Generic bidirectional typing for dependent type theories

Thiagо Felicissimo(✉)

Université Paris-Saclay, INRIA, LMF, ENS Paris-Saclay, Gif-sur-Yvette, France
thiago.felicissimo@inria.fr

Abstract. Bidirectional typing is a discipline in which the typing judgment is decomposed explicitly into inference and checking modes, allowing to control the flow of type information in typing rules and to specify algorithmically how they should be used. Bidirectional typing has been fruitfully studied and bidirectional systems have been developed for many type theories. However, the formal development of bidirectional typing has until now been kept confined to specific theories, with general guidelines remaining informal. In this work, we give a generic account of bidirectional typing for a general class of dependent type theories. This is done by first giving a general definition of type theories (or equivalently, a logical framework), for which we define declarative and bidirectional type systems. We then show, in a theory-independent fashion, that the two systems are equivalent. This equivalence is then explored to establish the decidability of typing for weak normalizing theories, yielding a generic type-checking algorithm that has been implemented in a prototype and used in practice with many theories.

Keywords: Type Theory · Bidirectional Typing · Logical Frameworks

1 Introduction

Algebraic [13,7] and logical framework [27,45,8,21] presentations of dependent type theories suffer from the verbosity of the required explicit type annotations, which destroys any hope of practical usability. In these settings, every type argument must be explicitly spelled out: an application is written as $t@_{A,x.B}u$, a dependent pair as $\langle t, u\rangle_{A,x.B}$, cons as $t ::_A l$ and the list goes on.

In order to restore usability, standard presentations of dependent type theories omit the majority of these annotations, so one writes $t\ u$ for application, $\langle t, u\rangle$ for a dependent pair, $t :: l$ for cons, etc. This unannotated syntax is so common that readers not familiar with algebraic presentations of type theory might not even realize that an omission is being made.

The omission of type arguments has nevertheless a cost: because knowing them is still important when typing terms, it becomes unclear how to do this algorithmically, even when decidability of conversion holds. Take for instance the typing rule for the dependent pair: in order to type $\langle t, u\rangle$ one needs to guess

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 143–170, 2024.
https://doi.org/10.1007/978-3-031-57262-3_6

the arguments A and B, as unlike for the fully-annotated version $\langle t, u\rangle_{A,x.B}$ these are not stored in the syntax.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u\rangle : \Sigma x : A.B}$$

A solution to this problem is provided by *bidirectional typing* [35,16,19,20,41]. In this typing discipline, the declarative typing judgment $\Gamma \vdash t : A$ is decomposed explicitly into inference $\Gamma \vdash t \Rightarrow A$, where Γ and t are inputs and A is an output, and checking $\Gamma \vdash t \Leftarrow A$, where Γ, t and A are all inputs. The important point is that, by using these new judgments to control the flow of type information in a typing rule, one can specify algorithmically how the rule should be used. For instance, the following rule clarifies how one should type $\langle t, u\rangle$: the types A and B are not to be guessed, but instead recovered from the type C, which should be given as input.

$$\frac{C \longrightarrow^* \Sigma x : A.B \qquad \Gamma \vdash t \Leftarrow A \qquad \Gamma \vdash u \Leftarrow B[t/x]}{\Gamma \vdash \langle t, u\rangle \Leftarrow C}$$

We therefore see that bidirectional typing is the natural companion for an unannotated syntax, as it allows to algorithmically explain how the missing information can be retrieved.

Bidirectional typing has been fruitfully studied and bidirectional systems have been developed for many type theories [16,33,25,1,38,2]. However, the formal development of bidirectional typing has until now remained confined to specific theories, with general guidelines remaining informal. One can then naturally wonder if it would be possible to define a framework in which bidirectional typing could be studied generically, putting its general theory in solid ground. This is exactly the goal of this paper.

We contribute a theory-independent account of bidirectional typing. For this, we start by giving a general definition of type theories (or equivalently, a logical framework), which differs from previous frameworks [45,29] by allowing for the usual unannotated syntaxes most often used in practice. Each such theory then defines a declarative type system, which is shown to satisfy good properties like weakening, substitution and, for well-behaved theories, subject reduction.

Then, to formulate our bidirectional system we first address the known problem that some unannotated terms cannot be algorithmically typed [18]. We propose generic notions of *inferable* and *checkable* terms which, for non-degenerate theories, coincide respectively with neutrals and normal forms, and then formulate our bidirectional system for this subset of terms. As argued in previous works [38,1], this restriction is reasonable because users of type theory usually only write terms in normal form. We then show, in a theory-independent fashion, that the bidirectional system is sound and complete with respect to the declarative system, establishing an equivalence between the two.

This equivalence is then explored to establish, for weak normalizing theories, the decidability of inference for inferable terms, and the decidability of checking for checkable terms. This proof gives rise to generic type inference and checking algorithms, which have been implemented in a prototype theory-independent checker, allowing them to be used in practice with multiple theories. This implementation is described in detail in an accompanying experience report [22].

Plan of the paper We start in Section 2 by giving our general definition of type theories. We then move to Section 3, in which we give the declarative type system and show it satisfies nice properties. Section 4 then gives the bidirectional type system and its proof of equivalence with the declarative one. In Section 5 we show various examples of theories which are instances of our framework. We finish by discussing related work in Section 6 and then we conclude with Section 7.

2 General type theories

In this section, we give a general definition of type theories (or equivalently, a logical framework) for which we will give declarative and bidirectional type systems in later sections. Our definition is inspired by recent proposals of general definitions of type theories [29,45] and by the logical framework literature [27,8,26,40]. Our definition however crucially departs from these works by allowing for unannotated syntaxes and by exploring the constructor/destructor symmetries of symbols and rules in type theories.

We start the section by defining the raw syntax of our theories. Then, after defining patterns and substitution, we give one of the central definitions of our work: the one of theory. We finish the section by describing the rewrite judgment used to specify the definitional equality of our type theories.

2.1 Raw Syntax

Scopes and signatures The basic ingredients of our raw expressions are variables, metavariables and symbols. These are specified by scopes and signatures, which we define by the following grammars.

$$\boxed{\mathsf{Scope}} \ni \qquad \gamma, \delta ::= \cdot \mid \gamma, x$$

$$\boxed{\mathsf{MScope}} \ni \qquad \theta, \xi ::= \cdot \mid \theta, \mathsf{x}\{\delta\}$$

$$\boxed{\mathsf{Sig}} \ni \qquad \Sigma ::= \cdot \mid \Sigma, c(\theta) \mid \Sigma, d(\theta)$$

Let us go through the definition. A *(variable) scope* γ is simply a list of variables, whereas a *metavariable scope* θ is a list of metavariables, each being accompanied by a variable scope δ explaining the arguments each metavariable expects. A *signature* Σ then assigns a metavariable scope θ to each symbol, also explaining the arguments it expects.

We see that, from the start, we split symbols in two classes: *constructors* c and *destructors* d. This separation will be justified by the fact that each of these two classes will play a different role in our theories: destructors are the source of computation and are bidirectionally typed in mode infer, whereas constructors are the results of computations and are bidirectionally typed in mode check.

In the following, we write $\gamma.\delta$ and $\theta.\xi$ for scope concatenation, and we write constructor names in blue and destructor names in orange.

Example 1. The following signature $\Sigma_{\lambda\Pi}$ defines the raw syntax of a minimalistic Martin-Löf Type Theory (MLTT) with only dependent functions.

$$\mathrm{Ty},\ \mathrm{Tm}(\mathsf{A}),\ \Pi(\mathsf{A},\mathsf{B}\{x\}),\ \lambda(\mathsf{t}\{x\}),\ @(\mathsf{u}) \tag{$\Sigma_{\lambda\Pi}$}$$

The entry for $@$ might seem a bit strange since application usually takes two arguments, t and u. As we will see, destructors automatically take an extra argument, so we do not need to specify one for t. The reason for including symbols Ty and Tm will also become clear later.

Terms and spines Given a fixed signature Σ, we then define the terms, (variable) substitutions and metavariable substitutions by the following grammars.

$$\begin{array}{rrll}
\boxed{\mathsf{Tm}\ \theta\ \gamma} \ni & t,u,T,U ::= & \mid x & \text{if } x \in \gamma \\
 & & \mid \mathsf{x}\{\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta\} & \text{if } \mathsf{x}\{\delta\} \in \theta \\
 & & \mid c(\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
 & & \mid d(t \in \mathsf{Tm}\ \theta\ \gamma; \mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi) & \text{if } d(\xi) \in \Sigma \\
\boxed{\mathsf{Sub}\ \theta\ \gamma\ \delta} \ni & \vec{t},\vec{u},\vec{s},\vec{v} ::= & \mid \varepsilon & \text{if } \delta = \cdot \\
 & & \mid \vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta', t \in \mathsf{Tm}\ \theta\ \gamma & \text{if } \delta = \delta', x \\
\boxed{\mathsf{MSub}\ \theta\ \gamma\ \xi} \ni & \mathbf{t},\mathbf{u},\mathbf{s},\mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \\
 & & \mid \mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}\ \theta\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}$$

We go through the definition step by step. First, we elect an intrinsically-scoped presentation of syntax, so the definitions of terms, substitutions and metavariable substitutions are each indexed by a scope γ and a metavariable scope θ, describing the variables and metavariables that can appear free.

A *term* is then either a variable x, a metavariable x applied to a substitution $\vec{t}$, a constructor c applied to a metavariable substitution $\mathbf{t}$ or a destructor d applied to a term t and a metavariable substitution $\mathbf{t}$. At the level of the syntax, the main difference between constructors and destructors is that the latter are automatically applied to a first argument, called the *principal argument*. Note also that we require each variable and metavariable to be in scope, and each metavariable or symbol to be applied to a substitution matching its scope of arguments, as specified by θ or Σ.

A *(variable) substitution* $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ is then either the empty substitution when δ is empty, or a substitution $\vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta'$ and a term $t \in \mathsf{Tm}\ \theta\ \gamma$ when $\delta = \delta', x$. Therefore, we see that the scope δ describes the output (or the domain of definition) of the substitution $\vec{t}$. Similarly, a *metavariable substitution* $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ is either empty when ξ is empty, or a metavariable substitution

$\mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi'$ and a term $t \in \mathsf{Tm}\ \theta\ \gamma.\delta$ when $\xi = \xi', \mathsf{x}\{\delta\}$. Unlike variable substitutions, metavariable substitutions are allowed to extend the scope of their arguments by binding the variables in δ, which we refer to by $\vec{x}_\delta$. This is used for instance in the cases of λ and Π in the following example.

Example 2. The terms defined by the signature $\Sigma_{\lambda\Pi}$ are given by the following grammar, where we omit the scope requirements for variables and metavariables.

$$t, u, A, B ::= x \mid \mathsf{x}\{\vec{t}\} \mid \mathrm{Ty} \mid \mathrm{Tm}(A) \mid \lambda(x.t) \mid \Pi(A, x.B) \mid @(t; u)$$

In the following, given a metavariable substitution $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ and $\mathsf{x}\{\delta\} \in \xi$, we write $\mathbf{t}_\mathsf{x} \in \mathsf{Tm}\ \theta\ \gamma.\delta$ for the term in $\mathbf{t}$ at the position pointed by x. Similarly, given a substitution $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ and $x \in \delta$, we write $t_x \in \mathsf{Tm}\ \theta\ \gamma$ for the term in $\vec{t}$ at the position pointed by x.

Contexts Given a fixed signature Σ, we define (variable) contexts and metavariable contexts by the following grammars.

$$\boxed{\mathsf{Ctx}\ \theta\ \gamma} \ni \quad \Gamma, \Delta ::= \cdot \mid \Gamma \in \mathsf{Ctx}\ \theta\ \gamma, x : T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma|$$

$$\boxed{\mathsf{MCtx}\ \theta} \ni \quad \Theta, \Xi ::= \cdot \mid \Theta \in \mathsf{MCtx}\ \theta, \mathsf{x}\{\Gamma \in \mathsf{Ctx}\ \theta.|\Theta|\ (\cdot)\} : T \in \mathsf{Tm}\ \theta.|\Theta|\ |\Gamma|$$

These are specified mutually with the *underlying scopes* $|\Gamma|$ of Γ and $|\Theta|$ of Θ, defined by the following clauses.

$$\begin{array}{ll} |\_| : \mathsf{Ctx}\ \theta\ \gamma \to \mathsf{Scope} & \qquad |\_| : \mathsf{MCtx}\ \theta \to \mathsf{MScope} \\ |\cdot| := \cdot & \qquad |\cdot| := \cdot \\ |\Gamma, x : T| := |\Gamma|, x & \qquad |\Theta, \mathsf{x}\{\Delta\} : T| := |\Theta|, \mathsf{x}\{|\Delta|\} \end{array}$$

A *context* $\Gamma \in \mathsf{Ctx}\ \gamma\ \theta$ is either empty, or composed by a context $\Gamma' \in \mathsf{Ctx}\ \gamma\ \theta$ and a variable x with a term $T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma'|$. The first important thing to note is that the term T does not live in scope γ, but in the extension of γ with the underlying scope of Γ', meaning that each entry in the context has access to the previously declared variables. Second, like in other frameworks [45], terms can also play the role of judgments, as illustrated by the following example.

Example 3. In MLTT we have two judgment forms: $\square\ \mathsf{type}$ for classifying types, and $\square : A$ for classifying terms. In our framework, these are represented by the constructors Ty and Tm. For instance, the context $A\ \mathsf{type}, x : A, y : (\Pi z : A.A)$ of MLTT is represented in our framework as

$$A : \mathrm{Ty}, x : \mathrm{Tm}(A), y : \mathrm{Tm}(\Pi(A, z.A))$$

which is syntactically well-formed in the signature $\Sigma_{\lambda\Pi}$. We can also write some strange contexts like $x : \lambda(z.z), y : x$, which will be eliminated later by typing.

The case of a *metavariable context* $\Theta \in \mathsf{MCtx}\ \theta$ is similar. We have either Θ empty or $\Theta = \Theta', \mathsf{x}\{\Delta\} : T$, where Δ has access to metavariables in θ and Θ, and T has moreover access to the variables in Δ.

Notation 1. We finish this subsection by establishing some notations.

– We write $e \in \mathsf{Expr}\ \theta\ \gamma$ as an informal abbreviation for any of the following: $e \in \mathsf{Tm}\ \theta\ \gamma$ or $e \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ or $e \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ or $e \in \mathsf{Ctx}\ \theta\ \gamma$.
– If the underlying signature is not clear from the context, we write Tm_Σ, Sub_Σ, MSub_Σ, Ctx_Σ, MCtx_Σ in order to make it explicit.
– We write $\mathsf{Ctx}\ \theta$ for $\mathsf{Ctx}\ \theta\ (\cdot)$, Ctx for $\mathsf{Ctx}\ (\cdot)\ (\cdot)$ and MCtx for $\mathsf{MCtx}\ (\cdot)$.

Remark 1. Because we work with a nameful syntax, we allow ourselves to implicitly weaken expressions: if $e \in \mathsf{Expr}\ \theta\ \gamma$ and θ is a subsequence of θ' and γ is a subsequence of γ' then we also have $e \in \mathsf{Expr}\ \theta'\ \gamma'$. Nevertheless, we expect that our proofs can be formally carried out using de Bruijn indices, by properly inserting weakenings whenever needed, and showing all the associated lemmata.

2.2 Substitution

Before defining the application of a substitution to a term, we first need to define the *identity substitutions*, by the following clauses. Note that, while the identity variable substitution id_γ is just the list of variables from γ, the identity metavariable substitution id_θ needs to eta-expand each metavariable $\mathsf{x}\{\delta\} \in \theta$ to $\vec{x}_\delta.\mathsf{x}\{\mathsf{id}_\delta\}$ in order for the result to be a valid metavariable substitution. In the following, we sometimes abuse notation and write id_Γ for $\mathsf{id}_{|\Gamma|}$ and id_Θ for $\mathsf{id}_{|\Theta|}$.

$$\begin{aligned} &\mathsf{id}_{\_} : (\gamma \in \mathsf{Scope}) \to \mathsf{Sub}\ (\cdot)\ \gamma\ \gamma \\ &\mathsf{id}_{(\cdot)} := \varepsilon \\ &\mathsf{id}_{\gamma,x} := \mathsf{id}_\gamma, x \end{aligned} \qquad \begin{aligned} &\mathsf{id}_{\_} : (\theta \in \mathsf{MScope}) \to \mathsf{MSub}\ \theta\ (\cdot)\ \theta \\ &\mathsf{id}_{(\cdot)} := \varepsilon \\ &\mathsf{id}_{\theta,\mathsf{x}\{\gamma\}} := \mathsf{id}_\theta, \vec{x}_\gamma.\mathsf{x}\{\mathsf{id}_\gamma\} \end{aligned}$$

We can now define in Figure 1 the application of a substitution to an expression. Given a variable substitution $\vec{v} \in \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2$ its application to an expression $e \in \mathsf{Expr}\ \theta\ \gamma_2$ gives $e[\vec{v}] \in \mathsf{Expr}\ \theta\ \gamma_1$, and given a metavariable substitution $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ its application to an expression $e \in \mathsf{Expr}\ \theta_2\ \gamma$ gives $e[\mathbf{v}] \in \mathsf{Expr}\ \theta_1\ \delta.\gamma$. The main case of the definition is when we substitute $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ in the term $\mathsf{x}\{\vec{t}\} \in \mathsf{Tm}\ \theta_2\ \gamma$. If $\mathsf{x}\{\gamma_\mathsf{x}\} \in \theta_2$, then by recursively substituting $\mathbf{v}$ in $\vec{t} \in \mathsf{Sub}\ \theta_2\ \gamma\ \gamma_\mathsf{x}$ we get $\vec{t}[\mathbf{v}] \in \mathsf{Sub}\ \theta_1\ \delta.\gamma\ \gamma_\mathsf{x}$. We moreover have $\mathbf{v}_\mathsf{x} \in \mathsf{Tm}\ \theta_1\ \delta.\gamma_\mathsf{x}$, so by substituting the variables in γ_x by $\vec{t}[\mathbf{v}]$ and the ones in δ by id_δ we get $\mathbf{v}_\mathsf{x}[\mathsf{id}_\delta, \vec{t}[\mathbf{v}]] \in \mathsf{Tm}\ \theta_1\ \delta.\gamma$ as the final result.

Example 4. If $t \in \mathsf{Tm}\ (\cdot)\ (\gamma, x)$ and $u \in \mathsf{Tm}\ (\cdot)\ \gamma$, then by applying $x.t, u \in \mathsf{MSub}\ (\cdot)\ \gamma\ (\mathsf{t}\{x\}, \mathsf{u})$ to $@(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \in \mathsf{Tm}\ (\mathsf{t}\{x\}, \mathsf{u})\ (\cdot)$ we get the term $@(\lambda(x.t); u) \in \mathsf{Tm}\ (\cdot)\ \gamma$. Note in particular that, compared to frameworks derived from contextual modal type theory [37], our metavariable substitutions are not required to be closed and can introduce new variables in the scope of the resulting term. For instance, while the inital term $@(\lambda(x.\mathsf{t}\{x\}); \mathsf{u})$ lives in an empty variable scope, the resulting term $@(\lambda(x.t); u)$ lives in the variable scope γ.

Substitution application satisfies all the expected laws, such as $e[\mathbf{v}][\mathbf{u}] = e[\mathbf{v}[\mathbf{u}]]$, $e[\mathsf{id}_\gamma] = e$ and $\mathsf{id}_\theta[\mathbf{v}] = \mathbf{v}$ — we refer to the technical report [24] for a detailed account of these properties, and for the full definition of substitution.

$$
\begin{array}{ll}
\_[\_] : \mathsf{Tm}\ \theta\ \gamma_2 \to \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 & \_[\_] : \mathsf{Tm}\ \theta_2\ \gamma \to \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \\
\quad \to \mathsf{Tm}\ \theta\ \gamma_1 & \quad \to \mathsf{Tm}\ \theta_1\ \delta.\gamma \\
x[\vec{v}] := v_x & x[\mathbf{v}] := x \\
\mathsf{x}\{\vec{t}\}[\vec{v}] := \mathsf{x}\{\vec{t}[\vec{v}]\} & \mathsf{x}\{\vec{t}\}[\mathbf{v}] := \mathbf{v}_{\mathsf{x}}[\mathsf{id}_\delta, \vec{t}[\mathbf{v}]] \\
c(\mathbf{t})[\vec{v}] := c(\mathbf{t}[\vec{v}]) & c(\mathbf{t})[\mathbf{v}] := c(\mathbf{t}[\mathbf{v}]) \\
d(t; \mathbf{u})[\vec{v}] := d(t[\vec{v}]; \mathbf{u}[\vec{v}]) & d(t; \mathbf{u})[\mathbf{v}] := d(t[\mathbf{v}]; \mathbf{u}[\mathbf{v}])
\end{array}
$$

Fig. 1. Application of a variable or metavariable substitution (excerpt)

2.3 Patterns

There will be a need to isolate a special class of expressions that will be shown later to support decidable and unitary matching. This will be needed both to define the rewrite rules of our theories and to determine when omitted arguments can be recovered. For this, given a fixed signature Σ, we define the *term patterns* and *metavariable substitution patterns* by the following grammars.

$$
\begin{array}{lll}
\boxed{\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma} \ni t, u ::= & \mid \mathsf{x}\{\mathsf{id}_\gamma\} & \text{if } \theta = \mathsf{x}\{\gamma\} \\
 & \mid c(\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
\boxed{\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi} \ni \mathbf{t}, \mathbf{u} ::= & \mid \varepsilon & \text{if } \xi = \cdot \text{ and } \theta = \cdot \\
 & \mid \mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta_1\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}^{\mathsf{P}}\ \theta_2\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\} \\
 & & \quad \text{and } \theta = \theta_1.\theta_2
\end{array}
$$

As we can see, the only symbols that are allowed to appear in patterns are constructors. This will be essential later to ensure that patterns do not only support syntactic matching, but also matching modulo rewriting. Moreover, our patterns are linear, and so each metavariable in scope occurs exactly once. Finally, our patterns are *fully-applied*, meaning that each metavariable occurrence is fully applied to all variables in scope.

Example 5. In the signature $\Sigma_{\lambda\Pi}$ we can build the pattern

$$\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \in \mathsf{Tm}^{\mathsf{P}}\ (\mathsf{A},\ \mathsf{B}\{x\})\ (\cdot)$$

We have the inclusions $\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma \subset \mathsf{Tm}\ \theta\ \gamma$ and $\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi \subset \mathsf{MSub}\ \theta\ \gamma\ \xi$, which we use to implicitly coerce patterns into regular expressions when needed.

2.4 Theories

We now come to a central definition in our work, that of a *theory* $\mathbb{T}$. We define inductively how a theory is built, simultaneously with its underlying signature $|\mathbb{T}|$ — technically, our definition is by small induction-recursion. The base case covers the empty theory $\mathbb{T} = (\cdot)$, so assuming now a theory $\mathbb{T}$ is given we can extend it with two types of entries: *schematic typing rules* and *rewrite rules*. We start with the first, which come in three kinds: sort, constructor and destructor rules.

Sort rules In our framework, a *sort* T is a term that can appear to the right of a colon.[1] As hinted in Example 3, and following [13,45,44], sorts are used to specify the judgment forms of a theory. For instance, the two judgment forms of MLTT "$\square$ type" and "$\square : A$" are defined by the following.

$$\frac{}{\vdash \mathrm{Ty}\ \mathsf{sort}} \qquad \frac{\vdash \mathsf{A} : \mathrm{Ty}}{\vdash \mathrm{Tm}(\mathsf{A})\ \mathsf{sort}}$$

Formally, a sort rule is of the form

$$c(\Xi \in \mathsf{MCtx}_{|\mathbb{T}|})\ \mathsf{sort}$$

and the previously shown rules are just an informal notation for $\mathrm{Ty}(\cdot)$ sort and $\mathrm{Tm}(\mathsf{A} : \mathrm{Ty})$ sort. In the following, we will make use of such informal representations in order to enhance readability of schematic rules. Note also that the premises (e.g. $\vdash \mathsf{A} : \mathrm{Ty}$) correspond to entries in the metavariable context of the rule, and henceforth we will use these two points of view interchangeably.

Constructor rules Like most works in bidirectional typing [35,19,1,16], our framework imposes that constructors are to be bidirectionally typed in mode check, and thus the sort of the conclusion can be used to recover arguments which are not recorded in the syntax. To capture this, premises are split into two metavariable contexts Ξ_1 and Ξ_2, where Ξ_1 is erased and Ξ_2 is stored in the term. The sort T is then required to be a pattern containing the metavariables of Ξ_1, leading to constructor rules of the following form.

$$c(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|};\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ |\Xi_1|) : U \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot)$$

Two examples of constructor rules are the ones for Π and λ — note however that the one for Π is slightly degenerate, given that we have $\Xi_1 = \cdot$ and thus no erased premises.

$$\frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\vdash \Pi(\mathsf{A}, \mathsf{B}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathtt{t} : \mathrm{Tm}(\mathsf{B}\{x\})\end{array}}{\vdash \lambda(\mathtt{t}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

These are just informal notations for $\Pi(\cdot;\ \mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}$ and $\lambda(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathtt{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$.

Destructor rules As for constructor rules, in destructor rules premises are also separated into erased Ξ_1 and non-erased Ξ_2 parts. However, unlike constructors, destructors are bidirectionally typed in inference mode. In this case, the erased arguments in Ξ_1 are not to be recovered from the sort of the conclusion, but instead by inferring the sort of the *principal argument* which is required to be

[1] We avoid calling them "types" to prevent a name clash with the types of the theories we define. Still, we allow ourselves to say "t is typed by sort T" to mean $t : T$.

a pattern containing the metavariables of Ξ_1. The destructor rules are therefore of the following form.

$$d(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|};\ \mathsf{x} : T \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot);\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x})) \\ : U \in \mathsf{Tm}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x}, |\Xi_2|)\ (\cdot)$$

The main example of destructor rule is the application rule

$$\frac{\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \vdash \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \vdash \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{\vdash @(\mathsf{t}; \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

which is just an informal notation for

$$@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

Rewrite rules Finally, the last type of rules are rewrite rules, which allow us to specify the definitional equality (also known as conversion) of the theory. As suggested by our constructor/destructor separation of symbols, the left-hand side of rewrite rules are to be headed by destructors. Moreover, to ensure the decidability of rewriting, we also ask both arguments t and $\mathbf{u}$ of the left hand side $d(t; \mathbf{u})$ to be patterns. The right-hand side of the rule is then a term containing only the metavariables introduced by t and $\mathbf{u}$. The rewrite rules are hence of the following form, where $d(\xi) \in |\mathbb{T}|$.

$$\theta_1; \theta_2 \Vdash d(t \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta_1\ (\cdot); \mathbf{u} \in \mathsf{MSub}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta_2\ (\cdot)\ \xi) \longmapsto r \in \mathsf{Tm}_{|\mathbb{T}|}\ \theta_1.\theta_2\ (\cdot)$$

We shall also ask for a supplementary condition: in order to extend a theory $\mathbb{T}$ with a rule $\theta_1; \theta_2 \Vdash l \longmapsto r$, there can be no rule $\theta_1'; \theta_2' \Vdash l' \longmapsto r'$ in $\mathbb{T}$ such that we have $l[\mathbf{v}] = l'[\mathbf{v}']$ for some $\mathbf{v}$ and $\mathbf{v}'$. As discussed in the next subsection, this will ensure that the rewrite system is confluent by construction.

The prototypical example of a rewrite rule is the computation rule for functions: the β-rule from the λ-calculus.

$$\mathsf{t}\{x\}; \mathsf{u} \Vdash @(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$$

In the following we allow ourselves to omit the metavariable scopes $\theta_1; \theta_2$ as these can be easily reconstructed by inspecting the rewrite rule's left hand side.

Underlying signature Finally, the recursive definition of the underlying signature of a theory is given by the following clauses, where we write Thy for the set of theories. As we can see, in both constructor and destructor rules the metavariable context of erased premises Ξ_1 is omitted from the syntax. Moreover, rewrite rules are simply ignored when computing the underlying signature.

$$\begin{array}{ll}
|_| : \mathsf{Thy} \to \mathsf{Sig} & |\mathbb{T}, c(\Xi_1; \Xi_2) : U| := |\mathbb{T}|, c(|\Xi_2|) \\
|\cdot| := \cdot & |\mathbb{T}, d(\Xi_1; \mathsf{x} : T; \Xi_2) : U| := |\mathbb{T}|, d(|\Xi_2|) \\
|\mathbb{T}, c(\Xi)\ \mathsf{sort}| := |\mathbb{T}|, c(|\Xi|) & |\mathbb{T}, \theta_1; \theta_2 \Vdash d(t; \mathbf{u}) \longmapsto r| := |\mathbb{T}|
\end{array}$$

Example 6. By putting together all of the rules previously seen in this subsection, we get the following theory $\mathbb{T}_{\lambda\Pi}$ defining a basic version of MLTT with only dependent functions.

$$\begin{aligned}
&\mathrm{Ty}(\cdot)\ \mathsf{sort},\quad \mathrm{Tm}(\mathsf{A} : \mathrm{Ty})\ \mathsf{sort},\quad \Pi(\cdot;\ \ \mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty}, \qquad (\mathbb{T}_{\lambda\Pi})\\
&\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\quad \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\\
&@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\quad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\quad \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}),\\
&@(\lambda(x.\mathsf{t}\{x\}); \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}
\end{aligned}$$

When computing its underlying signature $|\mathbb{T}_{\lambda\Pi}|$ we get the signature $\Sigma_{\lambda\Pi}$.

2.5 Rewriting

The rewrite rules of a theory $\mathbb{T}$ are used to define the *rewrite relation* $e \longrightarrow e'$ for expressions $e, e' \in \mathsf{Expr}\ \theta\ \gamma$, which is given by the context closure of the following rule — see the technical report [24] for the full definition.

$$\frac{(\theta_1; \theta_2 \Vdash d(t; \mathbf{u}) \longmapsto r) \in \mathbb{T} \qquad \mathbf{v}_1 \in \mathsf{MSub}\ \theta\ \gamma\ \theta_1 \qquad \mathbf{v}_2 \in \mathsf{MSub}\ \theta\ \gamma\ \theta_2}{d(t[\mathbf{v}_1]; \mathbf{u}[\mathbf{v}_2]) \longrightarrow r[\mathbf{v}_1, \mathbf{v}_2] \in \mathsf{Tm}\ \theta\ \gamma}$$

The relations $\longrightarrow^{+}$, $\longrightarrow^{*}$ and $\equiv$ are then defined as usual, respectively as the transitive, reflexive-transitive and reflexive-symmetric-transitive closures of $\longrightarrow$. The relation $\equiv$ is called the *definitional equality* (or *conversion*) of the theory.

Rewriting satisfies the key property of being stable under substitution: if $e \longrightarrow^{*} e'$ and $\vec{v} \longrightarrow^{*} \vec{v}'$ then $e[\vec{v}] \longrightarrow^{*} e'[\vec{v}']$, and if $e \longrightarrow^{*} e'$ and $\mathbf{v} \longrightarrow^{*} \mathbf{v}'$ then $e[\mathbf{v}] \longrightarrow^{*} e'[\mathbf{v}']$. This implies in particular that definitional equality is also stable under substitution.

Finally, recall that when defining theories we asked that no two different left-hand sides should unify. Because this is the only way two rule can overlap, this means that there are no *critical pairs* [11]. Therefore, because rules are also all *left-linear*, it follows that the rewrite system of any theory is *orthogonal*, hence confluent by [34, Theorem 6.11].

Proposition 1 (Confluence). *If $e' \ {}^{*}\!\!\longleftarrow e \longrightarrow^{*} e''$ then there is some e''' such that $e' \longrightarrow^{*} e''' \ {}^{*}\!\!\longleftarrow e''$. In particular, this implies that whenever $e \equiv e'$ then we have $e \longrightarrow^{*} e'' \ {}^{*}\!\!\longleftarrow e'$ for some e''.*

Notation 2. Whenever the underlying theory is not clear from the context we write $\longrightarrow_{\mathbb{T}}$ and $\longrightarrow^{+}_{\mathbb{T}}$ and $\longrightarrow^{*}_{\mathbb{T}}$ and $\equiv_{\mathbb{T}}$ to make it explicit.

3 Declarative type system

In the previous section we have given a general definition of type theories. As explained in the introduction, each theory also defines a declarative type system, which can be seen as the platonic type system, and a bidirectional type system, which is the one that can be algorithmically used in practice.

In this section we introduce the declarative type system. This system is then used to define the *valid theories*, a class of theories which are well-behaved. We then conclude the section by showing that the declarative system satisfies nice properties, and in particular satisfies subject reduction when the theory is valid.

3.1 Declarative typing rules

Given a fixed theory $\mathbb{T}$, the declarative type system is defined by the rules in Figure 2. The system is split in 6 judgments:

- $\Theta \vdash$: Well-formedness of metavariable context Θ.
- $\Theta; \Gamma \vdash$: Well-formedness of variable context Γ under metavariable context Θ.
- $\Theta; \Gamma \vdash T\ \mathsf{sort}$: Well-formedness of sort T under contexts $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash t : T$: Typing of a term t by T under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \vec{t} : \Delta$: Typing of a variable substitution $\vec{t}$ by Δ under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \mathbf{t} : \Xi$: Typing of a metavariable substitution $\mathbf{t}$ by Ξ under context $\Theta; \Gamma$.

The most important rules are the ones which instantiate schematic typing rules: Cons, Dest and Sort. For instance, in order to use Dest to type $d(t; \mathbf{u})$ a metavariable substitution $\mathbf{t}$ not stored in the syntax must be "guessed", and then we must show that $\mathbf{t}, t, \mathbf{u}$ is typed by $\Xi_1.(\mathsf{x} : T).\Xi_2$. The rules for typing a metavariable substitution can then be applied, which has the effect of unfolding the judgment $\mathbf{t}, t, \mathbf{u} : \Xi_1.(\mathsf{x} : T).\Xi_2$ into regular term typing judgments. At the end of this unfolding process, the resulting "big-step derivation" has basically the same shape as the schematic typing rule for d, and it can be understood as its instantiation. Let us look at a concrete example of this.

Example 7. Suppose we want to show that $@(t; u)$ is well-typed in the theory $\mathbb{T}_{\lambda\Pi}$. Because $@$ is a destructor symbol with schematic rule

$$@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty};\ \ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}));\ \ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

by guessing some A and B we can start the derivation with rule Dest, giving

$$\frac{\Theta; \Gamma \vdash A, x.B, t, u : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\ \mathsf{u} : \mathrm{Tm}(\mathsf{A}))}{\Theta; \Gamma \vdash @(t; u) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u]}$$

If we note that $\mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u] = \mathrm{Tm}(B[\mathsf{id}_\Gamma, u])$, and we continue by applying the rules defining the judgment $\Theta; \Gamma \vdash \mathbf{t} : \Xi$, we get

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash \qquad \Theta; \Gamma \vdash A : \mathrm{Ty} \qquad \Theta; \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty} \\ \Theta; \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Theta; \Gamma \vdash u : \mathrm{Tm}(A)\end{array}}{\Theta; \Gamma \vdash @(t; u) : \mathrm{Tm}(B[\mathsf{id}_\Gamma, u])}$$

which can be understood as the instantiation of the schematic rule for $@$. This also corresponds to the usual application rule of MLTT, as the first 3 premises can be shown admissible by inversion of typing and the results of Subsection 3.3.

$$\boxed{\Theta \vdash \quad (\Theta \in \mathsf{MCtx})}$$

$$\boxed{\Theta;\Gamma \vdash \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|)}$$

$$\textsc{EmptyMCtx}\ \frac{}{\cdot \vdash} \qquad \textsc{ExtMCtx}\ \frac{\Theta;\Gamma \vdash T\ \mathsf{sort}}{\Theta, \mathsf{x}\{\Gamma\} : T \vdash} \qquad \textsc{EmptyCtx}\ \frac{\Theta \vdash}{\Theta;\cdot \vdash} \qquad \textsc{ExtCtx}\ \frac{\Theta;\Gamma \vdash T\ \mathsf{sort}}{\Theta;\Gamma, x : T \vdash}$$

$$\boxed{\Theta;\Gamma \vdash T\ \mathsf{sort} \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{Sort}\ c(\Xi)\ \mathsf{sort} \in \mathbb{T}\ \frac{\Theta;\Gamma \vdash \mathbf{t} : \Xi}{\Theta;\Gamma \vdash c(\mathbf{t})\ \mathsf{sort}}$$

$$\boxed{\Theta;\Gamma \vdash t : T \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|;\ t \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{Cons}\ c(\Xi_1;\Xi_2) : T \in \mathbb{T}\ \frac{\Theta;\Gamma \vdash \mathbf{t}, \mathbf{u} : \Xi_1.\Xi_2}{\Theta;\Gamma \vdash c(\mathbf{u}) : T[\mathbf{t}]} \qquad \textsc{Var}\ x : T \in \Gamma\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash x : T}$$

$$\textsc{Dest}\ d(\Xi_1; \mathsf{x} : T; \Xi_2) : U \in \mathbb{T}\ \frac{\Theta;\Gamma \vdash \mathbf{t}, t, \mathbf{u} : \Xi_1.(\mathsf{x} : T).\Xi_2}{\Theta;\Gamma \vdash d(t;\mathbf{u}) : U[\mathbf{t}, t, \mathbf{u}]} \qquad \textsc{MVar}\ \mathsf{x}\{\Delta\} : T \in \Theta\ \frac{\Theta;\Gamma \vdash \vec{t} : \Delta}{\Theta;\Gamma \vdash \mathsf{x}\{\vec{t}\} : T[\vec{t}]}$$

$$\textsc{Conv}\ T \equiv_{\mathbb{T}} U\ \frac{\Theta;\Gamma \vdash t : T \qquad \Theta;\Gamma \vdash U\ \mathsf{sort}}{\Theta;\Gamma \vdash t : U}$$

$$\boxed{\Theta;\Gamma \vdash \vec{t} : \Delta \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Delta \in \mathsf{Ctx}\ |\Theta|;\ \vec{t} \in \mathsf{Sub}\ |\Theta|\ |\Gamma|\ |\Delta|)}$$

$$\textsc{EmptySub}\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtSub}\ \frac{\Theta;\Gamma \vdash \vec{t} : \Delta \qquad \Theta;\Gamma \vdash t : T[\vec{t}]}{\Theta;\Gamma \vdash \vec{t}, t : (\Delta, x : T)}$$

$$\boxed{\Theta;\Gamma \vdash \mathbf{t} : \Xi \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Xi \in \mathsf{MCtx};\ \mathbf{t} \in \mathsf{MSub}\ |\Theta|\ |\Gamma|\ |\Xi|)}$$

$$\textsc{EmptyMSub}\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtMSub}\ \frac{\Theta;\Gamma \vdash \mathbf{t} : \Xi \qquad \Theta;\Gamma.\Delta[\mathbf{t}] \vdash t : T[\mathbf{t}]}{\Theta;\Gamma \vdash \mathbf{t}, \vec{x}_\Delta.t : (\Xi, \mathsf{x}\{\Delta\} : T)}$$

Fig. 2. Declarative typing rules

Notation 3. We finish this subsection by establishing some notations.

1. We write $\Theta;\Gamma \vdash \mathcal{J}$ for any of the following: $\Theta;\Gamma \vdash$ or $\Theta;\Gamma \vdash T \ \mathsf{sort}$ or $\Theta;\Gamma \vdash t : T$ or $\Theta;\Gamma \vdash \vec{t} : \Delta$ or $\Theta;\Gamma \vdash \mathbf{t} : \Xi$.
2. We write $\mathbb{T} \triangleright \Theta;\Gamma \vdash \mathcal{J}$ when $\mathbb{T}$ is not clear from the context.
3. We write $\Theta \vdash \mathcal{J}$ for $\Theta; \cdot \vdash \mathcal{J}$ and $\Gamma \vdash \mathcal{J}$ for $\cdot;\Gamma \vdash \mathcal{J}$.

3.2 Valid theories

Our definition of theories given in Subsection 2.4 is a bit too permissive, and we would like to isolate a class of theories for which we can show nicer properties. These are the *valid theories*, defined by the following inference rules.

$$\frac{}{\cdot \ \mathsf{valid}} \qquad \frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi_1 \vdash T\ \mathsf{sort} \qquad \mathbb{T} \triangleright \Xi_1.\Xi_2 \vdash}{\mathbb{T}, c(\Xi_1;\Xi_2) : T\ \mathsf{valid}}$$

$$\frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi \vdash}{\mathbb{T}, c(\Xi)\ \mathsf{sort}\ \mathsf{valid}} \qquad \frac{\mathbb{T}\ \mathsf{valid} \qquad \mathbb{T} \triangleright \Xi_1.(\mathsf{x} : U).\Xi_2 \vdash T\ \mathsf{sort}}{\mathbb{T}, d(\Xi_1; \mathsf{x} : U; \Xi_2) : T\ \mathsf{valid}}$$

$$\frac{\begin{array}{c}\mathbb{T}\ \mathsf{valid} \qquad d(\Xi_1;\mathsf{x} : U;\Xi_2) : V \in \mathbb{T} \qquad \text{for some } \Theta_1, \Theta_2 \text{ with } |\Theta_1| = \theta_1, |\Theta_2| = \theta_2 : \\ \mathbb{T} \triangleright \Xi_1.\Theta_1.\Theta_2 \vdash (\mathsf{id}_{\Xi_1}, t, \mathbf{u}) : \Xi_1.(\mathsf{x} : U).\Xi_2 \qquad \mathbb{T} \triangleright \Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1}, t, \mathbf{u}]\end{array}}{\mathbb{T}, \theta_1;\theta_2 \Vdash d(t;\mathbf{u}) \longmapsto r\ \mathsf{valid}}$$

Intuitively, the definition of valid theories ensures that each time we extend a theory $\mathbb{T}$ with a schematic typing or rewrite rule, $\mathbb{T}$ can justify that the extension is well-behaved. For sort rules $c(\Xi)\ \mathsf{sort}$ this means ensuring that the metavariable context Ξ is well-formed in $\mathbb{T}$, and for destructor rules $d(\Xi_1; \mathsf{x} : U; \Xi_2) : T$ this means ensuring that T is a well-formed sort in metavariable context $\Xi_1.(\mathsf{x} : U).\Xi_2$ and in the theory $\mathbb{T}$. The rule for a constructor $c(\Xi_1;\Xi_2) : T$ does not only ask $\Xi_1.\Xi_2$ to be a well-formed metavariable context, but also for the term T to be a well-formed sort for Ξ_1 — recall that T can only contain metavariables from Ξ_1.

The most complicated case is for a rewrite rule $\theta_1;\theta_2 \Vdash d(t;\mathbf{u}) \longmapsto r$, in which we must find metavariable contexts Θ_1, Θ_2 with $|\Theta_1| = \theta_1$ and $|\Theta_2| = \theta_2$ allowing to type $\mathsf{id}_{\Xi_1}, t, \mathbf{u}$ and r. This technical condition is essential to prove subject reduction of our valid theories (Theorem 1).

Example 8. It is tedious but uncomplicated to see that the theory $\mathbb{T}_{\lambda\Pi}$ is valid. The most interesting part of the proof is when we add the β-rule

$$\mathsf{t}\{x\};\mathsf{u} \Vdash @(\lambda(x.\mathsf{t}\{x\});\mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$$

If we write $\mathbb{T}'$ for the part of $\mathbb{T}_{\lambda\Pi}$ preceding this declaration, and (for space reasons) we abbreviate $(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(A)\} : \mathrm{Ty})$ as Θ_{AB}, we have to show

$$\mathbb{T}' \triangleright \Theta_{\mathsf{AB}}.\Theta_1.\Theta_2 \vdash (\mathsf{A}, x.\mathsf{B}\{x\}, \lambda(x.\mathsf{t}\{x\}), \mathsf{u}) : \Theta_{\mathsf{AB}}.(\mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})), \mathsf{u} : \mathrm{Tm}(\mathsf{A}))$$

and

$$\mathbb{T}' \triangleright \Theta_{\mathsf{AB}}.\Theta_1.\Theta_2 \vdash \mathsf{t}\{\mathsf{u}\} : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

which can both be easily shown if we chose $\Theta_1 = \mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})$ and $\Theta_2 = \mathsf{u} : \mathrm{Tm}(\mathsf{A})$, which indeed satisfy $|\Theta_1| = \mathsf{t}\{x\}$ and $|\Theta_2| = \mathsf{u}$.

3.3 Basic metaproperties

We now show some basic metaproperties satisfied by the declarative type system. The assumption that the theory is valid is not necessary for all results, and will be stated explicitly when needed. We give proof sketches for some of the properties, and refer to the technical report [24] for the proofs.

Proposition 2 (Contexts are well-formed). *The following rules are admissible.*

$$\frac{\Theta; \Gamma \vdash \mathcal{J}}{\Theta; \Gamma \vdash} \qquad \frac{\Theta; \Gamma \vdash \mathcal{J}}{\Theta \vdash}$$

Proposition 3 (Weakening). *Let us write $\Gamma \sqsubseteq \Delta$ if Γ is a subsequence of Δ, and $\Theta \sqsubseteq \Xi$ if Θ is a subsequence of Ξ. The following rules are admissible.*

$$\Gamma \sqsubseteq \Delta \frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Theta; \Delta \vdash}{\Theta; \Delta \vdash \mathcal{J}} \qquad \Theta \sqsubseteq \Xi \frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Xi \vdash}{\Xi; \Gamma \vdash \mathcal{J}}$$

In order to state the substitution property, given $\Theta; \Delta \vdash \mathcal{J}$ we define the notations $(\Theta; \Delta \vdash \mathcal{J})[\vec{v}]$ and $(\Theta; \Delta \vdash \mathcal{J})[\mathbf{v}]$ by the following table.

$\Theta; \Delta \vdash \mathcal{J}$	$(\Theta; \Delta \vdash \mathcal{J})[\vec{v}]$ where $\Theta; \Gamma \vdash \vec{v} : \Delta$	$(\Theta; \Delta \vdash \mathcal{J})[\mathbf{v}]$ where $\Xi; \Gamma \vdash \mathbf{v} : \Theta$
$\Theta; \Delta \vdash$	$\Theta; \Gamma \vdash$	$\Xi; \Gamma.\Delta[\mathbf{v}] \vdash$
$\Theta; \Delta \vdash T\ \mathsf{sort}$	$\Theta; \Gamma \vdash T[\vec{v}]\ \mathsf{sort}$	$\Xi; \Gamma.\Delta[\mathbf{v}] \vdash T[\mathbf{v}]\ \mathsf{sort}$
$\Theta; \Delta \vdash t : T$	$\Theta; \Gamma \vdash t[\vec{v}] : T[\vec{v}]$	$\Xi; \Gamma.\Delta[\mathbf{v}] \vdash t[\mathbf{v}] : T[\mathbf{v}]$
$\Theta; \Delta \vdash \vec{t} : \Delta'$	$\Theta; \Gamma \vdash \vec{t}[\vec{v}] : \Delta'$	$\Xi; \Gamma.\Delta[\mathbf{v}] \vdash \mathsf{id}_\Gamma, \vec{t}[\mathbf{v}] : \Gamma.\Delta'[\mathbf{v}]$
$\Theta; \Delta \vdash \mathbf{t} : \Xi'$	$\Theta; \Gamma \vdash \mathbf{t}[\vec{v}] : \Xi'$	$\Xi; \Gamma.\Delta[\mathbf{v}] \vdash \mathbf{t}[\mathbf{v}] : \Xi'$

Proposition 4 (Substitution property). *The following rules are admissible.*

$$\frac{\Theta; \Gamma \vdash \vec{v} : \Delta \qquad \Theta; \Delta \vdash \mathcal{J}}{(\Theta; \Gamma \vdash \mathcal{J})[\vec{v}]} \qquad \frac{\Xi; \Gamma \vdash \mathbf{v} : \Theta \qquad \Theta; \Delta \vdash \mathcal{J}}{(\Theta; \Delta \vdash \mathcal{J})[\mathbf{v}]}$$

Proof. We illustrate the main case of the second statement's proof, which is by induction on $\Theta; \Delta \vdash \mathcal{J}$. Suppose the derivation ends with the rule MVar.

$$\mathsf{x}\{\Delta'\} : T \in \Theta \ \frac{\Theta; \Delta \vdash \vec{t} : \Delta'}{\Theta; \Delta \vdash \mathsf{x}\{\vec{t}\} : T[\vec{t}]}$$

By i.h. we have $\Xi; \Gamma.\Delta[\mathbf{v}] \vdash \mathsf{id}_\Gamma, \vec{t}[\mathbf{v}] : \Gamma.\Delta'[\mathbf{v}]$. Moreover, from $\Xi; \Gamma \vdash \mathbf{v} : \Theta$ we can derive $\Xi; \Gamma.\Delta'[\mathbf{v}] \vdash \mathbf{v}_\mathsf{x} : T[\mathbf{v}]$, so by the first statement (the substitution property for variable substitutions) we get $\Xi; \Gamma.\Delta[\mathbf{v}] \vdash \mathbf{v}_\mathsf{x}[\mathsf{id}_\Gamma, \vec{t}[\mathbf{v}]] : T[\mathbf{v}][\mathsf{id}_\Gamma, \vec{t}[\mathbf{v}]]$. Because $\mathsf{x}\{\vec{t}\}[\mathbf{v}] = \mathbf{v}_\mathsf{x}[\mathsf{id}_\Gamma, \vec{t}[\mathbf{v}]]$ and $T[\vec{t}][\mathbf{v}] = T[\mathbf{v}][\mathsf{id}_\Gamma, \vec{t}[\mathbf{v}]]$ we are done.

Proposition 5 (Sorts are well-typed). *The following rule is admissible when the underlying theory is valid.*

$$\frac{\Theta; \Gamma \vdash t : T}{\Theta; \Gamma \vdash T \ \mathsf{sort}}$$

Proof. By case analysis on $\Theta; \Gamma \vdash t : T$, and using Proposition 4. For rules CONS and DEST we use the validity of the theory to deduce $\Xi_1 \vdash T \ \mathsf{sort}$ from $c(\Xi_1; \Xi_2) : T \in \mathbb{T}$ and $\Xi_1.(\mathsf{x} : T).\Xi_2 \vdash U \ \mathsf{sort}$ from $d(\Xi_1; \mathsf{x} : T; \Xi_2) : U \in \mathbb{T}$.

Proposition 6 (Conversion in context). *The following rule is admissible.*

$$\Gamma \equiv \Delta \frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Theta; \Delta \vdash}{\Theta; \Delta \vdash \mathcal{J}}$$

3.4 Subject reduction

A key property that all of our valid theories satisfy is subject reduction. Aside from ensuring that well-typed programs cannot go wrong, this property will be vital to establish the soundness of the bidirectional type system.

In order to show subject reduction, we first need to prove some important properties of patterns. The first of them is injectivity with respect to conversion.

Lemma 1 (Injectivity of patterns). *If* $t \in \mathsf{Tm}^{\mathsf{P}} \ \theta \ \gamma$ *and* $t[\mathbf{v}] \equiv t[\mathbf{v}']$ *for some* $\mathbf{v} \in \mathsf{MSub} \ \theta' \ \delta \ \theta$ *and* $\mathbf{v}' \in \mathsf{MSub} \ \theta' \ \delta \ \theta$ *then* $\mathbf{v} \equiv \mathbf{v}'$.

Proof. By induction on the pattern, generalizing the statement also to metavariable substitution patterns. The key step is in case $c(\mathbf{t}[\mathbf{v}]) \equiv c(\mathbf{t}[\mathbf{v}'])$ in which we crucially rely on Proposition 1 to get $\mathbf{t}[\mathbf{v}] \equiv \mathbf{t}[\mathbf{v}']$ and invoke the i.h. to conclude.

Given a well-typed term t such that the result of substituting $\mathbf{v}$ in t is also well-typed, generally we cannot conclude that the metavariable substitution $\mathbf{v}$ is well-typed. This reasoning is however valid when t is a pattern, as shown by the following result. Differently from the previous lemma, its proof is more intricate so instead of trying to give a proof sketch we refer to the technical report [24].

Proposition 7 (Inversion of pattern typing). *Let* $\mathbf{v} \in \mathsf{MSub} \ (\cdot) \ |\Delta| \ |\Theta|$.

- *If* $t \in \mathsf{Tm}^{\mathsf{P}} \ |\Theta| \ (\cdot)$ *and* $\Theta \vdash t : T$ *and* $\Delta \vdash t[\mathbf{v}] : T[\mathbf{v}]$ *then* $\Delta \vdash \mathbf{v} : \Theta$
- *If* $T \in \mathsf{Tm}^{\mathsf{P}} \ |\Theta| \ (\cdot)$ *and* $\Theta \vdash T \ \mathsf{sort}$ *and* $\Delta \vdash T[\mathbf{v}] \ \mathsf{sort}$ *then* $\Delta \vdash \mathbf{v} : \Theta$
- *If* $\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}} \ |\Theta| \ (\cdot) \ |\Xi|$ *and* $\Theta \vdash \mathbf{t} : \Xi$ *and* $\Delta \vdash \mathbf{t}[\mathbf{v}] : \Xi$ *then* $\Delta \vdash \mathbf{v} : \Theta$

We are now ready to show subject reduction.

Theorem 1 (Subject reduction). *Suppose that the underlying theory is valid.*

- *If* $\Gamma \vdash T \ \mathsf{sort}$ *and* $T \longrightarrow T'$ *then* $\Gamma \vdash T' \ \mathsf{sort}$
- *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : T$
- *If* $\Gamma \vdash \mathbf{t} : \Xi$ *and* $\Xi \vdash$ *and* $\mathbf{t} \longrightarrow \mathbf{t}'$ *then* $\Gamma \vdash \mathbf{t}' : \Xi$

Proof. By induction on the rewrite judgment. We show only the main case:

$$\frac{\theta_1;\theta_2 \Vdash d(t;\mathbf{u}) \longmapsto r \qquad \mathbf{v}_1 \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ \theta_1 \qquad \mathbf{v}_2 \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ \theta_2}{d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) \longrightarrow r[\mathbf{v}_1,\mathbf{v}_2]}$$

Let $d(\Xi_1;\mathsf{x}:U;\Xi_2):V \in \mathbb{T}$ be the rule for d in $\mathbb{T}$. Because $\mathbb{T}$ is valid, there are Θ_1 and Θ_2 with $|\Theta_1| = \theta_1$ and $|\Theta_2| = \theta_2$ such that

$$\Xi_1.\Theta_1.\Theta_2 \vdash (\mathsf{id}_{\Xi_1},t,\mathbf{u}) : \Xi_1.(\mathsf{x}:U).\Xi_2 \qquad \text{and} \qquad \Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1},t,\mathbf{u}]$$

By inversion on $\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T$ we get

$$T \equiv V[\mathbf{v}_0,t[\mathbf{v}_1],\mathbf{u}[\mathbf{v}_2]]\frac{\dfrac{\Gamma \vdash \mathbf{v}_0,t[\mathbf{v}_1],\mathbf{u}[\mathbf{v}_2] : \Xi_1.(\mathsf{x}:U).\Xi_2}{\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : V[\mathbf{v}_0,t[\mathbf{v}_1],\mathbf{u}[\mathbf{v}_2]]}}{\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T}$$

Therefore, we have $\Gamma \vdash (\mathsf{id}_{\Xi_1},t,\mathbf{u})[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] : \Xi_1.(\mathsf{x}:U).\Xi_2$, and because $\mathsf{id}_{\Xi_1},t,\mathbf{u} \in \mathsf{MSub}^{\mathsf{P}}\ |\Xi_1.\Theta_1.\Theta_2|\ (\cdot)\ |\Xi_1.(\mathsf{x}:U).\Xi_2|$, then by Proposition 7 we get $\Gamma \vdash \mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2 : \Xi_1.\Theta_1.\Theta_2$. By applying Proposition 4 with $\Xi_1.\Theta_1.\Theta_2 \vdash r : V[\mathsf{id}_{\Xi_1},t,\mathbf{u}]$ we get $\Gamma \vdash r[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] : V[\mathsf{id}_{\Xi_1},t,\mathbf{u}][\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2]$. Finally, we have $\Gamma \vdash T\ \mathsf{sort}$ by Proposition 5 applied to $\Gamma \vdash d(t[\mathbf{v}_1];\mathbf{u}[\mathbf{v}_2]) : T$, and because $r[\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] = r[\mathbf{v}_1,\mathbf{v}_2]$ and $V[\mathsf{id}_{\Xi_1},t,\mathbf{u}][\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2] = V[\mathbf{v}_0,t[\mathbf{v}_1],\mathbf{u}[\mathbf{v}_2]] \equiv T$ then by the conversion rule we conclude $\Gamma \vdash r[\mathbf{v}_1,\mathbf{v}_2] : T$.

4 Bidirectional type system

In the previous section we have defined the declarative type system. We now move to the bidirectional type system. We start the section by discussing the problem of matching modulo, which is needed for recovering missing arguments. We then introduce the inferable and checkable terms, for which the bidirectional system is defined. We then give the bidirectional typing rules and prove they are sound and complete with respect to the declarative type system. Finally, we use this equivalence to deduce some important properties of the declarative system.

4.1 Matching modulo rewriting

Suppose we want to type $@(t;u)$ by first inferring the sort of t, yielding T. We know that the sort of the principal argument in the rule for $@$ is the pattern $\mathrm{Tm}(\Pi(\mathsf{A},x.\mathsf{B}\{x\}))$, so we could hope to recover A and B by matching T against this pattern. However, because of the conversion rule, in dependent type theories we cannot expect T to be syntactically equal to an instance of this pattern, but only convertible to it. Therefore, our goal is instead to find A and B satisfying $\mathrm{Tm}(\Pi(\mathsf{A},x.\mathsf{B}\{x\}))[A,x.B] \equiv T$. This shows that the process of recovering missing arguments in bidirectional typing is actually an instance of *matching modulo rewriting* — a connection that seems not to have been noted before in the literature. This also explains why we were careful in Subsection 2.4 to require the sort of a constructor rule and the sort of the principle argument of a destructor rule to be patterns, as they need to support decidable and unitary matching.

In order to explain how to solve matching modulo problems, let us first recall some concepts about rewriting theory. A *(functional) strategy* $\mathfrak{S}$ [11] is defined by a subrelation $\longrightarrow_{\mathfrak{S}} \subseteq \longrightarrow^{+}$ which has the same normal forms as $\longrightarrow$ and is functional in the sense that $t \longrightarrow_{\mathfrak{S}} u_1$ and $t \longrightarrow_{\mathfrak{S}} u_2$ imply $u_1 = u_2$. Let $\mathsf{m/o}$ be the *maximal outermost* strategy, which contracts all outermost redexes in one step, and write $t \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u})$ if $c(\mathbf{u})$ is the first term headed by a constructor to which t reduces by $\longrightarrow^{*}_{\mathsf{m/o}}$.

We can now define in Figure 3 an inference system for matching modulo rewriting, which given a pattern t and a term u tries to compute a metavariable substitution $\mathbf{v}$ such that $t[\mathbf{v}] \equiv u$. Note that the use of a specific rewriting strategy is necessary to ensure the functionality of the inference system.

$$\boxed{t \prec u \rightsquigarrow \mathbf{v} \quad (t \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma;\ u \in \mathsf{Tm}\ (\cdot)\ \delta.\gamma;\ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta)}$$

$$\texttt{Rigid}_{\prec}\ \frac{u \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u}) \qquad \mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}}{c(\mathbf{t}) \prec u \rightsquigarrow \mathbf{v}} \qquad \texttt{Flex}_{\prec}\ \frac{}{\mathsf{x}\{\mathsf{id}_{\gamma}\} \prec u \rightsquigarrow \vec{x}_{\gamma}.u}$$

$$\boxed{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v} \quad (\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi;\ \mathbf{u} \in \mathsf{MSub}\ (\cdot)\ \delta.\gamma\ \xi;\ \mathbf{v} \in \mathsf{Sub}\ (\cdot)\ \delta\ \theta)}$$

$$\texttt{EmptyMSub}_{\prec}\ \frac{}{\varepsilon \prec \varepsilon \rightsquigarrow \varepsilon} \qquad \texttt{ExtMSub}_{\prec}\ \frac{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}_1 \qquad t \prec u \rightsquigarrow \mathbf{v}_2}{\mathbf{t}, \vec{x}.t \prec \mathbf{u}, \vec{x}.u \rightsquigarrow \mathbf{v}_1, \mathbf{v}_2}$$

Fig. 3. Inference system for matching modulo

Let us now establish the correctness of this inference system in three steps. We first show its soundness, which follows by an easy induction on the derivation.

Proposition 8 (Soundness of matching).

- *If* $t \prec u \rightsquigarrow \mathbf{v}$ *then* $u \longrightarrow^{*} t[\mathbf{v}]$.
- *If* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ *then* $\mathbf{u} \longrightarrow^{*} \mathbf{t}[\mathbf{v}]$.

In order to show completeness we first have to answer the following question: if $t \equiv c(\mathbf{u})$, are we sure that by reducing t we eventually reach a term headed by c? The answer would be no, had we taken for instance an innermost strategy. Thankfully, because the rewrite systems of our theories are both orthogonal and *fully-extended*, it follows by [39, Theorem 2] that the maximal-outermost strategy we are using is *head-normalizing*, and so we have the following property.

Lemma 2. *If* $u \equiv c(\mathbf{t})$ *then* $u \longrightarrow^{\mathsf{h}}_{\mathsf{m/o}} c(\mathbf{u})$ *with* $\mathbf{t} \equiv \mathbf{u}$.

Using Lemma 2, we can now show completeness by induction on the pattern.

Proposition 9 (Completeness of matching).

– *If* $t[\mathbf{v}] \equiv u$ *for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$ *then* $t \prec u \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$
– *If* $\mathbf{t}[\mathbf{v}] \equiv \mathbf{u}$ *for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$ *then* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$

Recall that an expression is weak normalizing if it reduces to a normal form. We can now show that the inference system is decidable when being used with weak normalizing expressions. The proof is by induction on the pattern, using the fact that $\mathsf{m/o}$ is *normalizing* [42, Theorem 10], so that reducing a weak normalizing term with $\mathsf{m/o}$ always terminates.

Proposition 10 (Decidability of matching).

– *If* u *is weak normalizing, then* $\exists \mathbf{v}.\ t \prec u \rightsquigarrow \mathbf{v}$ *is decidable for all* t.
– *If* $\mathbf{u}$ *is weak normalizing, then* $\exists \mathbf{v}.\ \mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ *is decidable for all* $\mathbf{t}$.

4.2 Inferable and checkable terms

Before giving the bidirectional typing rules, we first have to address the problem that some terms without annotations cannot be algorithmically typed. Suppose for instance that we want to type the term $@(\lambda(x.t); u)$ by inferring the sort of the principal argument of $@$ to recover A and B. But because $\lambda(x.t)$ is headed by a constructor it can only be bidirectionally typed in mode check, so we are stuck. One could think that this limitation is specific to bidirectional typing, however a famous result by Dowek shows that, in a dependently-typed setting, the problem of typing unannotated terms is actually undecidable in its full generality [18].

To solve this issue, we have two options. We could proceed as in the Coq literature [33] and add extra annotations to terms so that they can always be inferred. For instance, we would then need to annotate abstraction with its domain by writing $\lambda(A, x.t)$ instead of $\lambda(x.t)$. However, not only this solution makes the syntax heavier, but by abandoning the constructor/destructor separation in which constructors are always typed in mode check, it yields typing rules which are much less symmetric and whose form seems difficult to specify in a generic way.

We instead take the choice made by most of the dependent bidirectional typing literature [2,38,1,16,3,4] and define our bidirectional system only for a subset of expressions, the *checkable and inferable terms* and the *checkable metavariable substitutions*. Given a fixed signature Σ, they are defined as follows.

$$
\begin{array}{rrll}
\boxed{\mathsf{Tm}^{\mathsf{i}}\ \gamma} \ni & t, u, v ::= & \mid x & \text{if } x \in \gamma \\
 & & \mid d(t \in \mathsf{Tm}^{\mathsf{i}}\ \gamma; \mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } d(\xi) \in \Sigma \\
\boxed{\mathsf{Tm}^{\mathsf{c}}\ \gamma} \ni & t, u, v ::= & \mid c(\mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
 & & \mid \underline{t} \in \mathsf{Tm}^{\mathsf{i}}\ \gamma & \\
\boxed{\mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi} \ni & \mathbf{t}, \mathbf{u}, \mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \\
 & & \mid \mathbf{t}' \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}^{\mathsf{c}}\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}
$$

Let us go through the definition. An inferable term is either a variable or a destructor whose principal argument is inferable, and whose other arguments are given by a checkable metavariable substitution. Imposing the principal argument to be inferable is the key restriction to rule out terms like $@(\lambda(x.t);u)$. A checkable term is then either a constructor applied to a checkable metavariable substitution, or an underlined inferable term. Finally, a checkable metavariable substitution is just a metavariable substitution containing only checkable terms.

Example 9. The inferrable and checkable terms for the signature $\Sigma_{\lambda\Pi}$ are given respectively by the following grammars, where we omit the scope information.

$$t^{\mathsf{i}}, u^{\mathsf{i}} ::= x \mid @(t^{\mathsf{i}}; u^{\mathsf{c}})$$
$$t^{\mathsf{c}}, u^{\mathsf{c}}, A^{\mathsf{c}}, B^{\mathsf{c}} ::= \mathrm{Ty} \mid \mathrm{Tm}(A^{\mathsf{c}}) \mid \Pi(A^{\mathsf{c}}, x.B^{\mathsf{c}}) \mid \lambda(x.t^{\mathsf{c}}) \mid \underline{t^{\mathsf{i}}}$$

One could wonder if restricting the terms that can be algorithmically typed is a significant limitation. For most usual theories (like $\mathbb{T}_{\lambda\Pi}$ and those in Section 5) the checkable terms coincide with the normal forms, and the inferable terms coincide with the neutrals. As argued in other works [38,1], users of type theory almost only write terms in normal form, and in the few cases writing a redex is more convenient, in actual implementations the principal argument can always be lifted to a top-level definition. Therefore, this restriction, also present in most of the dependent bidirectional typing literature [2,38,1,16,3,4], does not pose a serious limitation in practice.

Note also that inferable and checkable terms have no metavariables. Even if metavariables are needed in the declarative system to be able to say which theories are valid, when writing terms in a fixed theory metavariables are generally not needed, and hence they are omitted from usual presentations of type theories. It is therefore reasonable to leave them out of the bidirectional syntax, as they will be of no use for users.

Given $t \in \mathsf{Tm}^{\mathsf{c}}\ \gamma$ or $t \in \mathsf{Tm}^{\mathsf{i}}\ \gamma$ we write $\ulcorner t \urcorner \in \mathsf{Tm}\ (\cdot)\ \gamma$ for its underlying term, and for $\mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ \gamma\ \xi$ we also write $\ulcorner \mathbf{t} \urcorner \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ for its underlying metavariable substitution.

4.3 Bidirectional typing rules

Given a theory $\mathbb{T}$, we can now define its bidirectional type system by the rules in Figure 4. The system is split in 4 judgments:

- $\Gamma \vdash T \Leftarrow \mathsf{sort}$: Checking that a checkable term T is a well-formed sort
- $\Gamma \vdash t \Leftarrow T$: Checking that a checkable term t has sort T
- $\Gamma \vdash t \Rightarrow T$: Inferring a sort T for an inferable term t
- $\Gamma \mid \mathbf{v} : \Xi \vdash \mathbf{t} \Leftarrow \Theta$: Checking that a checkable metavariable substitution $\mathbf{t}$ can be typed by Θ "to the right" of $\mathbf{v} : \Xi$

As in the declarative system, the most important rules are the one that instantiate the schematic typing rules: Const, Dest and Sort. However, differently from the declarative system, no more guessing is needed when building

$$\boxed{\Gamma \vdash t \Leftarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{Tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$c(\Xi_1;\Xi_2) : T \in \mathbb{T}\ \dfrac{T \prec U \rightsquigarrow \mathbf{v} \qquad \Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{u} \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{u}) \Leftarrow U}\ \textsc{Cons} \qquad T \equiv_{\mathbb{T}} U\ \dfrac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash \underline{t} \Leftarrow U}\ \textsc{Switch}$$

$$\boxed{\Gamma \vdash t \Rightarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{Tm}^{\mathsf{i}}\ |\Gamma|)}$$

$$d(\Xi_1;\mathsf{x}:T;\Xi_2) : U \in \mathbb{T}\ \dfrac{\Gamma \vdash t \Rightarrow V \qquad T \prec V \rightsquigarrow \mathbf{v} \qquad \Gamma \mid (\mathbf{v}, \ulcorner t \urcorner) : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{u} \Leftarrow \Xi_2}{\Gamma \vdash d(t;\mathbf{u}) \Rightarrow U[\mathbf{v}, \ulcorner t \urcorner, \ulcorner \mathbf{u} \urcorner]}\ \textsc{Dest} \qquad x : T \in \Gamma\ \dfrac{}{\Gamma \vdash x \Rightarrow T}\ \textsc{Var}$$

$$\boxed{\Gamma \vdash T \Leftarrow \mathsf{sort} \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$c(\Xi)\ \mathsf{sort} \in \mathbb{T}\ \dfrac{\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t} \Leftarrow \Xi}{\Gamma \vdash c(\mathbf{t}) \Leftarrow \mathsf{sort}}\ \textsc{Sort}$$

$$\boxed{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \quad \begin{pmatrix}\Gamma \in \mathsf{Ctx};\ \Theta \in \mathsf{MCtx};\ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ |\Theta|; \\ \Xi \in \mathsf{MCtx}\ |\Theta|;\ \mathbf{t} \in \mathsf{MSub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|\end{pmatrix}}$$

$$\dfrac{}{\Gamma \mid \mathbf{v} : \Theta \vdash \varepsilon \Leftarrow (\cdot)}\ \textsc{EmptyMSub} \qquad \dfrac{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \qquad \Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash t \Leftarrow T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]}{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}_\Delta.t \Leftarrow (\Xi, \mathsf{x}\{\Delta\} : T)}\ \textsc{ExtMSub}$$

Fig. 4. Bidirectional typing rules

a type derivation. For instance, when using rule Dest with $d(t;\mathbf{u})$ the omitted arguments are no longer guessed, but instead recovered by inferring the sort of the principal argument t and then matching it against the associated pattern.

Example 10. Suppose we want to infer a sort for $@(t;u)$ in the theory $\mathbb{T}_{\lambda\Pi}$. To use rule Dest, we start by inferring a sort V for t, and then we try to match it against the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$. If matching succeeds, we recover the arguments A and B, which together with t are then used in

$$\Gamma \mid (A, x.B, x.\ulcorner t \urcorner) : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathsf{t} : ...) \vdash (u) \Leftarrow (\mathsf{u} : \mathrm{Tm}(\mathsf{A}))$$

where we omit the sort of t for lack of space. By applying the rules that define the judgment $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$, we see that this amounts to showing just $\Gamma \vdash u \Leftarrow \mathrm{Tm}(A)$, and so the final shape of this "big-step derivation" is the following, which corresponds to the usual bidirectional rule for application.

$$\dfrac{\Gamma \vdash t \Rightarrow V \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \prec V \rightsquigarrow A, x.B \qquad \Gamma \vdash u \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t;u) \Rightarrow \mathrm{Tm}(B[\mathsf{id}_\Gamma, \ulcorner u \urcorner])}$$

4.4 Equivalence with declarative typing

We now establish the equivalence between the declarative and bidirectional type systems. This is done in two steps, the first one being soundness:

Theorem 2 (Soundness). *Suppose that the underlying theory $\mathbb{T}$ is valid.*

1. *If* $\Gamma \vdash$ *and* $\Gamma \vdash t \Rightarrow T$ *then* $\Gamma \vdash \ulcorner t \urcorner : T$
2. *If* $\Gamma \vdash T\ \mathsf{sort}$ *and* $\Gamma \vdash t \Leftarrow T$ *then* $\Gamma \vdash \ulcorner t \urcorner : T$
3. *If* $\Gamma \vdash$ *and* $\Gamma \vdash T \Leftarrow \mathsf{sort}$ *then* $\Gamma \vdash \ulcorner T \urcorner\ \mathsf{sort}$
4. *If* $\Gamma \vdash \mathbf{v} : \Xi_1$ *and* $\Xi_1.\Xi_2 \vdash$ *and* $\Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{t} \Leftarrow \Xi_2$ *then* $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Xi_1.\Xi_2$.

Proof. By induction on the derivation. We illustrate one of the interesting cases.

$$c(\Xi_1;\Xi_2) : T \in \mathbb{T}\ \frac{T \prec U \rightsquigarrow \mathbf{v} \qquad \Gamma \mid \mathbf{v} : \Xi_1 \vdash \mathbf{u} \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{u}) \Leftarrow U}$$

By Proposition 8 we have $U \longrightarrow^* T[\mathbf{v}]$, so because we have $\Gamma \vdash U\ \mathsf{sort}$ then by Theorem 1 we get $\Gamma \vdash T[\mathbf{v}]\ \mathsf{sort}$. We have $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$, and validity of the theory also gives $\Xi_1 \vdash T\ \mathsf{sort}$, therefore by Proposition 7 we get $\Gamma \vdash \mathbf{v} : \Xi_1$. By validity of the theory we also have $\Xi_1.\Xi_2 \vdash$, therefore by applying the i.h. to the second premise we get $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{u} \urcorner : \Xi_1.\Xi_2$. We can then derive $\Gamma \vdash c(\ulcorner \mathbf{u} \urcorner) : T[\mathbf{v}]$, and because $T[\mathbf{v}] \equiv U$ and $\Gamma \vdash U\ \mathsf{sort}$, by conversion we conclude $\Gamma \vdash c(\ulcorner \mathbf{u} \urcorner) : U$.

Completeness then asserts that checkable/inferable terms typable in the declarative system can also be typed in the bidirectional system.

Theorem 3 (Completeness). *Suppose that the underlying theory $\mathbb{T}$ is valid.*

1. *If t is inferable and* $\Gamma \vdash \ulcorner t \urcorner : T$ *then* $\Gamma \vdash t \Rightarrow T'$ *with* $T \equiv T'$
2. *If t is checkable and* $\Gamma \vdash \ulcorner t \urcorner : T$ *then we have* $\Gamma \vdash t \Leftarrow T$
3. *If T is checkable and* $\Gamma \vdash \ulcorner T \urcorner\ \mathsf{sort}$ *then* $\Gamma \vdash T \Leftarrow \mathsf{sort}$
4. *If $\mathbf{t}$ is checkable and* $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$ *then we have* $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$

Proof. The proof requires us to strengthen the statement, so the two occurrences of the context Γ in points 1-4, of the sort T in point 2 and of the substitution $\mathbf{v}$ in point 4 are not required to be syntactically equal, but only convertible (see the technical report [24] for the exact statement). The proof is then by induction on the checkable/inferable term or checkable substitution.

We illustrate the case $t = d(u; \mathbf{t})$, in which we have to show that for all $\Gamma' \equiv \Gamma$ we have some $T' \equiv T$ such that $\Gamma' \vdash t \Rightarrow T'$. By inversion on $\Gamma \vdash \ulcorner t \urcorner : T$ we have

$$T \equiv V[\mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]\ \frac{d(\Xi_1; \mathsf{x} : U; \Xi_2) : V \in \mathbb{T}\ \dfrac{\Gamma \vdash \mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner : \Xi_1.(\mathsf{x} : U).\Xi_2}{\Gamma \vdash d(\ulcorner u \urcorner; \ulcorner \mathbf{t} \urcorner) : V[\mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]}}{\Gamma \vdash d(\ulcorner u \urcorner; \ulcorner \mathbf{t} \urcorner) : T}$$

Let $\Gamma' \equiv \Gamma$. From $\Gamma \vdash \mathbf{v}, \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner : \Xi_1.(\mathsf{x} : U).\Xi_2$ we get $\Gamma \vdash \ulcorner u \urcorner : U[\mathbf{v}]$, so because u is inferrable, by the i.h. we obtain that for some $U' \equiv U[\mathbf{v}]$ we have $\Gamma' \vdash u \Rightarrow U'$. By Proposition 9 we then get $U \prec U' \rightsquigarrow \mathbf{v}'$ with $\mathbf{v} \equiv \mathbf{v}'$. Then, because $\mathbf{t}$ is checkable and $\Gamma \equiv \Gamma'$ and $\mathbf{v}, \ulcorner u \urcorner \equiv \mathbf{v}', \ulcorner u \urcorner$, by the i.h. we derive $\Gamma' \mid (\mathbf{v}', \ulcorner u \urcorner) : (\Xi_1, \mathsf{x} : U) \vdash \mathbf{t} \Leftarrow \Xi_2$. Putting all this together, we conclude $\Gamma' \vdash d(u; \mathbf{t}) \Rightarrow V[\mathbf{v}', \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner]$, where we have $V[\mathbf{v}', \ulcorner u \urcorner, \ulcorner \mathbf{t} \urcorner] \equiv T$ as required.

4.5 Consequences of the equivalence

We now explore the established equivalence in order to show two important properties: decidability of typing and uniqueness of sorts.

We say that a theory $\mathbb{T}$ is weak normalizing if for all expressions e with $\Gamma \vdash e$ sort or $\Gamma \vdash e : T$ or $\Gamma \vdash e : \Xi$ we have that e is weak normalizing.

Theorem 4 (Decidability of typing). *Suppose that the underlying theory $\mathbb{T}$ is valid and weak normalizing.*

1. *If t is inferable and $\Gamma \vdash$ then the statement $\exists T.\ (\Gamma \vdash \ulcorner t \urcorner : T)$ is decidable.*
2. *If t is checkable and $\Gamma \vdash T$ sort then the statement $\Gamma \vdash \ulcorner t \urcorner : T$ is decidable.*
3. *If T is checkable and $\Gamma \vdash$ then the statement $\Gamma \vdash \ulcorner T \urcorner$ sort is decidable.*
4. *If $\mathbf{t}$ is checkable and $\Theta.\Xi \vdash$ and $\Gamma \vdash \mathbf{v} : \Theta$ then the statement $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$ is decidable.*

Proof. We first show the corresponding statement for the bidirectional system, using Proposition 10, Theorem 2 and the decidability of conversion for well-typed terms (which follows from weak normalization). By Theorems 2 and 3 we can then conclude. We refer to the technical report [24] for the proof.

We now move to uniqueness of sorts. First note that, because our terms are non-annotated, uniqueness of sorts does not hold in general: for instance, $\lambda(x.x)$ can be typed by $\mathrm{Tm}(\Pi(A, x.A))$ in context Γ for any A with $\Gamma \vdash A : \mathrm{Ty}$. Nevertheless, we can still show uniqueness of sorts for inferable terms:

Theorem 5 (Uniqueness of sorts). *Suppose that the underlying theory $\mathbb{T}$ is valid. If t is inferable and $\Gamma \vdash \ulcorner t \urcorner : T$ and $\Gamma \vdash \ulcorner t \urcorner : U$ then $T \equiv U$.*

Proof. By Theorem 3 we get $\Gamma \vdash t \Rightarrow T'$ with $T \equiv T'$ from $\Gamma \vdash \ulcorner t \urcorner : T$ and $\Gamma \vdash t \Rightarrow U'$ with $U \equiv U'$ from $\Gamma \vdash \ulcorner t \urcorner : U$. We can show type inference to be functional, so we get $T' = U'$ and thus $T \equiv U$.

5 More examples

In the previous sections we have illustrated our framework with the theory $\mathbb{T}_{\lambda\Pi}$, defining a basic Martin-Löf Type Theory with dependent products. We now show other examples of valid theories to showcase the generality of our framework. Throughout this section, we use the informal notation for schematic typing rules discussed in Subsection 2.4, for readability purposes. We refer to the files of the implementation [23] for more details.

Lists We can define lists by extending $\mathbb{T}_{\lambda\Pi}$ with the following.

$$\frac{\vdash \mathtt{A} : \mathrm{Ty}}{\vdash \mathrm{List}(\mathtt{A}) : \mathrm{Ty}} \qquad \frac{\vdash \mathtt{A} : \mathrm{Ty}}{\vdash \mathrm{nil} : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))} \qquad \frac{\vdash \mathtt{A} : \mathrm{Ty} \qquad \vdash \mathtt{x} : \mathrm{Tm}(\mathtt{A}) \qquad \vdash \mathtt{l} : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))}{\vdash \mathrm{cons}(\mathtt{x}, \mathtt{l}) : \mathrm{Tm}(\mathrm{List}(\mathtt{A}))}$$

$$\frac{\begin{array}{c}\vdash \mathtt{A} : \mathrm{Ty} \qquad \vdash \mathtt{l} : \mathrm{Tm}(\mathrm{List}(\mathtt{A})) \qquad x : \mathrm{Tm}(\mathrm{List}(\mathtt{A})) \vdash \mathtt{P} : \mathrm{Ty} \qquad \vdash \mathtt{pnil} : \mathrm{Tm}(\mathtt{P}\{\mathrm{nil}\}) \\ x : \mathrm{Tm}(\mathtt{A}), y : \mathrm{Tm}(\mathrm{List}(\mathtt{A})), z : \mathrm{Tm}(\mathtt{P}\{y\}) \vdash \mathtt{pcons} : \mathrm{Tm}(\mathtt{P}\{\mathrm{cons}(x, y)\})\end{array}}{\vdash \mathrm{ListRec}(\mathtt{l}; \mathtt{P}, \mathtt{pnil}, \mathtt{pcons}) : \mathrm{Tm}(\mathtt{P}\{\mathtt{l}\})}$$

$$\mathrm{ListRec}(\mathrm{nil}; x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\}) \longmapsto \mathtt{pnil}$$

$$\mathrm{ListRec}(\mathrm{cons}(\mathsf{x}, \mathtt{l}); x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\}) \longmapsto$$
$$\mathtt{pcons}\{\mathsf{x}, \mathtt{l}, \mathrm{ListRec}(\mathtt{l}; x.\mathsf{P}\{x\}, \mathtt{pnil}, xyz.\mathtt{pcons}\{x, y, z\})\}$$

Like one would wish, the constructors nil and cons indeed do not store the type annotation A, which is recovered from the sort. This annotation is also elided in the destructor ListRec, where it is recovered from the principal argument.

In general, we can extend the theory with any (non-indexed) inductive type. For instance, see the file `mltt.bitt` where we add dependent sums and W types.

Universes We can define *Tarski-style* universes by extending $\mathbb{T}_{\lambda\Pi}$ with a type U and a decoding function mapping each inhabitant a of U into a type $\mathrm{El}(a; \varepsilon)$.

$$\frac{}{\vdash \mathrm{U} : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U})}{\vdash \mathrm{El}(\mathsf{a}; \cdot) : \mathrm{Ty}}$$

We then add a code for each type of the theory, with an associated rewriting rule stating that the code is decoded by El into the appropriate type.

$$\frac{}{\vdash \mathrm{u} : \mathrm{Tm}(\mathrm{U})} \qquad \frac{\vdash \mathsf{a} : \mathrm{Tm}(\mathrm{U}) \qquad x : \mathrm{Tm}(\mathrm{El}(\mathsf{a}; \varepsilon)) \vdash \mathsf{b} : \mathrm{Tm}(\mathrm{U})}{\vdash \pi(\mathsf{a}, \mathsf{b}) : \mathrm{Tm}(\mathrm{U})}$$

$$\mathrm{El}(\mathrm{u}; \varepsilon) \longmapsto \mathrm{U} \qquad \mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\}); \varepsilon) \longmapsto \Pi(\mathrm{El}(\mathsf{a}; \varepsilon), x.\mathrm{El}(\mathsf{b}\{x\}; \varepsilon))$$

This specifies a type in type universe, which is known to be inconsistent [15]. This can however be easily fixed by stratifying universes into a hierarchy. By doing this, we can then define a Tarski-style variant of (functional) *Pure Type Systems* [9]. Alternatively, instead of using Tarski-style we can also define (weak) *Coquand-style universes* [17,30,32,6] which require replacing the sorts Ty and Tm by indexed families Ty_i and Tm_i — see the file `mltt-coquand.bitt` for a definition also featuring (weak) cumulativity and universe polymorphism.

Higher-order logic We have seen how to extend $\mathbb{T}_{\lambda\Pi}$ with various type formers, however we can also define logics. To define higher-order logic (HOL) we first declare a type of propositions and a sort rule to represent the judgment "P is true".

$$\frac{}{\vdash \mathrm{Prop} : \mathrm{Ty}} \qquad \frac{\vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})}{\vdash \mathrm{Prf}(\mathsf{P})\ \mathsf{sort}}$$

We can then add connectors or quantifiers such as the universal quantifier $\forall$ — see the file `hol.bitt` for more details.

$$\frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})\end{array}}{\vdash \forall(\mathsf{A}, \mathsf{P}) : \mathrm{Tm}(\mathrm{Prop})} \qquad \frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{p} : \mathrm{Prf}(\mathsf{P})\end{array}}{\vdash \forall_{\mathrm{in}}(\mathsf{p}) : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\}))}$$

$$\frac{\begin{array}{c}\vdash \mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ \vdash \mathsf{r} : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\})) \qquad \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{A})\end{array}}{\vdash \forall_{\mathrm{el}}(\mathsf{r}; \mathsf{t}) : \mathrm{Prf}(\mathsf{P}\{\mathsf{t}\})} \qquad \forall_{\mathrm{el}}(\forall_{\mathrm{in}}(x.\mathsf{p}\{x\}); \mathsf{t}) \longmapsto \mathsf{p}\{\mathsf{t}\}$$

6 Related work

Our general definition of dependent type theories draws much inspiration from other frameworks for type theory, such as GATs/QIITs [13,7,31], SOGATs [45], FTTs [29], and logical frameworks such as Dedukti [8,12] and Harper's Equational LF [27]. However, we differ from these works by supporting non-annotated syntaxes and enforcing a constructor/destructor separation of symbols and rules, both of which seem to be important ingredients for bidirectional typing.

Another point of divergence from these frameworks is that most of them allow the use of arbitrary equations when defining the definitional equality of theories. However, it then becomes hard to give an implementation, as it would require deciding arbitrary equational theories. We instead take the approach of Dedukti of supporting only rewrite rules, which allows to decide the definitional equality of theories in a uniform manner, and makes it possible to implement our framework. A different approach is taken in Andromeda, an implementation of FTTs, where they also allow for extensionality rules [10]. They however provide no proof of completeness for their equality-checking algorithm.

Our proposal also draws inspiration from the works of McBride, a main advocate of dependent bidirectional typing. His ongoing work on a framework for bidirectional typing [35,36] shares many similarities with ours, for instance by adopting a constructor/destructor separation of rules. However, an important difference with our framework is that he takes the bidirectional type system as the definition of the theory. Therefore, there is no discussion on how to show soundness and completeness with respect to a declarative system, as the bidirectional one is the only type system defined in his setting. This approach differs from most of the literature on dependent bidirectional typing [19,2,33,1,3,4], in which one first defines the type theory by a "platonic" declarative type system and then shows it equivalent to a bidirectional system which can be implemented. Finally, this choice also makes the metatheoretic study of theories quite different: for instance, in order for the bidirectional system to satisfy subject reduction he is obliged to introduce type ascriptions in the syntax.

Another work from which ours drew inspiration is the one of Reed [43], where he proposes a variant of the Edinburgh Logical Framework in which arguments can be omitted. Crucially, these arguments are not elaborated through global unification, but instead locally recovered by annotating each declaration with modes to guide a bidirectional algorithm. However, his framework does not allow for extending the definitional equality, meaning that one cannot define dependent type theories directly, but instead has to encode its derivations trees (as in [28]). This also means that his system does not need to deal with some complications that arise in our more general setting, such as matching modulo.

Finally, concurrently to our work, Chen and Ko [14] have proposed a framework for simply-typed bidirectional typing. They also define declarative and bidirectional systems and establish a correspondence between them. Compared to our work, their restriction to simple types removes many of the complexities that appear in dependent type theories. For instance, while their types are first-order terms with no notion of computation or typing, our sorts are higher-order terms considered modulo a set of rewrite rules and subject to typing judgments, making the process of recovering missing arguments much more intricate. They however formalize all their proofs in Agda.

7 Conclusion and future work

In this work we have given a generic account of bidirectional typing for a general class of dependent type theories. Our main results, Theorems 2 and 3, establish an equivalence between declarative and bidirectional type systems for a general class of theories. The underlying algorithm of Theorem 4, establishing the decidability of typing for weak normalizing theories, has been implemented in a prototype further described in an accompanying experience report [22]. Compared to other theory-independent typecheckers, such as Dedukti, its support for unannotated syntaxes should allow for better performances, which can make it a good candidate for cross-checking real proof libraries.

Regarding future work, the most important omission that we would like to address is that of inductive families. Indeed, these do not fit our definitions because their constructor's sorts either are non-linear patterns (as in the constructor for equality) or contain metavariables for arguments that are computationally relevant and thus cannot be omitted (as in the cons constructor for vectors).

Moreover, even if our system builds heavily on the constructor/destructor distinction in type theory, some few constructions do not respect this separation. For instance, to define Russell universes we need the rewrite rule $\mathrm{Tm}(\mathrm{U}) \longmapsto \mathrm{Ty}$ [44], which is not valid as Tm is not a destructor. Whether there is a way of accommodating these constructions without fully abandoning the constructor/destructor separation is something we would like to investigate in future work.

A long term goal is also to extend our framework to account for type-directed equality rules, which are needed for handling η-laws and definitional proof irrelevance. Even if it is well known how to design complete equality checking algorithms for specific theories with type-directed equalities [5], doing so in a general setting like ours seems to be an important challenge. We could take inspiration from the customizable equality-checking algorithm implemented in Andromeda [10]. However, as mentioned in the previous section, their algorithm is not proven complete, so further research in this direction seems to be needed.

Acknowledgements The author thanks Gilles Dowek, Vincent Moreau, Théo Winterhalter for helpful remarks on a preliminary version of this paper, Meven Lennon-Bertrand, Jesper Cockx, András Kovács for informative discussions about bidirectional typing, and Frédéric Blanqui for valuable remarks about this work.

Data-Availability Statement The implementation associated with this work is available online [23] and is described in an accompanying experience report [22].

References

1. Abel, A., Altenkirch, T.: A partial type checking algorithm for type: Type. Electronic Notes in Theoretical Computer Science **229**(5), 3–17 (2011)
2. Abel, A., Coquand, T.: Untyped algorithmic equality for martin-löf's logical framework with surjective pairs. In: Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005. Proceedings 7. pp. 23–38. Springer (2005)
3. Abel, A., Coquand, T., Pagano, M.: A modular type-checking algorithm for type theory with singleton types and proof irrelevance. Logical Methods in Computer Science **7** (2011)
4. Abel, A., Vezzosi, A., Winterhalter, T.: Normalization by evaluation for sized dependent types. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–30 (2017)
5. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (Jan 2018). https://doi.org/10.1145/3158111, https://dl.acm.org/doi/10.1145/3158111
6. Altenkirch, T., Boulier, S., Kaposi, A., Tabareau, N.: Setoid type theory—a syntactic translation. In: Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7–9, 2019, Proceedings 13. pp. 155–196. Springer (2019)
7. Altenkirch, T., Kaposi, A.: Type theory in type theory using quotient inductive types. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 18–29. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837638, https://doi.org/10.1145/2837614.2837638
8. 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 (2016), unpublished
9. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda calculus with types. Cambridge University Press (2013)
10. Bauer, A., Komel, A.P.: An extensible equality checking algorithm for dependent type theories. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:17)2022, https://doi.org/10.46298/lmcs-18(1:17)2022
11. Bezem, M., Klop, J., de Vrijer, R., Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2003), https://books.google.fr/books?id=7QQ5u-4tRUkC
12. Blanqui, F., Dowek, G., Grienenberger, É., Hondet, G., Thiré, F.: Some axioms for mathematics. In: Kobayashi, N. (ed.) 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference). LIPIcs, vol. 195, pp. 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.20, https://doi.org/10.4230/LIPIcs.FSCD.2021.20
13. Cartmell, J.: Generalised algebraic theories and contextual categories. Annals of Pure and Applied Logic **32**, 209–243 (1986). https://doi.org/https://doi.org/10.1016/0168-0072(86)90053-9, https://www.sciencedirect.com/science/article/pii/0168007286900539
14. Chen, L.T., Ko, H.S.: A formal treatment of bidirectional typing (2024)
15. Coquand, T.: An analysis of Girard's paradox. Tech. Rep. RR-0531, INRIA (May 1986), https://inria.hal.science/inria-00076023

16. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1-3), 167–177 (1996)
17. Coquand, T.: Canonicity and normalisation for dependent type theory. arXiv preprint arXiv:1810.09367 (2018)
18. Dowek, G.: The undecidability of typability in the lambda-pi-calculus. In: International Conference on Typed Lambda Calculi and Applications. pp. 139–145. Springer (1993)
19. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Computing Surveys (CSUR) **54**(5), 1–38 (2021)
20. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 281–292. POPL '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/964001.964025, https://doi.org/10.1145/964001.964025
21. Felicissimo, T.: Adequate and Computational Encodings in the Logical Framework Dedukti. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.25, https://drops.dagstuhl.de/opus/volltexte/2022/16306
22. Felicissimo, T.: Artifact report: Generic bidirectional typing for dependent type theories (2024)
23. Felicissimo, T.: BiTTs (Jan 2024). https://doi.org/10.5281/zenodo.10500598, https://doi.org/10.5281/zenodo.10500598
24. Felicissimo, T.: Generic bidirectional typing for dependent type theories (Technical Report) (2024), https://inria.hal.science/hal-04270368/file/tech.pdf
25. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
26. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (1993)
27. Harper, R.: An equational logical framework for type theories. arXiv preprint arXiv:2106.01484 (2021)
28. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of functional programming **17**(4-5), 613–673 (2007)
29. Haselwarter, P.G., Bauer, A.: Finitary type theories with and without contexts. arXiv preprint arXiv:2112.00539 (2021)
30. Kaposi, A., Huber, S., Sattler, C.: Gluing for type theory. In: 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
31. Kaposi, A., Kovács, A., Altenkirch, T.: Constructing quotient inductive-inductive types. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290315, https://doi.org/10.1145/3290315
32. Kovács, A.: Staged compilation with two-level type theory. Proceedings of the ACM on Programming Languages **6**(ICFP), 540–569 (2022)
33. Lennon-Bertrand, M.: Complete Bidirectional Typing for the Calculus of Inductive Constructions. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 24:1–24:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.24, https://drops.dagstuhl.de/opus/volltexte/2021/13919

34. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical computer science **192**(1), 3–29 (1998)
35. McBride, C.: Basics of bidirectionalism. https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/ (2018)
36. McBride, C.: Types who say ni. https://github.com/pigworker/TypesWhoSayNi/ (2022)
37. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic (TOCL) **9**(3), 1–49 (2008)
38. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
39. van Oostrom, V.: Normalisation in weakly orthogonal rewriting. In: International Conference on Rewriting Techniques and Applications. pp. 60–74. Springer (1999)
40. Pfenning, F.: Logical frameworks. Handbook of automated reasoning **2**, 1063–1147 (2001)
41. Pierce, B.C., Turner, D.N.: Local type inference. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1), 1–44 (2000)
42. van Raamsdonk, F.: Outermost-fair rewriting. In: International Conference on Typed Lambda Calculi and Applications. pp. 284–299. Springer (1997)
43. Reed, J.: Redundancy elimination for lf. Electronic Notes in Theoretical Computer Science **199**, 89–106 (2008)
44. Sterling, J.: Algebraic type theory and universe hierarchies. arXiv preprint arXiv:1902.08848 (2019)
45. Uemura, T.: Abstract and concrete type theories. Ph.D. thesis, University of Amsterdam (2021)