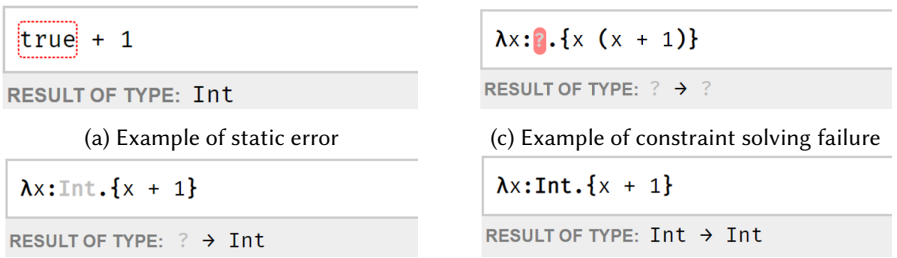
Type Hole Inference

ZHIYI PAN*, University of Michigan

1 INTRODUCTION

Bidirectional typing and *constraint-based type inference* are common approaches to deducing types for partially annotated programs. *Bidirectional typing* is a simple algorithmic system with type information propagating from the outside in [Dunfield and Krishnaswami 2019]. This produces clear error messages at the cost of requiring explicit type annotations in certain situations, e.g. top-level functions. *Type inference* allows programmers to omit most or all type annotations, but requires complex constraint solving for type checking, making it difficult for users to reason about types and producing complex error messages [Loncaric et al. 2016]. This paper develops *type hole inference*, which combines the benefits of bidirectional typing and type inference.

In recent work on the Hazel programming environment, Omar et al. [2017] assign formal meaning to incomplete programs, i.e. programs with holes, by developing a bidirectional type system with support for expression and type holes. Type holes operate as the unknown types from gradual type theory [Siek and Taha 2007]. Our approach takes this type system and adds constraint solving as a layer on top to suggest type hole fillings, without making constraint solving necessary for typing. Take the following Hazel programs extended with type hole inference as examples:



(a) Example of static error

(b) Example of constraint solving success

(c) Example of constraint solving failure

(d) Example of filling type holes

Fig. 1. Programs in Hazel environment

Figure 1a fails type checking at bidirectional typing, so our system reports a static error. Bidirectional judgements easily trace back the error and indicate the source in the editor with a red box, showing that the operand should be *Int* type.

Figure 1b is an example succeeding in both static checking and constraint solving. The bidirectional typing requires that a type annotation be provided on the lambda, but it is left as a hole. The system synthesizes the "$? \rightarrow Int$" type, with "?" standing for hole type. The constraint solver then generates a solution, $[(?,\ Int)]$, for the hole. Hence, the type hole is filled with a gray "*Int*" in the editor. The programmer can use a keyboard shortcut to replace the type hole with the suggested type as shown in Figure 1d, and the expression is synthesized to "$Int \rightarrow Int$" after replacement.

Figure 1c is an example failing in constraint solving. Constraints, $(? \sim Int)$ and $(? \sim ? \rightarrow ?)$, conflict since the type hole can not be *Int* and *arrow type* at the same time. In this case, the expression still remains well-typed since x is of hole type. The system postpones type-checking to runtime. However, the failure to infer a type is visually reported to indicate the issue.

*Research advisor: Cyrus Omar; ACM student member number: 5024951; Category: undergraduate

Author's address: Zhiyi Pan, University of Michigan, zhiyipan@umich.edu.

Type hole inference has a simple typing algorithm, simple error messages, and explicit annotations in the same situations that bidirectional typing does. However, programmers don't have to actually fill out those explicit annotations, most of which are type holes inserted and filled by the *type hole inference* automatically as shown in Figure 1b. The algorithm treats type holes as unknown types, and regards expressions well-typed no matter how constraint solving goes.

Contributions. The contributions of this paper are: (1) a new bidirectional typing system extended with type constraints in section 2; (2) a type inference algorithm to handle type holes in section 3.

2 TYPE SYSTEM FOR TYPE HOLE INFERENCE

Figure 2 defines the syntax of H-types and H-expressions. We start from the definition given in Omar et al. [2017] except two modifications. First, we add an annotated lambda. Second, we attach a unique *type hole identifier*, drawn as a superscript capital letter, to each type hole, such as $(\!|)^A$. In this way, type holes are identified as type variables. There are two types of expression holes: empty holes, $(\!|)$, standing for missing parts of an incomplete program, and non-empty holes, $(\!|e|\!)$, operating as membranes around static and dynamic type inconsistencies [Omar et al. 2019].

$$
\begin{array}{lcl}
HTyp\ \tau & ::= & \mathsf{num} \mid \tau \to \tau \mid (\!|)^A \\
HExp\ e & ::= & x \mid (\lambda x.e) \mid (\lambda x : \tau.e) \mid e(e) \mid \underline{n} \mid (e + e) \mid e : \tau \mid (\!|) \mid (\!|e|\!)
\end{array}
$$

Fig. 2. Syntax of H-types, H-expressions, and Type Constraint Set

Figure 3 defines a bidirectional typing system extended with type constraint sets. *Type constraint set* C is a set of type consistency equations, namely *type constraints*. Type consistency $\tau \sim \tau$ is a symmetric and reflective but not transitive relation defined in figure 4 [Omar et al. 2017]. We use a union operation, $C \cup C$, corresponding to mathematical set union operation to generate constraints inductively through bidirectional propagation. The type constraint set is solved by the unification algorithm in section 3, but importantly, constraint solving is not necessary for typing to succeed. The typing context, Γ, maps a set of expression variables to their types. Rule 1e and 1f synthesize expression hole to hole type, with premise, (A fresh), indicating generation of a new *type identifier*. Rule 2a and 2b have type consistency in their premises, generating new constraints and merging them into constraint sets in the conclusion. Rule 1h, 2b and 2c have *matched arrow type judgements* defined in figure 5. They leave the arrow type unchanged and assign the type hole the matched arrow type $(\!|)^B \to (\!|)^C$ with fresh identifiers and constraint generation [Omar et al. 2017].

3 TYPE HOLE INFERENCE ALGORITHM

Type hole inference takes two steps: (1) use bidirectional typing for type checking, type synthesis and constraint generation; (2) solve the constraint set by unification to infer types for type holes [Pierce 2002]. In contrast to *type inference*, our approach has the following features: (1) we separate type checking and constraint solving into two steps. The system only does type checking and triggers static errors at the first step; (2) expressions remain well-typed when the constraint solver can not find a solution for type variables, for instance when one type variable is equal to multiple types. The system postpones the remaining type checking to runtime. Since type variables may be unconstrained after unification, we can generalize those type holes by introducing polymorphism in the future work.

Constraint Generation. Constraints are generated through bidirectional judgements in figure 3. Take $(\lambda x : (\!|)^A.x + 1)\ (\!|)$ as an example. We apply rule 1h, 1g, 1b, 2a and 1e to derive a constraint set: $\{(\!|)^A \sim \mathsf{num};\ (\!|)^A \sim (\!|)^B\}$, where $(\!|)^B$ is a fresh type hole generated in rule 1e.

$\boxed{\Gamma \vdash e \Rightarrow \tau \mid C}$ e synthesizes τ

$$\frac{}{\Gamma, x : \tau \vdash x \Rightarrow \tau \mid \{\}} \quad (1a)$$

$$\frac{}{\Gamma \vdash \underline{n} \Rightarrow \mathsf{num} \mid \{\}} \quad (1b)$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \mathsf{num} \mid C_1 \qquad \Gamma \vdash e_2 \Leftarrow \mathsf{num} \mid C_2}{\Gamma \vdash (e_1 + e_2) \Rightarrow \mathsf{num} \mid C_1 \cup C_2} \quad (1c)$$

$$\frac{\Gamma \vdash e \Leftarrow \tau \mid C}{\Gamma \vdash (e : \tau) \Rightarrow \tau \mid C} \quad (1d)$$

$$\frac{(A \text{ fresh})}{\Gamma \vdash (\!|\,|\!) \Rightarrow (\!|\,|\!)^A \mid \{\}} \quad (1e)$$

$$\frac{(A \text{ fresh}) \qquad \Gamma \vdash e \Rightarrow \tau \mid C}{\Gamma \vdash (\!|e|\!) \Rightarrow (\!|\,|\!)^A \mid C} \quad (1f)$$

$$\frac{\Gamma, x : \tau_{in} \vdash e \Rightarrow \tau_{out} \mid C}{\Gamma \vdash (\lambda x : \tau_{in}.e) \Rightarrow \tau_{in} \to \tau_{out} \mid C} \quad (1g)$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid C_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_{in} \to \tau_{out} \mid C_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_{in} \mid C_3}{\Gamma \vdash e_1(e_2) \Rightarrow \tau_{out} \mid C_1 \cup C_2 \cup C_3} \quad (1h)$$

$\boxed{\Gamma \vdash e \Leftarrow \tau \mid C}$ e analyzes against τ

$$\frac{\Gamma \vdash e \Rightarrow \tau' \mid C_1 \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \mid C_1 \cup \{\tau \sim \tau'\}} \quad (2a)$$

$$\frac{\tau \blacktriangleright_{\to} \tau_{in} \to \tau_{out} \mid C_1 \qquad \Gamma, x : \tau_{in} \vdash e \Leftarrow \tau_{out} \mid C_2}{\Gamma \vdash (\lambda x.e) \Leftarrow \tau \mid C_1 \cup C_2} \quad (2b)$$

$$\frac{\tau \blacktriangleright_{\to} \tau_{in} \to \tau_{out} \mid C_1 \qquad \Gamma, x : \tau'_{in} \vdash e \Leftarrow \tau_{out} \mid C_2 \qquad \tau_{in} \sim \tau'_{in}}{\Gamma \vdash (\lambda x : \tau'_{in}.e) \Leftarrow \tau \mid C_1 \cup C_2 \cup \{\tau_{in} \sim \tau'_{in}\}} \quad (2c)$$

Fig. 3. H-type synthesis and analysis.

$\boxed{\tau \sim \tau}$ τ is consistent to τ

$$\frac{}{\mathsf{num} \sim \mathsf{num}} \qquad \frac{}{(\!|\,|\!)^A \sim \tau} \qquad \frac{}{\tau \sim (\!|\,|\!)^A} \qquad \frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4} \quad (3a\text{-}d)$$

Fig. 4. H-type consistency.

$\boxed{\tau \blacktriangleright_{\to} \tau_{in} \to \tau_{out} \mid C}$ τ has matching arrow type $\tau_{in} \to \tau_{out}$

$$\frac{}{\tau_{in} \to \tau_{out} \blacktriangleright_{\to} \tau_{in} \to \tau_{out} \mid \{\}} \quad (4a) \qquad \frac{(B \text{ fresh}) \qquad (C \text{ fresh})}{(\!|\,|\!)^A \blacktriangleright_{\to} (\!|\,|\!)^B \to (\!|\,|\!)^C \mid \{(\!|\,|\!)^A \sim (\!|\,|\!)^B \to (\!|\,|\!)^C\}} \quad (4b)$$

Fig. 5. Matched arrow types.

Constraint Solver. Type constraint sets are solved by a standard unification algorithm [Robinson 1965]. The solution is a list of pairs, (*type hole identifier*, H-type), with substituting each type hole with corresponding H-type unifying a given constraint set [Pierce 2002]. For example, the solution for the constraint set in the previous paragraph is [("A",num); ("B", num)].

4 CONCLUSION

This paper develops *type hole inference* which takes advantage of bidirectional typing and type inference. *Type hole inference* is based on simple bidirectional typing for straightforward type checking and clear error messages, then adopts the hole typing system and constraint solving to insert and fill type annotations automatically. In the future work, the constraint solver will support returning a list of potential H-types for type holes when unification can not find a solution. This will be implemented into the Hazel environment, so programmers can make better type decisions during development.

REFERENCES

Joshua Dunfield and Neel Krishnaswami. 2019. Bidirectional Typing. https://arxiv.org/pdf/1908.05839.pdf

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 781âĂŞ799. https://doi.org/10.1145/2983990.2983994

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Benjamin C. Pierce. 2002. Type Reconstruction. In *Types and Programming Languages*. MIT Press, 317–338.

J Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM (JACM)* 12, 1 (1965), 23âĂŞ41. https://doi.org/10.1145/321250.321253

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Functional Languages. (2007), 81–92.