Type Inference Logics

DENIS CARNIER, KU Leuven, Belgium
FRANÇOIS POTTIER, Inria, France
STEVEN KEUCHEL, Vrije Universiteit Brussel, Belgium

Type inference is essential for statically-typed languages such as OCaml and Haskell. It can be decomposed into two (possibly interleaved) phases: a generator converts programs to constraints; a solver decides whether a constraint is satisfiable. Elaboration, the task of decorating a program with explicit type annotations, can also be structured in this way. Unfortunately, most machine-checked implementations of type inference do not follow this phase-separated, constraint-based approach. Those that do are rarely executable, lack effectful abstractions, and do not include elaboration.

To close the gap between common practice in real-world implementations and mechanizations inside proof assistants, we propose an approach that enables modular reasoning about monadic constraint generation in the presence of elaboration. Our approach includes a domain-specific base logic for reasoning about metavariables and a program logic that allows us to reason abstractly about the meaning of constraints. To evaluate it, we report on a machine-checked implementation of our techniques inside the Coq proof assistant. As a case study, we verify both soundness and completeness for three elaborating type inferencers for the simply typed λ-calculus with Booleans. Our results are the first demonstration that type inference algorithms can be verified in the same form as they are implemented in practice: in an imperative style, modularly decomposed into constraint generation and solving, and delivering elaborated terms to the remainder of the compiler chain.

CCS Concepts: • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: program verification, type inference, elaboration

ACM Reference Format:
Denis Carnier, François Pottier, and Steven Keuchel. 2024. Type Inference Logics. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 346 (October 2024), 31 pages. https://doi.org/10.1145/3689786

1 Introduction

The goal of *type inference* is to determine whether a program is well-typed, and if it is, to compute its most general type. Type inference is often accompanied or followed by *elaboration*, whose purpose is to construct an explicitly-typed program. Naturally, type inference has long been the subject of formal study [Curry and Feys 1958]. In his seminal paper on the programming language ML, Milner [1978] presents two type inference algorithms, $\mathcal{J}$ and $\mathcal{W}$, and proves that $\mathcal{W}$ is *sound*: the type inferred by $\mathcal{W}$ is a valid type for this program. Four years later, Damas and Milner [1982] prove that $\mathcal{W}$ is *complete*: if there is a valid type for this program then $\mathcal{W}$ infers this type or a more general type [Damas 1984]. At the time, all proofs were carried out by hand, and were therefore prone to imprecision and mistakes. Since then, higher assurance has been obtained by employing proof assistants to mechanically verify the desired properties [Dubois and Ménissier-Morain 1999; Naraschewski and Nipkow 1999; Nazareth and Nipkow 1996; Urban and Nipkow

Authors' Contact Information: Denis Carnier, KU Leuven, Leuven, Belgium, denis.carnier@kuleuven.be; François Pottier, Inria, Paris, France, francois.pottier@inria.fr; Steven Keuchel, Vrije Universiteit Brussel, Brussels, Belgium, steven.keuchel@vub.be.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/10-ART346
https://doi.org/10.1145/3689786

2009]. Mechanization can reveal missing details or incorrect proofs, as illustrated, for instance, by Zhao et al. [2019].

Although it is clear, very early, that type inference involves solving equations [Milner 1978; Cardelli 1987; Wand 1987], the idea of explicitly binding type variables via existential quantifiers[1] appears only later on: Jouannaud and Kirchner [1991] explain first-order unification as a constraint rewriting process, where constraints involve equations, conjunctions, and existential quantification. Later still, Odersky et al. [1999] present a constraint-based reformulation of ML's type system. Today, some form of constraint-based type inference is in use in several production-grade compilers, including Helium [Heeren et al. 2003] and GHC [Vytiniotis et al. 2011].

Constraint-based type inference consists of two (possibly interleaved) steps, namely constraint generation and constraint solving. These steps are performed by two *separate* pieces of code that communicate via *constraints* whose logical meaning is clearly defined. In many paper presentations of constraint-based type inference, the constraint solver produces a Boolean result, namely "satisfiable" or "unsatisfiable". If the constraint is satisfiable then the source program is well-typed. Naturally, in this event, it is then desirable or necessary to perform a third step, *elaboration*, which constructs a program that carries explicit type annotations. Pottier [2014] proposes a method where these three steps are described by just two pieces of code. The generator and elaborator form one module; the solver forms another; the two parts communicate via *constraints with "semantic values"*. This separation of concerns seems particularly appealing because it is expected to make each of the two components easier to explain, evolve, re-use, and verify.

Yet, the constraint-based approach has seldom been mechanized. To the best of our knowledge, PureCake [Kanabar et al. 2023; Kanabar 2023] is the only compiler that includes a verified constraint-based type inference algorithm. While we applaud this achievement, we must also highlight some of its limitations. In particular, PureCake's type inference algorithm is proven sound, but not complete. The fact that type variables must be chosen "fresh" is not formalized or exploited, as it is not necessary for soundness. Also, while the proof of soundness exists, it is described neither in the paper [Kanabar et al. 2023] nor in Kanabar's dissertation [2023].

Realistic type inferencers (whether verified or unverified; whether constraint-based or monolithic) are usually expressed in an imperative style, either in an impure programming language, or in a pure language equipped with effectful abstractions for failure and state, such as applicatives or monads. Yet, when a monadic type inferencer is verified [Garrigue 2015; Tan et al. 2019], the monadic abstraction is often not exploited: the definitions of the monadic combinators are unfolded, and one reasons about *a concrete run* of the monadic program.

To reason about monadic programs without breaking the monadic abstraction, an alternative approach is to develop a *program logic* that associates a logical reasoning rule with every operation of the monad [Swierstra 2009; Maillard et al. 2019; Swierstra and Baanen 2019]. For instance, in their proofs of correctness and completeness of algorithm $\mathcal{W}$ and of a unification algorithm, Silva et al. [2020] use a program logic for the failure-and-state monad. However, while this program logic facilitates reasoning about failure and state, it ignores domain-specific concerns such as reasoning about the freshness of type variables. As a result, all statements have to deal with these concerns explicitly. To fully embrace this approach, it is necessary to develop a monad and a program logic that are tailored for the application domain: as argued by Nigron and Dagand [2021], "every monad deserves a dedicated program logic".

[1] An *existentially quantified constraint* $\exists\alpha.C$ expresses the idea that α stands for an as-yet-undetermined type, that is, a type whose concrete value can be guessed by the constraint solver, subject to the constraint C. This must not be confused with the concept of an *existential type* $\exists\alpha.\tau$, where the variable α represents an abstract type, that is, a type whose concrete value has intentionally been hidden. Existential types do not appear in this paper.

$$\begin{array}{lcl} e \in \text{Exp} & ::= & x \mid \lambda x.e \mid \lambda x{:}\tau.e \mid e\ e \mid \text{true} \\ & \mid & \text{false} \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \end{array} \qquad \begin{array}{lcl} \tau \in \text{Ty} & ::= & \tau \Rightarrow \tau \mid \text{bool} \\ \Gamma \in \text{Ctx} & ::= & \varepsilon \mid \Gamma, x : \tau \end{array}$$

Fig. 1. The simply typed λ-calculus with Booleans ($\lambda_{\mathbb{B}}$)

In summary, we argue that, in machine-checked implementations of type inference and elaboration, there is still an unmet need for an elegant and modular approach, one that retains and exploits the best practices of real-world implementations, including a constraint-based phase separation and the use of domain-specific monadic abstractions.

In this paper, we make several initial steps towards meeting this need. From first principles, we develop a domain-specific approach to the mechanical verification of *type inference* and *elaboration* algorithms based on constraints with semantic values. Our approach meets the following goals: (1) the programming interface uses functional abstractions; (2) the algorithms can be *extracted*, therefore executed outside of the proof assistant; (3) the implementation and proofs are *modular* and *composable* thanks to constraints; and (4) several kinds of output are supported: the output of an algorithm might be a Boolean result ("yes, the program is well-typed"), an inferred type, or an explicitly-typed program.

This paper contributes a new understanding and semantics of "constraints with semantic values". An ordinary constraint, whose meaning is a truth value (either the constraint holds, or it does not), can also be viewed as a monadic program, which either succeeds (with a unit result) or fails. This view of constraints as programs can be extended to constraints with semantic values: in this more general setting, a constraint is a monadic program which either succeeds (and produces a result of type A) or fails. Because a constraint can have several solutions, it is naturally a *non-deterministic* program, which has multiple possible results. In this setting, it seems natural to define the meaning of constraints by interpreting a constraint as a function of type $(A \to \mathbb{P}) \to \mathbb{P}$, which maps a postcondition to a truth value. This is known as a *predicate transformer semantics*. We use this semantics as the basis of a *program logic* for high-level reasoning about the meaning of constraints.

In summary, we make the following contributions:

- We specify an abstract interface for type inference monads that enforces correct by construction handling of metavariables (§3.1). We develop a constraint-based approach by providing a free monad instance (§3.1), that represents the syntax of constraints, and implement a constraint generator with elaboration for the simply typed λ-calculus with Booleans (§3.3).
- We develop a domain-specific *base logic* for reasoning about metavariables and define abstractions tailored for reasoning about weakening substitutions (§4.2).
- We define the semantics of constraints (§4.3) on top of the base logic and derive rules that form a *program logic* (§4.4) for reasoning about constraint generating functions which we use to prove the correctness of our generator (§4.5).
- We mechanize the results of this paper, and a solver based on first-order unification in the Coq proof assistant and report on the extraction to Haskell (§5).

2 Overview

In this section, we present an overview of our approach. The subject of our exposition is the simply typed λ-calculus with Booleans ($\lambda_{\mathbb{B}}$). We perform type inference and elaboration for this object language. Figure 1 describes $\lambda_{\mathbb{B}}$. Besides variables, abstraction, and application, we introduce an *if-then-else* construct and two primitive values: true and false. There are two forms of λ-abstraction, namely an implicitly-typed abstraction ($\lambda x.e$) and an explicitly-typed one ($\lambda x{:}\tau.e$).

$$\frac{(x:\tau)\in\Gamma}{\Gamma\vdash_D x:\tau\rightsquigarrow x}\ \textsc{T-Var}$$

$$\frac{\Gamma\vdash_D e_1:\tau_1\Rightarrow\tau_2\rightsquigarrow e_1' \qquad \Gamma\vdash_D e_2:\tau_1\rightsquigarrow e_2'}{\Gamma\vdash_D e_1\ e_2:\tau_2\rightsquigarrow e_1'\ e_2'}\ \textsc{T-App}$$

$$\frac{\Gamma,x:\tau_1\vdash_D e:\tau_2\rightsquigarrow e'}{\Gamma\vdash_D \lambda x.e:\tau_1\Rightarrow\tau_2\rightsquigarrow\lambda x{:}\tau_1.e'}\ \textsc{T-Abs-Implicit}$$

$$\frac{\Gamma,x:\tau_1\vdash_D e:\tau_2\rightsquigarrow e'}{\Gamma\vdash_D \lambda x{:}\tau_1.e:\tau_1\Rightarrow\tau_2\rightsquigarrow\lambda x{:}\tau_1.e'}\ \textsc{T-Abs-Explicit}$$

$$\textsc{T-True}\quad \Gamma\vdash_D \mathsf{true}:\mathsf{bool}\rightsquigarrow\mathsf{true}$$

$$\textsc{T-False}\quad \Gamma\vdash_D \mathsf{false}:\mathsf{bool}\rightsquigarrow\mathsf{false}$$

$$\frac{\Gamma\vdash_D e_1:\mathsf{bool}\rightsquigarrow e_1' \qquad \Gamma\vdash_D e_2:\tau\rightsquigarrow e_2' \qquad \Gamma\vdash_D e_3:\tau\rightsquigarrow e_3'}{\Gamma\vdash_D \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3:\tau\rightsquigarrow \textbf{if } e_1' \textbf{ then } e_2' \textbf{ else } e_3'}\ \textsc{T-If}$$

Fig. 2. Declarative typing and elaboration rules for $\lambda_{\mathbb{B}}$

$$
\begin{array}{l}
\mathsf{check}\ (\Gamma:\mathsf{Ctx})\ (e:\mathsf{Exp})\ (\tau:\mathsf{Ty}):\mathbb{P} := \textbf{match}\ e\ \textbf{with}\\
\quad |\ x \longrightarrow \textbf{if}\ (x:\tau')\in\Gamma\ \textbf{then}\ \tau=\tau'\ \textbf{else}\ \bot\\
\quad |\ \lambda x.e \longrightarrow \exists\tau_1.\exists\tau_2.\ (\tau=\tau_1\Rightarrow\tau_2)\wedge \mathsf{check}\ (\Gamma,x:\tau_1)\ e\ \tau_2\\
\quad |\ \lambda x{:}\tau_1.e \longrightarrow \exists\tau_2.\ (\tau=\tau_1\Rightarrow\tau_2)\wedge \mathsf{check}\ (\Gamma,x:\tau_1)\ e\ \tau_2\\
\quad |\ e_1\ e_2 \longrightarrow \exists\tau'.\ \mathsf{check}\ \Gamma\ e_1\ (\tau'\Rightarrow\tau)\wedge \mathsf{check}\ \Gamma\ e_2\ \tau'\\
\quad |\ \mathsf{false} \longrightarrow \tau=\mathsf{bool} \qquad |\ \mathsf{true}\longrightarrow\tau=\mathsf{bool}\\
\quad |\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \longrightarrow \mathsf{check}\ \Gamma\ e_1\ \mathsf{bool}\ \wedge\\
\qquad\qquad \mathsf{check}\ \Gamma\ e_2\ \tau\wedge \mathsf{check}\ \Gamma\ e_3\ \tau
\end{array}
$$

Fig. 3. Typing as a proposition generator

$$
\begin{array}{l}
\textbf{class}\ \mathsf{CstrM}\ (M:\mathsf{Type}\longrightarrow\mathsf{Type})\\
\quad \mathsf{pure} : A\longrightarrow M\ A\\
\quad (\gg\!=) : M\ A\longrightarrow\\
\qquad\qquad (A\longrightarrow M\ B)\longrightarrow M\ B\\
\quad \mathsf{fail} : M\ A\\
\quad (\sim) : \mathsf{Ty}\longrightarrow\mathsf{Ty}\longrightarrow M\ ()\\
\quad \mathsf{pick} : M\ \mathsf{Ty}
\end{array}
$$

Fig. 4. Monad interface for constraints with semantic values

Figure 2 defines the 4-place judgement $\Gamma\vdash_D e:\tau\rightsquigarrow e'$. This judgement, known as *declarative typing with elaboration*, states that in the typing context Γ the expression e admits the type τ and that its elaborated form is the expression e'. Here, e' is a fully type-annotated expression in the same object language: thus, in this paper, elaboration is just *type reconstruction.* In other settings, elaboration can also involve a translation of the source program into an intermediate language.

In the remainder of this overview, we first define a generator that produces constraints, viewed as propositions (of type $\mathbb{P}$) in the metalanguage (§2.1). These constraints express just the typability problem; they do not encode the elaboration process. The generator is a recursive function that translates a candidate typing judgement (Γ, e, τ) to a proposition. Subsequently, we introduce a monadic API for *constraints with semantic values* (§2.2). This allows the generator to produce constraints whose result is not just a truth value, but can be a more complex object, such as an elaborated expression. We provide a concrete implementation of this API in the form of a free monad (§2.3). We conclude this overview with a predicate transformer semantics for constraints, which we use to express the soundness and the completeness of the generator (§2.4).

2.1 Propositional Constraints

The *type checking* problem has a straightforward formulation: when Γ, e, τ are given, one must decide whether the typing judgement $\Gamma\vdash_D e:\tau$ holds. A constraint-based type-checker traverses the input program e to produce a constraint that is logically equivalent to the typing judgement $\Gamma\vdash_D e:\tau$. In this subsection, a constraint is just a meta-level proposition of type $\mathbb{P}$.

Figure 3 gives the implementation of a constraint generator check. It is defined by induction over the input program e. It does not analyze the shape of the candidate type τ. This is intentional:

beginning in the next section (§3), types may contain unification variables, whose shape is not yet known. Instead, check uses existentially quantified variables, equality assertions, and conjunction to express under what condition the candidate typing judgement holds. To illustrate this, consider the case of an implicitly-typed abstraction $\lambda x.e$. Such an abstraction is well-typed only when the candidate type τ is a function type. This is expressed by requiring τ to be of the form $\tau_1 \Rightarrow \tau_2$ for some types τ_1, τ_2. Furthermore, the function body e is required to admit the type τ_2 in the extended context $(\Gamma, x : \tau_1)$.

This constraint generator is sound and complete: that is, the constraint that it constructs is valid (true) if and only if the candidate typing judgement holds. This is stated by the following theorem:

Theorem 2.1 (Propositional generator correctness). $\text{check}\ \Gamma\ e\ \tau \leftrightarrow \Gamma \vdash_D e : \tau$.

Proof. Soundness ($\rightarrow$) is established by induction over the expression e, unpacking conjunctions and existentials in induction hypotheses, and applying the introduction rules of the typing relation. Completeness ($\leftarrow$) is established by induction over the typing judgement. Thus, there is no need for typing-judgement-inversion lemmas. □

2.2 Constraints with Semantic Values: Monadic API

Whereas the type checking problem calls for a Boolean answer (either the program is well-typed, or it is not), the *type synthesis* problem and the *elaboration* problem call for more complex answers, namely an inferred type and an explicitly-typed term. To address these problems, Pottier [2014] proposes *constraints with semantic values.*[2] In this setting, a constraint of type $\mathbb{CV}\ A$ represents at the same time a constraint in the ordinary sense—that is, a logical problem that must be solved—and a program whose result, or "semantic value", has type A. The semantic value may depend on the solution of the constraint: in particular, when a constraint involves an existentially quantified variable, its semantic value may depend on the value assigned to this variable by the constraint solver. We make this more precise in §3.

In this paper, we adopt and adapt Pottier's approach [2014]. However, whereas his constraints offer an applicative interface, ours offer a monadic interface. In Figure 3, to build constraints, we have used truth and falsity, equality between object-language types, conjunction, and existential quantification. This suggests that *constraints with semantic values* should also support these features. We achieve this by proposing an abstract interface for *constraint monads.* This interface takes the form of a type class, CstrM; it appears in Figure 4. The standard monadic combinators are pure, which expresses truth and allows returning a semantic value, and ($\gg=$), which can express conjunction. The failure combinator fail expresses falsity. The equality combinator ($\sim$) imposes an equation between two object-language types: if its arguments are equal, then it succeeds and returns a unit value; otherwise, it fails. The non-deterministic angelic choice combinator pick chooses an object-language type in such a way that subsequent equalities hold and failure is avoided. Together with ($\gg=$), it can simulate existential quantification.

Using this interface, we write a monadic constraint generator for $\lambda_{\mathbb{B}}$ (Figure 5). It synthesizes a type and performs elaboration. Just like the propositional constraint generator in Figure 3, it is defined by induction over the expression, and avoids pattern matching on object-language types.

2.3 Constraints with Semantic Values: Implementation as a Free Monad

In Figure 6, we define an instance of the type class CstrM. It is a free monad: it defines the syntax of the operations and does not specify their semantics. Following a standard recipe [Hancock and Setzer 2000], the operations pure, fail, ($\sim$), and pick become constructors of the inductive data

[2] The terminology "semantic value" is borrowed from the world of parsing.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{synth}\ (\Gamma : \text{Ctx})\ (e : \text{Exp}) : M\ (\text{Ty} \times \text{Exp}) := \textbf{match}\ e\ \textbf{with}} \\
\mid x \longrightarrow \textbf{if}\ (x : \tau) \in \Gamma\ \textbf{then}\ \text{pure}\ (\tau,\ x)\ \textbf{else}\ \text{fail} & \mid \lambda x.e \longrightarrow \tau_1 \leftarrow \text{pick};\ \tau_2, e' \leftarrow \text{synth}\ (\Gamma, x : \tau_1)\ e;\ \text{pure}\ (\tau_1 \Rightarrow \tau_2,\ \lambda x{:}\tau_1.e') \\
\mid \lambda x{:}\tau_1.e \longrightarrow \tau_2, e' \leftarrow \text{synth}\ (\Gamma, x : \tau_1)\ e;\ \text{pure}\ (\tau_1 \Rightarrow \tau_2,\ \lambda x{:}\tau_1.e') & \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \longrightarrow \\
\mid e_1\ e_2 \longrightarrow \tau_1, e_1' \leftarrow \text{synth}\ \Gamma\ e_1 & \quad \tau_1, e_1' \leftarrow \text{synth}\ \Gamma\ e_1 \\
\quad \tau_2, e_2' \leftarrow \text{synth}\ \Gamma\ e_2 & \quad \tau_2, e_2' \leftarrow \text{synth}\ \Gamma\ e_2 \\
\quad \tau_3 \leftarrow \text{pick} & \quad \tau_3, e_3' \leftarrow \text{synth}\ \Gamma\ e_3 \\
\quad \tau_1 \sim \tau_2 \Rightarrow \tau_3 & \quad \tau_1 \sim \text{bool} \\
\quad \text{pure}\ (\tau_3,\ e_1'\ e_2') & \quad \tau_2 \sim \tau_3 \\
\mid \text{true} \longrightarrow \text{pure}\ (\text{bool},\ \text{true}) & \quad \text{pure}\ (\tau_2,\ \textbf{if}\ e_1'\ \textbf{then}\ e_2'\ \textbf{else}\ e_3') \\
\mid \text{false} \longrightarrow \text{pure}\ (\text{bool},\ \text{false}) &
\end{array}
$$

Fig. 5. Monadic constraint generation with synthesis and elaboration

$$
\begin{array}{l}
\textbf{data}\ \text{Free}\ (A : \text{Type}) : \text{Type} := \\
\quad \text{Pure}\ (a : A) \\
\quad \text{Fail} \\
\quad \text{Eq}\ (\tau_1\ \tau_2 : \text{Ty})\ (k : \text{Free}\ A) \\
\quad \text{Pick}\ (k : \text{Ty} \rightarrow \text{Free}\ A)
\end{array}
$$

Fig. 6. Free monad definition

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{WP} :\ \text{Free}\ A \rightarrow (A \rightarrow \mathbb{P}) \rightarrow \mathbb{P} :=} \\
\text{WP}\ (\text{Pure}\ a)\ Q & := Q\ a \\
\text{WP}\ (\text{Fail})\ Q & := \bot \\
\text{WP}\ (\text{Eq}\ \tau_1\ \tau_2\ k)\ Q & := \tau_1 = \tau_2 \land \text{WP}\ k\ Q \\
\text{WP}\ (\text{Pick}\ k)\ Q & := \exists \tau. \text{WP}\ (k\ \tau)\ Q
\end{array}
$$

Fig. 7. Weakest preconditions for the free monad

type Free A. Each constructor carries the operation's inputs as well as a continuation k, which expects the operation's output as an argument. (In Fail, the continuation has type $0 \rightarrow \text{Free}\ A$, so we remove it altogether. In Eq, the continuation has type $1 \rightarrow \text{Free}\ A$, which we simplify to just Free A.) The "bind" operation is not a constructor; it is a function, which is defined by induction over its first argument.

The higher-order combinator Pick expects a function of the metalanguage as an argument. In essence, we arrive at a higher-order-abstract-syntax representation [Pfenning and Elliott 1988] of constraints, where metalanguage variables are used to represent existentially quantified object-language types. This representation may seem elegant, but is in fact quite impractical: to inspect the body of an existentially quantified constraint (that is, the continuation of a Pick operation), one must first supply a type! This makes it difficult to define a constraint solver. A naïve solver would enumerate all possible types at each Pick: it would be very impractical, and not even guaranteed to terminate. To work around this problem, in §3, we propose a first-order representation for existential variables.

2.4 Generator Correctness

To state and prove correctness of the generator, we wish to assign a semantics to a constraint of type Free A. In other words, we want to interpret a constraint as a proposition that expresses that the constraint holds, and describes what semantic value the constraint might produce.

The function WP in Figure 7 offers such an interpretation. If C is a constraint of type Free A and if Q is a postcondition of type $A \rightarrow \mathbb{P}$ then the proposition WP C Q means that the non-deterministic choices inherent in the constraint C *can* be settled in such a way that failure is avoided and the constraint's semantic value satisfies Q. In other words, WP C Q means that there exists a value

$a : A$ such that C can produce a and a satisfies Q. More succinctly, one might say that "some result of C satisfies Q" or that "C can satisfy Q".

The function WP is sometimes referred to as a *predicate transformer semantics* [Swierstra and Baanen 2019], because if f has type $A \to \text{Free } B$ then the function $\lambda Q\ a.\ \text{WP}\ (f\ a)\ Q$, whose type is $(B \to \mathbb{P}) \to (A \to \mathbb{P})$, maps a postcondition to a precondition. This explains why it is named WP, for *weakest precondition*.

We combine the function WP and the constraint generator synth to define an *algorithmic* variant of the typing relation:

$$\Gamma \vdash_A e : \tau_1 \rightsquigarrow e'_1 \;:=\; \text{WP}\ (\text{synth}\ \Gamma\ e)\ (\lambda(\tau_2, e'_2).\ \tau_1 = \tau_2 \land e'_1 = e'_2)$$

This judgement states that the constraint $\text{synth}\ \Gamma\ e$ can produce the semantic value (τ_1, e'_1). In other words, it states that one of its successful execution paths produces exactly the desired type and elaborated expression. The correctness of the constraint generator can then be stated as a logical equivalence between declarative and algorithmic typing:

Theorem 2.2 (Constraint generator correctness). $\Gamma \vdash_A e : \tau_1 \rightsquigarrow e' \leftrightarrow \Gamma \vdash_D e : \tau_1 \rightsquigarrow e'$

In the completeness direction ($\leftarrow$), the proof is by induction over the declarative typing judgement. The goal is a *total correctness* statement: under the assumption that the typing judgement holds, the computation must be able to succeed and to satisfy its postcondition. Because the judgement $\text{WP}\ C\ Q$ is in fact a Hoare logic, the proof requires Hoare-style reasoning rules for WP, including a sequencing rule, a consequence rule, and so on. These rules are presented later on.

The soundness direction ($\to$) is best dealt with by first reformulating the statement. The original statement has the shape $\text{WP}\ C\ P \to Q$, which means "if there exists an execution of C that satisfies P, then Q". This is equivalent to "every execution of C satisfies $P \to Q$".[3] This statement can be written under the form $\text{WLP}\ C\ (P \to Q)$, where the judgement $\text{WLP}\ C\ Q$ means that the constraint C *must* produce a value that satisfies Q, or in other words, that *every* successful execution of the program C produces a value that satisfies Q.

A direct definition of this judgement appears in Figure 8. It is an alternative predicate transformer semantics; the name WLP stands for *weakest liberal precondition*. The two semantics are related via the equivalence $(\text{WP}\ C\ P \to Q) \;\leftrightarrow\; (\text{WLP}\ C\ (P \to Q))$, which is proved by induction on C.

Thus, the soundness direction of Theorem 2.2 is reformulated as follows:

Lemma 2.3 (Constraint generator soundness). $\text{WLP}\ (\text{synth}\ \Gamma\ e)\ (\lambda(\tau, e').\ \Gamma \vdash_D e : \tau \rightsquigarrow e')$

The proof of this reformulated statement is more pleasant than a direct attempt at proving the original statement: indeed, instead of *deconstructing* a WP hypothesis, we must now *construct* a WLP statement. This can be done with the help of suitable Hoare-style reasoning rules for WLP.

A WLP judgement is a *partial correctness* statement: if the computation succeeds, then the postcondition must hold. Since our programs always terminate, here, the word "partial" refers only to the possibility of failure, as opposed to the possibility of divergence. Thus, perhaps it would be more accurate to refer to WLP as a *weak total correctness* judgement [Apt 1983]. In general, it is not possible to establish an equivalence between partial and total correctness judgements, as done above; in our setting, however, such an equivalence holds.

Noteworthy is that we have just established the correctness of a constraint generator (which performs type synthesis and elaboration) in isolation—that is, independently of a constraint solver. However, to actually produce a value, an executable solver is necessary. In the rest of this paper, we define a solver for a different variable representation, and integrate it with the generator to obtain a verified generator-solver combination.

[3] We take the liberty of writing $P \to Q$ for $\lambda a.(P\ a \to Q)$.

$$
\begin{aligned}
&\mathsf{WLP} : \mathsf{Free}\ A \rightarrow (A \rightarrow \mathbb{P}) \rightarrow \mathbb{P} := \\
&\mathsf{WLP}\ (\mathsf{Pure}\ a)\ Q \quad := Q\ a \\
&\mathsf{WLP}\ (\mathsf{Fail})\ Q \quad := \top \\
&\mathsf{WLP}\ (\mathsf{Eq}\ \tau_1\ \tau_2\ k)\ Q := \tau_1 = \tau_2 \rightarrow \mathsf{WLP}\ k\ Q \\
&\mathsf{WLP}\ (\mathsf{Pick}\ k)\ Q \quad := \forall \tau. \mathsf{WLP}\ (k\ \tau)\ Q
\end{aligned}
$$

Fig. 8. Weakest liberal preconditions for Free

$$
\begin{aligned}
&\alpha, \beta : \mathsf{Evar} \\
&w \quad : \mathsf{World} ::= \epsilon \mid w, \alpha \\
&\hat{\tau} \quad : \hat{\mathsf{Ty}}\ w \quad ::= \alpha\ (\text{if } \alpha \in w) \mid \hat{\mathsf{bool}} \mid \hat{\tau} \Rightarrow \hat{\tau} \\
&\hat{\Gamma} \quad : \hat{\mathsf{Ctx}}\ w ::= \varepsilon \mid \hat{\Gamma}, x : \hat{\tau}\ (\text{if } \hat{\tau} \in \hat{\mathsf{Ty}}\ w) \\
&\lceil _ \rceil : \forall w.\ A \rightarrow \hat{A}\ w
\end{aligned}
$$

Fig. 9. World-indexed types

3 Monadic Constraint Generation

In the previous section, we used higher-order abstract syntax to express existential quantification, but we remarked that this prevents us from implementing a constraint solver (§2.3). We now remedy this problem and develop a *first-order* abstract syntax of constraints with semantic values. Based on this abstract syntax, we implement a new constraint generator with elaboration for $\lambda_{\mathbb{B}}$. From the previous section (§2), we keep just the definition of $\lambda_{\mathbb{B}}$; we redefine everything else from scratch.

In implementations of type inference, constraints must be *well-scoped*: all type variables must be bound by an existential quantifier. The fact that constraints are well-scoped can be either established *a posteriori* or guaranteed *a priori*, by construction. In this paper, we choose the latter approach: we impose a discipline of intrinsic well-scoping [Altenkirch and Reus 1999; Benton et al. 2012]. While our exposition makes use of this intrinsically-scoped representation, it is not a requirement for reproducing our approach. Moreover, we keep the choice of the underlying variable representation opaque. For a more in-depth discussion of the benefits and drawbacks, we refer the reader to §5.

To mix monadic computations with intrinsic scoping of generated existential variables, we build on existing ideas to index monadic computations [McBride 2011] to keep track of dynamic allocations, specifically existing work to model dynamic allocation of reference cells [Bach Poulsen et al. 2017; Rouvoet 2021] and symbolic variables [Keuchel et al. 2022]. The following exposition adapts these ideas to our setting.

In the remainder of this section, we develop intrinsically-scoped constraints with semantic values. In §3.1, we define open versions of object-language types, typing contexts, and constraints. (An "open" syntactic object can contain type variables.) As before, constraints form a free monad. We use the structure of the free monad to motivate a more general abstract interface. In §3.2, we introduce the type constructor Open. This type constructor turns a closed type, such as the type of expressions, into an open version of this type, without requiring a duplication of its definition. Finally, we present the constituents of a type inference and elaboration algorithm: a constraint generator (§3.3), a conversion to prenex normal form (§3.4), and a constraint solver (§3.5). We compose these components (§3.6) to obtain an end-to-end algorithm.

3.1 Constraints with Semantic Values

For the intrinsically-scoped representation, we index our types with sets of existential variables,[4] called *worlds* (Figure 9). We write $\hat{\mathsf{Type}}$ as a short-hand for $\mathsf{World} \rightarrow \mathsf{Type}$. Figure 9 defines open object-language types $\hat{\mathsf{Ty}} : \hat{\mathsf{Type}}$ and open typing contexts $\hat{\mathsf{Ctx}} : \hat{\mathsf{Type}}$. We use a circumflex to visually differentiate these types of open objects from their closed counterparts Ty and Ctx. The difference between Ty and $\hat{\mathsf{Ty}}\ w$ is that an inhabitant of Ty is a closed object-language type, whereas an inhabitant of $\hat{\mathsf{Ty}}\ w$ can be a type variable $\alpha \in w$. The function $\lceil _ \rceil$ translates a closed type (resp. context) to an open type (resp. context).

[4] The order in which existential variables appear is not important in this simple setting, but would matter if the constraint language was richer and involved both universally and existentially bound type variables.

$$\begin{aligned}
&\textbf{data}\ \hat{\text{Free}}\ (A : \hat{\text{Type}})\ (w : \text{World}) : \text{Type} := \\
&\quad \hat{\text{Pure}}\ (a : A\ w) \\
&\quad \hat{\text{Fail}} \\
&\quad \hat{\text{Eq}}\ (\hat{\tau}_1\ \hat{\tau}_2 : \hat{\text{Ty}}\ w)\ (k : \hat{\text{Free}}\ A\ w) \\
&\quad \hat{\text{Pick}}\ (\alpha : \text{Evar})\ (k : \hat{\text{Free}}\ A\ (w, \alpha))
\end{aligned}$$

Fig. 10. Definition of the free monad

$$\begin{aligned}
(\sqsubseteq) &: \text{World} \to \text{World} \to \text{Type} \\
\text{new}_\alpha &: w \sqsubseteq w, \alpha \\
\text{refl} &: w \sqsubseteq w \\
\text{trans} &: w_0 \sqsubseteq w_1 \to w_1 \sqsubseteq w_2 \to w_0 \sqsubseteq w_2 \\
_[_] &: A\ w \to w \sqsubseteq w' \to A\ w'
\end{aligned}$$

Fig. 11. Parallel substitutions

$$\begin{aligned}
\models A &:= \forall\{w\}.\ A\ w \\
A \twoheadrightarrow B &:= \lambda w.\ A\ w \to B\ w \\
\Box A &:= \lambda w.\ \forall\{w'\}. w \sqsubseteq w' \to A\ w' \\
\Diamond A &:= \lambda w.\ \exists\{w'\}. w \sqsubseteq w' \times A\ w' \\
\text{Const}\ A &:= \lambda w.\ A
\end{aligned}$$

Fig. 12. Notations

$$\begin{aligned}
&(\gg\!=_{\text{F}}) : \models \hat{\text{Free}}\ A \twoheadrightarrow \Box(A \twoheadrightarrow \hat{\text{Free}}\ B) \twoheadrightarrow \hat{\text{Free}}\ B \\
&\hat{\text{Pure}}\ a \gg\!=_{\text{F}} f := f\ \text{refl}\ a \\
&\hat{\text{Fail}} \gg\!=_{\text{F}} f := \hat{\text{Fail}} \\
&\hat{\text{Eq}}\ \hat{\tau}_1\ \hat{\tau}_2\ k \gg\!=_{\text{F}} f := \hat{\text{Eq}}\ \hat{\tau}_1\ \hat{\tau}_2\ (k \gg\!=_{\text{F}} f) \\
&\hat{\text{Pick}}\ \alpha\ k \gg\!=_{\text{F}} f := \hat{\text{Pick}}\ \alpha\ (k \gg\!=_{\text{F}} f[\text{new}_\alpha])
\end{aligned}$$

Fig. 13. Free monad bind

We assume the existence of a function fresh : World $\to$ Evar which deterministically computes a fresh name: that is, fresh $w \notin w$. We explicitly use this function whenever we introduce a new existential variable. However, we leave corresponding freshness side-conditions implicit.

Free Monad. Figure 10 gives a new definition of the free monad. With respect to Figure 6, the main difference is that the $\hat{\text{Pick}}$ constructor now carries a variable α, which is considered bound in k. The "continuation" k is no longer a meta-level function: it is just a constraint.

Non-deterministically coming up with a type is implemented by generating a fresh variable:

$$\hat{\text{pick}}_{\text{F}} : \forall w.\ \hat{\text{Free}}\ \hat{\text{Ty}}\ w := \lambda w.\ \textbf{let}\ \alpha := \text{fresh}\ w\ \textbf{in}\ \hat{\text{Pick}}\ \alpha\ (\hat{\text{Pure}}\ \alpha)$$

As it turns out, defining a bind operation for this monad proves to be more subtle. Consider, defining it with the usual type

$$\forall w.\ \hat{\text{Free}}\ A\ w \to (A\ w \to \hat{\text{Free}}\ B\ w) \to \hat{\text{Free}}\ B\ w.$$

Unfortunately, this definition is not possible. The value of type A in the left-hand side may appear under some binders ($\hat{\text{Pick}}$), and may refer to those variables. Hence, bringing it into world w to pass to the continuation would mean those variables escape their scope. We circumvent this issue by also passing the world w' of A to the continuation, and moreover, a weakening substitution $\theta : w \sqsubseteq w'$ to transport other values to this new world. The type of bind becomes [Bach Poulsen et al. 2017; Keuchel et al. 2022; Rouvoet 2021]:

$$\forall w.\ \hat{\text{Free}}\ A\ w \to (\forall w'. w \sqsubseteq w' \to A\ w' \to \hat{\text{Free}}\ B\ w') \to \hat{\text{Free}}\ B\ w.$$

In this paper, we mainly consider parallel substitutions, i.e. that simultaneously substitute all type variables in world w by types in world w'. Figure 11 shows the interface that we are assuming. In particular, new_α represents a weakening that brings a new variable α into scope. We write the compositions more succinctly by juxtapositions. Moreover, we use post-application (_[_]) of substitutions to open types A that have a substitution function defined, which include $\hat{\text{Ty}}$, $\hat{\text{Ctx}}$ etc. We do not require that all indexed types $\hat{\text{Type}}$ admit a substitution function.

As is evident from the type of bind above, the explicit world passing creates a lot of noise. We therefore define some type constructors in Figure 12 that hide the *plumbing*. Concretely, validity

$$
\begin{aligned}
&\textbf{class}\ \hat{\text{C}}\text{strM}\ (M : \hat{\text{Type}} \to \hat{\text{Type}})\\
&\quad \hat{\text{pure}} \ : \models A \twoheadrightarrow M\ A\\
&\quad (\ggg\!=) : \models M\ A \twoheadrightarrow \Box(A \twoheadrightarrow M\ B) \twoheadrightarrow M\ B\\
&\quad \hat{\text{fail}} \ : \models M\ A\\
&\quad (\sim) \ : \models \hat{\text{Ty}} \twoheadrightarrow \hat{\text{Ty}} \twoheadrightarrow M\ ()\\
&\quad \hat{\text{pick}} \ : \models M\ \hat{\text{Ty}}
\end{aligned}
$$

Fig. 14. Monadic interface for constraint generation

$$
\begin{aligned}
&\text{Open}\ (A : \text{Type}) : \hat{\text{Type}} := \text{Assignment} \twoheadrightarrow \text{Const}\ A\\
&\text{pure} \quad : \models \ \text{Const}\ A \twoheadrightarrow \text{Open}\ A := \lambda a\ \iota.\ a\\
&(\langle\$\rangle) \ : \models \ \text{Const}\ (A \to B) \twoheadrightarrow \text{Open}\ A \twoheadrightarrow \text{Open}\ B\\
&\qquad\qquad := \lambda f\ a\ \iota.\ f\ (a\ \iota)\\
&(\langle * \rangle) \ : \models \ \text{Open}\ (A \to B) \twoheadrightarrow \text{Open}\ A \twoheadrightarrow \text{Open}\ B\\
&\qquad\qquad := \lambda f\ a\ \iota.\ f\ \iota\ (a\ \iota)\\
&\lfloor _ \rfloor \quad : \models \ \hat{\text{Ty}} \twoheadrightarrow \text{Open Ty} := \lambda \hat{\tau}\ \iota.\ \hat{\tau}[\iota]
\end{aligned}
$$

Fig. 15. The Open modality and its applicative interface

$\models A$ expresses that a computation of type A may be used in any world. $A \twoheadrightarrow B$ describes functions between open types A, B as families of functions, and $\Box A$ (“box” A) denotes that a computation is abstracted over a substitution. We discuss the dual $\Diamond A$ (“diamond” A) in §3.4. Const turns a closed type A into an open type by ignoring the world. Type constructors such as box, diamond and Const are also commonly called *modalities*. As indicated by the braces, for a boxed computation $\Box A$, we pass the substitution explicitly, but pass the world implicitly, since it is determined by the substitution, and we also leave the world implicit when defining a computation $\models A$.

Figure 13 gives the definition of the bind for the free monad. Concretely, bind recurses on the left-hand side computation until it reaches a leaf. In the pure case, the continuation is “unboxed” by applying it to the identity substitution refl, before applying it to the semantic value. We push the boxed continuation under an existential quantifier by substituting with new_α. Note that substitution is in particular defined for boxed computations, because we can precompose the given substitution with the parameterized one.

Abstract monad interface. Figure 14 defines an abstract interface that we use for the implementation of the constraint generator. Defining the rest of the free monad instance is straightforward and left to the reader. In the remainder, we assume the use of an arbitrary but fixed monad M that implements the interface. When writing code, we make use of a *do* notation for the monad that gives us an explicit handle on the substitution

$$[\theta]\ x \leftarrow m\,; b \ := \ m \ggg\!= (\lambda \theta\ x.\ b)$$

3.2 The Open Modality

Before defining the constraint generator, we have to answer the question of what the type of the semantic value should be. Since we want to define type reconstruction, it should represent an object language expression, but so far we have not defined an open version of expressions that may refer to existentials. Defining copies, like we did for object language types, becomes cumbersome, especially if we consider more complex elaborations that produce expressions of a different language.

In order to manage the scalability of our approach, Figure 15 defines the Open modality that like Const turns a closed type into an open type. The difference is, that Open A may depend on a solution of the constraints. In particular, we represent a solution by an (Assignment : $\hat{\text{Type}}$) of closed object language types to every existential in a world w. Open quite literally represents closed semantic values A that depend on the solution as a function from a solution to A. In a sense, this modality enables a kind of staged computation [Davies and Pfenning 2001], with the first stage being constraint generation before solving, and the second stage after solving.

$$\hat{\text{synth}} : \models \hat{\text{Ctx}} \twoheadrightarrow \text{Const Exp} \twoheadrightarrow M\ (\hat{\text{Ty}} \times \hat{\text{Exp}})$$

$$\begin{array}{ll}
\hat{\text{synth}}\ \hat{\Gamma}\ (\lambda x.e) := & \hat{\text{synth}}\ \hat{\Gamma}\ (e_1\ e_2) := \\
\quad [\theta_1] \quad \hat{\tau}_1 \leftarrow \hat{\text{pick}} & \quad [\theta_1]\ \hat{\tau}_1, \hat{e}_1 \leftarrow \hat{\text{synth}}\ \hat{\Gamma}\ e_1 \\
\quad [\theta_2]\ \hat{\tau}_2, \hat{e} \leftarrow \hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e & \quad [\theta_2]\ \hat{\tau}_2, \hat{e}_2 \leftarrow \hat{\text{synth}}\ \hat{\Gamma}[\theta_1]\ e_2 \\
\qquad \hat{\text{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e}) & \quad [\theta_3] \quad \hat{\tau}_3 \leftarrow \hat{\text{pick}} \\
 & \quad [\theta_4] \quad _ \leftarrow \hat{\tau}_1[\theta_2\theta_3] \sim \hat{\tau}_2[\theta_3] \Rightarrow \hat{\tau}_3 \\
 & \qquad \hat{\text{pure}}\ (\hat{\tau}_3[\theta_4],\ \hat{e}_1[\theta_2\theta_3\theta_4]\ \hat{e}_2[\theta_3\theta_4])
\end{array}$$

Fig. 16. Selected cases of constraint generation for synthesising an open type and a type reconstructed expression

The modality allows for the composition of a second stage computation during the first stage, by means of an applicative interface [McBride and Paterson 2008][5] that is defined in Figure 15. To extract open object language types as a semantic output, we define the $\lfloor _ \rfloor$ function. Its implementation simply instantiates the existentially quantified variables by applying an assignment (ι : Assignment w) to the given open object language type ($\hat{\tau}$: $\hat{\text{Ty}}\ w$), which we also denote with post-application.

As a convenience, and to prepare for elaboration, we define an open type of object language expressions $\hat{\text{Exp}}$:= Open Exp. The applicative can be used to define smart constructors, e.g.

$$\hat{\lambda} x{:}_._ : \models \hat{\text{Ty}} \twoheadrightarrow \hat{\text{Exp}} \twoheadrightarrow \hat{\text{Exp}} := \lambda \hat{\tau}\ \hat{e}.\ (\lambda x{:}_._)\ \langle\$\rangle\ \lfloor \hat{\tau} \rfloor\ \langle * \rangle\ \hat{e}.$$

and we assume the availability of smart constructors for all the expression cases.

As a side note, Pottier [2014] defines a constraint generator using an applicative-only interface that is roughly comparable to the specialization $\hat{\text{Free}}$ (Open A), i.e. he considers closed semantic values only. Martinot and Scherer [2020] discuss the possibility of decomposing it into its constituents, but ultimately do not pursue this idea. That is precisely what we have done here.

3.3 Constraint Generation

Figure 16 implements a new constraint generator using the interface defined in Figure 14. Since the structure is almost identical to the one in Figure 5, apart from the variable representation and the explicit passing of substitutions, we only show the unannotated λ-case and the application case.

Each method and recursive call now explicitly introduces a substitution, signaling potential allocations of existentials. This requires careful composition of these substitutions to transfer an object from the from the world in which it was introduced to the world where it is used. For example, in the untyped lambda case, after invoking $\hat{\text{pick}}$, we apply substitution θ_1 to the typing context $\hat{\Gamma}$. The same applies to transporting $\hat{\tau}_1$ across the recursive call. However, this explicit transportation, while necessary, tends to create a lot of noise without adding interesting information. In our mechanization (§5), this is managed implicitly through Coq's type class system, but we keep substitutions explicit here for clarity.

3.4 Prenex Normal Form Conversion

As a first step towards solving the generated constraints, we transform them into a *prenex normal form* (PNF). Concretely, all existential quantifiers appear on the outside of a quantifier-free part.

[5] The applicative functor terminology is usually used for endofunctors. Since Open is not endo, the precise designation would be a *lax monoidal functor*.

This normal form corresponds to the grammar

$$\bot \mid \exists\overline{\alpha}.\ \overline{\hat{\tau} = \hat{\tau}}.$$

To model the quantifiers we make use of the $\diamond$ modality defined in Figure 12. The diamond is a monad resembling the writer monad, but instead of accumulating monoid values, it collects and composes substitutions. In essence, we represent a vector of existentials $\overline{\alpha}$ quantifiers, by a weakening substitution ($\theta : w \sqsubseteq w, \overline{\alpha}$). The prenex form is thus

$$\text{Prenex } A := \hat{\text{Option}}\ \diamond(\hat{\text{List}}\ (\hat{\text{Ty}} \times \hat{\text{Ty}}) \times A)$$

Parallel substitutions represent more than weakenings though so that Prenex represents more than constraints in PNF. In our implementation, we define a type of *weakening only substitutions* to model PNFs precisely. To convert a $\hat{\text{Free}}$ constraint to an equivalent one in prenex form, we can implement a function $\text{prenex} : \models \hat{\text{Free}}\ A \twoheadrightarrow \text{Prenex}\ A$. This is not very interesting, and we therefore omit it. Predictably, Prenex is a monad as well, which also instantiates the interface in Figure 14.

3.5 Constraint Solving

As part of the solver, we implemented McBride's [2000; 2003a] first-order unification algorithm, albeit with a different termination proof and rebased to the context variable removal of Keller and Altenkirch [2010]. Due to a lack of space, we omit a description of the full implementation.

The algorithm employs a traditional nested-recursive structure of an outer recursion over worlds, with one variable eliminated at each recursive step, and an inner structural recursion over two object language types. Additionally, it follows an accumulator-passing style [McBride 2003a; Kumar and Norrish 2010], progressively building up a substitution in triangular form [Baader et al. 2001].

The solver can fail, or otherwise produce a most-general unifier in the $\diamond$ monad, i.e. the interface of the solver is

$$\text{solve} : \models \hat{\text{List}}\ (\hat{\text{Ty}} \times \hat{\text{Ty}}) \twoheadrightarrow \text{Solved}\ () \qquad \text{Solved}\ A := \hat{\text{Option}}\ \diamond A$$

in other words, it produces *constraints with semantics values in solved form*. The solver defines what can essentially be seen as the *dynamic semantics* for *constraints with semantic values*, e.g. it allows us to "run" the $\hat{\text{Free}}$ monad computations

$$\begin{aligned}&\text{run} : \models \hat{\text{Free}}\ A \twoheadrightarrow \text{Solved}\ A := \lambda m.\\ &\quad [\theta_1]\ eqs, a \leftarrow \text{prenex}\ m;\ [\theta_2]\ _ \leftarrow \text{solve}\ eqs;\ \hat{\text{pure}}\ a[\theta_2]\end{aligned}$$

3.6 Putting It Together

We now bring the different parts together and define an end-to-end type reconstruction algorithm that composes them. Note that the do notation here is for the regular option monad:

$$\begin{aligned}&\text{reconstruct}\ (\Gamma : \text{Ctx})\ (e : \text{Exp}) : \text{Option}\ (\exists w. \hat{\text{Ty}}\ w \times \hat{\text{Exp}}\ w) :=\\ &\quad (\{w\}, \theta, \hat{\tau}, \hat{e}) \leftarrow \text{run}\ (\hat{\text{synth}}\ \lceil \Gamma \rceil\ e);\ \text{Some}\ (w, \hat{\tau}, \hat{e})\end{aligned}$$

The result of this algorithm is a world w of unconstrained existentials, together with an open object language type and elaborated expression that can still refer to the variables in w. This highlights that constraint solving is non-grounding: any unification variable whose shape is not deduced from traversing the program remains in the output of the solver. If we wish to, we may choose to ground the remaining variables (for example to bool) to arrive at a concrete type-reconstructed term. Clearly, this grounding is incomplete: a specific type has been chosen in the annotation.

$$\begin{array}{l} \Gamma \vdash_A e : \tau \rightsquigarrow e' := \\ \quad \textbf{match } \text{reconstruct } \Gamma\ e \textbf{ with} \\ \quad \mid \text{Some } (w, \hat{\tau}, \hat{e}) \rightarrow \\ \quad\quad \exists (\iota : \text{Assignment } w). \\ \quad\quad\quad \hat{\tau}[\iota] = \tau \wedge \hat{e}[\iota] = e' \\ \quad \mid \text{None} \quad \rightarrow \bot \end{array}$$

Fig. 17. Closed algorithmic typing relation

$$\begin{array}{lcl} \text{Pred } w & := & \text{Assignment } w \rightarrow \mathbb{P} \\ P \dashv\vdash Q & := & \forall \iota : \text{Assignment } w,\ P\ \iota \leftrightarrow Q\ \iota \\ P \vdash Q & := & \forall \iota : \text{Assignment } w,\ P\ \iota \rightarrow Q\ \iota \\ u \approx v & := & \lambda \iota. u[\iota] = v[\iota] \\ \hat{\Gamma} \vdash_{\hat{D}} e : \hat{\tau} \rightsquigarrow \hat{e} & := & \lambda \iota. \hat{\Gamma}[\iota] \vdash_D e : \hat{\tau}[\iota] \rightsquigarrow \hat{e}[\iota] \\ P[\theta] & := & \lambda \iota. P\ (\iota \circ \theta). \end{array}$$

Fig. 18. Assignment predicates

4 Type Inference Logics

We turn our attention now to the formal verification of the functional correctness of the type reconstruction algorithm from §3. In particular, we want to show that the effectful reconstruct function (§3.6) *decides* the typing rules for the simply typed λ-calculus with Booleans (Figure 2).

Our proof strategy consists of two parts: a *base logic* that abstracts away assignments to existentially quantified variables, and a *program logic* that defines reasoning rules for constraints with semantic values in terms of this base logic. This approach allows us to prove a reformulation of the generator correctness statement of Theorem 2.2.

We describe our approach in a top-down fashion. First, §4.1 will state the desired end-to-end correctness theorem. Then, §4.2 will motivate and describe a domain-specific base logic over existentially quantified variables (evars) and describe *predicate transformer semantics* for substitutions over evars as useful modalities. In §4.3, we first define an *exact predicate transformer semantics* for our constraints with values, and subsequently define an interface (§4.4) with derivable rules that we use as our program logic. Finally, we address the correctness of the constraint generator and solver in §4.5, highlighting the modularity and proof automation of our approach.

4.1 The Correctness Statement

In order to mirror Theorem 2.2, Figure 17 defines a *closed algorithmic typing relation.* This definition states that if reconstruct succeeds with a set or *world* of residual, unconstrained unification variables w, an open type $\hat{\tau}$ and an open reconstructed expression $\hat{e}$, then we require that these can be instantiated simultaneously, i.e. with the same assignment, to the given closed type and reconstructed expression. In other words, the calculated outputs of the algorithm are *more general*: different instantiations of these unconstrained variables lead to different concrete types. The overall correctness theorem now follows as an equivalence between algorithmic and declarative typing:

Theorem 4.1 (End-to-end correctness). $\Gamma \vdash_A e : \tau \rightsquigarrow e' \leftrightarrow \Gamma \vdash_D e : \tau \rightsquigarrow e'$.

4.2 Assignment Predicates

Just like in the implementation, we can modularly decompose Theorem 4.1 into constraint generation and constraint solving. The key insight is that this statement is between *closed* relations: they both deal with a concrete object type and elaborated expression, while the outputs of the generator and the input of the solver are *open*—the object type and expressions quantify over possible instantiations of the existential variables. This mismatch is rectified in the definition of the algorithmic typing relation in Figure 17 through an existential quantification over an *assignment* to the unconstrained variables, thereby linking open and closed pieces of data. What is needed to compose the proofs effectively is a logic that abstracts over these assignments. This avoids a lot of bookkeeping in the correctness statements for the individual phases that occurs for instance due to world updates—and hence assignments—during constraint generation and solving.

This section introduces the notion of *assignment predicates* as the carrier of a *base logic* that we use to state and reason about assertions involving existential variables. On top of this base logic, we will later (§4.4) define a *program logic* for reasoning about open monadic computations.

The Pred *abstraction.* Figure 18 contains the definition of *assignment predicates* Pred as a map from assignments of a given world to metalevel propositions. In other words, we use a world-indexed family of predicates as the carrier of our logic. One way to think about assignments is that they form a kind of *ghost state* or *logical state* [Jung et al. 2016; Owicki and Gries 1976] which is used to keep track of additional facts in the proof of a program, but is not part of the concrete state of the computation. For constraint generation, the concrete state is represented by the accumulated existential variables and equality constraints, while the logical state additionally keeps track of an assignment to every existential variable. In the next section, we will develop predicate transformers for our monads, which enforce that at every point the concrete state is compatible with the ghost state, i.e. that the ghost state satisfies the equality constraints.

Figure 18 additionally defines entailment ($\vdash$) and bientailment ($\dashv\vdash$). To save space, we omit the other logical connectives, but they, too, are lifted with their usual introduction and elimination rules from $\mathbb{P}$ to Pred. For example, implication is defined as follows:

$$(\rightarrow) : \ \models \mathrm{Pred} \twoheadrightarrow \mathrm{Pred} \twoheadrightarrow \mathrm{Pred} := \lambda P\ Q\ \iota.\ P\ \iota \rightarrow Q\ \iota$$

The rest of Figure 18 defines an *internal equality* $u \approx v$ which expresses that u and v become equal after instantiation, an open version of the declarative typing relation derived from the closed one, and a substitution operator for predicates $P[\theta]$, which composes the substitution with the assignment. Substitution distributes over all connectives and the equality and typing relation.

Substitutions as assignment predicate transformers. Interestingly, we can also interpret substitutions as *assignment predicate transformers*. Figure 19 defines both a total and a partial correctness interpretation of weakest preconditions. It is worth noting that they change the world of the predicates contravariantly. The key ingredient is the equation $\iota_1 \circ \theta = \iota_0$ involving a preassignment ι_0 and an existentially (wp) or universally (wlp) quantified postassignment ι_1. The predicate transformers are then the usual total and partial correctness interpretations of an equality assertion.

The equation can be understood in different ways. When viewing θ as a system of equations, it expresses that ι_0 satisfies them. When interpreting ι_0 as a substitution, it says that θ is more general than ι_0.

For constraint generation, let us consider the introduction of a new fresh variable. Suppose θ is a weakening that only introduces new existential variables, i.e. the domain of ι_0 is a subset of the domain of ι_1, then the equation denotes that by restricting ι_1 to the domain of ι_0, we get ι_0 back, i.e. ι_1 consistently extends ι_0. Said differently, in wp it represents an existential quantification over closed types for all variables in w_1 that do not appear in w_0. Dually, wlp represents a universal quantification.

These predicate transformers enjoy a wealth of properties, some of which are shown in the bottom half of Figure 19. The properties follow directly from similar ones for the underlying composition of substitutions and assignments, and for the underlying existential and universal quantification. The reflexivity and transitivity properties correspond to the classical rules for skip and seq statements of imperative languages. The frame rule allows us to focus on one proof obligation involving a weakest precondition simply by moving other obligations Q into the postcondition. As usual, when wp appears to the left of an implication, we can switch to wlp instead. The adjoint rules show a connection between the weakest preconditions and the substitution operator. Of particular interest is SUB-WLP-ADJOINT because it serves as an introduction rule whenever the head of an obligation is wlp. Dually, the adjoint rule for wp can be used as an elimination rule, but we have not used

$$\mathrm{wp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0 :=$$
$$\lambda(\iota_0 : \mathrm{Assignment}\ w_0).\ \exists(\iota_1 : \mathrm{Assignment}\ w_1).\ \iota_1 \circ \theta = \iota_0 \land Q\ \iota_1$$
$$\mathrm{wlp}\ \{w_0\ w_1\}\ (\theta : w_0 \sqsubseteq w_1)\ (Q : \mathrm{Pred}\ w_1) : \mathrm{Pred}\ w_0 :=$$
$$\lambda(\iota_0 : \mathrm{Assignment}\ w_0).\ \forall(\iota_1 : \mathrm{Assignment}\ w_1).\ \iota_1 \circ \theta = \iota_0 \to Q\ \iota_1$$

SUB-WLP-REFL* :	$\mathrm{wlp}\ \mathrm{refl}\ P \dashv\vdash P$
SUB-WLP-TRANS* :	$\mathrm{wlp}\ (\theta_1\theta_2)\ Q \dashv\vdash \mathrm{wlp}\ \theta_1\ (\mathrm{wlp}\ \theta_2\ Q)$
SUB-WP-FALSE :	$\mathrm{wp}\ \theta\ \bot \dashv\vdash \bot$
SUB-WP-FRAME :	$\mathrm{wp}\ \theta\ P \land Q \dashv\vdash \mathrm{wp}\ \theta\ (P \land Q[\theta])$
SUB-WP-WLP :	$(\mathrm{wp}\ \theta\ P \to Q) \dashv\vdash \mathrm{wlp}\ \theta\ (P \to Q[\theta])$
SUB-WP-MONO :	$\mathrm{wlp}\ \theta\ (P \to Q) \vdash \mathrm{wp}\ \theta\ P \to \mathrm{wp}\ \theta\ Q$
SUB-WLP-MONO :	$\mathrm{wlp}\ \theta\ (P \to Q) \vdash \mathrm{wlp}\ \theta\ P \to \mathrm{wlp}\ \theta\ Q$

SUB-WP-ADJOINT
$$\frac{P \vdash Q[\theta]}{\mathrm{wp}\ \theta\ P \vdash Q}$$

SUB-WLP-ADJOINT
$$\frac{P[\theta] \vdash Q}{P \vdash \mathrm{wlp}\ \theta\ Q}$$

Fig. 19. Select rules for substitution predicate transformers. Rules with (∗) apply identically to wp.

$$\hat{\mathrm{WP}}_{\mathrm{F}} : \models \hat{\mathrm{Free}}\ A \twoheadrightarrow \Box(A \twoheadrightarrow \mathrm{Pred}) \twoheadrightarrow \mathrm{Pred}$$
$$\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Pure}}\ a)\ Q := Q\ \mathrm{refl}\ a \qquad \hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Eq}}\ \hat{\tau}_1\ \hat{\tau}_2\ m)\ Q := \hat{\tau}_1 \approx \hat{\tau}_2 \land \hat{\mathrm{WP}}_{\mathrm{F}}\ m\ Q$$
$$\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Fail}})\ Q := \bot \qquad \hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Pick}}\ \alpha\ m)\ Q := \mathrm{wp}\ \mathrm{new}_\alpha\ (\hat{\mathrm{WP}}_{\mathrm{F}}\ m\ Q[\mathrm{new}_\alpha])$$

Fig. 20. Predicate transformer semantics of the $\hat{\mathrm{Free}}$ monad

it in this way in our proofs. In Figure 19 we define strong monotonicity rules. A weaker version, i.e. $(P \vdash Q) \to (\mathrm{wp}\ \theta\ P \vdash \mathrm{wp}\ \theta\ Q)$ is insufficient for our proofs, because we may need equalities or typing judgements that only hold in the ghost state hidden in the right hand side entailment, to prove that P implies Q. However, the weaker version requires us to show that P implies Q for any ghost state. Notice that we use wlp on the left-hand side of the monotonicity rules in Figure 19 to express in the present world and ghost state, i.e. before θ, that P implies Q in the future, i.e. after θ.

4.3 Predicate Transformer Semantics

We define the weakest preconditions $\hat{\mathrm{WP}}_{\mathrm{F}}$ for the free monad in Figure 20. The weakest liberal preconditions $\hat{\mathrm{WLP}}_{\mathrm{F}}$ can be defined dually. We use uppercase letters to distinguish them from the wp of substitution from §4.2. Just like the continuation of the bind operator in §3, $\hat{\mathrm{WP}}_{\mathrm{F}}$ and $\hat{\mathrm{WLP}}_{\mathrm{F}}$ take a boxed postcondition $\Box(A \twoheadrightarrow \mathrm{Pred})$ as input, i.e. it can depend on an output world, the substitution into that world, and the semantic value.

In the case of a $\hat{\mathrm{Pure}}$, the postcondition is first *unboxed* by applying it to the identity substitution refl before applying it to the semantic value. The $\hat{\mathrm{Fail}}$ and $\hat{\mathrm{Eq}}$ rules should be familiar from Figure 7. The $\hat{\mathrm{Pick}}$ case is the most interesting. Notice that we are reusing the wp for substitutions. The reader may have expected an existential quantification over a closed (or open) type which is then substituted for the last introduced variable, such as:

$$\hat{\mathrm{WP}}_{\mathrm{F}}\ (\hat{\mathrm{Pick}}\ \alpha\ m)\ Q \dashv\vdash \exists\tau : \mathrm{Ty}.(\hat{\mathrm{WP}}_{\mathrm{F}}\ m\ Q[\mathrm{new}_\alpha])[\alpha \mapsto \lceil\tau\rceil].$$

This rule is equivalent, but complicates proofs because we would need to generalize our theorems to account for an arbitrary substitution that is applied to the $\hat{\mathrm{WP}}_{\mathrm{F}}$ in the head of a goal. This kind of generalization still occurs, albeit for an arbitrary closing assignment instead of a substitution, and it is completely hidden in the definition of (bi)entailments. For our proofs, we integrate a derived

$$
\begin{array}{lll}
\text{WP-BIND}: & \hat{\mathrm{WP}}\ m\ (\lambda\theta\ a.\ \hat{\mathrm{WP}}\ (f\ \theta\ a)\ Q[\theta]) & \vdash\ \hat{\mathrm{WP}}\ (m \gg\!= f)\ Q \\
\text{WP-EQUALS}: & \hat{\tau}_1 \approx \hat{\tau}_2 \wedge \blacksquare(\lambda\theta.Q\ \theta\ ()) & \vdash\ \hat{\mathrm{WP}}\ (\hat{\tau}_1 \sim \hat{\tau}_2)\ Q \\
\text{WP-PICK}: & \exists\tau.\blacksquare(\lambda\theta.\ \forall\hat{\tau}.\ \lceil\tau\rceil \approx \hat{\tau} \rightarrow Q\ \theta\ \hat{\tau}) & \vdash\ \hat{\mathrm{WP}}\ (\hat{\mathrm{pick}})\ Q \\
\text{WP-MONO}: & \blacksquare(\lambda\theta.\ \forall a.\ P\ \theta\ a \rightarrow Q\ \theta\ a) & \vdash\ \hat{\mathrm{WP}}\ m\ P \rightarrow \hat{\mathrm{WP}}\ m\ Q \\
\text{WLP-BIND}: & \hat{\mathrm{WLP}}\ m\ (\lambda\theta\ a.\ \hat{\mathrm{WLP}}\ (f\ \theta\ a)\ Q[\theta]) & \vdash\ \hat{\mathrm{WLP}}\ (m \gg\!= f)\ Q \\
\text{WLP-EQUALS}: & \hat{\tau}_1 \approx \hat{\tau}_2 \rightarrow \blacksquare(\lambda\theta.Q\ \theta\ ()) & \vdash\ \hat{\mathrm{WLP}}\ (\hat{\tau}_1 \sim \hat{\tau}_2)\ Q \\
\text{WLP-PICK}: & \blacksquare(\lambda\theta.\ \forall\hat{\tau}.\ Q\ \theta\ \hat{\tau}) & \vdash\ \hat{\mathrm{WLP}}\ (\hat{\mathrm{pick}})\ Q \\
\text{WLP-MONO}: & \blacksquare(\lambda\theta.\ \forall a.\ P\ \theta\ a \rightarrow Q\ \theta\ a) & \vdash\ \hat{\mathrm{WLP}}\ m\ P \rightarrow \hat{\mathrm{WLP}}\ m\ Q \\
\text{WP-WLP}: & \hat{\mathrm{WLP}}\ m\ (\lambda\theta.\ P\ \theta\ a \rightarrow Q[\theta]) & \dashv\vdash\ \hat{\mathrm{WP}}\ m\ P \rightarrow Q
\end{array}
$$

Fig. 21. Program logic interface for constraint monads

rule to introduce the wp of new_α into the rule for $\hat{\mathrm{pick}}$:

$$
\textsc{sub-wp-new}\ \frac{\exists\tau : \mathrm{Ty}.\ P[\mathrm{new}_\alpha] \vdash \lceil\tau\rceil \approx \alpha \rightarrow Q}{P \vdash \mathrm{wp}\ \mathrm{new}_\alpha\ Q}
$$

This also uses an existential quantification, but instead of substituting the last introduced variable, we introduce a hypothesis that states that the variable is equal to the existentially quantified type. Notice that the world also changes, in the consequent it is w but we moved to world (w, α) in the antecedent, i.e. the variable is not substituted away. The $\hat{\mathrm{WP}}_\mathrm{F}$ and therefore also this rule are used in the completeness proof, where we get preexisting closed types to fill in the existentials.

4.4 Program Logic

Since the constraint generator is implemented against an abstract monad interface, we also want to define an abstract reasoning interface. We could distill the specific definition of the $\hat{\mathrm{WP}}_\mathrm{F}$ for the free monad into *bientailment rules* for the type class methods. However, that would result in an overspecification because it would leak implementation-specific details of the free monad into the interface. The free monad does not solve equalities immediately and implements $\hat{\mathrm{pick}}$ by choosing a unification variable name via fresh. With bientailments, we could not allow any other behavior. As we lay out in the next section, proving a bientailment directly is difficult. Consequently, we split up the correctness into soundness and completeness directions, for which reasoning with entailments is enough. Therefore, we define an interface with weaker entailment rules, that allows more implementations. In particular, we can also define an instance for the Solved monad.

Allowing arbitrary behavior for $(\sim)$ specifically means that we need to allow an arbitrary substitution to be passed to the continuation, not just the identity substitution like in the case of the free monad. In other words, we need to introduce universal quantifications over substitutions at specific points. This is in fact similar to the box modality, albeit on the level of predicates:

$$
\blacksquare :\ \models \Box\mathrm{Pred} \twoheadrightarrow \mathrm{Pred} := \lambda\{w\}\ P.\ \forall\{w'\}\ (\theta : w \sqsubseteq w').\ \mathrm{wlp}\ \theta\ (P\ \theta)
$$

It expresses that the given boxed predicate P holds in any future world. We use wlp again to talk about the future in the present. We define elimination, introduction, and substitution rules that follow from the SUB-WLP-REFL, SUB-WLP-ADJOINT, and SUB-WLP-TRANS rules for substitutions:

$$
\blacksquare P \vdash P\ \mathrm{refl} \qquad (\forall\{w'\}\ (\theta : w \sqsubseteq w').\ Q[\theta] \vdash P\ \theta) \rightarrow (Q \vdash \blacksquare P) \qquad (\blacksquare P)[\theta] \dashv\vdash \blacksquare(P[\theta])
$$

With all the ingredients in place, we finally define the program logic interface in Figure 21. WP-BIND is a sequencing rule, i.e. it becomes the $\hat{\mathrm{WP}}$ of the first subcomputation m which yields a

substitution θ and a value a. We then require Q to hold after applying f. Note that we also apply the substitution θ to the postcondition Q, i.e. transporting it to the world after executing m, similar to how continuations are transported in the definition of bind. The $(\sim)$ and $\hat{\text{pick}}$ rules have been generalized with the $\blacksquare$ modality, but otherwise stay the same. Crucially, we also use the $\blacksquare$ modality in the monotonicity rules. Moreover, if $\hat{\text{WP}}$ appears to the left of an implication, we can switch to the $\hat{\text{WLP}}$ via WP-WLP.

4.5 Correctness of the Generator

With the program logic rules at hand, we can turn to the correctness proof of the constraint generator. First, we derive an *open algorithmic typing relation* from the generator

$$\hat{\Gamma} \vdash_{\hat{A}} e : \hat{\tau} \rightsquigarrow \hat{e} := \hat{\text{WP}}\ (\hat{\text{synth}}\ \hat{\Gamma}\ e)\ (\lambda\theta\ (\hat{\tau}', \hat{e}').\ \hat{\tau}[\theta] \approx \hat{\tau}' \wedge \hat{e}[\theta] \approx \hat{e}').$$

Akin to the end-to-end theorem, we state correctness of the constraint generator as an equivalence of the algorithmic and declarative typing relations, in this case between open typing relations:

Theorem 4.2 (Generator correctness). $\hat{\Gamma} \vdash_{\hat{A}} e : \hat{\tau} \rightsquigarrow \hat{e} \dashv\vdash \hat{\Gamma} \vdash_{\hat{D}} e : \hat{\tau} \rightsquigarrow \hat{e}$

A direct proof by induction over e is difficult. We did indeed define bientailment rules for the free monad, but in contrast to Figure 20, the postcondition is specific instead of arbitrary. At the point in the proof where we want to use an induction hypothesis, the postconditions will not be aligned, and massaging them into the correct shape is burdensome. Hence, just like in §2.4, we can split this equivalence into a soundness and a completeness direction, using the weaker program logic rules. For soundness, we can switch from the $\hat{\text{WP}}$ to the $\hat{\text{WLP}}$ via rule WP-WLP and simplify the equalities in the postcondition away.

Lemma 4.3 (Generator soundness). $\vdash \hat{\text{WLP}}\ (\hat{\text{synth}}\ \hat{\Gamma}\ e)\ (\lambda\theta\ (\hat{\tau}, \hat{e}).\ \hat{\Gamma}[\theta] \vdash_{\hat{D}} e : \hat{\tau} \rightsquigarrow \hat{e})$

Proof. By induction over the expression. Consider the case $\lambda x.e_1$, for which we have to prove

$$\vdash \hat{\text{WLP}} \begin{pmatrix} [\theta_1] & \hat{\tau}_1 \leftarrow \hat{\text{pick}} \\ [\theta_2] & \hat{\tau}_2, \hat{e}_1 \leftarrow \hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e_1 \\ & \hat{\text{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e}_1) \end{pmatrix} (\lambda\theta\ (\hat{\tau}, \hat{e}).\ \hat{\Gamma}[\theta] \vdash_{\hat{D}} \lambda x.e_1 : \hat{\tau} \rightsquigarrow \hat{e})$$

After applying the WLP-Bind and WLP-Pick, we end up with $\blacksquare$ at the head. From the introduction rule for $\blacksquare$ and the rule for pick, we get a universal quantification over some θ_1 and over a $\hat{\tau}_1$, respectively. We match these to the names in the do notation. It remains to show

$$\vdash \hat{\text{WLP}} \begin{pmatrix} [\theta_2] & \hat{\tau}_2, \hat{e}_1 \leftarrow \hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e_1 \\ & \hat{\text{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e}_1) \end{pmatrix} (\lambda\theta\ (\hat{\tau}, \hat{e}).\ \hat{\Gamma}[\theta_1\theta] \vdash_{\hat{D}} \lambda x.e_1 : \hat{\tau} \rightsquigarrow \hat{e})$$

Using WLP-Bind and introducing our IH for e_1 with $(\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)$ it remains to show

$$\begin{aligned} \vdash\ & \hat{\text{WLP}}\ \big(\hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e_1\big)\ (\lambda\theta_2\ (\hat{\tau}_2, \hat{e}_1).\ \ (\hat{\Gamma}[\theta_1\theta_2], x : \hat{\tau}_1[\theta_2]) \vdash_{\hat{D}} e_1 : \hat{\tau}_2 \rightsquigarrow \hat{e}_1) \rightarrow \\ & \hat{\text{WLP}}\ \big(\hat{\text{synth}}\ (\hat{\Gamma}[\theta_1], x : \hat{\tau}_1)\ e_1\big)\ (\lambda\theta_2\ (\hat{\tau}_2, \hat{e}_1).\ \ \hat{\text{WLP}}\ \big(\hat{\text{pure}}\ (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2,\ \hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e}_1)\big) \\ & \qquad\qquad (\lambda\theta\ (\hat{\tau}, \hat{e}).\ \hat{\Gamma}[\theta_1\theta_2\theta] \vdash_{\hat{D}} \lambda x.e_1 : \hat{\tau} \rightsquigarrow \hat{e})) \end{aligned}$$

at which point we use WLP-Mono. We introduce the resulting $\blacksquare$ and get a universal quantification over θ_2. We have to show the implication between the postconditions, which after applying the pure rule and simplifying the identity substitution away becomes

$$\vdash \big(\hat{\Gamma}[\theta_1\theta_2], x : \hat{\tau}_1[\theta_2] \vdash_{\hat{D}} e_1 : \hat{\tau}_2 \rightsquigarrow \hat{e}_1\big) \rightarrow \big(\hat{\Gamma}[\theta_1\theta_2] \vdash_{\hat{D}} (\lambda x.e_1) : (\hat{\tau}_1[\theta_2] \Rightarrow \hat{\tau}_2) \rightsquigarrow (\hat{\lambda} x{:}\hat{\tau}_1[\theta_2].\hat{e}_1)\big).$$

We have not defined introduction rules for the open typing relation, but we can drop down to the closed typing relation where the implication follows from T-Abs-Implicit. □

For completeness, we also transform the statement to make it easier to prove. Ideally, we perform the proof by induction over the given open declarative typing judgment, i.e. the right-hand side of Theorem 4.2. However, the open relation itself is not inductively defined. Rather, it is defined in terms of the inductive closed typing relation. It is not clear how to define a general enough induction principle for the open relation. Instead, we induct over the underlying closed relation. The statement itself has to be massaged to bring it into a shape for the induction. Specifically, non-variable positions of the relation need to abstracted over with equality constraints similar to the *Elim* tactic of McBride [2002] and Coq's *dependent induction* tactic.

Lemma 4.4 (Generator completeness).

$$\begin{aligned} &\forall \Gamma\ e\ \tau\ e'.\ \Gamma \vdash_D e : \tau \rightsquigarrow e' \rightarrow \forall w\ (\hat{\Gamma} : \hat{\mathrm{C}}\mathrm{tx}\ w). \\ &\quad \vdash \lceil \Gamma \rceil \approx \hat{\Gamma} \rightarrow \hat{\mathrm{W}}\mathrm{P}\ (\hat{\mathrm{s}}\mathrm{ynth}\ \hat{\Gamma}\ e)\ (\lambda\theta\ (\hat{\tau}, \hat{e}').\lceil \tau \rceil \approx \hat{\tau} \wedge \lceil e' \rceil \approx \hat{e}') \end{aligned}$$

Proof. By induction on the closed typing judgment. The reasoning is similar to the soundness direction. Each case is simplified using the $\hat{\mathrm{W}}\mathrm{P}$ rules from Figure 21. When the $\hat{\mathrm{W}}\mathrm{P}$ of a recursive call is at the head position, we introduce it using monotonicity and the IH for that call. □

4.6 Putting It Together

We can define similar weakest preconditions operators $\hat{\mathrm{W}}\mathrm{P}_\mathrm{P}$ for Prenex, and $\hat{\mathrm{W}}\mathrm{P}_\mathrm{S}$ for Solved, and use them to prove the correctness of the rest of the pipeline.

Lemma 4.5 (Prenex conversion correctness). $\hat{\mathrm{W}}\mathrm{P}_\mathrm{P}\ (\mathrm{prenex}\ m)\ Q \dashv\vdash \hat{\mathrm{W}}\mathrm{P}_\mathrm{F}\ m\ Q$

Lemma 4.6 (Constraint solver correctness). $\hat{\mathrm{W}}\mathrm{P}_\mathrm{S}\ (\mathrm{solve}\ eqs)\ (\lambda_.\top) \dashv\vdash [\![eqs]\!]$

We use denotation brackets to interpret a list of equalities as a predicate. All of the correctness lemmas can be combined into the end-to-end Theorem 4.1. We refer the reader to our mechanization (§5) for details. Having a sound and complete algorithm means we can effectively decide typing.

Corollary 4.7 (Decidability of typing). *The three place relation* $\Gamma \vdash_D e : \tau$ *is decidable.*

Proof. After running the algorithm on the inputs Γ, e, we get a set of unconstrained variables w, an open type $\hat{\tau}$ and an open expression $\hat{e}$. Solving the additional constraint $\lceil \tau \rceil \approx \hat{\tau}$ will reduce this set further and the remainder can be grounded to bool. If the algorithm and the constraint solver succeed, then the judgment holds by Theorem 4.1 and Lemma 4.6. Otherwise it does not. □

Unfortunately, we cannot decide the four place relation with this definition of our algorithm. For the decidability of the relation with elaboration, we need to decide if the output of the algorithm can be instantiated to the given elaborated expression. The adoption of the Open modality for expressions, in favor of a separate, first-order representation of open expressions to be used in the unification algorithm, means we cannot. Since the "fix" is very minor—essentially, just redefine $\hat{\mathrm{E}}\mathrm{xp}$ to something other than Open Exp—and because we argue that the open modality is very interesting and useful in its own right, we omit these details for presentation purposes.

4.7 Correctness via Logical Relations

In this section we use an alternative proof method, proposed by Keuchel et al. [2022] in the context of symbolic execution, to reduce the correctness (Theorem 4.2) of the constraint generator with the first-order representation of existential variables from Figure 16, to the correctness (Theorem 2.2) of the constraint generator with the higher-order representation from Figure 5. The general idea is to define a *binary logical relation* between closed and open types that express a form of equivalence, then show that the two constraint generator implementations are logically related, and finally use the relatedness to reduce the correctness statements.

$$
\begin{aligned}
&\mathcal{R}[\![A, \hat{A}]\!]_w : A \times \hat{A}\ w \to \mathrm{Pred}\ w \\
&\mathcal{R}[\![\mathrm{Ty}, \hat{\mathrm{Ty}}]\!]_w(\tau, \hat{\tau}) := \lceil \tau \rceil \approx \hat{\tau} \\
&\mathcal{R}[\![A, \mathrm{Open}\ A]\!]_w(a, \hat{a}) := \lceil a \rceil \approx \hat{a} \\
&\mathcal{R}[\![\mathbb{P}, \mathrm{Pred}]\!]_w(P, \hat{P}) := \lceil P \rceil \leftrightarrow \hat{P} \\
&\mathcal{R}[\![A \to B, \hat{A} \twoheadrightarrow \hat{B}]\!]_w(f, \hat{f}) := \\
&\quad \forall a\ \hat{a},\ \mathcal{R}[\![A, \hat{A}]\!]_w(a, \hat{a}) \to \\
&\qquad \mathcal{R}[\![B, \hat{B}]\!]_w(f\ a, \hat{f}\ \hat{a}) \\
&\mathcal{R}[\![A, \Box\hat{A}]\!]_w(a, \hat{a}) := \\
&\quad \blacksquare(\lambda(\theta : w \sqsubseteq w').\mathcal{R}[\![A, \hat{A}]\!]_{w'}(a, \hat{a}\ \theta))
\end{aligned}
$$

Fig. 22. Logical Relation

$$
\begin{aligned}
&\mathcal{R}[\![\mathrm{Free}\ A, \hat{\mathrm{Free}}\ \hat{A}]\!]_w(m, \hat{m}) := \\
&\quad \textbf{match}\ m, \hat{m}\ \textbf{with} \\
&\quad \mid \mathrm{Pure}\ a, \hat{\mathrm{Pure}}\ \hat{a} \to \mathcal{R}[\![A, \hat{A}]\!]_w(a, \hat{a}) \\
&\quad \mid \mathrm{Fail}, \hat{\mathrm{Fail}} \to \top \\
&\quad \mid \mathrm{Eq}\ \tau_1\ \tau_2\ m, \hat{\mathrm{Eq}}\ \hat{\tau}_1\ \hat{\tau}_2\ \hat{m} \to \\
&\qquad \mathcal{R}[\![\mathrm{Ty}, \hat{\mathrm{Ty}}]\!]_w(\tau_1, \hat{\tau}_1) \wedge \mathcal{R}[\![\mathrm{Ty}, \hat{\mathrm{Ty}}]\!]_w(\tau_2, \hat{\tau}_2) \wedge \\
&\qquad \mathcal{R}[\![\mathrm{Free}\ A, \hat{\mathrm{Free}}\ \hat{A}]\!]_w(m, \hat{m}) \\
&\quad \mid \mathrm{Pick}\ f, \hat{\mathrm{Pick}}\ \alpha\ k \to \\
&\qquad \mathrm{wlp\ new}_\alpha\ (\forall \tau : \mathrm{Ty}, \lceil \tau \rceil \approx \alpha \to \\
&\qquad\qquad \mathcal{R}[\![\mathrm{Free}\ A, \hat{\mathrm{Free}}\ \hat{A}]\!]_{w,\alpha}(f\ \tau, k)) \\
&\quad \mid _, _ \to \bot
\end{aligned}
$$

Fig. 23. Logical Relation for the free monad

Logical Relation. Figure 22 contains the definition of the logical relation for some of the simple cases. Note that it is a relation defined using predicates of a given world w as opposed to regular propositions like in [Keuchel et al. 2022]. We will also use a proposition-valued relation for computations that are valid, i.e. that can be used in any world:

$$
\mathcal{R}[\![A, \hat{A}]\!] : A \times (\models \hat{A}) \to \mathbb{P} \qquad \mathcal{R}[\![A, \hat{A}]\!](a, \hat{a}) := \forall w.\mathcal{R}[\![A, \hat{A}]\!]_w(a, \hat{a}\ \{w\})
$$

For most of the base cases, like object language types and typing contexts, and types in the Open modality, the logical relation is defined using the internal equality, or equivalently, equality after instantiating with the assignment that makes up the ghost state. As usual, functions are pointwise related: they send related inputs to related outputs. A pure proposition is related to a predicate if they are equivalent after injecting the proposition into predicates. More interesting is the case of a boxed open computation $\hat{a}$. It is related to a closed computation a whenever they are related in every future as expressed by the $\blacksquare$ modality. We apply the boxed computation $\hat{a}$ to an explicit witness θ given by $\blacksquare$, after which $\hat{a}\ \theta$ becomes an unboxed computation.

Logically Related Computations. Figure 23 relates the higher-order with the first-order abstract syntax variant of the free monad. It proceeds by recursion over two computations and requires that at each step the same constructor is used. The cases for Pure, Fail and Eq follow immediately. The interesting case Pick is similar to the function case above. After introducing α via ($\mathrm{wlp\ new}_\alpha$) and τ via universal quantification, we assume that they are logically related as witnessed by the internal equality, and require that the two bodies $f\ \tau$ and k of the existential quantifications are logically related.

Here we only defined the logical relation concretely for the free monads, but we can also reason about two abstract monads $M, \hat{M}$ that implement the interfaces in Figures 4 and 14, respectively. For this, we require, e.g. in a new type class, that every corresponding pair of methods of the monad type classes are logically related at their respective types. We omit the details.

Lemma 4.8 (Relatedness of constraint generators). *The constraint generators* synth *of Figure 5 and* $\hat{\mathrm{synth}}$ *of Figure 16 are logically related, i.e. the following holds:*

$$
\forall (e : \mathrm{Exp}),\ \mathcal{R}[\![\mathrm{Ctx} \to M\ (\mathrm{Ty} \times \mathrm{Exp}), \hat{\mathrm{Ctx}} \twoheadrightarrow \hat{M}\ (\hat{\mathrm{Ty}} \times \hat{\mathrm{Exp}})]\!](\mathrm{synth}\ e, \hat{\mathrm{synth}}\ e)
$$

Proof. By induction over the expression e. Since both algorithms call the respective methods from the type classes in the same order, we go through both computations in lockstep and use the relatedness of all monad operations. □

Generator Correctness via the Logical Relation. We now turn towards reducing Theorem 4.2 to Theorem 2.2 via the logical relation. The first step is to realize that logically related computations have logically related weakest precondition predicate transformer semantics.

Lemma 4.9 (Relatedness of Weakest Preconditions).

$$\mathcal{R}[\![\mathrm{Free}\ A \to (A \to \mathbb{P}) \to \mathbb{P}, \hat{\mathrm{Free}}\ \hat{A} \twoheadrightarrow \Box(\hat{A} \twoheadrightarrow \mathrm{Pred}) \twoheadrightarrow \mathrm{Pred}]\!](\mathrm{WP}, \hat{\mathrm{WP}})$$

Since we defined algorithmic typing using the weakest preconditions of the generators, we immediately get relatedness of closed and open typing relations which is the missing link for the reduction of the correctness theorems:

Corollary 4.10 (Relatedness of algorithmic typing). *The closed and open algorithmic typing relations are related, i.e. the following relation holds (omitting the type of the closed relation):*

$$\mathcal{R}[\![_, \hat{\mathrm{Ctx}} \twoheadrightarrow \mathrm{Const\ Exp} \twoheadrightarrow \hat{\mathrm{Ty}} \twoheadrightarrow \hat{\mathrm{Exp}} \twoheadrightarrow \mathrm{Pred}]\!](_ \vdash_A _ : _ \rightsquigarrow _,\ _ \vdash_{\hat{A}} _ : _ \rightsquigarrow _)$$

Alternative Proof of Theorem 4.2. After unfolding the bientailment and introducing the assignment ι, the theorem follows from the following equivalences:

$$(\hat{\Gamma} \vdash_{\hat{A}} e : \hat{\tau} \rightsquigarrow \hat{e})\ \iota \overset{4.10}{\longleftrightarrow} \hat{\Gamma}[\iota] \vdash_A e : \hat{\tau}[\iota] \rightsquigarrow \hat{e}[\iota] \overset{2.2}{\longleftrightarrow} \hat{\Gamma}[\iota] \vdash_D e : \hat{\tau}[\iota] \rightsquigarrow \hat{e}[\iota] \overset{\text{Fig.}18}{\longleftrightarrow} (\hat{\Gamma} \vdash_{\hat{D}} e : \hat{\tau} \rightsquigarrow \hat{e})\ \iota.$$

□

5 Mechanization and Evaluation

We have mechanically formalized our approach in the Coq proof assistant. This section discusses choices and trade-offs, as well as practical infrastructure for proof automation.

Modularity and phase separation. In our mechanization, we defined an abstract constraint monad type class as outlined in §3.1. We also implemented instances for the $\hat{\mathrm{Free}}$, Prenex and the Solved monad. When instantiating a generator programmed against the interface with the Solved instance, the phases are only separated logically but not in the dynamic execution. We implemented and mechanized the correctness of three constraint generators: the synthesizer from §3.3; a checker in the style of Figure 3; and a bidirectional generator combining synthesizing and checking. This demonstrates that our approach is flexible: it allows for different styles of constraint generation. Furthermore, for each constraint generator we have two correctness proofs: a direct one and one via our logical relation.

Existential variable representation. We left the specific variable representation for existentials open in the paper. Concerns related to capture avoidance and α-equivalence do not arise in our setting, since we never push substitutions under binders, and we never compare terms containing binders. Our code base does not contain these definitions. We represent existentials using a hybrid approach based on well-scoped de Bruijn indices and names, comprehensible to human readers. They are exclusively used for computations and "decoration", respectively. We do generate fresh names but we do not prove that names are sufficiently fresh, i.e. that no shadowing occurs.

In our experience, the use of intrinsic scoping benefits the discovery of (correct) abstractions. Specifically, this approach makes sure that we do not use incidental properties of a variable representation, such as the invariance of a term under weakening. An intrinsic scoping prevents us from using that simply because terms in different scopes have different types. Instead, each scope change is an explicit operation on syntactic terms with an explicit witness.

The downside is of course that every subcomputation can potentially induce a scope change, and thus we have to deal with a lot of witnesses and need to use them transport values from a point where they are introduced to where they are used. We found that this is more problematic on the

Table 1. Overview of the mechanisation effort in terms of lines of code

CATEGORY		Spec	Proof
I	**Generic**	**1583**	**1035**
	Base logic	533	325
	Infrastructure	282	154
	Unification	186	129
	Monad interface	126	146
	Logical relation	139	89
	Free monad	67	57
	Monad interface (HOAS)	99	16
	Open modality	44	54
	Free monad (HOAS)	51	4
	Prenex monad	22	31
	Solved monad	14	21
	Prenex conversion	20	9

CATEGORY		Spec	Proof
II	**Specific ($\lambda_{\mathbb{B}}$)**	**777**	**557**
	Generators (HOAS)	213	160
	Generator (bidir)	110	77
	Relatedness	63	87
	Generator (check)	76	50
	Infrastructure	58	66
	Generator (synth)	68	49
	Composition	58	37
	Specification	70	4
	Unification	46	27
	Extraction	15	0
TOTAL		**2360**	**1592**

programming side, since the user has to deal with that. During proofs the intrinsic scoping enforces correct handling without manual intervention, so one can largely gloss over the explicit witnesses.

Proof handling. Typically, a proof assistant's existing infrastructure for handling hypotheses does not extend to an embedded logic's entailments or implications. To improve proof handling, we instantiate the MoSeL [Krebbers et al. 2017, 2018] framework with our base logic, stubbing spatial connectives and modalities when necessary. In addition to the usual context of hypotheses and local variables of the proof assistant, MoSeL provides the user with a context of hypotheses of the embedded logic. Furthermore, it implements high-level tactics for introduction and elimination of the embedded logic's connectives.

One of the highlight features of MoSeL is its adaptable support for modalities, which is not restricted to a specific hardwired set of modalities, but is instead extensible through type classes. The tactics it provides are parametric in the modalities they work with. We employ this machinery for the specific modalities of our logic, particularly for *introducing* the wlp of a substitution. That is, whenever the proof goal is of the form $P_1 \wedge \ldots \wedge P_n \vdash \mathrm{wlp}\ \theta\ Q$, we "introduce" the wlp θ part of the head by means of a tactic that is in essence implementing rule SUB-WLP-ADJOINT. The effect of this introduction is that the substitution is applied to all hypotheses P_i and the wlp vanishes from the head. Moreover, we hook (extensible) type class-based logic programming into the modality introduction to further distribute the substitution automatically over connectives inside the hypotheses to reduce the need for rewriting steps. Dually, we use the *modality elimination* facilities if $\blacksquare$ appears at the head of a hypothesis.

We further automate proofs by providing reusable tactics to automatically apply the weakest precondition reasoning rules of Figure 19 at the base logic level and the rules of Figure 21 at the program logic level.

Extraction. We have extracted the end-to-end reconstruct function from §3.6 to Haskell. To demonstrate the applicability of our work, we provide a small implementation combining the extracted type inference and elaboration algorithm with a non-verified parser and pretty printer. We provide a set of example $\lambda_{\mathbb{B}}$ programs as part of our development.

Engineering effort. Table 1 provides an overview of the mechanization effort in terms of source lines of code as produced by the coqwc tool. We draw a distinction between generic (I) and language-specific code (II). We deem definitions and proofs as generic if they are, in some sense, object language-agnostic or sufficiently modular to be compatible with and reusable for a different object language. In principle, the code could be parameterized over a language instantiation, but we have not done so. 'Infrastructure' includes the definition of worlds, fresh name generation, substitution etc. The base logic comprises the definition of predicates, the predicate transformers for substitutions, and the instantiation of the MoSeL framework. The generic part of unification includes reusable recursion and induction schemes for successive variable elimination and the variable cases for an occurs check, as well as the cases for *flex-flex* and *flex-rigid* unifications. The monad interface consists of the constraint monad type class, the program logics interface and derived reasoning rules. The implementation of the individual monads also contain the program logic instantiations. We implemented both higher-order and first-order abstract syntax variants.

The language-specific code includes infrastructure like traversals to implement substitutions and instantiation of open types to closed types and injecting closed types into open types. This includes various proofs for the interaction between the traversals. The language specific unification machinery consists of the occurs check traversal, and the structural recursion over two open types. The variable cases are generally not part of the language-specific code, but are handled by the generic code.

Overall, the language-specific parts are dominated by the constraint generators and the infrastructure. Proofs of the main lemmas that recurse over the syntax of the language are quite small, in the order of 3 to 5 lines of code, where only the variable case is handled explicitly, and all other cases are handled by automatic proof scripts. The bulk of the generator-specific proofs is to massage the statements into the correct form. This is slightly more work for the completeness direction of the direct proof and less work for the logical relation proof. Overall the proof effort for both proof methods is comparable, but of course the logical relation proof requires the implementation of a second HOAS-based generator.

As evidenced by Table 1, the base logic forms one of the biggest parts of our development, but it is also the one with the biggest potential for reuse. The constraints and their program logic are still somewhat specific to the features of the type system and thus need to be extended on a case by case basis. However, even richer type systems will need a form of worlds and assignments. We hope to develop a sufficiently generic base logic that can be instantiated and reused for multiple constraint languages, including richer forms of constraints.

5.1 Synthetic Benchmark

We report on a small synthetic benchmark that captures the execution time for executing the extracted Haskell code. The benchmark is comprised of two parts. The former part simply infers types for increasingly large Church numerals, e.g. $\lambda f.\lambda x.f\ (f\ (f\ x))$ for $n = 3$. The latter benchmark consists of what we dub a "worst-case term": for $n = 3$, a lambda expression is generated of the form $\lambda f.\lambda x_1.\lambda x_2.\lambda x_3.f\ x_1\ x_2\ x_3$. Figure 24 plots execution times for both Church numerals (left) and the worst-case terms (right) for values of n up to 500. The execution time measurements were obtained on an Apple MacBook Air with M2 processor and 16 GB of memory, using Coq version 8.18 and GHC version 9.8.1.

For both benchmarks, the $\hat{\text{Free}}$ monad suffers from the usual problem of a linear-time bind operator [Voigtländer 2008]. The Prenex monad is not very efficient either, because the variable representation uses unary natural numbers to represent de Bruijn indices: each increment in the Church numeral is introducing one additional existential, in turn requiring all indices to be incremented, incurring a quadratic runtime overhead.

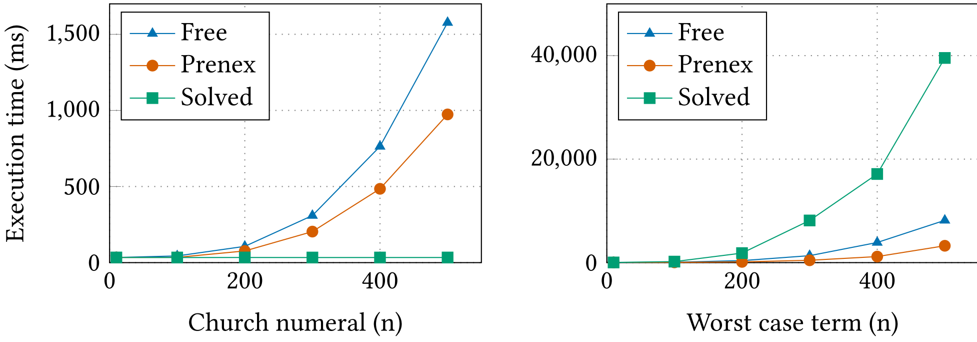

Fig. 24. Execution times of the two synthetic benchmarks. The x-axis plots the input size of the program, n representing the number of unification variables for that term. Note that for the worst-case term, the `Solved` monad (sym. ■) performs significantly worse due to (among other factors) the application of a quadratic weakening substitution for every fresh existential variable introduction.

For the Church numerals the Solved monad is more efficient. Most generated existentials are immediately eliminated by a unification. This allows the computation to scale to somewhat large terms: Church number 1,000,000 can be checked in about 7.3s. However, for the worst-case term the Solved monad cannot immediately eliminate existentials. Each new λ-bound variable also generates a new existential, yet equality constraints for unification are only introduced by the application case, not the lambda abstractions. Consequently, even the Solved monad will have to weaken the indices for all generated existentials at each lambda abstraction. Each weakening is implemented via a parallel substitution generated "on the fly", which has quadratic runtime in the number of existentials. The $\hat{\text{Free}}$ and Prenex monads outperform Solved in this case, since in our implementation they do not use parallel substitutions, but a weakening only substitution that implements a single variable weakening in constant time. We discuss efficient algorithms for solving in the Future work section (§7.1).

6 Related Work

Our contributions build on a vast literature of related work. Here we focus on related work that concerns different aspects of formal reasoning about inference.

Mechanized reasoning about type inference. In §1, we already provided a short survey of specifications and mechanizations of type inference. Reasoning about type inference inside a proof assistant dates back to seminal work by Dubois and Ménissier-Morain [1999]; Naraschewski and Nipkow [1999]; Nazareth and Nipkow [1996]; Urban and Nipkow [2009]. Many of these early works axiomatize (part of) unification. More recent examples include CakeML [Tan et al. 2015], PureCake [Kanabar et al. 2023], and work by Zhao et al. [2019]. It should be stressed that many of these related works are concerned with object languages that are more expressive than the simply typed lambda calculus. They include language features such as polymorphism that we do not discuss in this paper. For a more thorough discussion, see Future work (§7.1).

Handling of metavariables. The original works by Damas and Milner [1978; 1982] handle unification variables somewhat implicitly. Follow up work places more emphasis on keeping track of where type variables are bound and what information is associated with them. The ordered contexts of [Dunfield 2009; Dunfield and Krishnaswami 2013; Bosman et al. 2023] explicitly track solved and unsolved existential variables. The ordering restricts which variables can appear in a solution.

Moreover, the typing relation is extended with an output context and a substitution is encoded in the derivations themselves. This is comparable to the passing of a world and a substitution to the continuation in our definition of bind. In a similar spirit, Gundry et al. [2010] present an implementation of polymorphic type inference which emphasizes managing ordered contexts. Similarly, Zhao et al. [2018, 2019] keep track of them explicitly in algorithmic worklists.

Comparable to our approach Asai and Kadowaki [2017] opt for an intrinsically-scoped representation for type inference. Like in our settings, this unfortunately creates a lot of noise. Similarly, Bach Poulsen et al. [2017] and van der Rest et al. [2022] use intrinsic scoping for dealing with metavariables in monadic computations. They use a monadic strength to hide explicit witnesses. This brings its own kind of awkwardness and noise to it, but more crucially, this concerns the programming side only, and that line of work does not include the development of program logics for their computations and it is not clear how to adapt this to reasoning in our setting. Finally, Rouvoet [2021] complements the work of Bach Poulsen et al. [2017] by introducing notations that hide worlds in the type signature, similar to Figure 12.

Program logics for monadic programs. Existing work has treated implementing and reasoning about monadic programs in proof assistants. Cock et al. [2008] develop a Hoare logic for the state monad with failures in Isabelle/HOL. The triples for basic monadic operations are structured in weakest-precondition form, in order to to allow for automation when reasoning. Swierstra [2009] develops the Hoare state monad, which indexed with a pre- and postcondition in an *intrinsic style*. Furthermore, Swierstra and Baanen [2019] explore predicate transformer semantics for computations represented by a free monad for, amongst others, the state, exception and non-determinism effects. An important realization is that (weakest precondition) predicate transformers can be seen as computations in a continuation monad with propositions as the answer type, also called the *backwards predicate transformer monad* [Swamy et al. 2013; Ahman et al. 2017]. Dijkstra monads [Swamy et al. 2013; Ahman et al. 2017] are monads indexed by their predicate transformers semantics, which can, for some effects, be generically constructed [Maillard et al. 2019]. This is an inherently intrinsic style, but it can also be used extrinsically [Silver and Zdancewic 2021].

Logics for type inference, freshness and constraints. McBride [2000, 2003b] develops a small base logic for the correctness proof of his unification algorithm [McBride 2003a], with parallel substitutions as the ghost state and triangular substitutions as the actual state. This includes the definition of a few connectives (conjunction, negation, equality), but not all connectives of predicate logic. More interestingly, McBride also defines a maximality modality for his base logic, to express that his algorithm yields most general unifiers. We hope to reuse this modality in future work,

Silva et al. [2020] use Swierstra's Hoare state monad with exceptions to mechanize a monadic formulation of Algorithm $\mathcal{W}$ in Coq. This is the only prior work we are aware of that provides a mechanized proof of a type inference algorithm and retains some effectful abstraction when reasoning. They use a single program logic that allows for failure, much like our $\widehat{\text{WLP}}$, and use it to establish soundness and principal typing[6]. However, they do not build a domain-specific base logic for their reasoning, but rather, use the Hoare state exception monad as a general purpose program logic for its effects, which means that "base logic details" shine through in their statements.

Nigron and Dagand [2021] develop a small base separation logic exploiting the properties of separation logic to enforce disjointness and thus freshness of identifiers. Building on their base logic, they define program logics for two different free monads involving freshness. Moreover, they also instantiate the MoSeL framework [Krebbers et al. 2018] for proof handling.

[6] Silva et al. [2020] claim proof of completeness, however, our definition of completeness is that the algorithm should not fail on typable inputs, which they do not show. Rather, we interpret their completeness statement as principal typing.

Correctness via logical relations. Keuchel et al. [2022] mechanize a verification condition generator based on symbolic execution. They define a base logic using symbolic propositions, i.e. deeply-embedded propositions, which only features restricted quantification over object language types, and directly implement the symbolic executor in a predicate transformer monad based on symbolic propositions. However, rather than to define elaborate reasoning rules for the base logic, or a program logic for their monad, they reduce the soundness of their symbolic executor, via a logical relation argument, to a higher-order abstract syntax variant similar to §4.7. They then proceed to show the soundness of this second program using conventional techniques.

However, Keuchel et al. [2022] only consider this proof method, while we also establish how to reason about a program that handles symbolic terms directly. Keuchel et al. [2022] only prove a soundness statement while we prove both soundness and completeness. Moreover, we provide several improvements to this method. Keuchel et al. [2022] use normal propositions, and makes the assignment part of the logical relation. Their logical relation is comparable, but all of our abstractions appear unfolded. As a result, their relatedness proofs are often interrupted my manual rewriting steps that apply infrastructure lemmas. These steps are hidden in our case, because they happen for instance transparently in the background during a modality introduction. We believe part of our automation is also applicable in their setting and could reduce their proof effort.

7 Conclusions and Future Work

In this paper, we presented proof techniques for the mechanization of type inference algorithms that extend to the problem of elaboration. In particular, we demonstrated that constraint-based, phase-separated type inference, employed by many functional programming language implementations, also exhibits modular implementation benefits at the level of specification. We described an approach that relied, among other things, on *weakest precondition predicate transformers* as a denotation of *constraints with semantic values*. To evaluate our approach, we developed a mechanized implementation of monadic, executable, phase separated type inference inside the Coq proof assistantand extracted it to Haskell. We applied this machinery to $\lambda_{\mathbb{B}}$, implementing and verifying the functional correctness of an executable type checker, type synthesizer and type reconstruction algorithm while leveraging proof automation to keep proof script size to a minimum.

7.1 Limitations and Future Work

At present, a major limitation of our approach is the expressiveness of the object language. In many ways the simply typed lambda calculus does not adequately represent realistic programming languages. Such languages typically include Hindley-Milner typing with numerous extensions including subtyping, higher-ranked polymorphism, impredicative polymorphism, GADTs, and others. Moreover, this limitation makes it difficult to judge how our approach compares to existing mechanical formalizations that target Hindley-Milner or higher-ranked polymorphic systems. We hope to rectify this in follow-up work that scales our implementation to support a richer set of language features. To this end, we outline several independent ideas on how to generalize the method to different polymorphic type systems, and what constitutes some of the necessary changes to make this work.

Hindley-Milner polymorphism. To scale the approach to include HM let-generalizations, several definitions have to be adapted. The presentation in this paper makes exclusive use of parallel substitutions, which do not scale to allow for generalization, because, as discussed in the final paragraph of §3.4, they represent more than weakenings. For HM-style polymorphism we believe it necessary to work with several subrelations, including the box and diamond modalities induced by each of the subrelations. Concretely, for generalization, we need to calculate the correct set of

variables at each generalization point. The parallel substitution only represents instantiations and therefore does not really carry the required information. Using a weakening-only relation, which we implemented and is the basis for $\hat{\text{Free}}$ and Prenex monads, we can represent variables that have been introduced after a generalization point.

Furthermore, our constraint language needs to be extended to account for generalization of monomorphic types to type schemes and instantiation of types schemes. To handle instantiations, one first needs to know which type scheme to instantiate. Alas, this depends on the result of the solver on a subproblem. To remedy this problem, and keep constraint generation and solving separate, Gustavsson and Svenningsson [2001] and Pottier and Rémy [2005] elegantly enrich the syntax of constraints with so-called *constraint abstractions* or *let constraints*, which take the place of type schemes. This representation thus remains first order and can be extended to support semantic values [Pottier 2014]. We expect that similar extensions to our constraint language are required to handle HM-style polymorphism.

On the solver side, we expect the unification algorithm to be largely reusable for monotypes, but the solver needs to account for generalizations. This amounts to calculating a *canonical solved form* from a *solved form* [Pottier and Rémy 2005]. As part of this, one usually needs to communicate information "to the left" and across generalization points. This is typically tackled by replacing existentials in the context with a set of new bindings [Bosman et al. 2023], or by moving existentials and enriching information in an ordered context [Gundry et al. 2010]. Again, parallel substitutions do not have enough structure to describe this, thus requiring us to adopt a bespoke subrelation.

Top level polymorphism. A weaker form of polymorphism is to only generalize top level definitions but not local `let`-bindings. This is for instance implemented in CakeML [Tan et al. 2015]. This approach typically interleaves constraint generation and solving by first generating the constraints for a single binding, solving those, then generalizing to construct a type scheme for the binding before continuing with the remaining top-level bindings. For the constraint language, we still expect something like *constraint abstractions* to be necessary, albeit with similar top-level restrictions. The benefits, however, lie in simplifying the generalization part of constraint solving.

Rigid universals. Another possible extension is the inclusion of so-called *rigid universal* or *Skolem* type variables [Vytiniotis et al. 2011]. These arise for instance from explicitly given type annotations in a language with `let`-polymorphism (even without generalization), and the Odersky-Läufer polymorphic subtyping system [Odersky and Läufer 1996]. Within the defining expression of the universal, the universal itself can be referenced, but it can never be substituted by a type. Hence, it acts more like a constant than a variable.

In the constraint language, we need to allow subconstraints to reference an additional set of universals that are in scope in this subconstraint only. We expect that this can be modeled by way of a new modality (akin to contextual modalities [Nanevski et al. 2008]) with its own set of properties. Furthermore, to pass a semantic value of such a subconstraint to a continuation, this extension would require an additional form of conjunction (or monadic bind). The continuation then receives the result also under this modality. This is similar to what is observed by Martinot and Scherer [2020], i.e. that *outputs turn into modal inputs*.

Efficient implementation. Many real-world implementations rely on highly space and time-efficient implementations for constraint solvers to minimize compile time. Of equal interest to us is therefore the mechanization of more efficient algorithms. On the one hand we wish to allow the (computationally relevant parts of) weakening (§5.1) to be done in constant time, and on the other hand we are interested in developing an *efficient union-find-based constraint solver* that leverages Coq's primitive arrays [Armand et al. 2010]. For this purpose, we hope to reuse our base logic

(which we believe is sufficiently modular) and build atop it a program logic that simply enforces consistency between the ghost and concrete state of the algorithm, i.e. a union-find data structure.

Object language parameterization. An obstacle to modularly reusing the code of our development is that the object language types are fixed to be that of our chosen variant of the simply typed lambda calculus and are currently not configurable. To address this we envision that the code needs to be abstracted over the object language types and related methods. This includes the injection of existential variables and closed types into open types, and traversals like substitution, an occurs check, and unification. This can be achieved by either directly abstracting over these elements and their proofs, or by adopting a datatype generic approach [Benke et al. 2003; Hinze 2000] in which the traversals and their properties are implemented by recursing over the structure of types. Datatype genericity has already been used in existing work to deal with intrinsically-scoped syntax with binders [Allais et al. 2018; Asai and Kadowaki 2017; Keuchel and Jeuring 2012]. For our purposes, a particular large universe of types that could be supported is that of fixed points of finitary containers [Abbott et al. 2003; Gambino and Hyland 2004; Jaskelioff and Rypacek 2012; Moggi et al. 1999] with the additional requirement that shapes have decidable equality. Both the restriction to a finite number of fields per shape (constructor) and decidable equality of shapes are required for unification.

Programming with presheaves. Next, we want to discuss the choice of working with indexed sets and monotone functions by means of a box modality. Ideally, we can work with presheaves in which every function is already monotone. We found that, without further support, working with presheaves aggravates the problem of passing explicit witnesses. Chiefly, the problem we are facing is that we do not know *how* to conveniently program in the internal language of the presheaf model inside the proof assistant. The switch to a more explicit modal-logic approach rather than an intuitionistic approach is a compromise, because we can be economic and make functions monotonic only when needed. We are aware of other recent work to program in presheaf models [Ceulemans et al. 2022], but we have not found anything that is practical enough for our needs. Nevertheless, we believe that more insights in this direction may lead to more concise formulations worth investigating.

Removing explicit weakening. Finally, related to the previous paragraph, our motivation for the world-indexing and explicit weakenings revolved around the discovery of necessary abstractions and their properties. Indeed, without it we would not have found the WP modalities of Figure 19 and in turn the $\blacksquare$ modality. Unfortunately, this created a lot of noise and boilerplate. Now that we have identified the abstraction, we will focus on a representation for variables that remains invariant under extension, i.e. when new existential variables are brought into scope. This will banish explicit witnesses entirely from the code. For example using de Bruijn levels, one can drop the world indexing and hide worlds entirely in the base logic. We hope to use our definitions to define program logics with modalities that remain silent, i.e. without explicit weakenings.

Acknowledgements

We thank Didier Rémy, Dominique Devriese, Georgios Karachalias, Justus Fasse, Roger Bosman, and the anonymous OOPSLA reviewers for their comments, feedback and suggestions on earlier drafts. This research is partially funded by the Research Fund KU Leuven, the Cybersecurity Research Program Flanders, and by ERC grant (UniversalContracts, 101040088). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Data-Availability Statement

Our results are publicly available [Carnier et al. 2024a,b]. The artifact contains the mechanization in the Coq proof assistant of all theorems and proofs discussed in the paper. Moreover, we provide example programs, Haskell sources, and a script to reproduce the benchmark results from §5.1.

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computation Structures (FoSSaCS'03)*, Andrew D. Gordon (Ed.). Springer, 23–38.

Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. In *Principles of Programming Languages (POPL)*. 515–529.

Ki Yung Ahn and Andrea Vezzosi. 2016. Executable Relational Specifications of Polymorphic Type Systems Using Prolog. In *Functional and Logic Programming (Lecture Notes in Computer Science, Vol. 9613)*. Springer, 109–125.

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (jul 2018), 30 pages.

Thorsten Altenkirch and Bernhard Reus. 1999. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In *Computer Science Logic*, Jörg Flum and Mario Rodriguez-Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–468.

Krzysztof R. Apt. 1983. Ten years of Hoare's logic: A survey—part II: Nondeterminism. *Theoretical Computer Science* 28, 1 (1983), 83–109.

Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. 2010. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–98.

Kenichi Asai and Kyoko Kadowaki. 2017. A Type Theoretic Specification of Type Inference. (2017). Unpublished.

Franz Baader, Wayne Snyder, Paliath Narendran, Manfred Schmidt-Schauss, and Klaus Schulz. 2001. Chapter 8 - Unification Theory. In *Handbook of Automated Reasoning*, Alan Robinson and Andrei Voronkov (Eds.). North-Holland, Amsterdam.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (dec 2017), 34 pages.

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing* 10, 4 (Dec. 2003), 265–289.

Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reason.* 49, 2 (aug 2012), 141–159.

Roger Bosman, Georgios Karachalias, and Tom Schrijvers. 2023. No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31-August 4, 2023, Białystok, Poland (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Luca Cardelli. 1987. Basic polymorphic typechecking. *Science of Computer Programming* 8, 2 (1987), 147–172.

Denis Carnier, François Pottier, and Steven Keuchel. 2024a. *Type Inference Logics - Artifact*.

Denis Carnier, François Pottier, and Steven Keuchel. 2024b. *Type Inference Logics - Software Repository*.

Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2022. Sikkel: Multimode Simply Type Theory as an Agda Library. In *Workshop on Mathematically Structured Functional Programming (MSFP)*. 93–112.

David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–182.

Haskell B. Curry and Robert M. Feys. 1958. *Combinatory Logic*. Vol. 1. North-Holland Publishing Company, Amsterdam.

Luis Damas. 1984. *Type Assignment in Programming Languages*. Ph.D. Dissertation. University of Edinburgh.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Principles of Programming Languages (POPL)*. 207–212.

Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (may 2001), 555–604.

Catherine Dubois and Valérie Ménissier-Morain. 1999. Certification of a Type Inference Tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning* 23, 3–4 (Nov. 1999), 319–346.

Jana Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML* (Edinburgh, Scotland) *(ML '09)*. Association for Computing Machinery, New York, NY, USA, 15–26.

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming (ICFP)*. 429–442.

Nicola Gambino and Martin Hyland. 2004. Wellfounded Trees and Dependent Polynomial Functors. In *Types for Proofs and Programs (LNCS, Vol. 3085)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer, 210–225.

Jacques Garrigue. 2015. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science* 25, 4 (2015), 867–891.

Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In *Workshop on Mathematically Structured Functional Programming (MSFP)*. 43–54.

Jörgen Gustavsson and Josef Svenningsson. 2001. Constraint Abstractions. In *Symposium on Programs as Data Objects (Lecture Notes in Computer Science, Vol. 2053)*. Springer.

Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic*, Peter G. Clote and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 317–331.

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the type inference process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 3–13.

Ralf Hinze. 2000. Generic programs and proofs. Habilitation thesis. Universität Bonn.

Mauro Jaskelioff and Ondrej Rypacek. 2012. An Investigation of the Laws of Traversals. In *Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming,* MSFP'12 *(Electronic Proceedings in Theoretical Computer Science, Vol. 76)*, James Chapman and Paul Blain Levy (Eds.). Open Publishing Association, 40–49.

Jean-Pierre Jouannaud and Claude Kirchner. 1991. Solving equations in abstract algebras: a rule-based survey of unification. In *Computational Logic. Essays in honor of Alan Robinson*, Jean-Louis Lassez and Gordon Plotkin (Eds.). MIT Press, Chapter 8, 257–321.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *International Conference on Functional Programming (ICFP)*. 256–269.

Hrutvik Kanabar. 2023. *Verified compilation of a purely functional language to a realistic machine semantics*. Ph.D. Dissertation. School of Computing, University of Kent.

Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 952–976.

Chantal Keller and Thorsten Altenkirch. 2010. Hereditary Substitutions for Simple Types, Formalized. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming* (Baltimore, Maryland, USA) *(MSFP '10)*. Association for Computing Machinery, New York, NY, USA, 3–10.

Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming). *Proc. ACM Program. Lang.* 6, ICFP, Article 97 (aug 2022), 31 pages.

Steven Keuchel and Johan T. Jeuring. 2012. Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming,* WGP '12. ACM, 57–68. Copenhagen, Denmark, September 12, 2012.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 77:1–77:30.

Robert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*.

Ramana Kumar and Michael Norrish. 2010. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *Interactive Theorem Proving*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–66.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and éric Tanter. 2019. Dijkstra monads for all. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 104:1–104:29.

Olivier Martinot and Gabriel Scherer. 2020. Quantified Applicatives: API design for type-inference constraints. Presented at the ML Family Workshop.

Conor McBride. 2000. *Dependently Typed Functional Programs and their Proofs.* Ph.D. Dissertation. University of Edinburgh.

Conor McBride. 2002. Elimination with a Motive. In *Types for Proofs and Programs*, Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–216.

Conor McBride. 2003a. First-Order Unification by Structural Recursion. *Journal of Functional Programming* 13, 6 (2003), 1061–1075.

Conor McBride. 2003b. *First-Order Unification by Structural Recursion – Correctness Proof.* Accessed: 2023-10-26.

Conor McBride. 2011. Functional pearl: Kleisli arrows of outrageous fortune. (2011). Available at https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf.

Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375.

E. Moggi, G. Bellè, and C.B. Jay. 1999. Monads, Shapely Functors and Traversals. *Electronic Notes in Theoretical Computer Science* 29 (1999), 187–208. CTCS '99, Conference on Category Theory and Computer Science.

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

Wolfgang Naraschewski and Tobias Nipkow. 1999. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23 (1999), 299–318.

Dieter Nazareth and Tobias Nipkow. 1996. Formal Verification of Algorithm W: The Monomorphic Case. In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science, Vol. 1125)*. Springer, 331–345.

Pierre Nigron and Pierre-évariste Dagand. 2021. Reaching for the Star: Tale of a Monad in Coq. In *Interactive Theorem Proving (ITP) (Leibniz International Proceedings in Informatics, Vol. 193)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:19.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations To Work. In *Principles of Programming Languages (POPL)*. 54–67.

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. . *Theory and Practice of Object Systems* 5, 1 (1999), 35–55.

Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (may 1976), 279–285.

Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation (PLDI)*. 199–208.

François Pottier. 2014. Hindley-Milner elaboration in applicative style. In *International Conference on Functional Programming (ICFP)*.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 284–298.

A. J. Rouvoet. 2021. *Correct by Construction Language Implementations*. Ph.D. Dissertation. Delft University of Technology.

Rafael Castro G. Silva, Cristiano D. Vasconcellos, and Karina Girardi Roggia. 2020. Monadic W in Coq. In *Brazilian Symposium on Programming Languages (SBLP)*. 25–32.

Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. *ACM SIGPLAN Notices* 48, 6 (2013), 387–398.

Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science, Vol. 5674)*. Springer, 440–451.

Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 103:1–103:26.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2.

Yong Kiam Tan, Scott Owens, and Ramana Kumar. 2015. A verified type system for CakeML. In *Implementation of Functional Languages (IFL)*. 7:1–7:12.

Aaron Tomb and Cormac Flanagan. 2005. Automatic type inference via partial evaluation. In *Principles and Practice of Declarative Programming (PPDP)*. 106–116.

Christian Urban and Tobias Nipkow. 2009. Nominal verification of algorithm W. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin (Eds.). Cambridge University Press, 363–382.

Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-Typed Definitional Interpreters à La Carte. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 192 (oct 2022), 30 pages.

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

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

Mitchell Wand. 1987. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticæ* 10 (1987), 115–122.

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. Formalization of a Polymorphic Subtyping Algorithm. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 604–622.

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 112:1–112:29.

Received 2024-04-06; accepted 2024-08-18