a type synthesiser Now that we have the definition $\Lambda\Leftrightarrow$D for the bidirectional type system $(\Sigma_\supset, \Omega^\Leftrightarrow)$ with mc$\Lambda\Leftrightarrow$D for its mode-correctness, we can instantiate its type synthesiser (Corollary 5.11) just by importing the module

```
open import Theory.Trichotomy Λ⇔D mcΛ⇔D
```

where the type synthesiser is defined:

```
synthesise
  : (Γ : Cxt) (r : Raw (length Γ))
  → Dec (∃[ A ] Γ ⊢ r ꞉ A) ⊎ ¬ Pre Syn r
```

Running a type synthesiser Every statement in our development so far has been formally proved without any postulates, so the proof synthesiser can actually compute as a program! The type of raw terms for $(\Sigma_\supset, \Omega^\Leftrightarrow)$ are defined in the module Syntax.Typed.Raw.Term with the mode-erased signature erase $\Lambda\Leftrightarrow$D:

```
open import Theory.Erasure
open import Syntax.Typed.Raw.Term (erase Λ⇔D)
```

As raw terms for operations are defined generically using a single constructor op, it is not so convenient to manipulate them directly. Hence, we use pattern synonyms and define raw terms in a form close to the informal presentation:

```
infixl 8 _·_
infixr 7 λ_

pattern _·_ r s = op (`app , s , r , _)
pattern λ_ r = op (`abs , r , _)
```

```
S : Raw n
S = ƛ ƛ ƛ ` suc (suc zero) · ` zero · (` suc zero · ` zero)
```

Then, invoking the program synthesise with S and its required type annotation

```
⊢S? = synthesise [] (((b ⊃ b ⊃ b) ⊃ (b ⊃ b) ⊃ b ⊃ b) ∋ S)
```

gives us a typing derivation as expected.

B Detailed definitions

Definition B.1. 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 $[\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n}$, the set of *synthesised type variables* is defined by

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

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

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

Rules	Operations
$\dfrac{}{\Gamma \vdash \mathtt{z} \Leftarrow \mathtt{nat}}$	$\mathsf{z}\colon \cdot \rhd \cdot \to \mathtt{nat}^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Leftarrow \mathtt{nat}}{\Gamma \vdash \mathtt{s}(t) \Leftarrow \mathtt{nat}}$	$\mathsf{s}\colon \cdot \rhd \mathtt{nat}^{\Leftarrow} \to \mathtt{nat}^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Rightarrow \mathtt{nat} \quad \Gamma \vdash t_0 \Leftarrow A \quad \Gamma, x : \mathtt{nat} \vdash t_1 \Leftarrow A}{\Gamma \vdash \mathtt{ifz}(t_0; x.t_1)(t) \Leftarrow A}$	$\mathsf{ifz}\colon A \rhd \mathtt{nat}^{\Rightarrow}, A^{\Leftarrow}, A^{\Leftarrow} \to A^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Leftarrow A \quad \Gamma \vdash u \Leftarrow B}{\Gamma \vdash (t, u) \Leftarrow A \times B}$	$\mathsf{pair}\colon A, B \rhd A^{\Leftarrow}, B^{\Leftarrow} \to A \times B^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Rightarrow A_1 \times A_2}{\Gamma \vdash \mathtt{proj}_i(t) \Rightarrow A_i}$ for $i = 1, 2$	$\mathsf{proj}_i\colon A_1, A_2 \rhd A_1 \times A_2^{\Rightarrow} \to A_i^{\Rightarrow}$ for $i = 1, 2$
$\dfrac{\Gamma \vdash t \Leftarrow A_i}{\Gamma \vdash \mathtt{inj}_i(t) \Leftarrow A_1 + A_2}$ for $i = 1, 2$	$\mathsf{inj}_i\colon A_1, A_2 \rhd A_i^{\Leftarrow} \to A_1 + A_2^{\Leftarrow}$ for $i = 1, 2$
$\dfrac{\Gamma \vdash u \Rightarrow A + B \quad \Gamma, x_1 : A \vdash t_1 \Leftarrow C \quad \Gamma, x_2 : B \vdash t_2 \Leftarrow C}{\Gamma \vdash \mathtt{case}(u; x_1.t_1; x_2.t_2) \Leftarrow C}$	$\mathsf{case}\colon A, B, C \rhd A + B^{\Rightarrow}, [A]C^{\Leftarrow}, [B]C^{\Leftarrow} \to C^{\Rightarrow}$
$\dfrac{\Gamma, x : A \vdash t \Leftarrow A}{\Gamma \vdash \mu x.\, t \Leftarrow A}$	$\mathsf{mu}\colon A \rhd [A]A^{\Leftarrow} \to A^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Rightarrow A \quad \Gamma, x : A \vdash u \Leftarrow B}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u \Leftarrow B}$	$\mathsf{let}\colon A, B \rhd A^{\Rightarrow}, [A]B^{\Leftarrow} \to B^{\Leftarrow}$
$\dfrac{\Gamma \vdash t \Rightarrow A}{\Gamma \vdash \mathtt{ret}(t) \Rightarrow T(A)}$	$\mathsf{ret}\colon A \rhd A^{\Rightarrow} \to T(A)^{\Rightarrow}$
$\dfrac{\Gamma \vdash t \Rightarrow T(A) \quad \Gamma, x : A \vdash u \Rightarrow T(B)}{\Gamma \vdash \mathtt{bind}(t; x.u) \Rightarrow T(B)}$	$\mathsf{bind}\colon A, B \rhd T(A)^{\Rightarrow}, [A]T(B)^{\Rightarrow} \to T(B)^{\Rightarrow}$

Table B.1. A computational calculus with naturals, products, sums, and general recursion