Foundation-Independent Type Reconstruction

Florian Rabe*

Jacobs University Bremen, Germany

Abstract

MMT is a framework for designing and implementing formal systems in a way that systematically abstracts from theoretical and practical aspects of their type theoretical and logical foundations. Thus, definitions, theorems, and algorithms can be stated independently of the foundation, and language designers can focus on the essentials of a particular foundation and inherit a large scale implementation from MMT at low cost. Going beyond the similarly-motivated approach of meta-logical frameworks, MMT does not even commit to a particular meta-logic—that makes MMT level results harder to obtain but also more general.

We present one such result: a foundation-independent type reconstruction algorithm. It realizes the foundation-independent aspects generically relative to a set of rules that supply the foundation-specific knowledge. Maybe surprisingly, we see that the former covers most of the algorithm, including the most difficult details. Thus, we can easily instantiate our algorithm with rule sets for several important language features including, e.g., dependent function types. Moreover, our design is modular such that we obtain a type reconstruction algorithm for any combination of these features.

1 Introduction and Related Work

1.1 Motivation

We use the phrase *type reconstruction* for the variant of a type checking/type inference algorithm that additionally solves unknowns parts of the checked term. Very well-known examples are

- Type-checking a λ-abstraction like $\lambda x.x + 1 : int \to int$ where the type of the bound variable is omitted.
- Inferring the type of $Cons(1, Nil)$ to be $List[int]$ where the type argument int of $Cons$ and Nil is omitted.

Intuitively, these omitted terms existed in the human user's mind but were omitted when writing the term, and it is now the system's task to reconstruct them from the context.

Type reconstruction is among the most difficult algorithms about formal systems to understand, implement, or document, especially if dependent types are used. At the same time, type reconstruction is critical to make a formal system practical. An implementation without good type reconstruction support often makes it infeasible to conduct case studies that go beyond toy examples.

This gravely harms the development and evaluation of formal systems that are used as programming languages and logics. This bottleneck is particularly painful for experimental systems. It is not unusual at all to find an entire PhD thesis that designs one new formal system, and often implementations that support major case studies are accomplished only years later (if at all). Indeed, in practice we find that only very few, widely used systems have strong support. Therefore, it is desirable to implement type reconstruction generically for many formal systems at once.

1.2 Type Reconstruction in Individual Systems

Unique Reconstruction Our goal is to infer omitted subexpressions such as implicit arguments (e.g., type parameters) and the types (or kinds etc.) of bound variables. These unknown subexpressions can be formally represented as meta-variables that are existentially quantified on the outside of the expression. Then, instead of checking a typing judgment $t : A$, we have to prove $\exists \vec{X}.t(\vec{X}) : A(\vec{X})$, and the witnesses found for the X_i are the solutions for the unknown subexpressions.

*The author was supported by DFG grant RA-18723-1 OAF.

This problem is particularly difficult in expressive type theories where unification is undecidable. These employ sophisticated algorithms to obtain practical solutions. Most of these systems are dependently-typed, and we can roughly distinguish two families. Firstly, systems with universe hierarchy include Coq [Coq15], Matita [ACTZ06], Lean [dMKA+15], and Agda [Nor05]. Secondly, systems without universe hierarchy include Twelf [PS99], Delphin [PS08], and Beluga [PD10]. Abella [Gac08] is similar to the latter family but uses a two-leveled logic.

These algorithms are so complicated that they are often understood by only a few people working closely with the original developers. Their formal description or documentation may be incomplete, outdated, or lacking altogether. For example, [Nor05] describes the algorithm of Agda for a fragment of the language, and [Lut01] describes an algorithm for the calculus of constructions. An (apparently unpublished) description of an algorithm used in Lean (there called *elaboration*) is given in [dMAKR15]. Only recently two major *formal* presentations were accomplished: [ARCT12] for the algorithm implemented in Matita (there called *refinement*), and [Pie13] for the algorithm implemented in Beluga [PD10] (which is representative of the algorithms used in the Twelf family).

The basic idea of these algorithms is to apply a standard bidirectional type checking algorithm to $t(\vec{X}) : A(\vec{X})$. Here, whenever possible, the expected type A is carried along while recursing into the term t. At some point this recurses into equality constraints about the X_i, most importantly when checking the equality between an inferred type and an expected type. Eventually this yields constraints of the form $X_i = t_i$, which are used to solve X_i as t_i.

Non-Unique Reconstruction Type reconstruction is easiest if the existential quantification about $\vec{X}$ is a *unique* existential. This corresponds to the reasonable requirement that the user should only omit subexpressions that can be unambiguously reconstructed by the system. Any ambiguity should be flagged as an error.

Recently type reconstruction algorithms have been extended to allow for non-unique existential quantification. Here the system takes context knowledge into account to guess which out of multiple possible reconstructions the user most likely meant. This feature is usually coupled with a form of non-logical declaration that allows the user to guide this guessing in a predictable way. For example, consider the monoid composition operator $\circ$ of type $\Pi M : Monoid.\, M.univ \to M.univ \to M.univ$ where $M.univ$ is the underlying type of a monoid M. If M is an implicit argument, we cannot uniquely reconstruct it in the expression $1 \circ 1$ because there can be multiple monoids M with $M.univ = int$. However, the user may declare a *unification hint* in the context that tells the system to prefer the additive monoid $(int, +, 0)$ in such a situation.

Such unification hints were first used in [GM08] in Coq and later implemented in Matita in [ARCT09] and Lean [dMAKR15].

1.3 Type Reconstruction in Logical Frameworks

Due to its importance and difficulty, type reconstruction is a natural candidate for logic-*independent* algorithms in a meta-logical framework. This allows implementing the algorithm once at the framework level and then instantiating it for various formal systems. This has been pursued in multiple ways.

Declarative Frameworks Logical frameworks like LF [HHP93], Abella [Gac08], or Isabelle [Pau94] were introduced specifically to exploit the potential of logic-independent solutions. Logic-independent type reconstruction algorithms were not necessarily the primary motivation (LF and Abella are driven by meta-logical reasoning and Isabelle by theorem proving.) but a welcome consequence.

A logical framework fixes a meta-logic F and then encodes the operators, notations, and typing rules of an object logic L in terms of their counterparts of F. Typically encodings can be shallow in the sense that typing judgments in L can be represented as certain typing judgments in F, in which case L can inherit type reconstruction from F.

A similar effect is obtained if F is so expressive that it already subsumes L. In this case, L does not have to be encoded in F but can simply be embedded in it. In that sense, type reconstruction can be inherited from systems like Coq as well, even if they were not designed as logical frameworks.

Computational Frameworks Despite some successes [AHMP92, HST94, KMR09] the logical framework approach has proved insufficient for two reasons. Firstly, formal systems often use novel, experimental, or idiosyncratic features that may or may not admit elegant encodings in logical frameworks, and often the most interesting features are exactly the ones that do not. If the encoding becomes inelegant, the overhead introduced by the framework becomes the bottleneck and may eliminate the gains of logic-independent algorithms. Secondly, designing logical frameworks that allow non-unique reconstruction tends to be much harder, at least for now while the relevant trade-offs are still being investigated.

The first reason is a major driver of the author's MMT framework [RK13], in which the present contribution is developed. MMT fixes only the high-level design of the type reconstruction algorithm but is completely agnostic in the formal system, i.e., does not even fix the logical framework. The low-level structure of the algorithm is supplied as a set of rules that are programmed directly in the underlying programming language.

The second reason drove the development of ELPI [DGCT15], which can be seen as an intermediate between fixing a declarative logical framework and the completely free approach of MMT. It uses λ-Prolog as a meta-logic to allow users to program type reconstruction algorithms declaratively.

Both approaches share the use of an untyped representation language, in which the syntax of the object logic is embedded, and a Turing-complete programming language, in which the type reconstruction rules of the object logic are formulated. The advantage of ELPI is that the framework exerts more control over what the user is allowed to do and avoids the awkwardness of requiring users to program. The advantage of MMT is that it gives users more flexibility and allows the use of existing programming language IDEs.

The type reconstruction algorithms of MMT and ELPI have been developed at roughly the same time for similar reasons. Over the next few years, it will be of great interest to observe how these two approaches develop and what lessons can be learned for future trade-offs.

1.4 Our Approach

Type Reconstruction in MMT Our approach is similar to the use of a logical framework. Like logical frameworks, MMT represents the operators, notations, and rules of an object logic L as declarations in an MMT theory for L. But the rules are not spelled out declaratively. Instead, the rule declarations just point to objects in the underlying programming language of MMT.

MMT does not fix the type system, and neither does our type reconstruction algorithm. Instead, it is parametrized by a set of rules. When checking a judgment, the set of available rules is collected from the context.

This gives us more freedom: the rules can use arbitrary case distinctions, side conditions, auxiliary computations, or even state and I/O. They can also provide more informative log messages (for debugging rules) and error messages (for users debugging their formalizations).

A drawback is that it becomes harder to supply inference rules in practice: rules can no longer be specified declaratively but must be programmed. But our implementation mitigates the pain of programming rules by offering simple plugin interfaces. Most importantly, rules can be compiled separately, i.e., changing a rule does not require recompiling the MMT system itself. The programmed rules can also be stored and maintained along-side normal MMT content, and MMT can compile and load them dynamically and automatically.

On the positive side, because rules are reified, they can themselves be the result of arbitrary computations. For example, the author's case studies now routinely use

- parametric rules that are instantiated differently for different object logics,
- rules that are automatically generated from declarative descriptions written by the user.

Finally, another drawback is that our foundation-independent algorithm is not as fast as foundation-specific ones. This is not surprising because the algorithm must check at every step which rules are applicable. It remains future work to investigate whether specific fixed sets of rules can be compiled in a way that allows for being competitive in terms of speed.

A Meta-Meta-Logical Framework The most appealing use of MMT is to use it as a framework for implementing logical frameworks. Then we might call it a meta-meta-logical framework, which gave rise to the abbreviation MMT (with T ambiguously referring to the theory and the tool).

This use of MMT combines two advantages. Firstly, it allows full flexibility to design logical frameworks and makes it fast to experiment with and implement them. Secondly, once we have at least one logical framework, we can use it to declare object logics declaratively.

In this scenario, the difficult part of programming individual rules is primarily carried out by the logical framework developer. Individual users only have to choose an appropriate framework and can then define their object logic declaratively.

Contribution The main challenge for our approach is to design an interface layer that at the same time

- fully abstracts from individual foundations,
- has enough structure to admit meaningful foundation-independent definitions, theorems, and algorithms.

Most critically, by allowing an arbitrary set of rules, our type reconstruction algorithm must work with an open-world assumption: at no point do we know what meaningful terms are around and what their meanings are. This open-world assumption affects the design in interesting ways, making some aspects more and others less difficult. For example, in MMT it becomes very natural to use meta-variables both for terms and for types because it does not distinguish between terms and types anyway. This is in contrast with, e.g., Twelf [PS99] and [Pie13], which must avoid type variables. As a negative example, it becomes difficult to conclude fast that two terms cannot be equal (which is often the critical step that triggers backtracking); this is because MMT cannot predict what the canonical forms are (if any exists) and what equality rules might become applicable later.

We give a type reconstruction algorithm at the MMT level that can be instantiated with a wide range of formal systems by supplying appropriate sets of rules. Moreover, because rules are taken from the current context, the MMT module system [RK13] can be used to design formal systems and their type reconstruction algorithms modularly.

Our algorithm achieves a very good separation of concerns. The fixed foundation-independent part of the algorithm handles all the bureaucracy of type reconstruction such as applying structural rules, maintaining and solving the meta-variables, error reporting, timeouts (we allow for undecidable typing), or backtracking. The flexible foundation-specific rules can focus on the foundational details. In particular, to a formal system designer, programming rules in MMT feels as natural and easy as defining a standalone type *checking* algorithm. The complexity of type *reconstruction* and other features such as the module system remains hidden.

This separation of concerns has the additional advantage that our foundation-independent algorithm may in fact be easier to understand than existing foundation-specific ones.

Evaluation We have performed multiple case studies that define formal systems in MMT. Most of them are driven by the desire to design more expressive extensions of LF modularly.

In this paper, we present three simple examples of such modular features. Firstly, we define LF [HHP93] itself, i.e., a dependent type theory with function types. This requires only the straightforward implementation of 9 simple rules. Secondly, we give product types in analogy to function types. Incidentally, this analogy reveals an elegant symmetry between typing rules that appears not to have been observed before and that provides a valuable guide for designing future formal systems. Thirdly, we extend LF with shallow polymorphism by adding only a single rule.

We have also conducted a few more advanced case studies. We will only describe these briefly in Sect. 6.4 because they would require a considerably longer paper. They are nevertheless noteworthy because they constitute younger or more experimental formal systems for which there was no prior type reconstruction support.

Finally, as part of his ongoing PhD thesis, Dennis Müller[1] has implemented several more formal systems in this style. This includes predicate subtypes, record types, and homotopy type theory. Even though these case studies have not been prepared for publication yet, this is remarkable for two reasons. Firstly, our type reconstruction algorithm enabled a first year PhD student with no dedicated background in formal systems to produce new implementations of formal systems on a daily basis. Secondly, he does so not as his main research objective but as a necessary requirement—he now implements new formal systems routinely as tools for his primary research.

1.5 Foundation-Independent Solutions

Orthogonally to the discussion so far, it is interesting to relate our results to other solutions that make no commitment to a particular logical foundation and can be instantiated for arbitrary formal systems. Indeed, this paper presents only a small step in a larger research agenda of designing and implementing as many aspects of formal systems as possible foundation-independently. The author developed MMT as the theoretical and practical framework for this project.

At small scales, designing and implementing a new formal system is quite simple. It usually consists of a grammar, an inference system, and a kernel implementing them. However, to be practical, we have to

[1] University of Erlangen-Nuremberg, Germany, advised by Michael Kohlhase

supplement this with a number of advanced features: a human-friendly concrete syntax with user-defined notations, a type reconstruction algorithm, a module system that enables reuse, a smart user interface, and a theorem prover that discharges proof obligations. These features have two things in common: they are essential for practical applications at large scales, and they require orders of magnitude more work than the initial implementation.

This impedes evolution: problems in the initial design may become apparent only in large case studies, which can only be carried out after a major investment into advanced features. Moreover, existing advanced implementations can often not be reused for new languages: as these implementations evolve and acquire users, they tend to become locked to a certain language so that adding a new language feature often requires a reimplementation.

MMT's long-term objective is to investigate the hypothesis that (i) the basic design (e.g., grammar and rules) of formal systems are highly specific, but (ii) many advanced features can be realized generically. This motivates a separation of concerns where small scale definitions of logics are combined with generic large-scale implementations.

MMT has previously been used to validate this hypothesis for several features including module system [RK13], querying [Rab12], change management [IR12], and user interface [Rab14a]. Similar positive results have been obtained using other frameworks including substitution-based search [KŞ06], machine learning–based premise selection for theorem proving [KU15], or the integration of automated proof tools in interactive provers [MP08].

1.6 Overview

It is very difficult to give instructive, pleasant-to-read descriptions of type reconstruction algorithms, and our presentation is the result of multiple rewrites. The resulting description is remarkable for being simple enough to understand easily, formal enough to reason about, and algorithmic enough to implement directly.

We present the MMT language in Sect. 2 and 3. Then we describe the type reconstruction algorithm in Sect. 4 and 5 and instantiate it with example systems in Sect. 6. In Sect. 7, we describe how our implementation of type reconstruction interacts with other parts of the MMT system such as parsing and user interface. In Sect. 8, we conclude and discuss future work.

The implementation of the MMT system including our algorithm and all our rules are available at `https://uniformal.github.io/`.

2 Syntax

Theory	Σ	$::=$	$\cdot \mid \Sigma,\ c[:E][=E][\#N]$
Expression	E	$::=$	$c \mid c(\Sigma; E^*)$
Notation	N	$::=$	$(\mathcal{V}_n \mid \mathcal{A}_n \mid \mathtt{string})^*$

Figure 1: MMT Grammar

In this section, we summarize the syntax of the small fragment of MMT that is needed for our purposes. In particular, we omit the module system, which is entirely orthogonal to type reconstruction. The grammar is given in Fig. 1, and the intuitions are explained in the remainder of this section.

Notations are used by the MMT system for parsing and presentation only. They are not relevant for our purposes here except that we will heavily use them in the examples.

Theories and Constants A **theory** Σ is a list of constant declarations. We use commas to separate the elements of such a list and to concatenate lists, and $\cdot$ denotes the empty list. Each constant may occur in the subsequent declarations.

A **constant** declaration is of the form $c[: A][= t][\#N]$ where c is an identifier; A is its **type**, t its **definiens**, and N its **notation**, all of which are optional. For example, $c : A = t$ introduces c as an abbreviation of the expression t of type A, whereas $c : A$ introduces a fresh expression of type A. Declarations with neither type nor definiens are used to get off the ground because no single constant in built into MMT.

Both A and t (if present) are arbitrary expressions not subject to any built-in type system. The only structural requirement is that each expression uses only previously declared constants.

To reduce the number of case distinctions, we will occasionally write the type or definiens of c as

- $\perp$ to emphasize that the type/definiens is absent, or
- _ to state that the type/definiens is irrelevant and may or may not be present.

Contexts and Variables Formal systems often distinguish between theories and contexts. Both are lists of declarations. But theories declare constants, which are global identifiers with an arbitrary but fixed meaning, whereas contexts declare variables, which are local identifiers with a flexible but unknown value.

For our purposes, the distinction is not essential, and we formally merge the two concepts in order to simplify the language. Nonetheless, to support our intuitions, we use the word "variable" and write x instead of c whenever an identifier is considered local. Similarly, we say "context" instead of theory and write Γ instead of Σ for a list of variable declarations.

Expressions Relative to a theory, we form **expressions** inductively. MMT uses an extremely simple but expressive abstract syntax for expressions, which minimizes the number of case distinctions: expressions are constants/variables c and complex expressions $c(\Gamma; E_1, \ldots, E_n)$. As we see in the examples below, the complex expressions subsume most relevant expression-formers including binders and operators.

In $c(\Gamma; E_1, \ldots, E_n)$, we call c the **constructor**, Γ the list of **bound variables**, and the E_i the **arguments**. E_1 (if present) is called the **head** of the expression.

The variables declared in Γ are **bound** in all subsequent variable declarations and in the arguments E_i. The usual definition of α-equality and capture-avoiding **substitution** can be generalized easily. If E is an expression using the free variables $x_1, \ldots, x_n$ and $\gamma = t_1, \ldots, t_n$ is a list of terms, we write $E[\gamma]$ for the result of substituting each x_i with t_i.

Both Γ and the argument list may be empty. The special case $\Gamma = \cdot$ yields the case of n-ary non-binding operators. The special case where the length of Γ is 1 and $n = 1$ yields the usual binders such as λ, which bind one variable and take one argument.

Notations If a constant c has a notation N, then N is used for the concrete representation of complex expressions with constructor c. Notations are lists of three kinds of objects: $\mathcal{V}_n$ refers to the declaration of the n-th bound variable, $\mathcal{A}_n$ to the n-th argument, and arbitrary strings provide delimiters between and around them.

Notations are irrelevant for the essentials of our type reconstruction algorithm. We use notations for two reasons only:

- Notations are used in examples.
- Notations are a convenient way to tell the parser, which subexpressions will be omitted by the human user and should be reconstructed. We discuss this in the next paragraph.

We allow two kinds of omitted expressions that have to filled in by type reconstruction. Firstly, if the notation contains $\mathcal{V}_i$ but the concrete syntax only has an identifier x, the parser inserts a meta-variable for the **omitted type**. Secondly, if a notation mentions, e.g., argument $\mathcal{A}_2$ but not $\mathcal{A}_1$, then $\mathcal{A}_1$ is deemed an **implicit argument**. If an implicit argument is not part of the concrete syntax, the parser inserts a fresh meta-variable for it.

Note that MMT does distinguish bound variables and meta-variables syntactically. Instead, we distinguish them by carrying around two different contexts: one for meta-variables and one for bound variables.

Representing Languages as MMT Theories The syntax of MMT is generic in the sense that MMT has no built-in constants c. Therefore, to form any expressions at all, we first have to declare some constants for the primitive constructors of the desired language:

> *Example* 2.1 (λ-Calculus as an MMT Theory). The upper half of Fig. 2 gives a simple MMT theory for the dependently-typed λ-calculus LF [HHP93]. It declares one constant for each primitive constructor and a notation for it.
>
> The table in the lower half exemplifies the notations. `lambda` constructs λ-abstractions: in the expression $\texttt{lambda}(x : A; t)$, `lambda` is the constructor, $x : A$ the single variable binding, and t the single argument. The notation declares $[x : A]t$ as its concrete representation.

`type`	#	`type`
`kind`	#	`kind`
`Pi`	#	$\{\,\mathcal{V}_1\,\}\,\mathcal{A}_1$
`lambda`	#	$[\,\mathcal{V}_1\,]\,\mathcal{A}_1$
`apply`	#	$\mathcal{A}_1\ \ \mathcal{A}_2$
`arrow`	#	$\mathcal{A}_1 \to \mathcal{A}_2$

Constructor	Abstract syntax	Concrete syntax
formation	$\texttt{Pi}(x : A; B)$	$\{x : A\}B$
introduction	$\texttt{lambda}(x : A; t)$	$[x : A]t$
application	$\texttt{apply}(\cdot; f\,t)$	$f\,t$

Figure 2: A Theory for LF in MMT

`tp`	$: \texttt{type}$	$\#\ \texttt{tp}$
`tm`	$: \texttt{tp} \to \texttt{type}$	$\#\ \texttt{tm}\,\mathcal{A}_1$
`Sigma`	$: \{A\}(\texttt{tm}\,A \to \texttt{tp}) \to \texttt{tp}$	$\#\ \Sigma\,\mathcal{A}_2$
`pair`	$: \{A, B\}\{a : \texttt{tm}\,A\}\texttt{tm}\,(B\,a) \to \texttt{tm}\,(\Sigma[x]B\,x)$	$\#\ (\mathcal{A}_3, \mathcal{A}_4)$
`pi1`	$: \{A, B\}\texttt{tm}\,(\Sigma[x : \texttt{tm}\,A]B\,x) \to \texttt{tm}\,A$	$\#\ \texttt{pi1}\,\mathcal{A}_3$
`pi2`	$: \{A, B\}\{t : \texttt{tm}\,(\Sigma[x : \texttt{tm}\,A]B\,x)\}\texttt{tm}\,(B\,(\texttt{pi1}\,t))$	$\#\ \texttt{pi2}\,\mathcal{A}_3$

Figure 3: A Theory for Product Types in MMT/LF

`apply` constructs function applications: in the expression $\texttt{apply}(\cdot; f, t)$, `apply` is the constructor, no variables are bound, and f and t are the arguments. f is the head, which corresponds to the usual definition of head. The notations declares $f\ t$ as its concrete representation.

In addition to the function type constructor `Pi`, we declare `arrow` as a separate symbol in order to attach a different notation to it. It is not possible to formally declare `arrow` as an abbreviation for a `Pi` expression because we do not have any syntactic material with which to state that property. That is understandable because LF is our most primitive MMT theory that we use to get off the ground. Instead, we have to declare a rule later on that transforms `arrow`-expressions into `Pi`-expressions.

Now we can represent specific LF-theories as extensions of the theory for LF:

Example 2.2 (Sigma Types as an MMT/LF Theory). Fig. 3 gives a straightforward intrinsically typed encoding of Σ-types in LF. These declarations can now be typed: we use the constants previously declared in LF to form the types.

Firstly, `tp` and `tm` introduce the basics of an intrinsic encoding of a typed object language inside LF. Expressions $A : \texttt{tp}$ represent object language types, and expressions $t : \texttt{tm}\,A$ represent object language terms of object language type A. (The abstract syntax of $\texttt{tm}\,A$ is $\texttt{apply}(\cdot; \texttt{tm}, A)$.)

Secondly, we introduce `Sigma` as an object language type operator that forms Σ-types with introductory form `pair` and two elimination forms `pi1` and `pi2`. This is a well-known logic definition using LF as a logical framework.

This example already uses omitted types and implicit arguments that must be reconstructed later. All reconstructible variable types are omitted, and the object language type arguments A and B are implicit in `Sigma`, `pair`, `pi1`, and `pi2`.

3 Inference System

3.1 Judgments

MMT uses the **judgments** given in Fig. 4. The primary judgment is that for valid theories, which includes the well-typedness of every declaration.

Expressions are subject to the **typing** judgment $E : E'$ and the **equality** judgment $E \equiv E'$ relative to a theory Σ. We do not fix a universe hierarchy or other language-specific aspects of the type system, i.e.,

$\vdash \Sigma$	Σ is a valid theory
$\Sigma \vdash t : A$ $\Sigma \vdash E \equiv E'$ $\Sigma \vdash A\,\mathtt{inh}$	t has type A E and E' are equal A is inhabitable

Figure 4: Judgments

typing is simply a binary relation between expressions, and expression uniformly represent terms, types, kinds, universes, etc.

Similarly, we do not fix the details of the equality judgment such as decidability or the existence of canonical forms. It is also just a binary relation between expressions.

Finally, we use one unusual judgment: The unary judgment $\Sigma \vdash A\,\mathtt{inh}$ on expressions, which we call **inhabitability**. Its intended meaning is to specify those expressions A that may occur on the right-hand side of the typing judgment. In particular, a declaration $c : A$ is well-typed only if $\Sigma \vdash A\,\mathtt{inh}$. We will get back to this in Sect. 3.2 when giving the rules for valid theories.

Abbreviations We will often fix a theory Σ and then only consider theories Σ, Γ that extend Σ. In that case, we may want to think of Γ as the context, and it can be convenient to make that explicit in the notation:

Notation 3.1 (Contexts). For any of the judgments about expressions, we abbreviate

$$\Gamma \vdash_\Sigma J \qquad \text{for} \qquad \vdash_{\Sigma,\Gamma} J$$

As usual, we will omit the antecedent Γ if it is the empty context. Moreover, we may omit the theory Σ if it is fixed in the surrounding text.

3.2 Rules

MMT fixes only a small number of rules. We think of them as structural rules because they do not mention any specific constant and thus apply to arbitrary theories.

The **rules for theories** are given in Fig. 5. Theories are valid if each declaration $c[: A][= t]$ is valid relative to the ones preceding it. If a type A is provided, it must be inhabitable. If additionally a definiens t is provided, t must be typed by A. If only a definiens t is provided but no type, t must simply be typed by *some* expression A.

The lookup rules formalize how to *use* a declaration $c : A = t$: The constant c becomes a well-formed expression of type A and equal to t. For simplicity, we allow declarations to shadow previous declarations, in which lookup retrieves the right-most one.

The **rules for expressions** only formalize two fundamental principles: α-renaming of bound variables and congruence of equality. The rules given in Fig. 6.

valid theories:

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \qquad [\vdash_\Sigma A\,\mathtt{inh}] \qquad [\vdash_\Sigma t : A]}{\vdash \Sigma,\ c[: A][= t]}$$

lookup in theories:

$$\frac{\vdash \Sigma \qquad c : A[= \_]\text{ in }\Sigma \qquad (*)}{\vdash_\Sigma c : A} \qquad \frac{\vdash \Sigma \qquad c[: \_] = t\text{ in }\Sigma \qquad (*)}{\vdash_\Sigma c \equiv t}$$

$(*)$ right-most declaration if multiple declarations for c in Σ

Figure 5: Rules for Theories

α-conversion:

$$\frac{A'_i = A_i[x'_1, \ldots, x'_{i-1}] \qquad E'_i = E_i[x'_1, \ldots, x'_n]}{\Gamma \vdash_\Sigma c(\overrightarrow{x_m : A_m}; \overrightarrow{E_n}) \equiv c(\overrightarrow{x'_m : A'_m}; \overrightarrow{E'_n})}$$

equality is equivalence:

$$\frac{}{\Gamma \vdash_\Sigma E \equiv E} \qquad \frac{\Gamma \vdash_\Sigma E \equiv E'}{\Gamma \vdash_\Sigma E' \equiv E}$$

$$\frac{\Gamma \vdash_\Sigma E \equiv E' \qquad \Gamma \vdash_\Sigma E' \equiv E''}{\Gamma \vdash_\Sigma E' \equiv E''}$$

equality is congruence:

$$\frac{\Gamma, \overrightarrow{x_{i-1} : A_{i-1}} \vdash_\Sigma A_i \equiv A'_i \qquad \Gamma, \overrightarrow{x_m : A_m} \vdash_\Sigma E_i \equiv E'_i}{\Gamma \vdash_\Sigma c(\overrightarrow{x_m : A_m}; \overrightarrow{E_n}) \equiv c(\overrightarrow{x_m : A'_m}; \overrightarrow{E'_n})}$$

$$\frac{\Gamma \vdash_\Sigma t : A \qquad \Gamma \vdash_\Sigma t \equiv t' \qquad \Gamma \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma t' : A'}$$

$$\frac{\Gamma \vdash_\Sigma A\ \mathtt{inh} \qquad \Gamma \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma A'\ \mathtt{inh}}$$

Notations: $\overrightarrow{x_m : A_m} = x_1 : A_1, \ldots, x_m : A_m$
$\overrightarrow{E_n} = E_1, \ldots, E_n$

Figure 6: Rules for Equality

These fixed rules say nothing language-specific. For example, there are no rules that determine the type of a complex expression $c(\Gamma; E_1, \ldots, E_n)$. These rules must be provided explicitly when declaring c. More precisely, we define:

Definition 3.2 (Logical Framework)**.** A **logical framework** consists of an MMT theory T and a set $\mathcal{R}$ of inference rules for the three judgments on expressions (i.e., typing, equality, and inhabitability). The rules may use side-conditions and auxiliary judgments but may not mention any constants other than those declared in T.

A logical framework instantiates MMT with a specific syntax (the theory T) and semantics (the rules in $\mathcal{R}$). Once a logical framework is fixed, MMT understands the semantics of any theory that extends T.

Example 3.3 (Rules of LF). We extend Ex. 2.1 with the following rules where U ranges over $\{\mathtt{type}, \mathtt{kind}\}$.

The left rule below makes `type` a `kind`. The right rule tells MMT which expressions are inhabitable: A is inhabitable if it is a type or a kind. This has the effect that LF theories may declare typed constants and kinded constants.

$$\frac{}{\vdash_{\mathtt{LF}} \mathtt{type} : \mathtt{kind}}\mathtt{type} \qquad \frac{\Gamma \vdash_{\mathtt{LF}} A : U}{\Gamma \vdash_{\mathtt{LF}} A\,\mathtt{inh}}\mathtt{univ}$$

Moreover, we add the well-known typing rules for dependent function types

$$\frac{\Gamma \vdash_{\mathtt{LF}} A : \mathtt{type} \qquad \Gamma, x : A \vdash_{\mathtt{LF}} B : U}{\Gamma \vdash_{\mathtt{LF}} \{x : A\}B : U}\mathtt{Pi}$$

$$\frac{\Gamma \vdash_{\mathtt{LF}} \{x : A\}B : U \qquad \Gamma, x : A \vdash_{\mathtt{LF}} t : B}{\Gamma \vdash_{\mathtt{LF}} [x : A]t : \{x : A\}B}\mathtt{lambda}$$

$$\frac{\Gamma \vdash_{\mathtt{LF}} f : \{x : A\}B \qquad \Gamma \vdash_{\mathtt{LF}} t : A}{\Gamma \vdash_{\mathtt{LF}} f\,t : B[t]}\mathtt{apply}$$

which correspond to the rules $(*, *)$ and $(*, \Box)$ of pure type systems [Ber90]. Note that these ensure that only typed but no kinded variables may be bound by `lambda` and `Pi`.

In addition to the equality rules from Fig. 6, we use the well-known rules for β and η-equality:

$$\frac{\Gamma \vdash_{\mathtt{LF}} a : A}{\Gamma \vdash_{\mathtt{LF}} ([x : A]t)\,a \equiv t[a]}\mathtt{beta} \qquad \frac{\Gamma \vdash_{\mathtt{LF}} f : \{x : A\}B}{\Gamma \vdash_{\mathtt{LF}} f \equiv [x : A]f\,x}\mathtt{eta}$$

MMT poses only very few constraints on the set $\mathcal{R}$. In particular, we do not specify the details of what an inference rule is. However, two meta-properties are important for our purposes:

Definition 3.4. A logical framework has **monotonicity** if the following rule is admissible

$$\frac{\vdash_\Sigma J \qquad \vdash \Sigma' \qquad \Sigma' > \Sigma}{\vdash_{\Sigma'} J}$$

where $\Sigma' > \Sigma$ means that Σ' arises from Σ by adding

- entire declarations
- types or definiens that are not yet present in Σ.

Monotonicity is also called weakening. It expresses that derivations cannot be invalidated by adding assumptions. Due to the structure of MMT's lookup rules, monotonicity holds almost automatically. But it is technically possible to break monotonicity by using very specific rules in $\mathcal{R}$, e.g., by using a premise $c : \_ = \bot$, which becomes false if a definiens is added to c.

Monotonicity is a special case of the preservation of judgments along theory morphisms—a fundamental invariant of the MMT language and system (see [Rab14b] for details). Therefore, instantiating MMT with non-monotonic languages is generally not practical.

Definition 3.5. A logical framework has **type unicity** if the following rule is admissible

$$\frac{\Gamma \vdash_\Sigma t : A \qquad \Gamma \vdash_\Sigma t : A'}{\Gamma \vdash_\Sigma A \equiv A'}$$

Type unicity is a very strong condition: it precludes any form of subtyping. MMT does not require this condition at all. Even the type reconstruction algorithm we present here has been used successfully for some languages with non-unique types, and we are currently extending it to systematically support subtyping.

However, we have so far worked out the correctness of the algorithm only for languages with unique types. We expect our algorithm to be correct in more general settings, but such a treatment would go beyond the scope of this paper.

We conclude this section with a few optional remarks that may be helpful to some readers:

Remark 3.6 (Substitution Preserves Judgments). Consider an arbitrary logical framework $(T, \mathcal{R})$. Under some reasonable and easy-to-establish assumptions about the rules of $\mathcal{R}$, we can prove that substitution preserves all judgments. The details can be found in [Rab14b].

Remark 3.7 (Substructural Rules). The lookup rules imply that weakening and exchange are admissible rules, i.e., MMT cannot represent substructural frameworks directly. This is an intentional trade-off to allow for more substantial results at the MMT level: because MMT is designed for building large modular libraries of interrelated theories, a substructural version of MMT would be very difficult to use.

Remark 3.8 (Definition Expansion). The lookup rule for the definiens implies that constants are always equal to their definiens. The presence of this rule only requires that definition expansion is always legal. It does not imply that the implementation always expands definitions. Indeed, it is usually desirable to delay definition expansion as much as possible.

Remark 3.9 (Subject Reduction). The congruence rule for typing implies subject reduction: If $t : A$ and $t \equiv t'$, then $t' : A$. Thus, MMT can only represent languages for which subject reduction is admissible.

Subject reduction is violated by some complex languages. But we require it to make intuitive equality reasoning sound: equal expressions can always be substituted for each other.

Remark 3.10 (The ξ-Rule). The congruence rule for complex terms permits equality conversion under a binder. Specialized to `lambda`, this is usually called the ξ-rule: $\vdash [x : A]t \equiv [x : A]t'$ if $x : A \vdash t \equiv t'$. ξ is sometimes rejected, but we argue that its nature as a congruence rule justifies assuming it.

Recall that in the presence of β, the rules η and ξ together are equivalent to extensionality. We hold that if extensionality is to be avoided, one should reject η but not ξ.

4 Preparations

Type reconstruction is so complex that a direct presentation of an algorithm quickly becomes unreadable. Therefore, we separate the presentation into parts:

- Sect. 4.1 specifies the overall problem and introduces five subalgorithms that are the main components of the overall algorithm.
- Sect. 4.2 gives the key ideas behind our algorithm.
- Sect. 4.3 introduce a novel kind of inference rule that is critical to present our algorithm declaratively.
- Then Sect. 5 describes the details of the main algorithm and the five subalgorithms.

4.1 Problem Statement

Our goal is to effectively check the validity of an MMT judgment relative to a fixed logical framework $(\Sigma, \mathcal{R})$. We do not need to worry about the judgment $\vdash \Sigma$ for valid theories: the rules in Fig. 5 define a straightforward algorithm for checking it (if we have algorithms for the other three judgments).

Thus, the problem consists of checking the three expression judgments $\Gamma \vdash_\Sigma J$ (for typing, equality, and inhabitability). As usual, we do this using a set of mutually recursive algorithms. These are listed in Fig. 7 and explained in the sequel.

Σ will remain fixed throughout this section.

Algorithm	Notation	Intuition
Checking		
type checking	$\Gamma \vdash_\Sigma t : A$	check $\Gamma \vdash_\Sigma t : A$
equality checking	$\Gamma \vdash_\Sigma t \equiv t' : A$	check $\Gamma \vdash_\Sigma t \equiv t'$
inhabitability checking	$\Gamma \vdash_\Sigma A\, \mathtt{inh}$	check $\Gamma \vdash_\Sigma A\, \mathtt{inh}$
Querying		
type inference	$\Gamma \vdash_\Sigma t \overset{\rightsquigarrow}{:} A$	compute A such that $\Gamma \vdash_\Sigma t : A$
simplification	$\Gamma \vdash_\Sigma t \overset{\rightsquigarrow}{\equiv} t'$	simplify t to t' such that $\Gamma \vdash_\Sigma t \equiv t'$

Figure 7: Mutually recursive algorithms

Overview of Subproblems We split the typing and equality judgment into two variants each: These differ in whether the second expression is given or to be found. Thus, we have to provide three checking algorithms for inhabitability, typing, and equality and two query algorithms for typing and equality.

We write $\Gamma \vdash_\Sigma t : A$ for the problem where t and A are given and a boolean is to be returned depending on whether $\Gamma \vdash_\Sigma t : A$ is valid. Correspondingly, we write $\Gamma \vdash_\Sigma t \equiv t' : A$ and $\Gamma \vdash E\, \mathtt{inh}$. Here the equality judgment takes an optional type A; if present, it helps direct the algorithm. We speak of **type checking**, **equality checking**, and **inhabitability checking**. When referring to all three, we simply speak of checking judgments.

We write $\Gamma \vdash_\Sigma t \overset{\rightsquigarrow}{:} A$ for the problem where t is given and A is to be returned such that $\Gamma \vdash_\Sigma t : A$ is valid. We speak of **type inference**. Correspondingly, we write $\Gamma \vdash_\Sigma t \overset{\rightsquigarrow}{\equiv} t'$ for the problem where t is given and an equal expression t' is to be returned. We call this **simplification** and it is meant to subsume any kind of definition expansion, computation, or rewriting. When referring to both, we speak of query judgments.

The result of simplification is usually not unique, but all correct results are equal due to symmetry and transitivity. The result of type inference is usually not unique either, but all correct results are equal because we assume unicity of types. Frameworks without unicity of types would require an additional subtyping judgment; MMT can be generalized in this way, but we omit that here.

Goal If all input expressions are fully known, the above 5 algorithms are usually very simple. Often all parts are decidable so that the algorithms always terminate. This is the case, e.g., for the languages in the λ-cube such as LF.

The type *reconstruction* problem is much harder and usually undecidable. Concretely, we consider the judgment $U, \Gamma \vdash_\Sigma J$, where the variables in U are the meta-variables that represent unknown subexpression. We will call the meta-variables *unknowns* in the sequel. Our goal is to find the unique solution to the unknowns that makes the judgment valid (and to prove validity). In our implementation, our type reconstruction algorithms additionally report an error if no solution exists (in which case the input is ill-typed) or multiple solutions exist (in which case the input is ambiguous).

It does not matter where the unknowns originate, but it is worth recalling that they usually represent

- omitted types of bound variables,
- omitted arguments that can be inferred from the types of other arguments (so-called implicit arguments),
- explicitly omitted subexpressions.

Formal Statement In the simplest case, U remains fixed throughout the algorithm. Then our goal is simply to add definitions to all unknowns in U. However, occasionally, it is convenient to add auxiliary unknowns to U during type reconstruction. Therefore, we introduce the following definitions:

Definition 4.1. For two contexts, we write $u > U$ if for every U-variable x, there is a u-variable x and if x has a type/definiens in U, it has the same type/definiens in u.

We write $u \gtrdot U$ if $u > U$ and every u-variable has a definiens.

We think of $u > U$ as a refinement of U: u subsumes all U-declarations but may add a type/definiens to them and may add new variables entirely. We think of $u \gtrdot U$ as a total refinement: all U-variables are assigned concrete values in u. Essentially our problem is to find $u \gtrdot U$ such that $u, \Gamma \vdash_\Sigma J$.

We allow u to declare more variables than U so that we can introduce auxiliary unknowns during type reconstruction. When we have found u, we still have to eliminate those auxiliary variables:

Definition 4.2. Consider a context $u \triangleright U$. Let u' arise from u by exhaustively replacing every occurrence of a u-variable with its definiens. We write $\overline{u}$ for the substitution that maps every U-variable to its u'-definiens

Then we can formulate our problem as finding $u \triangleright U$ such that $\Gamma[\overline{u}] \vdash_\Sigma J[\overline{u}]$. Of course, allowing auxiliary variables in u means that u will never be unique. Moreover, in any case, u can only be unique up to the equality judgment. Therefore, we define:

Definition 4.3. Consider contexts $u \triangleright U$ and $u' \triangleright U$. We write $u \equiv u'$ if $\Gamma[\overline{u}] \vdash_\Sigma x[\overline{u}] \equiv x[\overline{u'}]$ for every U-variable x.

Thus, we need to find $u \triangleright U$ that is unique up to $\equiv$ and satisfies $\Gamma[\overline{u}] \vdash_\Sigma J[\overline{u}]$.

Remark 4.4 (Unknowns with Free Variables). Our problem statement assumes that the solutions to the U are closed expressions: The solutions to unknowns may depend on Σ (which is globally fixed) but not on Γ (which contains the bound variables).

An elegant way to allow unknowns with free variables, is to have every meta-variable carry a context. A drawback of this approach is that we need explicit substitutions are necessary to delay substitutions in meta-variables. That is not a bad thing per se but would considerably complicate the algorithm and the presentation.

We simply assume that a logical framework with function types is used. In that case, we can mimic unknowns with free variables by using functions: We declare an unknown X and use, e.g., $X\ x\ y$ for an unknown expression in which x and y from Γ may occur free. Then unknowns are closed expressions and can be ignored during substitution.

Example Recall the example from Fig. 3. After successful type reconstruction, we expect the theory from Fig. 8 with all reconstructed subexpressions filled in.

$$
\begin{array}{lll}
\mathtt{tp} & : \mathtt{type} & \#\ \mathtt{tp} \\
\mathtt{tm} & : \mathtt{tp} \to \mathtt{type} & \#\ \mathtt{tm}\,\mathcal{A}_1 \\
\mathtt{Sigma} & : \{A : \mathtt{tp}\}(\mathtt{tm}\,A \to \mathtt{tp}) \to \mathtt{tp} & \#\ \Sigma\,\mathcal{A}_2 \\
\mathtt{pair} & : \{A : \mathtt{tp}, B : \mathtt{tm}\,A \to \mathtt{tp}\}\{a : \mathtt{tm}\,A\}\mathtt{tm}\,(B\,a) \to \mathtt{tm}\,(\Sigma\,A\,[x : \mathtt{tm}\,A]B\,x) & \#\ (\mathcal{A}_3, \mathcal{A}_4) \\
\mathtt{pi1} & : \{A : \mathtt{tp}, B : \mathtt{tm}\,A \to \mathtt{tp}\}\mathtt{tm}\,(\Sigma\,A\,[x : \mathtt{tm}\,A]B\,x) \to \mathtt{tm}\,A & \#\ \mathtt{pi1}\,\mathcal{A}_3 \\
\mathtt{pi2} & : \{A : \mathtt{tp}, B : \mathtt{tm}\,A \to \mathtt{tp}\}\{t : \mathtt{tm}\,(\Sigma\,A\,[x : \mathtt{tm}\,A]B\,x)\}\mathtt{tm}\,(B\,(\mathtt{pi1}\,A\,B\,t)) & \#\ \mathtt{pi2}\,\mathcal{A}_3
\end{array}
$$

Figure 8: Product Types after Type Reconstruction

To check whether this theory is valid, we have to check inhabitability of each declaration's type. Each time we start a new instance of the type reconstruction algorithm.

As an example, we consider the declaration of `pi1`. When it is checked, all preceding declarations already look like the ones in Fig. 8. After inserting meta-variables for the unknown subexpressions, the declaration is

$$\mathtt{pi1} : \{A : ?_1, B : ?_2\,A\}\mathtt{tm}\,(\Sigma\,?_3\,A\,B\,[x : \mathtt{tm}\,A]B\,x) \to \mathtt{tm}\,A$$

where the $?_i$ are the unknowns.

So we call type reconstruction on the judgment

$$\vdash \{A : ?_1, B : ?_2\,A\}\mathtt{tm}\,(\Sigma\,?_3\,A\,B\,[x : \mathtt{tm}\,A]B\,x) \to \mathtt{tm}\,A\ \mathtt{inh}$$

where the context U of unknowns is

$$?_1,\ ?_2,\ ?_3$$

After type reconstruction, we expect a solution $u \triangleright U$ like

$$?_1 : \mathtt{type} = \mathtt{tp},\ ?_2 : \mathtt{tp} \to \mathtt{type} = [x : \mathtt{tp}]\mathtt{tm}\,x \to \mathtt{tp},\ ?_3 : \{x : \mathtt{tp}\}\{y : \mathtt{tm}\,x \to \mathtt{tp}\}\mathtt{tp} = [x : \mathtt{tp}][y : \mathtt{tm}\,x \to \mathtt{tp}]x$$

After substituting each $?_i$ with its definiens in the declaration of `pi1` and simplifying the result, we obtain the declaration from Fig. 8.

4.2 Global State and Invariant

Due to the difficulty of type reconstruction, it is not easy to give the above-mentioned five algorithms directly. Instead, they become **subalgorithms** of a substantially more complex **main algorithm**. While the five subalgorithms do the logically relevant work, the main algorithm maintains global state and handles the bureaucracy.

Fig. 9 shows the input and output as well as the global state of the main algorithm. The individual aspects are described below.

Input	Σ, U, γ, j		
Problem	find $u \gg U$ such that $\gamma[u] \vdash_\Sigma j[u]$		
State	Intuition	Initial value	Success condition
u	context of unknowns	U	$u \gg U$
goals	delayed checking obligations	$\{\gamma \vdash j\}$	$goals = \varnothing$
Invariant	$u > U \wedge \exists^! v \gg u \bigwedge_{\Gamma \vdash J \in goals} \Gamma[\overline{v}] \vdash_\Sigma J[\overline{v}]$		
Output	u or failure		

Figure 9: Overview of main algorithm

Input The input consists of Σ, U, and γ, and a checking judgment j. Our goal is to solve $U, \gamma \vdash_\Sigma j$.

Because Σ contains the global declarations, i.e., those that do not change during a run of the algorithm, we can maintain Σ globally. Similarly, as we describe below, the unknowns are maintained as a mutable global variable u whose initial value is U. Therefore, we can drop both Σ and u from the notations. Thus, all judgments are simply written as $\Gamma \vdash J$ relative to globally maintained values for Σ and u.

Unknowns Formal descriptions of type reconstruction algorithms such as [ARCT12, Pie13] usually treat U as an immutable context and the solution u as a substitution that is to be found.

Our treatment is different: We maintain u as a global, mutable variable. It is initialized as U and is modified by the reconstruction algorithm along the way. Upon termination, u holds the needed solution.

This has the advantage that we can give a presentation of the algorithm that is close to simple implementations *and* relatively easy to read and reason about.

Delaying Goals In the presence of unknown variables, it is common for the subalgorithms to get stuck. For example, a syntax-directed type checking algorithm gets stuck when trying to check a term against an unknown type.

Therefore, the main algorithm maintains a delay-activate loop. If a subalgorithm is stuck on a checking judgment $\Gamma \vdash J$, that goal is **delayed**, and processing continues as if it had been discharged. Once an unknown variable that occurs in J has been solved, the goal becomes **activatable**, i.e., available for further processing.

The main algorithm maintains the set of currently delayed goals in the mutable variable *goals* and activates them when possible. Initially, *goals* contains only the judgment $\Gamma \vdash J$, and successful termination is possible if all delayed goals have been derived, i.e., if *goals* is empty.

Output The main algorithm repeatedly activates a goal from *goals* until

- all goals have been derived, i.e., *goals* is empty, or
- no open goal is activatable, or
- a subalgorithm signals failure.

The output in these cases is follows:

- *goals* is non-empty: Failure. The initial judgment may be provable but could not be proved.
- *goals* is empty:
  - all variables in u have a definiens: Success. u is returned.

- otherwise: Failure. The initial judgment may be provable but not all unknowns could be solved.[2][3]

- A subalgorithm signaled failure: Failure. This happens if the invariant was disproved, i.e., if the initial judgment is not provable.

For the purposes of this paper, we can simply assume that *failure* is signaled by raising an exception. That way we do not have to worry about *failure* being a possible output. In our implementation, we additionally keep a list of errors and do not interrupt the algorithm when failure is detected. That has the practical advantage of finding *all* typing errors, which is important for building user interfaces.

Invariant The invariant expresses the existence of a unique solution that satisfies all open goals. The invariant is implied by the success condition, but we do not require it to hold in the initial state. If it does not hold in the initial state, the input is ill-typed or ambiguous and should not be accepted successfully.

Therefore, we require that all transitions of the global state (such as applying a rule to reduce a goal to some subgoals) preserve and reflect the invariant. More precisely, we define:

Definition 4.5. A judgment $\gamma \vdash j$ holds in state $(u, goals)$ if

$$\left(\bigwedge_{\Gamma \vdash J \,\in\, goals} u, \Gamma \vdash J \right) \Rightarrow u, \gamma \vdash j$$

v is called a **solution** of $(u, goals)$ if $v \gtrdot u$ and all $g \in goals$ hold in state $(v, \varnothing)$.

Definition 4.6. A transition $(u, goals) \rightsquigarrow (u', goals')$ of the global state that satisfies $u' > u$ is

- **sound** if every solution v' of $(u', goals')$ is also a solution of $(u, goals)$,
- **complete** if for every solution v of $(u, goals)$ there is a unique solution v' of $(u', goals')$ such that $v' > v$.

It is called **faithful** if it is sound and complete.

In Def. 4.6, we assume that the state transition refines u to u'. This captures the intuition that a state transition should make progress towards a state $(v, \varnothing)$ where v is a solution. Because of $u' > u$, we have that $v \gtrdot u'$ implies $v \gtrdot u$. Therefore, solutions of $(u', goals')$ may also be solutions of $(u, goals)$. Soundness then requires that this is indeed the case for all solutions, i.e., the state transition does not introduce spurious solutions.

Conversely, completeness requires that the state transition does not lose any solutions v of $(u, goals)$. This is slightly trickier than soundness because u' may have more variables than u and thus $v \gtrdot u$ does not imply $v \gtrdot u'$. Therefore, we require that we can refine v to v' by adding uniquely determined definitions for the additional variables.

Faithful transitions preserve and reflect the invariant. But note that transitions that are only sound or only complete do not necessarily preserve or reflect the invariant.

Correctness The main theorem that motivates our algorithm is the following:

Theorem 4.7. *If we have a chain of faithful transitions*

$$(U, \{\gamma \vdash j\}) \rightsquigarrow \ldots \rightsquigarrow (u, \varnothing) \qquad \text{with } u \gtrdot U$$

then u is the unique solution such that $\gamma[\overline{u}] \vdash j[\overline{u}]$.

Proof. It is easy to see that faithfulness is reflexive and transitive. Thus the entire chain is faithful.

In the final state $(u, \varnothing)$, we can read off the unique solution directly because all variables have a definiens. Soundness guarantees that u is a solution of the initial state. Completeness guarantees uniqueness. □

Thus, a type reconstruction algorithm is correct if it applies faithful state transitions to the initial state until the success condition holds. Then the final state induces the solution.

[2] We could improve the algorithm here: If the types of all unsolved unknowns are known (and do not cyclically depend on each other), the original judgment j can be modified by universally quantifying over the remaining unsolved variables. This is the behavior of the Twelf system, where the last step is called *abstraction*.

[3] Another improvement, which is already implemented in MMT, is to start a theorem prover to generate possible solutions for an unsolved unknown whose type A is known. In particular, if A is a singleton type (e.g., if its values are proofs and we use proof irrelevance), this still yields a unique solution.

4.3 Inference Rules with Side Effects

Type checking algorithms are commonly presented as inference systems. This is desirable because it is easy to read, reason about, and implement. However, inference systems for type reconstruction can become very complicated because the delay-activate loop creates an interdependence between the different branches of a derivation tree. For example, if one branch solves an unknown, that solution must be propagated to all other branches.

In our case, this interdependence is modeled by the global state $(u, goals)$. A state transition triggered in one subtree must be visible to all other subtrees. Using shared global variables may sound too low-level for a research paper, but we actually obtain a rather elegant and concise formulation.

The key insight is to introduce a novel kind of inference rule: we allow hypotheses to perform state transitions when a rule is applied. To emphasize this statefulness, we will consistently speak of *s-rules* in the sequel:

Definition 4.8 (Rules)**.** An **s-rule** is of the form

$$L \quad \frac{P_1 \quad \dots \quad P_n}{P_0}$$

where L is an optional label.

The P_i may be:

- a **pure premise**: any of the five judgments from Fig. 7 (relative to the globally maintained Σ and u),
- a **effectful premise** $V > X : A \equiv t$ where
  - X is a variable in u, i.e., u is of the form $u_0,\ X : \_ = \_,\ u_1$,
  - V is a context of fresh variables relative to u_0,
  - A and t may be omitted, and if present are expressions relative to u_0, V.
- a **simple premise**: any other statement that can be decided immediately without effect on the state.

Example 4.9 (Simple Premises). Simple premises allow for all kinds of side conditions that occur frequently but do not require a formal treatment. Examples include

- lookups such as "$c : A = \_$ in Σ", which introduces A for a given c,
- abbreviations such as $C = A \to B$, which introduces C for given A and B,
- pattern matching such as $C = A \to B$, which introduces A and B for given C,
- freshness conditions such as $x \notin \Gamma$, which introduces some fresh name x for given Γ.

Most premises can be discharged immediately by simple computations. But some of them such as lookup or pattern matching can also fail.

Example 4.10 (Effectful Premise). The intended meaning of an effectful premise $V > X : A \equiv t$ is to perform a variable transformation that changes u in two ways:

- insert fresh unknown variables V before X in u,
- change the declaration of X in u by adding a type A and/or a definiens t.

Effectful premises allow the uniform treatment of several frequent state transitions. Most importantly, we can use $> X \equiv t$ for the special case where we have found the solution t of the unknown X.

The introduction of new unknowns is often necessary when unknowns are complex types. For example, assume we have determined that an unknown type X must be a simple function type but cannot yet determine its domain and codomain. Then we can use the variable transformation $A : \mathtt{type}, B : \mathtt{type} > X : \mathtt{type} \equiv A \to B$ to introduce fresh unknowns for domain and codomain and solve X relative to them. In Ex. 5.4, we need a similar variable transformation for type inference in LF.

Labels are useful to sort rules into different groups. In some situations, it is important to choose a rule from a specific group:

Example 4.11 (Labels). We will use the label `isol` to indicate that a rule can be used to isolate an unknown, i.e., to transform an equation into the form $X = E$ where X is an unknown and does not occur in E. Such equations can be used to solve X as E.

Isolation rules tend to be inverse to other equality rules. Therefore, they would easily lead to cycles if we applied them together with other equality rules. The label allows to avoid such cycles.

Because our s-rules may have side-effects, we have to be more careful when applying a rule. For example, the order of premises now matters. The following definitions make that precise:

Definition 4.12 (S-Function)**.** An **s-function** is a partial function that

- takes a state $(u, goals)$ and a judgment $\Gamma \vdash J$, and
- returns a successor state $(u', goals')$ as well as
- a result value, which is
  - when handling a checking judgment: **success** or **failure**,
  - when handling a query judgment: **success**(E) for a u'-expression E, **delay**, or **failure**.

If an s-function is defined, we say it is **applicable**.

S-functions will be the semantics of s-rules. The semantics of our five subalgorithms will also be s-functions.

Intuitively, applying an s-function to a checking judgment returns a boolean, i.e., **success** or **failure**. If the judgment has to be delayed, we still return **success** but also append it to *goals*.

Applying it to a query judgment returns the computed expression, i.e., **success**(E), or **failure** if no such expression exists. Additionally, it can return **delay** if the decision cannot be made at this point.

There is a big difference between delaying checking and query judgments. Delaying a checking judgment is easy: append it to *goals*, assume it succeeded, and continue. Delaying a query judgment is not possible because the result expression is usually needed to continue processing. Therefore, handling a query judgment may return **delay**, in which case we have to identify and delay the checking judgment that triggered the query.

Definition 4.13 (Semantics of Rules)**.** Consider an s-rule R with premises $P_1, \ldots, P_n$ and conclusion P_0.

Its semantics is the s-function that preforms a state transition and returns a result as follows:

1. If $\Gamma \vdash J$ is not of the form P_0, then R is not applicable. Otherwise, substitute the relevant parts of J into $P_1, \ldots, P_n$.
2. Process the premises $P_1, \ldots, P_n$ (in that order) as follows:
   - if P_i is pure: Call the corresponding subalgorithm on it. Let r_i be its result. If P_i is a query judgment and the subalgorithm returns **success**(E), substitute it into $P_{i+1}, \ldots, P_n, P_0$.
   - if P_i is effectful: Proceed as described in Def. 4.15.
   - if P_i is simple: Check the condition. If false, R is not applicable. If this computes new values, substitute them into $P_{i+1}, \ldots, P_n, P_0$.
3. If any r_i is **failure**, return **failure**.
4. Otherwise, if any r_i is **delay**, then
   - if P_0 is a query judgment: return **delay**,
   - if P_0 is a checking judgment: add P_0 to *goals* and return **success**.
5. Otherwise,
   - if P_0 is a query judgment: return **success**(E) where E is the right-hand side expression in P_0 after the substitutions from Step 2,
   - if P_0 is a checking judgment: return **success**.

Remark 4.14 (Backtracking Unapplicable Rules)*.* Inspecting Def. 4.13, we see that simple premises can make a rule inapplicable even if previous premises have already caused a state transition.

This happens for example, when the applicability of a rule depends on the result of a type inference. For example, we may want to treat $[x : A]E$ differently depending on the type of A. But inferring the type of A, may already solve unknowns that occur in A or trigger other pure premises that end up being delayed.

Therefore, implementations may have to backtrack to unroll these side effects. However, for many practical type systems, backtracking is redundant because the side effects are desired anyway. For example, even if we want to first infer the type of A to determine which rule to apply, the type of A has to be inferred either way. We see an example of this situation in Ex. 5.4.

It remains to define the semantics of effectful premises. Intuitively, we just change u according to the variable transformation. But we may have to perform some checks:

- If we try to solve a variable that has already been solved previously, we have to check equality of the two solutions.[4]

[4] Usually, this does not happen because once an unknown is solved, it can be substituted in all open goals.

- If both the type A and the definiens t of an unknown are solved, we additionally check $\vdash t : A$.

Formally, we obtain:

Definition 4.15 (Semantics of Effectful Premises)**.** We define the successor state and result r of processing the variable transformation $V > X : A_1 \equiv t_1$ in state (u, goals).

As an auxiliary concept, we define an option-expression to be either an expression E or $\bot$. Given two option-expressions O_1 and O_2, we define the option-expression $O_1 \vee O_2$ by:

- If $O_1 = \bot$, then O_2.
- If $O_2 = \bot$, then O_1.
- If $O_1 = E_1$ and $O_2 = E_2$, then E_1 and add $\vdash E_1 \equiv E_2$ to *goals*.

Now we proceed as follows:

1. Let u be of the form $u_0,\ X : A_2 = t_2,\ u_1$. If it is not, r is **failure**.
2. The A_i and t_i are option-expressions. Let $A = A_1 \vee A_2$ and $t = t_1 \vee t_2$.
3. If $t \neq \bot$ and $A \neq \bot$, add $\vdash t : A$ to *goals*.
4. Replace u with $u_0,\ V,\ X : A = t,\ u_1$.
5. r is **success**.

4.4 A Deeper Logical Formulation

This section is not needed for understanding the remainder of this paper. Instead, it gives an alternative formulation of the problem that some readers will find more and some less intuitive.

Sentences and Theory Morphisms We can extend MMT to a logic in a straightforward way: We use the judgments $\Gamma \vdash_\Sigma J$ as the sentences over Σ. Then we can allow MMT theories to be pairs $(\Sigma; K)$ where K is a set of Σ-sentences. Now we can define a theory morphism $\sigma : (\Sigma; K) \to (\Sigma'; K')$ as a list of maps $c \mapsto E$ such that

- σ contains exactly one map $c \mapsto E$ for every Σ-constant $c[: A][= t]$ where E must be a Σ'-expression such that (if present) $\vdash_{\Sigma'} E : \sigma(A)$ and $\vdash_{\Sigma'} E \equiv \sigma(t)$,
- for every $\Gamma \vdash_\Sigma J \in K$, the judgment $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(J)$ is derivable.

Here $\sigma(-)$ is the homomorphic extension of σ that replaces every Σ-constant c with the corresponding E.

State Transitions and Solutions are Theory Morphisms A state (U, goals) can be seen as a theory $(\Sigma, U; \mathit{goals})$. Then u being a solution of that state becomes equivalent to $\overline{u}$ being a theory morphism $(\Sigma, U; \mathit{goals}) \to (\Sigma, \varnothing)$. Indeed, in both cases, we have to provide a Σ-expression for all U-variables such that all sentences in K hold.

The condition $u' > u$ means that we have a theory morphism $i : (\Sigma, u; \varnothing) \to (\Sigma, u'; \varnothing)$ that maps all constants as $c \mapsto c$. A state transition $(u, \mathit{goals}) \rightsquigarrow (u', \mathit{goals}')$ is sound if i is also a theory morphism $(\Sigma, u; \mathit{goals}) \to (\Sigma, u'; \mathit{goals}')$. It is complete if there is a unique theory morphism in the opposite direction that is the identity on Σ, u. Thus, faithfulness becomes a special case of isomorphism. Moreover, if we call the solutions *models*, then faithfulness becomes a special of what is known as model-theoretical conservativity of theory morphisms.

We can use this intuition to generalize soundness and completeness as follows: A transition is sound if there is some theory morphism $m : (\Sigma, u; \mathit{goals}) \to (\Sigma, u'; \mathit{goals}')$ that is the identity on Σ. And it is complete if m is at least model-theoretically conservative—we may additionally require a uniqueness condition.

Connection to Other Transformations Using the intuitions from above, we obtain a broader perspective on what happens during type reconstruction. A theory is transformed into another theory by extending the language (going from u to u') and changing/extending the set of axioms. To be sound and complete, the inclusion of the old into the new language should yield a conservative theory morphism.

Typically, inference systems do not change the language along the way. The most important situation where it does happen is skolemization: This is a theory transformation $(\Sigma; K) \rightsquigarrow (\Sigma'; K')$ where $\Sigma' = \Sigma, f : A \to B$ and K' arises from K by replacing the axiom $\forall x : A \exists y : B.F(x, y)$ with $\forall x : A.F(x, f(x))$. The inclusion from Σ to Σ' is indeed a conservative theory morphism.

A similar theory transformation occurs when an occurrence of a description operator $\varepsilon x : A.F(x)$ is replaced with a fresh constant c and an axiom $F(c)$.

Main algorithm	Bureaucracy
Subalgorithms	Language-independent logical aspects
Rules from $\mathcal{S}$	Language-specific logical aspects

Figure 10: Separation of concerns in MMT type reconstruction

Such theory transformations can be difficult to describe declaratively in inference systems, especially if they are to be intertwined with other proof rules. Our s-rules may provide a formal framework for elegantly describing and reasoning about such transformations.

5 The Algorithm

Consider a logical framework $(T, \mathcal{R})$. Our type reconstruction algorithm is parametric in a set $\mathcal{S}$ of s-rules. The set $\mathcal{S}$ constitutes the algorithmic counterpart of the specification $\mathcal{R}$.

Conceptually, our type reconstruction algorithm consists of three levels as shown in Fig. 10. This structure yields two crucial abstraction barriers that separate the concerns involved in type reconstruction.

We describe the main algorithm in Sect. 5.1. It provides a first abstraction barrier by encapsulating all the bureaucracy that is critical for type reconstruction but has no deep logical relevance. Specifically, it maintains input, global state, and output as described in Sect. 4.2. It repeatedly chooses a checking judgment from *goals* and calls the respective subalgorithm on it until all goals are proved or failure occurs.

Afterwards we describe the five subalgorithms corresponding to the judgments in Fig. 7. Each one implements an s-function that arises by chaining some s-rule applications. The main work in the subalgorithms is to perform case distinctions that choose an appropriate s-rule to apply. Two kinds of s-rules are applied:

- a fixed set of language-independent s-rules that are justified by the rules given in Sect. 3.2,
- a parametric set $\mathcal{S}$ of language-specific s-rules that are justified by the rules in $\mathcal{R}$.

The distinction between these two kinds of rules provides a second abstraction barrier

As a running example, we will give a set of s-rules that yields a type reconstruction algorithm for the logical framework LF from Ex. 3.3.

5.1 Main Algorithm

The main algorithm is a relatively simple loop that picks judgments from *goals* and calls the respective subalgorithm on them until *goals* is empty. The only difficulty is that we have to avoid looping infinitely if we cannot make progress on any delayed judgment.

Algorithm 5.1 (Main Algorithm). A judgment $k \in goals$ is called **activatable** if the value of u has changed since k was added.

We repeat the following until *goals* contains no activatable judgments:

1. Remove an activatable judgment k from *goals*.
2. Let r be the result of calling the respective subalgorithm on k.
3. If r is **failure**, stop; otherwise, repeat.

Assuming all transitions were faithful, we can now read off the result from the final state $(u, goals)$ as described in Sect. 4.2.

Remark 5.2 (Selecting an Activatable Judgment). In Alg. 5.1, we use a very simple definition of *activatable* in order to simplify the presentation.

In practice, it makes sense to maintain for every $k \in goals$ a set of unknowns whose solution k is waiting for. Then k becomes activatable when one of those unknowns in solved.

5.2 Type Inference

Type inference handles the judgment $\Gamma \vdash t \stackrel{\leadsto}{:} A$ resulting in **success**(A), **failure**, or **delay**. It is implemented as a syntax-directed algorithm that proceeds by induction on t. We handle the base cases generically and apply s-rules from $\mathcal{S}$ otherwise:

Algorithm 5.3 (Type Inference). Consider the judgment $\Gamma \vdash t \overset{\leadsto}{:} A$.

- If t is a constant c declared in Σ, u, or Γ,
  - look up its type

$$\frac{c : A = \_ \text{ in } \Sigma, u, \Gamma}{\Gamma \vdash c \overset{\leadsto}{:} A}$$

  - if c has no type, infer the type of its definiens:

$$\frac{c : \bot = t \text{ in } \Sigma, u, \Gamma \qquad \Gamma \vdash t \overset{\leadsto}{:} A}{\Gamma \vdash c \overset{\leadsto}{:} A}$$

  - if c has neither type nor definiens, continue below.
- Otherwise, apply some applicable s-rule from $\mathcal{S}$.
- If no s-rule is applicable, try to simplify t:

$$\frac{\Gamma \vdash t \overset{\leadsto}{\equiv} t' \qquad \Gamma \vdash t' \overset{\leadsto}{:} A}{\Gamma \vdash t \overset{\leadsto}{:} A}$$

  If simplification returns $t' = t$, **delay**.

Example 5.4 (Type Inference for LF). For LF, we provide one type inference s-rule for each constant of `LF`:

$$\frac{}{\Gamma \vdash \mathtt{type} \overset{\leadsto}{:} \mathtt{kind}}$$

$$\frac{\Gamma \vdash A : \mathtt{type} \qquad \Gamma, x : A \vdash B \overset{\leadsto}{:} U}{\Gamma \vdash \{x : A\}B \overset{\leadsto}{:} U}$$

$$\frac{\Gamma \vdash A : \mathtt{type} \qquad \Gamma, x : A \vdash t \overset{\leadsto}{:} B}{\Gamma \vdash [x : A]t \overset{\leadsto}{:} \{x : A\}B}$$

$$\frac{\Gamma \vdash f \overset{\leadsto}{:} C \qquad C = \{x : A\}B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\, t \overset{\leadsto}{:} B[t]}$$

These are essentially the rules given in Ex. 3.3.

The s-rules from Ex. 5.4 are enough if there are no unknowns. In the presence of unknowns, we use one additional, unusual type inference s-rule. For example, if $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, we expect being able to infer the type of $[f](f\,0) + 1$ as $\{f : \mathbb{N} \to \mathbb{N}\}\mathbb{N}$. But recursively applying the type inference rules gets stuck at $f\,0$ because the type of f is omitted and thus an unknown. The following example adds that rule:

Example 5.5 (Type Inference for LF (Continued)). The LF rule for application needs a partner that handles the case where C is an unknown. Here we apply a variable transformation that decomposes the unknown type into a function type between fresh unknown types.

We first give a special case that establishes the intuition:

$$\frac{\Gamma \vdash f \overset{\leadsto}{:} C \qquad \begin{array}{l} C = X \\ X : \_ = \bot \text{ in } u \\ X_1, X_2, x \text{ fresh} \end{array} \qquad X_1, X_2 > X \equiv \{x : X_1\}X_2\, x \qquad \Gamma \vdash t : X_1}{\Gamma \vdash f\, t \overset{\leadsto}{:} X_2\, t}$$

This s-rule allows progress when the head of an expression has an unknown type: It decomposes the unknown type X into a function type with two fresh unknowns for domain and codomain. Relative to these new unknowns, type inference can proceed as usual.

In the general case, recalling Rem. 4.4, we have to allow for the type of f to contain variables from Γ. Then C is of the form $X\,\vec{y}$, and the new unknowns X_1 and X_2 may contain those variables, too.

Therefore, we use the following s-rule in general:

$$\frac{\Gamma \vdash f \mathrel{\overset{\leadsto}{:}} C \qquad \begin{array}{l} C = X\,\vec{y} \\ X : \_ = \bot \text{ in } u \\ X_1, X_2, x \text{ fresh} \end{array} \qquad \begin{array}{l} X_1, X_2 > X \\ \Gamma \vdash X\,\vec{y} \equiv \{x : X_1\,\vec{y}\} X_2\,\vec{y}\,x \end{array} \qquad \Gamma \vdash t : X_1\,\vec{y}}{\Gamma \vdash f\,t \mathrel{\overset{\leadsto}{:}} X_2\,\vec{y}\,t}$$

In this rule, the variable transformation only adds the new unknowns but does not solve X. Instead, an equality check captures the decomposition of X. We could alternatively solve X directly, but our s-rule is more convenient because equality checking will apply the appropriate s-rules for handling the variables $\vec{y}$ anyway.

Thus, we have two type inference rules that we can try to apply to $f\,t$. Both rules first infer the type of f, and depending on the result only one of the two rules is applicable.

5.3 Type Checking

Type checking handles the judgment $\Gamma \vdash t : A$ resulting in **success** or **failure**. It is implemented as a syntax-directed algorithm that proceeds by induction on A. We handle the base cases generically and apply s-rules from $\mathcal{S}$ otherwise:

Algorithm 5.6 (Type Checking). Consider the judgment $\Gamma \vdash t : A$.

- If t is a constant declared in Σ or Γ, look up its type and check equality:

$$\frac{c : A' = \_ \text{ in } \Sigma, \Gamma \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash t : A}$$

- If t is a constant X declared in u, add the type to u:

$$\frac{X : \_ = \_ \text{ in } u \qquad \text{no cycle} \qquad > X : A}{\Gamma \vdash X : A}$$

  Here the simple premise "no cycle" means that no unknown declared after X in u occurs in A. This check is necessary to make sure u remains well-formed.
- Otherwise, if the head of A is an unknown, delay.
- Otherwise, apply some applicable s-rule from $\mathcal{S}$.
- If no s-rule is applicable, try to simplify A:

$$\frac{\Gamma \vdash A \mathrel{\overset{\leadsto}{\equiv}} A' \qquad \Gamma \vdash t : A'}{\Gamma \vdash t : A}$$

  If simplification returns $A' = A$, infer the type of t and check equality:

$$\frac{\Gamma \vdash t \mathrel{\overset{\leadsto}{:}} A' \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash t : A}$$

  If type inference returns **delay**, delay.

Here "delay" means to add $\Gamma \vdash t : A$ to *goals* and return **success**.

Example 5.7 (Type Checking for LF). For LF, we need a single type checking s-rule that checks terms against a function type:

$$\frac{\Gamma, y : A \vdash f\,y : B[y]}{\Gamma \vdash f : \{x : A\}B} \quad y \text{ fresh}$$

This s-rule is not part of Ex. 3.3 because it is derivable by using `lambda` and `eta`. It is needed here as a separate s-rule to obtain better computational behavior of type reconstruction, especially as we will not use `eta`.

5.4 Inhabitability Checking

Inhabitability checking handles the judgment $\Gamma \vdash A\,\texttt{inh}$ resulting in **success** or **failure**. This subalgorithm is rather trivial because MMT provides no built-in s-rules for inhabitability:

Algorithm 5.8 (Inhabitability Checking). Consider the judgment $\Gamma \vdash t : A$.

- Apply some applicable s-rule from $\mathcal{S}$.
- If no s-rule is applicable, add the judgment to *goals* and return **success**.

Example 5.9 (Inhabitability for LF). For LF, we use exactly the rules given in Ex. 3.3:

$$\frac{\Gamma \vdash A \mathrel{\overset{\leadsto}{:}} U \qquad U \in \{\texttt{type}, \texttt{kind}\}}{\Gamma \vdash A\,\texttt{inh}}$$

5.5 Simplification

Simplification handles the judgment $\Gamma \vdash E \mathrel{\overset{\leadsto}{\equiv}} E'$ resulting in **success**(E'). Simplification never returns **failure** or **delay**, but it may default to $E' = E$ if no s-rule is applicable. We say that simplification makes **progress** if $E' \neq E$.

Simplification performs two kinds of steps: expand a definition or apply an s-rule from $\mathcal{S}$. To avoid an unnecessary blowup due to greedily expanding definitions, each call to simplification performs only a single step:

Algorithm 5.10 (Simplification). Consider the judgment $\Gamma \vdash E \mathrel{\overset{\leadsto}{\equiv}} E'$.

- Apply applicable s-rules from $\mathcal{S}$ until one of them makes progress.
- If none makes progress, if E is a constant with definiens t, return **success**(t):

$$\frac{c : \_ = t \text{ in } \Sigma, u, \Gamma}{\Gamma \vdash c \mathrel{\overset{\leadsto}{\equiv}} t}$$

- Otherwise, if E is a complex term $c(\Delta; \overrightarrow{E_n})$, apply the congruence rule from Fig. 6 to simplify some E_i. Specifically, try (in the order $i = 1, \dots n$)[5]

$$\frac{\Gamma, \Delta \vdash E_i \mathrel{\overset{\leadsto}{\equiv}} E_i' \qquad E_i \neq E_i'}{\Gamma \vdash c(\Delta; \overrightarrow{E_n}) \mathrel{\overset{\leadsto}{\equiv}} c(\Delta; E_1, \dots, E_{i-1}, E_i', E_{i+1}, \dots, E_n)}$$

- Otherwise, return **success**(E) (i.e., make no progress):

$$\frac{}{\Gamma \vdash E \mathrel{\overset{\leadsto}{\equiv}} E}$$

Example 5.11 (Simplification for LF). For LF, the only simplification s-rule is the rule `beta` from Ex. 3.3.

Because we apply only one simplification step at a time, the simplification algorithm for LF amounts to weak head normal form conversion known from λ-calculus.

5.6 Equality Checking

Equality checking handles the judgment $\Gamma \vdash E \equiv E' : A$ where A may or may not be provided. If A is provided, equality checking assumes that $\Gamma \vdash E : A$ and $\Gamma \vdash E' : A$ are implied by the invariant. It results in **success** or **failure**.

This is by far the most complex of the five subalgorithms because it must try different strategies and solve unknown variables. Therefore, we break it up into multiple parts:

[5] We could also simplify in the context Δ without any problems. We omit that here for simplicity. If a variable in Δ has a type, simplification in the type can occasionally be useful to make certain advanced rules applicable. For example, a decision procedure for the natural numbers might be applicable only to quantifiers over $x : nat$. If a variable x in Δ has a definiens, that definiens can anyway also be expanded where x occurs in some E_i.

Algorithm 5.12 (Equality Checking). Consider the judgment $\Gamma \vdash E \equiv E' : A$. We first describe the base cases:

- If $E = E'$, return **success**:

$$\frac{}{\Gamma \vdash E \equiv E : A}$$

- Otherwise, if E is an unknown X, solve it:

$$\frac{X : \_ = \_ \text{ in } u \qquad \text{no cycle} \qquad > X \equiv E'}{\Gamma \vdash X \equiv E'}$$

  Here the simple premise "no cycle" is as in Alg. 5.6.
  If E' is an unknown, we apply the dual rule.
- Otherwise, try to isolate an unknown as described in Alg. 5.14.

If isolation is not possible, we apply a syntax-directed algorithm that proceeds by induction on A:

- If A is not provided, determine it by type inference:

$$\frac{\Gamma \vdash E \overset{\rightsquigarrow}{:} A \qquad \Gamma \vdash E \equiv E' : A}{\Gamma \vdash E \equiv E'}$$

  Here, if $\Gamma \vdash E \overset{\rightsquigarrow}{:} A$ returns **delay**, we try $\Gamma \vdash E' \overset{\rightsquigarrow}{:} A$ accordingly. Only if both of them return **delay**, we add the input judgment to *goals* and return **success**.
- Otherwise, apply some applicable s-rule from $\mathcal{S}$.
- If no s-rule is applicable, try to simplify A:

$$\frac{\Gamma \vdash A \overset{\rightsquigarrow}{\equiv} A' \qquad \Gamma \vdash E \equiv E' : A'}{\Gamma \vdash E \equiv E' : A}$$

  If simplification returns $A' = A$, apply term-based equality checking as described in Alg. 5.16.

Example 5.13 (Equality Checking for LF). For LF, we need a single equality checking s-rule that checks the equality of two functions:

$$\frac{\Gamma, y : A \vdash f\, y \equiv f'\, y : B[y]}{\Gamma \vdash f \equiv f' : \{x : A\}B} \quad y \text{ fresh}$$

Note that this s-rule is very similar to the one for type checking from Ex. 5.7. This s-rule is known as the extensionality rule. In the presence of ξ (which is fixed in MMT, see Rem. 3.10) and β (which is part of LF, see Ex. 5.11), it is equivalent to `eta` (which we do not use in this example).

Isolating Unknowns Now we present the step skipped in the first part of Alg. 5.12. The basic idea is to generalize the algebraic equation solving technique that transforms, e.g., $X + a = b$ into $X = b - a$, i.e., to invert the toplevel operator (here: $+$) until an unknown is isolated.

The algorithm is rather simple because the difficulty of spotting when and how a variable can be isolated is delegated to isolation rules:

Algorithm 5.14 (Isolation). An **isolation s-rule** is an s-rule with the label `isol`.

Consider the judgment $\Gamma \vdash E \equiv E'$.

- If applying a sequence of isolation s-rules can transform this judgment into $\Gamma^* \vdash X \equiv E^*$, apply those s-rules.
- Otherwise, do nothing.

Here the label `isol` is used to indicate that an equality s-rule should only be used during isolation. That is necessary to avoid cycles because isolation s-rules tend to invert other equality s-rules. Therefore, isolation s-rules must only be applied if they actually allow isolating an unknown.

Moreover, it is important that Alg. 5.14 searches for sequences of isolation s-rules, not just individual isolation s-rules. For example, if we use isolation s-rules for arithmetic, we need to apply two different isolation s-rules to isolate X in $(X + a) * b = c$.

Example 5.15 (Isolation Rules for LF). For LF, we need one isolation s-rule that inverts application:

$$\texttt{isol} \quad \frac{\Gamma, \Gamma' \vdash E \equiv [x:A]E'}{\Gamma, x:A, \Gamma' \vdash E\,x \equiv E'} \qquad x \text{ not in } \Gamma', E$$

It is instructive to prove that the conclusion of this rule is equivalent to the premise. Soundness (top-to-bottom) follows by applying both sides to a fresh variable x (which is allowed because of congruence) and using `beta` on the right. Completeness (bottom-to-top) follows by λ-abstracting over x on both sides (which is allowed because of congruence) and applying `eta` on the left.

In particular, if $E = X\,\vec{x}$ is an unknown that is applied to distinct variables $x_1, \ldots, x_n$, we can iterate this s-rule n times to transform $X\,x_1\,\ldots\,x_n \equiv E'$ into $X \equiv [x_1 : A_1] \ldots [x_n : A_n]E'$. At this point, isolation has succeeded, and Alg. 5.12 applies the variable transformation $>X \equiv [x_1 : A_1] \ldots [x_n : A_n]E'$.

This treatment of expressions $X\,x_1\,\ldots\,x_n$ is essentially the one from pattern unification.

Note that the isolation rule from Ex. 5.15 is the only rule from the entire LF example that is aware of the existence of meta-variables. All other rules are the same as the ones needed anyway for type-*checking* without any reconstruction.

Term-Based Equality Checking Alg. 5.12 typically reaches a base case where A is an atomic type for which no applicable s-rule is provided. In this case, we have to perform equality reasoning by exploiting the shape of the terms E and E'.

There are several strategies, all of which are problematic in some cases. Therefore, even for languages where equality checking is decidable, different choices at this point can show vastly different performance. Here we only describe a naive solution (that is considerably weaker than our implementation) for the sake of completeness:

Algorithm 5.16 (Term-Based Equality Checking). Consider the judgment $\Gamma \vdash E \equiv E' : A$.

- As long as simplification makes progress, simplify E and E':

  $$\frac{\Gamma \vdash E \overset{\rightsquigarrow}{\equiv} E^* \qquad \Gamma \vdash E^* \equiv E' : A}{\Gamma \vdash E \equiv E' : A}$$

  and accordingly for E'.
- If simplification of E and E' makes no progress anymore, try to apply the following s-rule (which combines α-renaming and congruence):

  $$\frac{\Gamma, \overrightarrow{y_{i-1} : B_{i-1}} \vdash B_i \equiv B'_i \qquad \Gamma, \overrightarrow{y_m : B_m} \vdash E_i[\overrightarrow{y_m}] \equiv E'_i[\overrightarrow{y_m}]}{\Gamma \vdash c(\overrightarrow{x_m : A_m}; \overrightarrow{E_n}) \equiv c(\overrightarrow{x'_m : A'_m}; \overrightarrow{E'_n})} \qquad \begin{array}{l} E \overset{inj}{\equiv} E' \\ \overrightarrow{y_m} \text{ fresh} \end{array}$$

  $$\text{where } B_i = A_i[\overrightarrow{y_{i-1}}] \quad \text{and} \quad B'_i = A'_i[\overrightarrow{y_{i-1}}]$$

  where we use the notations from Fig. 6.
  The intuition behind this s-rule is to reduce the equality of two complex expressions with the same constructor c to the equality of their components. This is always sound but only complete if c is injective. Therefore, this s-rule is guarded by the premise $E \overset{inj}{\equiv} E'$ that is described below. Intuitively, it guarantees that the rule is only applicable if c is injective.
- Otherwise, if no unknowns occur in $\Gamma \vdash E \equiv E' : A$, return **failure**.
- Otherwise, append $\Gamma \vdash E \equiv E' : A$ to *goals* and return **success**.

Remark 5.17 (Exhaustive Simplification). The first part of Alg. 5.16 exhaustively simplifies E and E'. Here definition expansion may cause a dramatic blowup in expression size that is often unnecessary. It can even preclude finding a solution that would otherwise be easy to find.

Much better strategies must be (and are) used in practice. But the choice is difficult. For example, let us assume that we alternate between simplifying E and E' (starting with E) and compare again after each step. Then we end up comparing the following pairs of expressions (E, E'), (E_1, E'), (E_1, E'_1), (E_2, E'_1),.... Thus if $E_1 = E'$, we have to apply only a single step to derive the equality. But if $E = E'_1$, we may end up fully simplifying both expressions before deriving the equality.

Injectivity Rules The second part of Alg. 5.16 applies the congruence rule to reduce, e.g., $\Gamma \vdash c(\cdot; e) \equiv c(\cdot; e')$ to $\Gamma \vdash e \equiv e'$.

This s-rule is clearly sound. But it is complete only in special cases, namely if c is injective with respect to all argument positions in which the two expression do not agree. Injectivity depends on the constructor c and cannot be established generically by MMT. Therefore, Alg. 5.16 uses the special side condition $E \overset{inj}{\equiv} E'$, which must be established by constructor-specific injectivity s-rules.

Example 5.18 (Injectivity for LF). We provide one injectivity s-rule for each constructor of complex expressions.

The constructor $c = \texttt{Pi}$ is injective in all argument positions:

$$\texttt{inj} \quad \frac{}{\{x : A\}B \overset{inj}{\equiv} \{x : A'\}B'}$$

Indeed, $\{x : A\}B$ and $\{x : A'\}B'$ can only be equal if $A \equiv A'$ and $B \equiv B'$. The same applies for $c = \texttt{lambda}$

$$\texttt{inj} \quad \frac{}{[x : A]t \overset{inj}{\equiv} [x : A']t'}$$

but that s-rule is redundant because the equality of two LF-functions is already handled when Alg. 5.12 applies the extensionality s-rule from Ex. 5.13.

The situation is more difficult for the constructor $c = \texttt{apply}$. Clearly, we cannot have $f\,a \overset{inj}{\equiv} g\,b$ for all f, a, g, b. The best we can hope for is $f\,a \overset{inj}{\equiv} f\,b$ for all a, b—which exactly expresses the injectivity of f. But that is not always complete either. In any case, note that we only have to worry about the case where f is a constant—the only other well-typed possibility would be a λ-abstraction, in which case Alg. 5.16 applies β-reduction anyway.

For constants, we have the following:

$$\texttt{inj} \quad \frac{c : \_ = \bot \text{ in } \Sigma, \Gamma}{c\,a_1 \,\ldots\, a_n \overset{inj}{\equiv} c\,b_1 \,\ldots,\, b_n}$$

i.e., undefined constants declared in the global theory Σ or the context Γ (but not unknowns) are injective functions.

We do not have to consider constants with definiens because Alg. 5.16 expands them anyway. However, even if we use a smarter simplification algorithm that does not expand definitions aggressively, we are not allowed to consider constants with definiens: the definiens might not be injective.

Finally, c may not be an unknown. An unknow must be solved first before we can assess whether it is injective.

Example 5.19 (Strict Definitions). The Twelf implementation [PS99] of LF uses an additional injectivity s-rule. Defined constants are considered injective if their definiens is strict. Strictness is an easily-decidable condition about the occurrences of variables that guarantees injectivity.

Remark 5.20 (Canonical Forms). The injectivity statement for `apply` in Ex. 5.18 is a reformulation of what is called canonical forms for LF in [HHP93]. If LF-terms are fully simplified, they are either (i) introduction forms (formed with constructor `lambda`) or (ii) elimination forms (formed with constructor `apply`) whose head is a constant. These expressions are called the canonical forms, and they can only be equal if their constituents are.

This result can be generalized to other language features, e.g., to product types. In those cases, Alg. 5.12 becomes a decision procedure if all unknowns are solved.

Remark 5.21 (Brittleness of Injectivity). Even when injectivity holds, it is brittle: It depends on which rules are present in $\mathcal{R}$. For example, assume we add a type *unit* to LF. Now in the presence of the equality rule

$$\frac{}{\Gamma \vdash t \equiv t' : unit}$$

injective functions of type $A \to unit$ are suddenly not injective anymore.

This is in contrast to all other s-rules presented in this section, which remain faithful no matter what rules are added to $\mathcal{R}$. Consequently, the faithfulness of these injectivity rules must be established again whenever $\mathcal{R}$ is extended.

From a proof-theoretical perspective, this is because injectivity is not a rule that is derivable from $\mathcal{R}$. Instead, it is only admissible, and that must be proved by induction on expressions and derivations. Therefore, adding rules may break injectivity.

From a model-theoretical perspective, this is because we are using initial model semantics. In the initial model, terms are unequal unless their equality can be derived. Therefore, functions are injective by default. But if we refine the initial model (which corresponds to by adding equality rules to $\mathcal{R}$), those inequalities are not necessarily preserved.

6 Modular Rule Sets

Now we give an example of how to reap the benefits of MMT's generic treatment: because the type reconstruction algorithm is parametric in the set of s-rules, we can easily instantiate it with different type systems. Moreover, because the faithfulness of almost all rules is retained when extending the type system with new rules, any combination of faithful rules yields a correct implementation.

We exemplify the procedure by giving type reconstruction rules for function types and product types as separate, combinable modules.

6.1 Function Types

$$\frac{\Gamma \vdash A : \mathtt{type} \qquad \Gamma, x : A \vdash B \overset{\leadsto}{:} \mathtt{type}}{\Gamma \vdash \{x : A\}B \overset{\leadsto}{:} \mathtt{type}}
\qquad
\frac{\begin{array}{l}\Gamma \vdash f \overset{\leadsto}{:} X \\ X_1 : \mathtt{type}, X_2 : X_1 \to \mathtt{type} > \\ \quad > X : \mathtt{type} \equiv \{x : X_1\}X_2\, x \\ \Gamma \vdash t : X_1\end{array}}{\Gamma \vdash f\, t \overset{\leadsto}{:} X_2\, t}$$

$$\frac{\Gamma \vdash A : \mathtt{type} \qquad \Gamma, x : A \vdash t \overset{\leadsto}{:} B}{\Gamma \vdash [x : A]t \overset{\leadsto}{:} \{x : A\}B}
\qquad
\frac{\Gamma \vdash f \overset{\leadsto}{:} \{x : A\}B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\, t \overset{\leadsto}{:} B[t]}$$

$$\frac{\Gamma, y : A \vdash f\, y : B[y]}{\Gamma \vdash f : \{x : A\}B}
\qquad
\frac{\Gamma, \Gamma' \vdash f \equiv [x : A]t}{\Gamma, x : A, \Gamma' \vdash f\, x \equiv t}\ x \text{ not in } \Gamma', f$$

$$\frac{\Gamma, y : A \vdash f\, y \equiv f'\, y : B[y]}{\Gamma \vdash f \equiv f' : \{x : A\}B}
\qquad
\frac{\Gamma \vdash a : A}{\Gamma \vdash ([x : A]t)\, a \overset{\leadsto}{\equiv} t[a]}$$

Figure 11: Type Reconstruction Rules for Dependent Function Types

The examples throughout Sect. 4 have already introduced the s-rules for dependent function types and kinds. We repeat the ones for dependent function types (i.e., the pure type system rule $(*, *)$) in Fig. 11. For simplicity, we omit some side conditions.

Besides summarizing the s-rules, Fig. 11 systematically arranges them in a 4×2 table. This is no coincidence: Our arrangement indicates a general pattern that we can find in many type operators and that we consider an interesting and novel result in itself. The sequel discusses this pattern in more detail.

Upper Half Consider the following triplet of rules, which we find very often when giving the inference rules for a type constructor:

formation	
introduction	elimination

A type constructor is often defined by three symbols: the constructor itself, the introduction form, and the elimination form. And the inference system usually provides one rule for each of these: the formation rule creates the new type, the introduction rule creates elements of that type, and the elimination rule uses elements of the new type. For example, in Ex. 3.3, we have one rule each for `Pi`, `lambda`, and `apply`.

These 3 rules form 3 pairs out of which 2 have a special symmetry:

- Formation and introduction behave very similarly. Both the abstract and concrete syntax of `Pi` and `lambda` and the premise of the rule are systematically similar.
- Introduction and elimination are duals: One creates values, the other uses them. Concretely, the same type occurs in the conclusion of the `lambda` rule and in the first premise of the `apply` rule.

However, there is no similar symmetry between formation and elimination.

Our general pattern includes a fourth rule, which leads to the following table:

formation	decomposition
introduction	elimination

This quartet of rules is more appealing because all 4 pairs of neighbors share a symmetry:

- Formation and introduction: as discussed above.
- Introduction and elimination: as discussed above.
- Formation and decomposition are inverse to each other. Formation constructs the type from components; decomposition destructs it into components.
- Decomposition and elimination are two cases of a case distinction for type inference of the elimination form. The former is the case where the type is unknown, the latter is the case where the type is known.

Lower Half The four rules in the lower half are usually given as the following two normalization rules:

formation	
introduction	elimination
expansion/extensionality	computation

Expansion states that the introduction form is surjective. It is usually equivalent to a rule that states that the elimination form is injective. And computation reduces elimination of an introduction form. For example, for function types, the three rules are called η, extensionality, and β, respectively.

Our general pattern uses the following rules, which is more practical for type reconstruction:

formation	decomposition
introduction	elimination
type checking	isolation
equality checking	computation

The rules in the lower left quadrant are the two checking rules. Both use the elimination operator in the same way to check typing and equality. The type checking rule can be derived from the η-rule, and equality checking is the extensionality rule.

The two rules in the lower right quadrant define the meaning of the elimination form. The lower one (β) defines the meaning of applying a known function, the upper one of an unknown function.

Left Half The four rules on the left capture the key properties of the new type: forming the type, introducing values, checking values, and checking equality of values.

Right half The four rules on the right capture the properties of the elimination form. The isolation-computation symmetry in the lower right quadrant mirrors the decomposition-elimination symmetry in the upper right quadrant.

6.2 Product Types

Now we give the s-rules for product types. For simplicity, we restrict attention to the simply-typed case.

The MMT theory is given in Fig. 13. The rules are given in Fig. 13, arranged in the same 4×2 schema as in Fig. 11.

Fig. 13 omits the injectivity rules. These are similar to the ones for function types: We need an injectivity rule for `prod`, and `pair` is injective, too, but the rule is redundant. For the projections π^i, we can apply injectivity if they are applied to an undefined constant from Σ or Γ.

`type`	#	`type`
`prod`	#	$\mathcal{A}_1 \times \mathcal{A}_2$
`pair`	#	$(\,\mathcal{A}_1\, \mathcal{A}_2\,)$
`pi1`	#	$\pi^1\, \mathcal{A}_1$
`pi2`	#	$\pi^2\, \mathcal{A}_1$

Figure 12: A Theory for Product Types in Mmt

$$\frac{\Gamma \vdash A_1 \overset{\leadsto}{:} \mathtt{type} \qquad \Gamma \vdash A_2 \overset{\leadsto}{:} \mathtt{type}}{\Gamma \vdash A_1 \times A_2 \overset{\leadsto}{:} \mathtt{type}}$$

$$\frac{\Gamma \vdash a_1 \overset{\leadsto}{:} A_1 \qquad \Gamma \vdash a_2 \overset{\leadsto}{:} A_2}{\Gamma \vdash (a_1, a_2) \overset{\leadsto}{:} A_1 \times A_2}$$

$$\frac{\begin{array}{l}\Gamma \vdash t \overset{\leadsto}{:} X \\ X_1 : \mathtt{type}, X_2 : \mathtt{type} > \\ > X : \mathtt{type} \equiv X_1 \times X_2\end{array}}{\Gamma \vdash \pi^i\, t \overset{\leadsto}{:} X_i}$$

$$\frac{\Gamma \vdash t \overset{\leadsto}{:} A_1 \times A_2}{\Gamma \vdash \pi^i\, t \overset{\leadsto}{:} A_i}$$

$$\frac{\Gamma \vdash \pi^i\, t : A_i}{\Gamma \vdash t : A_1 \times A_2}$$

$$\frac{\Gamma \vdash \pi^i\, t \equiv \pi^i\, t' : A_i}{\Gamma \vdash t \equiv t' : A_1 \times A_2}$$

$$\text{head of } t \text{ is } X \quad \frac{Y > X \qquad \Gamma \vdash t \equiv \begin{cases} (t', Y) \text{ if } i=1 \\ (Y, t') \text{ if } i=2 \end{cases}}{\Gamma \vdash \pi^i\, t \equiv t'}$$

$$\frac{\Gamma \vdash a_i : A_i}{\Gamma \vdash \pi^i\, (a_1, a_2) \overset{\leadsto}{\equiv} a_i}$$

i in premises: two premises for $i = 1, 2$
i in conclusion: two rules for $i = 1, 2$

Figure 13: Type Reconstruction Rules for Product Types

Because product types use two elimination forms π^1 and π^2, all rules are stated for π^i for $i = 1, 2$. In the rules in the right half, π^i occurs in the conclusion—these actually abbreviate two rules each. In the rules in the lower left quadrant π^i occurs in the premise—these premises abbreviate a pair of premises.

All rules are straightforward except for the isolation rule. For example, in $\pi^1\, X \equiv t'$, we cannot solve X—we can only solve its first component. Therefore, the isolation rules introduces a fresh unknown Y for the second component and then solves X as (t', Y).

Remark 6.1 (Chaining Isolation Rules). Note that our isolation rules for function and product types are carefully formulated in such a way that they can be combined with other isolation rules.

For example, consider isolation for the judgment $\Gamma, x : A \vdash (\pi^1\, X)\, x \equiv E'$. It will first apply the isolation rule for function types resulting in $\Gamma \vdash \pi^1\, X \equiv [x : A]E'$. Then the isolation rule for product types yields $\Gamma \vdash X \equiv ([x : A]E', Y)$ with a fresh variable Y.

In general, the isolation algorithm can isolate an unknown whenever a sequence of elimination forms is applied to an unknown.

Remark 6.2 (Dependent Product Types). It is possible to generalize these rules to dependent product types.

We omit that here to avoid the subtleties raised by losing type unicity.

6.3 Shallow Polymorphism

Shallow polymorphism is not a type operator in the sense of function or product types. Instead, it allows all declarations to additionally have some free kinded variables, e.g., as in $\mathit{list}(a : \mathtt{type}) : \mathtt{type}$.

In the presence of function types, we can treat such constants as functions. Thus, we can use the declaration $\mathit{list} : \{a : \mathtt{type}\}\mathtt{type}$. Correspondingly, the instantiation of a polymorphic constants becomes a special case of function application. Now *shallow* means that expressions like $\{a : \mathtt{type}\}\mathtt{type}$ are allowed to declare a polymorphic constant but may not occur anywhere else. In particular, we do not have $\{a : \mathtt{type}\}\mathtt{type} : \mathtt{type}$.

We obtain a type reconstruction algorithm for polymorphic variants of LF by adding a single inhabitability rule:

$$\frac{\Gamma \vdash A \stackrel{\rightsquigarrow}{:} U \qquad U \in \{\mathtt{type}, \mathtt{kind}\} \qquad \Gamma, x : A \vdash B\,\mathtt{inh}}{\Gamma \vdash \{x : A\}B\,\mathtt{inh}}$$

It makes expressions such as $\{a : \mathtt{type}\}a \to a \to \mathtt{type}$ inhabitable and thus allows declaring, e.g., the *list* constant from above.

Note that even with this rule, according to Ex. 5.4, type inference of $\{a : \mathtt{type}\}a \to a \to \mathtt{type}$ does not succeed. That is intentional to preclude, e.g., $\{a : \mathtt{type}\}a \to a \to \mathtt{type} : \mathtt{type}$.

Thus, we can build polymorphic variants of languages modularly and immediately obtain a polymorphic type reconstruction algorithm from the monomorphic one. In particular, the type reconstruction algorithm can now solve implicit type arguments of polymorphic constants without any additional rules.

The formal systems of LF and Isabelle are quite similar, the main difference being that the former uses dependent types and the latter shallow polymorphism and higher-order logic. It is straightforward to define higher-order logic as a theory of LF with shallow polymorphism. Thus, we can obtain a logical framework that combines the features of LF and Isabelle.

6.4 Advanced Features

The above examples were intentionally chosen for their simplicity. It remains to investigate how our algorithm fares on more complex languages. We have conducted three case studies to that end. Notably, in all three cases no type reconstruction support existed before. The case studies are too complex to be included in this paper. But we briefly describe them in the sequel.

Rewriting LF modulo extends LF with rewriting [CD07]. It gained attention with the Dedukti system [BCH12], which demonstrated that (under the assumption that the user-provided rewrite rules are normalizing) LF modulo is as practical but more powerful than LF. Dedukti has been used in particular as a universal proof checker and is therefore optimized towards fast checking of large already-reconstructed libraries. Consequently the developers did not prioritize type reconstruction.

However, checking a library written in logic L always requires a fixed encoding of L in Dedukti. This is a manual once-per-logic effort. But if L is complex, it quickly reaches the threshold where reconstruction becomes desirable. Here, our algorithm can supplement Dedukti by allowing to encode the logic L in MMT and then generating the necessary Dedukti file.

Therefore, we represented LF modulo in MMT. Because this requires user-written rewrite rules, we introduced a special MMT plugin to generate rules. There are various ways to do that, e.g., with a symbol

$$\rightsquigarrow : \ \{a : \mathtt{type}\}a \to a \to \mathtt{type} \ \# \ \mathcal{A}_2 \rightsquigarrow \mathcal{A}_3$$

with the intuition that the existence of a term of type $\{\overrightarrow{x_m : A_m}\}l \rightsquigarrow r$ legitimizes rewriting l to r.

We implemented a plugin for MMT that watches for declarations of constant, whose type that form. Whenever one is found, the plugin automatically generates the simplification rule that turns l into r for arbitrary $\overrightarrow{x_m}$. Because our type reconstruction algorithm anyway uses all rules visible in the context, the generated rule is automatically used in the sequel.

Thus, users can add new rewrite rules declaratively and dynamically.

Sequences In previous work [HKR11], we designed a logical framework with native support for sequences. This allows using sequence arguments (i.e., operators that a flexible number of arguments) and sequence variables (i.e., binders that bind a flexible number of variables).

However, the language in [HKR11] only *specified* the well-typed terms (i.e., it only gave the set $\mathcal{R}$). Designing an algorithm for it that allows for type reconstruction proved elusive. Type reconstruction was so difficult that we started investigating how to simplify the language in a way that it retains the essentials of its expressivity while making type reconstruction easier.

Using the framework presented in this paper, the author was able to experiment with different variants efficiently and find a good trade-off within a few days. Without this framework, the same investigation would have been prohibitively expensive because every variant of the language would have required a separate implementation.

Locks LLF$_\mathcal{P}$ [HLMS17] is an extension of LF that adds *locks*: these are monadic type constructors $\mathcal{L}^{\mathcal{P}}_{t,A}[\cdot]$ indexed by meta-judgments $\mathcal{P}(\vdash t : A)$. We can think of $\mathcal{P}$ as the attitude via which $\vdash t : A$ should be established, e.g., $\mathcal{P}$ could indicate the use of an external prover. LLF$_\mathcal{P}$ was designed as a mechanism for extending LF with external features and is therefore inherently not definable inside LF.

The designers of LLF$_\mathcal{P}$ are very interested in obtaining tool support and—quite naturally—began a from-scratch implementation. While this was ongoing, the issue came up in a conversation of the present author with one of them—Ivan Scagnetto. Together we were able to instantiate MMT with LLF$_\mathcal{P}$ easily, thus not only implementing the language immediately but getting type reconstruction for free.

This required writing an MMT theory that used 4 (untyped) constants and 6 rules, whose implementation in MMT took about 100 lines of code. Importantly, this whole collaboration unfolded over only a few hours after dinner: building the entire implementation barely took any longer than it took to explain the rules in the first place.

7 Implementation

The MMT system [Rab09] implements the data structures for theories, contexts, terms, and judgments as well as the infrastructure for authoring, analyzing, and maintaining them. The system is designed to be maximally generic: all algorithms (e.g., parsing, type reconstruction, etc.) work in the same way for every MMT theory. Whenever language-specific knowledge (e.g., inference rules) is required during an algorithm, MMT encapsulates it in abstract interfaces.

MMT calls the instances of these abstract interfaces **rules**. Each rule is an object in the underlying programming language of MMT (Scala), whose methods are invoked during the corresponding MMT algorithm. In particular, all s-rules needed for type reconstruction are supplied in this way. Rules are directly programmed in Scala, and MMT provides support for compiling and loading them at run time. Alternatively, rules can be generated dynamically by MMT plugins.

A major result of MMT is to demonstrate that the language-independent parts of these algorithms can be dramatically larger than the language-specific ones. For example, to implement LF, we implement the rules given in Sect. 4. These require only about 200 lines of Scala code, less than 1% of the overall MMT code base.

In the following, we briefly describe some features of the MMT implementation that interact with type reconstruction.

Type Reconstruction The author implemented the algorithm presented in this paper as a part of the MMT system. The implementation supports several additional features, e.g.,

- a subalgorithm for undecidable subtyping,
- lazy expansion of definitions,
- tracing of dependencies to allow for change management,
- option for rules to try to derive a judgment and backtrack if the attempt fails,
- option to call a theorem prover to discharge undecidable side conditions,
- good support for user-friendly error messages.

But apart from these additional features, the presentation in this paper corresponds closely to the implementation.

At the highest level, type reconstruction can be called as a function that takes the context U of unknowns and a judgment and returns the following:

- a (possibly partial) substitution that provides solutions for the unknowns,
- a (possibly empty) list of typing errors,
- a (possibly empty) list of remaining delayed constraints that could not be proved.

The implementation uses a few optimizations that are noteworthy because they are specific to the nature of our algorithm: the strong abstraction barrier between our algorithm and the individual rules makes it non-trivial for them to share auxiliary knowledge.

An idiosyncrasy of Scala allows for an elegant solution to this problem: Because Scala compiles to the JVM, it does not offer native inductive types. Instead it codes inductive types as certain groups of class declarations. This has the—usually ignored—effect that the Scala objects representing MMT expressions can carry stateful data that is ignored by pattern-matching and equality function. We make use of this by allowing any MMT component or rule to attach arbitrary data to any subexpression.

In particular, our type reconstruction algorithm uses this feature in two ways.

Firstly, expressions cannot be fully simplified right away because

- exhaustive expansion of definitions would be inefficient,
- some simplification rules only become applicable once an unknown has been solved.

Therefore, expressions must be simplified step by step as needed. To avoid unnecessary repeated traversals, we attach a boolean values that marks whether a subexpression has already been fully simplified.

Secondly, type inference must be called in many places, often multiple times. For example, it may happen that type inference gets stuck due to an unsolved unknown after successfully inferring the types of some subexpressions. Or a rule may want to infer a type that has already been inferred by a different rule. To avoid having to re-infer those types, we attach to each subexpression its inferred type.

Lexing and Parsing Type reconstruction is implemented independently of parsing. But the MMT parser is the typical source of MMT terms that contain unknowns: the parser uses the notations to determine the positions of implicit arguments and omitted variable types and inserts one fresh meta-variables for each.

Occasionally, theories must declare lexing rules as well, most importantly when using literals (e.g., for integers). MMT literals are treated in [Rab15] and omitted them here, but our implementation of type reconstruction supports them as well.

Module System The MMT module system [RK13] permits building large theories using union, instantiation, and translation. The module systems can be used uniformly for the theories that represent logical frameworks (as LF in Ex. 2.1), those that represent object logics (as in Ex. 2.2), and those that represent developments in these object logics.

Even though rules are implemented as Scala objects, the information whether a rule should be used is carried by special declarations in theories. For example, the full MMT theory for LF contains not only the declarations from Ex. 2.1 but also one reference to each s-rule from Sect. 5. That way every MMT context gives rise to the set of rules that are in scope—these are the ones that are used whenever an algorithm is run.

Importantly, this makes it possible to build logical frameworks modularly: language features are represented as theories that declare rules, and the usual module system operations are used to combine features. It is not guaranteed that combining features yields reasonable sets of rules, e.g., two terminating sets of rules may become non-terminating when used together. But for example, dependent function types (LF), product types, shallow polymorphism, rewriting, sequences, and locks are all orthogonal framework features. They are defined in separate theories that can be combined arbitrarily, and we obtain type reconstruction immediately for each combination.

Moreover, the module system is orthogonal to type reconstruction: when implementing individual rules, modularity is transparent to the developer. That means the language designer defining a logic in MMT is not aware (and does not have to worry about the fact) that MMT immediately yields a module system for the new language.

Substitution Substitution is a primitive function in MMT, and every rule may call substitution as needed, e.g., for β-reduction. By default, MMT applies substitutions immediately, which is well-known to be inefficient. To optimize substitution, MMT does two things.

Firstly, every subterm caches its free variables so that substitution only recurses when necessary. MMT also supports structure-sharing. For example, if x occurs twice in $E(x)$, the term $E(t)$ is not duplicated when applying the substitution. Moreover, substitution respects structure-sharing: during a subsequent substitution into $E(t)$, t is only traversed once and not duplicated.

Secondly, MMT allows plugins to switch out the substitution function. MMT itself provides a few advanced implementations, e.g., to memoize substitution application. For example, it is possible to define a language for explicit substitutions in MMT so that substitution becomes a constant-time operation. This requires introducing additional equality rules that apply the explicit substitutions when necessary.

User Interface MMT includes an IDE-style user interface [Rab14a] based on the jEdit text editor. It includes several advanced features such as context-sensitive auto-completion, interactive type inference, and change management. The interface makes use of the results of type reconstruction in several ways.

Firstly, the IDE shows the list of all typing errors discovered during type reconstruction. Moreover, the reconstruction algorithm traces all steps, and individual rules may insert custom messages into the trace tree. For example, the inference rule for `apply` adds a message about the expected type when calling the corresponding check. The IDE uses that to show for each error, the entire history that led to it. This is important because type reconstruction errors are often discovered late in the derivation, and early messages on the trace are often more helpful to users than later ones.

Secondly, the outline viewer shows the abstract syntax tree corresponding to the concrete syntax in the buffer. After type reconstruction is performed, the solutions of all unknowns are visible as parts of the syntax tree even though they are not present in the concrete syntax.

Thirdly, inferred types and implicit arguments are shown as tooltips when hovering over the respective part of the text in the buffer. Moreover, hovering over a selected subterm dynamically runs type inference and displays the result as a tooltip.

The IDE functionality degrades gracefully: if a declaration has typing errors, IDE functionality still works for it as much as possible. In particular, if reconstruction cannot solve all unknowns, all solutions that were found are used in the interface. This is critical because users need to interact (e.g., by inspecting the abstract syntax tree or interactively inferring the type of a subexpression) especially with those declarations that have typing errors.

8 Conclusion and Future Work

8.1 Contribution

MMT emphasizes *foundation-independent* design. The MMT language and system define an interface layer that aims at the systematic separation of concerns between (i) the small scale definition of a formal system (the foundation) by giving operators, notations, and typing rules, and (ii) the large scale features that are required for an implementation to be practical. This enables a rapid prototyping approach where developers can focus on the type theoretical and logical foundation and obtain a major implementation at extremely low cost.

The value of foundation-independence hinges on the hypothesis that most large scale feature can be realized foundation-independently, i.e., once and for all at the MMT level. The present work shows that this is possible for type reconstruction.

Concretely, we have given a foundation-independent type reconstruction algorithm that works with an arbitrary MMT theory. The design is biform in the sense of [FvM03], i.e., it combines MMT theories with small pieces of code (the rules) in the underlying programming language. The latter is used to inject foundation-specific knowledge—the typing rules—into the foundation-independent algorithm.

We applied our system to obtain implementations of various features of formal systems, including in particular LF, product types, and shallow polymorphism, as well as LF modulo, sequences, and locks. Notably the module system of MMT automatically yields type reconstruction algorithms for any modular combinations of these features.

These examples demonstrate the benefit of foundation-independence: The foundation-specific code of the rules makes up only a small fraction of the overall foundation-independent code base of MMT. Moreover, our type reconstruction algorithm is easily integrated with other foundation-independent features of MMT such as parsing, module system, and IDE.

8.2 Future Work

Rules for Other Language Features We expect that our approach can be applied to much more complex formal systems than the examples given in this paper.

For example, we already started generalizing existing reconstruction algorithms that support undecidable subtyping to be foundation-independent. We have also given the typing rules for record types and are working on inductive types.

Unification Hints For checking the judgment $E = E' : A$ in our algorithm, many formal systems only need to supply rules for specific complex types A, e.g., the extensionality rule for function types. But our implementation also supports giving rules for specific pairs (E, E'). We have so far only used this for congruence reasoning, but it also subsumes unification rules in the style of canonical instances [GGMR09] or unification hints [ARCT09].

This opens the door to generalizing our algorithm to non-unique reconstruction, where hints choose among multiple possible solutions. We are currently investigating how to specify foundation-independently (i) what a unification hint is and (ii) how unifiers should be chosen.

Abstraction We plan to improve our algorithm with Twelf-style abstraction. The type reconstruction algorithms in the systems of the Twelf [PS99] family allow for input with free variables. These are abstracted at the outside once their types are inferred. For example, in Fig. 3, we have to tediously bind A and B in multiple declarations, whereas they could simply remain free in Twelf—the Π-binding at the outside would be introduced by abstraction. But abstraction is nontrivial because the inferred types may themselves contain unknowns (and so on) that must be abstracted as well.

We have experimentally added this feature to our algorithm. However, some details are unclear, most importantly the question in which order the variables should be bound. This order is unspecified in Twelf, and that choice interacts badly with a module system as we discovered in [RS09].

Theorem Proving The author conjectures that even theorem proving, which can be seen as the next big step after type reconstruction, is amenable to a foundation-independent solution. Indeed, existing systems have already demonstrated that critical aspects of theorem proving are foundation-independent. These include the tactic language in, e.g., Isabelle [Wen99], the integration of decision procedures in, e.g., PVS [ORS92], or the use of external "hammering" tools [BKPU16].

References

[ACTZ06] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Crafting a Proof Assistant. In T. Altenkirch and C. McBride, editors, *TYPES*, pages 18–32. Springer, 2006.

[AHMP92] A. Avron, F. Honsell, I. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

[ARCT09] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 84–98. Springer, 2009.

[ARCT12] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[BCH12] M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

[Ber90] S. Berardi. *Type dependence and constructive mathematics*. PhD thesis, Dipartimento di Matematica, Università di Torino, 1990.

[BKPU16] J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[CD07] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117. Springer, 2007.

[Coq15] Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

[DGCT15] C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: Fast, Embeddable, lambda-Prolog Interpreter. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 460–468, 2015.

[dMAKR15] L. de Moura, J. Avigad, S. Kong, and C. Roux. Elaboration in dependent type theory, 2015. https://arxiv.org/abs/1505.04324.

[dMKA+15] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In A. Felty and A. Middeldorp, editors, *Automated Deduction*, pages 378–388. Springer, 2015.

[FvM03] W. Farmer and M. von Mohrenschildt. An Overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38(1–3):165–191, 2003.

[Gac08] A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of IJCAR 2008*, pages 154–161, 2008.

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

[GM08] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HKR11] F. Horozal, M. Kohlhase, and F. Rabe. Extending OpenMath with Sequences. In A. Asperti, J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics, Work-in-Progress Proceedings*, pages 58–72. University of Bologna, 2011.

[HLMS17] F. Honsell, L. Liquori, P. Maksimovic, and I. Scagnetto. Llfp: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads. *Logical Methods in Computer Science*, 2017. to appear.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[IR12] M. Iancu and F. Rabe. Management of Change in Declarative Languages. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 325–340. Springer, 2012.

[KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see `https://trac.omdoc.org/LATIN/`.

[KŞ06] M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.

[KU15] C. Kaliszyk and J. Urban. HOL(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[Lut01] M. Luther. More On Implicit Syntax. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning*, pages 386–400. Springer, 2001.

[MP08] J. Meng and L. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[Nor05] U. Norell. The Agda WiKi, 2005. `http://wiki.portal.chalmers.se/agda`.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[PD10] B. Pientka and J. Dunfield. A Framework for Programming and Reasoning with Deductive Systems (System description). In *International Joint Conference on Automated Reasoning*, 2010. To appear.

[Pie13] B. Pientka. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *J. Funct. Program*, 23(1):1–37, 2013.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction*, pages 202–206, 1999.

[PS08] A. Poswolsky and C. Schürmann. System Description: Delphin - A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008.

[Rab09] F. Rabe. The MMT Language. 2009.

[Rab12] F. Rabe. A Query Language for Formal Mathematical Libraries. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 142–157. Springer, 2012.

[Rab14a] F. Rabe. A Logic-Independent IDE. In C. Benzmüller and B. Woltzenlogel Paleo, editors, *Workshop on User Interfaces for Theorem Provers*, pages 48–60. Elsevier, 2014.

[Rab14b] F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 2014. doi:10.1093/logcom/exu079.

[Rab15] F. Rabe. Generic Literals. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 102–117. Springer, 2015.

[RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

[Wen99] M. Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, pages 167–184. Springer, 1999.