Total Type Error Localization and Recovery with Holes

ANONYMOUS AUTHOR(S)

Type systems typically only define the conditions under which an expression is well-typed, leaving ill-typed expressions formally meaningless. This approach is insufficient as the basis for language servers driving modern programming environments, which are expected to recover from simultaneously localized errors and continue to provide a variety of downstream semantic services. This paper addresses this problem, contributing the first comprehensive formal account of total type error localization and recovery: the marked lambda calculus. In particular, we define a gradual type system for expressions with marked errors, which operate as non-empty holes, together with a total procedure for marking arbitrary unmarked expressions. We mechanize the metatheory of the marked lambda calculus in Agda and implement it, scaled up, as the new basis for Hazel, a full-scale live functional programming environment with, uniquely, no meaningless editor states.

The marked lambda calculus is bidirectionally typed, so localization decisions are systematically predictable based on a local flow of typing information. Constraint-based type inference can bring more distant information to bear in discovering inconsistencies but this notoriously complicates error localization. We approach this problem by deploying constraint solving as a type-hole-filling layer atop this gradual bidirectionally typed core. Errors arising from inconsistent unification constraints are localized exclusively to type and expression holes, i.e. the system identifies unfillable holes using a sytem of traced provenances, rather than localized in an *ad hoc* manner to particular expressions. The user can then interactively shift these errors to particular downstream expressions by selecting from suggested partially consistent type hole fillings, which returns control back to the bidirectional system. We implement this type hole inference system in Hazel.

1 INTRODUCTION

Modern programming environments provide developers with a collection of semantic services—for example, type hints, semantic navigation, semantic code completion, and automated refactorings—that require static reasoning about the type and binding structure of a program as it is being edited. The problem is that when the program being edited is ill-typed, these semantic services can become degraded or unavailable [Omar et al. 2017b]. These gaps in service are not always transient. For example, a change to a type definition might result in type errors at dozens of use sites in a large program, which might take hours or days to resolve, all without the full aid of these services.

These service gaps are fundamentally rooted in a definitional gap: a type system defined in the conventional way, e.g. in the tradition of the typed lambda calculus and its derivatives [Pierce 2002a], assigns meaning only to well-typed programs. If a type error appears *anywhere*, the program is formally meaningless *everywhere*.

This gap problem has prompted considerable practical interest in (1) **type error localization**: mechanisms for identifying the location(s) in a program that explain a type error, and (2) **type error recovery**: mechanisms that allow the system to optimistically recover from a localized type error and continue on to locate other errors and provide downstream semantic services, ideally at every location in the program and with minimal degradation in service. Essentially all widely-used programming systems have some support for type error localization, e.g. in compiler error messages or directly in the editor via markings decorating localized errors. Developers are known to attend to reported error locations when debugging type errors [Joosten et al. 1993]. Many systems also attempt recovery in certain situations, discussed below. However, type error localization and recovery mechanisms have developed idiosyncratically, in part as folklore amongst language and tool implementors. Different type checkers or language servers [Barros et al. 2022; Bour et al. 2018], even for the same language, localize and recover from type errors in different ways, with little in the way of unifying theory of the sort that grounds the design of modern type systems themselves.

Consider, for example, the ill-typed program below, which is shown as presented to the user in this paper's version of Hazel [Hazel Development Team 2023], a typed functional dialect of Elm [Czaplicki 2018]. Hazel supports local type inference, specified in the well-established bidirectional style [Dunfield and Krishnaswami 2019; Omar et al. 2017a; Pierce and Turner 2000a]:

```
let x =
  if f(y) then true else 1
in
if x then x + "abc" else 2
```

A type checker with no support for error localization or recovery—as students in an undergraduate course might write—would simply report that this program is ill-typed. A more practical approach, and one common even in production type checkers, is to localize the first error that causes local type inference to fail and emit an explanatory error message before terminating. In this example, the system might report that the variable `f` located on Line 2 is free, then terminate.

An implementation with support for type error recovery, like Elm's compiler or OCaml's `merlin` [Bour et al. 2018], would be tasked with continuing past this first error. The general difficulty here is that there is now missing semantic information, namely the type of `f`, that the bidirectional type system as specified would appear to demand in order to proceed, here in order to determine which type the argument, `y`, is to be analyzed against. Intuitively, however, it is clear that this knowledge is *unnecessary* to make an error localization decision about `y`: it is free, so a second error can be simultaneously localized despite the missing information about `f`.

To recover further, we might choose to also ignore the bidirectional type system's demand that the conditional guard be confirmed to be a boolean expression (because the type of `f(y)` is also not well-defined) and continue into its branches, observing that they have inconsistent types, `Bool` and `Int`. There are several ways to localize this error. One approach, taken e.g. by Elm and `merlin`, would be to assume, arbitrarily, that one branch is correct, localizing the error to the other branch. Heuristics have been developed to help make this choice less arbitrary, e.g. by observing recent editing history [Chuchem and Lotem 2019], or training a machine learning model [Seidel et al. 2017]. A less *ad hoc* approach, which Hazel takes above, is to localize the inconsistency to the conditional as a whole, reporting that the problem is that the branch types differ. When the cursor is on this conditional expression, the Hazel type inspector—a status bar that reports typing information, including error messages, about the term at the cursor [Potter and Omar 2020]—displays:

This localization decision affects how the system recovers as it proceeds into the let body. If localization had assumed that the **`then`** branch were correct, as for example in `merlin`, then `x : Bool` and an error should be reported on only the second use of `x`. If the **`else`** branch were chosen, then `x : Int` and an error would be reported on only the first use of `x`. In either case, this error might mislead the programmer if the earlier localization guess was incorrect. If the inconsistency were localized to the conditional expression as a whole, as in Hazel, then we again confront the problem of missing type information: `x` does not have a known type, though it *is* known to be bound. There is no definitive type or binding error at either use of `x`, so we do not report an error in Hazel. More specifically, we treat `x`'s type as the unknown a.k.a. dynamic type from gradual typing [Siek and Taha 2006]. In any of these cases, we would like to be able to recover and localize the type inconsistency on the string addend because the `+` operator is integer addition in Hazel. Recovering from this error, we can assume that both branch types in the second conditional will have `Int` type, so no error is marked on the conditional as a whole.

This informal exercise demonstrates that (1) localization choices can vary, particularly with regard to the extent to which they make *ad hoc* guesses about intent; (2) when combined with error recovery, one localization decision can influence downstream localization decisions; and (3) error recovery necessitates reasoning without complete knowledge about types and binding. We argue that such semantic subtleties call for a rigorous theoretical treatment of these topics.

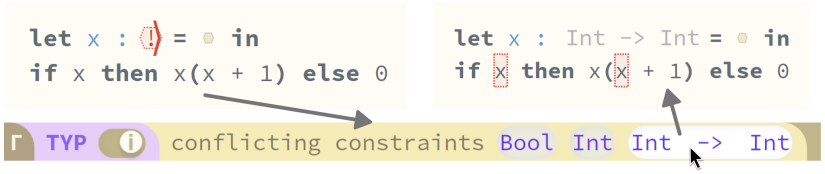
Section 2 of this paper develops the first comprehensive type-theoretic formulation of type error localization and recovery, called the *marked lambda calculus*. We bring together three individually well-studied ideas: (1) bidirectional typing, which specifies how type and binding information flows from type annotations to expressions [Dunfield and Krishnaswami 2019; Pierce and Turner 2000a], (2) gradual typing, which offers a principled approach for recovering from missing type information [Siek and Taha 2006; Siek et al. 2015], and (3) non-empty holes, which function as syntactic membranes marking erroneous terms [Omar et al. 2017a], operationalizing the "red lines" that an editor displays. The marked lambda calculus achieves *total error recovery*, meaning that *every* syntactically well-formed program sketch (a program structure with, optionally, empty holes [Solar-Lezama 2013]) can be marked, i.e. its errors can be simultaneously localized, such that the resulting program has a well-defined type and binding structure. We establish this and other properties as metatheorems that we mechanize in the Agda proof assistant. As we define the calculus, we consider a number of situations where error localization decisions are subtle and conclude by defining some extensions of the intentionally minimal core calculus, e.g. with destructuring patterns and System F-style polymorphism, that serve as case studies of the general approach that we hope this paper will be adopted by language designers defining practical type systems.

Section 3.1 describes our own effort in this direction, which is to scale up the marked lambda calculus as the new basis for Hazel, a typed functional programming environment that fully solves the semantic gap problem: Hazel's semantic services are available as long as the program sketch is syntactically well-formed. Prior work has separately considered mechanisms for maintaining syntactically well-formed program sketches during development: manual hole insertion (as in Agda [Norell 2007], GHC Haskell [HaskellWiki 2014], Idris [Brady 2013], and others), syntax error recovery [Queiroz de Medeiros et al. 2020; Sorkin and Donovan 2011], and structure editing [Omar et al. 2017a; Teitelbaum and Reps 1981]. Hazel has both a textual syntax and a structure editor [Moon et al. 2023; Omar et al. 2017a]. As a secondary contribution, Section 3.2 describes how (1) total marking can resolve the problem of undefined behavior in the Hazelnut structure editor calculus, and (2) integrating marking with typed structure editing allows us to incrementally re-mark only where necessary based on the edit location and the type and binding structure.

The developments in Section 2 support the intuition (remarked upon by Pierce and Turner [2000a] and Dunfield and Krishnaswami [2019]) that local type inference pairs well with type error localization and recovery because information flows systematically and locally through the tree. However, many functional languages including Elm, OCaml and Haskell feature constraint-based type inference, with constraints gathered globally. For type systems where inference is decidable, this is powerful, but it is also notorious for complicating error localization, because inconsistencies can arise through a confluence of constraints originating from any number of locations [Wand 1986].

For example, in the following Hazel expression (ignoring the error on the type hole for the moment), bidirectional error localization does not mark any uses of `x` as erroneous because a type for `x` cannot locally be inferred (the let-bound expression is an empty hole, which has unknown type). However, by gathering constraints from the three uses of `x` and attempting to unify, we see that the type hole is unfillable. Rather than arbitrarily privileging one of these uses, as is the case in languages like OCaml (and which has led to the development of complex heuristics, e.g. using

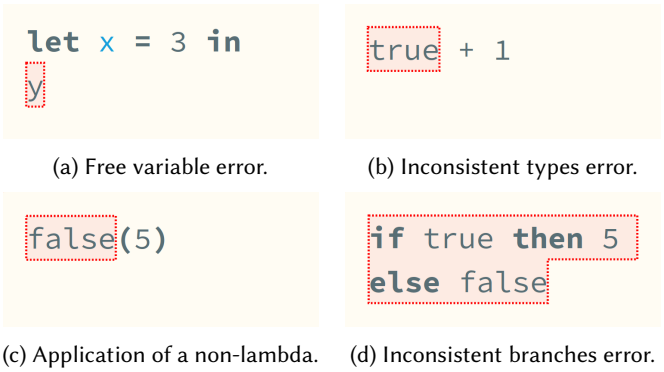
machine learning [Seidel et al. 2017], to avoid misleading programmers), this error is localized, with a !, to the type hole itself, with the error message providing partially consistent solutions as shown.



The user can interactively explore possible error localizations by hovering over a partially consistent suggestion, which (temporarily) fills the type hole and thus returns the localization decision to the local inference system. Here, each choice causes a different set of two uses of x to be marked, e.g. the first and third use for `Int -> Int` (shown above). Until the choice is finalized, the type hole continues to operate gradually. Section 4 describes this distinctively neutral approach to blending local and constraint-based type inference in Hazel, where local inference is used to systematically mark erroneous expressions in the program and constraint-based inference (using entirely standard algorithms, which we do not repeat in this paper) is used exclusively to locate unfillable holes.

2 THE MARKED LAMBDA CALCULUS

To begin to motivate the development of this section, consider Figure 1, which shows common type errors as they appear in the Hazel programming environment. These programs are syntactically well-formed but ill-typed.



(a) Free variable error.

(b) Inconsistent types error.

(c) Application of a non-lambda.

(d) Inconsistent branches error.

Fig. 1. Examples of common type errors.

The key contribution of this section is the *marked lambda calculus*, a calculus based on the gradually typed lambda calculus (GTLC) that formalizes the mechanism by which errors like these can be localized, and how to recover, in all cases, from such errors. Section 2.1 introduces the syntax, judgemental structure, guiding metatheory, and then goes through the rules, organized by syntactic form rather than judgement, to build intuition about how the various judgements relate to one another [Siek and Taha 2006]. All rules and theorems may be found organized by judgement form in the supplementary appendix for reference, alongside a complete mechanization in the Agda proof assistant [Norell 2007] (discussed in Section 2.2). We intentionally keep the marked lambda calculus minimal, because it is intended to capture the essential idea and introduce

a general pattern that language designers can employ to created mark variants of their own type systems. As an initial example, we consider a surprisingly subtle combination of features as a more substantial case study: the combination of destructing let expressions with granular type annotations, in Section 2.3. Finally, Section 2.4 briefly explores how the system might be extended to more complex judgemental structures, such as parametric polymorphism.

2.1 The Core Calculus

The marked lambda calculus is based on the gradually typed lambda calculus [Siek and Taha 2006] extended with numbers, booleans, pairs, and empty expression holes [Omar et al. 2017b]. Given in Figure 2, the syntax consists of two expression languages:

- The **unmarked language**, which is the original language. Expressions of this language are called *unmarked expressions*, denoted by the metavariable e.
- The **marked language**, which mirrors the structure of the unmarked language but is extended with explicit *error marks*. We call expressions of this language *marked expressions*, denoted $\check{e}$.

In this simple setting, we only need one sort for types, τ. The base types num and bool classify number and boolean expressions. The number literal corresponding to the mathematical number n is given by $\underline{n}$, and there is a single arithmetic operation, addition. tt and ff are the boolean values and if e_1 then e_2 else e_3 is the boolean conditional. Arrow and product types classify lambda abstractions and pairs, respectively, in the usual way.

? is a type hole, which we identify with the unknown type from gradual type theory [Siek and Taha 2006]. Finally, to model the edit state of a program in development, $(\!|\,|\!)$ denotes an *empty expression hole*, used to represent syntactically incomplete portions of the program à la Hazelnut [Omar et al. 2017a]. Note, however, that empty expression holes are not semantically critical to the calculus—they are included to emphasize that the system can handle program sketches.

$$
\begin{array}{rrcl}
\text{Type} & \tau & ::= & ? \mid \mathsf{num} \mid \mathsf{bool} \mid \tau \to \tau \mid \tau \times \tau \\
\text{UExp} & e & ::= & x \mid \lambda x : \tau.\ e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \underline{n} \mid e + e \\
& & \mid & \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e, e) \mid \pi_1 e \mid \pi_2 e \mid (\!|\,|\!) \\
\text{MExp} & \check{e} & ::= & x \mid \lambda x : \tau.\ \check{e} \mid \check{e}\ \check{e} \mid \mathsf{let}\ x = \check{e}\ \mathsf{in}\ \check{e} \mid \underline{n} \mid \check{e} + \check{e} \\
& & \mid & \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ \check{e}\ \mathsf{then}\ \check{e}\ \mathsf{else}\ \check{e} \mid (\check{e}, \check{e}) \mid \pi_1 \check{e} \mid \pi_2 \check{e} \mid (\!|\,|\!) \\
& & \mid & (\!|x|\!)_{\square} \mid (\!|\check{e}|\!)_{\not\to} \\
& & \mid & (\!|\lambda x : \tau.\ \check{e}|\!)_{:} \mid (\!|\lambda x : \tau.\ \check{e}|\!)^{\Leftarrow}_{\blacktriangleright\not\to} \mid (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright\not\to}\ \check{e} \\
& & \mid & (\!|\mathsf{if}\ \check{e}\ \mathsf{then}\ \check{e}\ \mathsf{else}\ \check{e}|\!)_{\not\mathsf{tt}\not\mathsf{ff}} \\
& & \mid & (\!|(\check{e}, \check{e})|\!)^{\Leftarrow}_{\blacktriangleright\not\times} \mid \pi_1 (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright\not\times} \mid \pi_2 (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright\not\times}
\end{array}
$$

Fig. 2. Syntax of the marked lambda calculus.

Before giving any of the rules, let us summarize the overall judgemental structure of the calculus. Note that colors in the judgement forms below are entirely redundant reading aids—color has no semantic significance.

As our starting point, types classify unmarked expressions by a completely standard *bidirectional type system* [Dunfield and Krishnaswami 2019; Pierce and Turner 2000a], which employs two mutually defined judgments. *Type synthesis*, written $\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau$, establishes that, under the typing context Γ, the expression e synthesizes or locally infers the type τ. *Type analysis*, written $\Gamma \vdash_{\mathcal{U}} e \Leftarrow \tau$, states that the expression e may appear where an expression of type τ is expected.

The key operation is *marking*, which transforms an unmarked expression into a marked expression, inserting error marks where appropriate. This corresponds to a type checking process with error localization and recovery. The marked language may then serve as a foundation for other semantic services, such as constraint-based inference (as discussed in Section 4). Intuitively, each of the mark forms corresponds to a different kind of error message that might be shown by an editor or emitted by a compiler. Note that we do not specify what the error message should say in this paper. Throughout the remainder of this section, as we formulate marking for the GTLC, we will motivate and give precise semantics for each error mark. Furthermore, when the core is extended, new marks may be needed, as we discuss below.

The marked language possesses its own type system, also formulated bidirectionally. We write $\Gamma \vdash_{\not M} \check{e} \Rightarrow \tau$ for synthesis and $\Gamma \vdash_{\not M} \check{e} \Leftarrow \tau$ for analysis (in addition to the color difference, note the subscript on the turnstile to distinguish marked from unmarked typing).

Finally, the key judgement, *marking*, is also of a bidirectional nature. The synthetic marking judgment $\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau$ establishes that, under the context Γ, the unmarked expression e is "marked into" the marked expression $\check{e}$, which synthesizes type τ. Analogously, the analytic marking judgment $\Gamma \vdash e \leadsto \check{e} \Leftarrow \tau$ states that e is marked into $\check{e}$, which analyzes against τ.

How can we ensure that the marking procedure is correctly defined? There are two critical metatheorems that guide us as we continue. The first is a *totality* of marking:

Theorem 2.1 (Marking Totality).

(1) For all Γ *and* e*, there exist* $\check{e}$ *and* τ *such that* $\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau$ *and* $\Gamma \vdash_{\not M} \check{e} \Rightarrow \tau$.

(2) For all Γ*,* e*, and* τ*, there exists* $\check{e}$ *such that* $\Gamma \vdash e \leadsto \check{e} \Leftarrow \tau$ *and* $\Gamma \vdash_{\not M} \check{e} \Leftarrow \tau$.

That is, we may mark *any* syntactically well-formed program in any context, resulting in a *well-typed* marked program.

Furthermore, since error marks are effectively annotations on top of the program, marking should preserve syntactic structure modulo those marks. Figure 3 gives selected clauses of the definition of *mark erasure*, which converts marked expressions back into unmarked ones by removing error marks.

$$
\begin{array}{rcl}
(\!|\,|\!)^{\square} & = & (\!|\,|\!) \\
x^{\square} & = & x \\
(\lambda x : \tau.\ \check{e})^{\square} & = & \lambda x : \tau.\ (\check{e}^{\square}) \\
(\check{e}_1\ \check{e}_2)^{\square} & = & (\check{e}_1^{\square})\ (\check{e}_2^{\square}) \\
 & \cdots & \\
(\!|x|\!)_{\square}^{\square} & = & x \\
(\!|\check{e}|\!)_{\nearrow}^{\square} & = & \check{e}^{\square} \\
(\!|\text{if } \check{e}_1 \text{ then } \check{e}_2 \text{ else } \check{e}_3|\!)_{\downarrow\!\!\!/}^{\square} & = & \text{if } (\check{e}_1^{\square}) \text{ then } (\check{e}_2^{\square}) \text{ else } (\check{e}_3^{\square})
\end{array}
$$

Fig. 3. Mark erasure (selected)

Then, Theorem 2.2 provides the necessary *well-formedness* criterion for marking.

Theorem 2.2 (Marking Well-Formedness).

(1) If $\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau$*, then* $\check{e}^{\square} = e$.

(2) If $\Gamma \vdash e \leadsto \check{e} \Leftarrow \tau$*, then* $\check{e}^{\square} = e$.

Together, these metatheorems imply that to go from a standard bidirectional system for the unmarked language to a marking system, we need to handle all possible failure modes with

appropriate marks and marking logic (otherwise totality would be violated) and without otherwise changing the program (otherwise well-formedness would be violated). Now, let us consider each form in turn.

2.1.1 Numbers. To start, consider the simple case of numbers. Because of *subsumption*, which is discussed in more detail next, we need only define a synthesis rule for unmarked numbers, in which number literals synthesize the type num:

$$\textsc{USNum}\ \frac{}{\Gamma \vdash_{\mathcal{U}} \underline{n} \Rightarrow \mathsf{num}}$$

How should numbers be marked? Straightforwardly, we simply give the same number as a marked expression, synthesizing again num. No type errors may occur in a just single number, so we only need the following rules for marking numbers and typing the results:

$$\textsc{MKSNum}\ \frac{}{\Gamma \vdash \underline{n} \hookrightarrow \underline{n} \Rightarrow \mathsf{num}} \qquad \textsc{MSNum}\ \frac{}{\Gamma \vdash_{\mathcal{M}} \underline{n} \Rightarrow \mathsf{num}}$$

For addition expressions, the type of both operands should be num. To denote this in a bidirectional system, they are analyzed against num, giving the following typing rule for unmarked addition expressions:

$$\textsc{USPlus}\ \frac{\Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \mathsf{num} \qquad \Gamma \vdash_{\mathcal{U}} e_2 \Leftarrow \mathsf{num}}{\Gamma \vdash_{\mathcal{U}} e_1 + e_2 \Rightarrow \mathsf{num}}$$

The marking rule parallels USPlus closely. Since an expected type for each operand is known, we shift the responsibility for any type errors to them. Hence, we recursively mark each operand in analytic mode and rebuild the marked addition expression. The typing rule for marked addition expressions then mirrors USPlus exactly.

$$\textsc{MKSPlus}\ \frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \mathsf{num} \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow \mathsf{num}}{\Gamma \vdash e_1 + e_2 \hookrightarrow \check{e}_1 + \check{e}_2 \Rightarrow \mathsf{num}} \qquad \textsc{MSPlus}\ \frac{\Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \mathsf{num} \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Leftarrow \mathsf{num}}{\Gamma \vdash_{\mathcal{M}} \check{e}_1 + \check{e}_2 \Rightarrow \mathsf{num}}$$

2.1.2 Subsumption. Only synthetic rules are necessary for typing numbers and addition because of *subsumption*, given below, which states that if an expression synthesizes a type, it may also be analyzed against that type or any consistent type.

$$\textsc{UASubsume}\ \frac{\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau' \qquad \tau \sim \tau' \qquad e\ \mathsf{subsumable}}{\Gamma \vdash_{\mathcal{U}} e \Leftarrow \tau}$$

We rely on the notion of *type consistency* from gradual type theory, which defines a reflexive and symmetric (but not transitive) relation between types, writing $\tau_1 \sim \tau_2$ to mean that τ_1 is consistent with τ_2. Defined in Figure 4, this replaces the notion of *type equality* to relate the unknown type to all other types.

Hence, when checking $\underline{n}$ against a known expected type, subsumption checks that the type is consistent with num, the type that $\underline{n}$ synthesizes. This succeeds for ? and num.

We also restrict the usage of subsumption to "subsumable" syntactic forms, written e subsumable. This judgment is defined for all syntactic forms except lambda abstractions, conditionals, and pairs, the only ones with both synthesis and analysis rules (see below). In other words, we restrict

$\boxed{\tau_1 \sim \tau_2}$ τ_1 is consistent with τ_2

$$\textsc{TCUnknown1}\ \frac{}{? \sim \tau} \qquad \textsc{TCUnknown2}\ \frac{}{\tau \sim ?} \qquad \textsc{TCRefl}\ \frac{}{\tau \sim \tau} \qquad \textsc{TCArr}\ \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Fig. 4. Type consistency.

subsumption to be the rule of "last resort", which is necessary to establish that marking and typing are deterministic (see Theorem 2.4).

Now, to define analytic marking on forms without explicit analytic typing rules, we also need subsumption rules for marking and marked expression typing. Note that we define an analogous notion of "subsumability" for marked expressions, written $\check{e}$ subsumable.

$$\textsc{MKASubsume}\ \frac{\Gamma \vdash e \rightsquigarrow \check{e} \Rrightarrow \tau' \qquad \tau \sim \tau' \qquad e \text{ subsumable}}{\Gamma \vdash e \rightsquigarrow \check{e} \Lleftarrow \tau} \qquad \textsc{MASubsume}\ \frac{\Gamma \vdash_{\mathbb{M}} \check{e} \Rrightarrow \tau' \qquad \tau \sim \tau' \qquad \check{e} \text{ subsumable}}{\Gamma \vdash_{\mathbb{M}} \check{e} \Lleftarrow \tau}$$

But what happens when the synthesized type of an expression is *not* consistent with the expected type, i.e. that the premise $\tau \sim \tau'$ fails? Recalling the example of Figure 1b, subsumption would be used to analyze tt against num when checking tt + 1, which fails since num $\nsim$ bool, i.e. UA$\textsc{Subsume}$ does not apply. At this point, traditional typing semantics would simply fail the type checking process. MKA$\textsc{Subsume}$ would also not be applicable, leaving marking undefined in those cases.

To satisfy marking totality, such a possibility motivates an *inconsistent type mark* $(\!|\check{e}|\!)_{\not\sim}$, which is applied when the synthesized type of $\check{e}$ is inconsistent with the expected type:

$$\textsc{MKAInconsistentTypes}\ \frac{\Gamma \vdash e \rightsquigarrow \check{e} \Rrightarrow \tau' \qquad \tau \not\sim \tau' \qquad e \text{ subsumable}}{\Gamma \vdash e \rightsquigarrow (\!|\check{e}|\!)_{\not\sim} \Lleftarrow \tau} \qquad \textsc{MAInconsistentTypes}\ \frac{\Gamma \vdash_{\mathbb{M}} \check{e} \Rrightarrow \tau' \qquad \tau \not\sim \tau' \qquad \check{e} \text{ subsumable}}{\Gamma \vdash_{\mathbb{M}} (\!|\check{e}|\!)_{\not\sim} \Lleftarrow \tau}$$

Observe that the premises of MKA$\textsc{InconsistentTypes}$ are identical to those of MKA$\textsc{Subsume}$, except that $\tau \not\sim \tau'$. By marking an error, the type checking process may carry on and provide semantic feedback for the rest of the program.

2.1.3 Variables. Let us now consider the case of variables. Typing in the unmarked language is standard, and because of subsumption, only a synthetic rule is required:

$$\textsc{USVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathbb{U}} x \Rrightarrow \tau}$$

That is, if x is bound to a type in the typing context, it synthesizes that type. Straightforwardly, marking converts an unmarked variable into the same variable via the following rules:

$$\textsc{MKSVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x \Rrightarrow \tau} \qquad \textsc{MSVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathbb{M}} x \Rrightarrow \tau}$$

However, consider the case that a variable, such as y in Figure 1a, is *not* bound. Similar to above, USVar would not apply. A total marking procedure should, however, report the error and continue,

motivating an *free variable mark* $(\!|x|\!)_{\square}$ and the accompanying rules:

$$\textsc{MKSFree}\ \frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash x \hookrightarrow (\!|x|\!)_{\square} \Rightarrow ?} \qquad \textsc{MSFree}\ \frac{x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\not M} (\!|x|\!)_{\square} \Rightarrow ?}$$

A free variable is marked as such, and since nothing may be said about their types, we synthesize the unknown type. As with the inconsistent type mark, this allows type checking to proceed, with usage of the free variable permitted in any expression.

2.1.4 Lambda Abstractions. Unlike numbers and variables, there are explicit synthesis and analysis rules for unmarked lambda abstractions. This is because expected input and output types are known, and we may verify that the type annotation and body match them.

$$\textsc{USLam}\ \frac{\Gamma, x : \tau \vdash_{\not U} e \Rightarrow \tau_2}{\Gamma \vdash_{\not U} \lambda x : \tau_1.\, e \Rightarrow \tau_1 \to \tau_2} \qquad \textsc{UALam}\ \frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \sim \tau_1 \qquad \Gamma, x : \tau \vdash_{\not U} e \Leftarrow \tau_2}{\Gamma \vdash_{\not U} \lambda x : \tau.\, e \Leftarrow \tau_3}$$

The synthesis rule is standard. Analysis employs the judgment $\tau \blacktriangleright_{\to} \tau_1 \to \tau_2$, which establishes that τ is a *matched arrow type* [Cimini and Siek 2016], i.e. it may be considered an arrow type. Defined in Figure 5, this notion is purely a technical mechanism to avoid duplication of rules related to arrow types [Siek et al. 2015].

$\boxed{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2}$ τ has matched arrow type $\tau_1 \to \tau_2$

$$\textsc{TMAUnknown}\ \frac{}{? \blacktriangleright_{\to} ? \to ?} \qquad \textsc{TMAArr}\ \frac{}{\tau \to \tau \blacktriangleright_{\to} \tau \to \tau}$$

Fig. 5. Matched arrow types.

The synthesis rule for marking intuitively follows USLam closely. We recursively mark the body with an extended context and construct a new marked lambda abstraction:

$$\textsc{MKSLam}\ \frac{\Gamma, x : \tau_1 \vdash e \hookrightarrow \check{e} \Rightarrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\, e \hookrightarrow \lambda x : \tau_1.\, \check{e} \Rightarrow \tau_1 \to \tau_2} \qquad \textsc{MSLam}\ \frac{\Gamma, x : \tau \vdash_{\not M} \check{e} \Rightarrow \tau_2}{\Gamma \vdash_{\not M} \lambda x : \tau_1.\, \check{e} \Rightarrow \tau_1 \to \tau_2}$$

Similarly, we construct corresponding analytic rules:

$$\textsc{MKALam1}\ \frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \sim \tau_1 \qquad \Gamma, x : \tau \vdash e \hookrightarrow \check{e} \Leftarrow \tau_2}{\Gamma \vdash \lambda x : \tau.\, e \hookrightarrow \lambda x : \tau.\, \check{e} \Leftarrow \tau_3} \qquad \textsc{MALam1}\ \frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \sim \tau_1 \qquad \Gamma, x : \tau \vdash_{\not M} \check{e} \Leftarrow \tau_2}{\Gamma \vdash_{\not M} \lambda x : \tau.\, \check{e} \Leftarrow \tau_3}$$

However, recalling totality, we look again to the premises of UALam to see what type errors may arise. First, what if τ_3 is not a matched arrow type? This would be the case if it were num, for example. The lambda abstraction's synthesized type would indeed be inconsistent with the expected type, but this case is slightly distinct from that of the inconsistent types mark: the lambda expression is in analytic position. Instead, we mark $(\!|\check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\not\to}}$ to indicate that an expression of the non-matched arrow type τ_3 was expected, but a lambda abstraction was encountered.

$$\textsc{MKALam2}\ \frac{\tau_3 \blacktriangleright_{\not\to} \qquad \Gamma, x : \tau \vdash e \hookrightarrow \check{e} \Leftarrow ?}{\Gamma \vdash \lambda x : \tau.\, e \hookrightarrow (\!|\lambda x : \tau.\, \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\not\to}} \Leftarrow \tau_3} \qquad \textsc{MALam2}\ \frac{\tau_3 \blacktriangleright_{\not\to} \qquad \Gamma, x : \tau \vdash_{\not M} \check{e} \Leftarrow ?}{\Gamma \vdash_{\not M} (\!|\lambda x : \tau.\, \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\not\to}} \Leftarrow \tau_3}$$

Though no expected output type is known, we still need to check and mark the body; it is thus analyzed against the unknown type in MKALAM2.

Note that in this example language, the distinction from the inconsistent type mark is of reduced significance—all expressions synthesize a type. However, the addition of unannotated lambda abstractions, for example, would necessitate such a distinction.

Another error arises when $\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2$, but $\tau \nsim \tau_1$, i.e. the actual type annotation for x is inconsistent with the expected input type. The *inconsistent ascription mark* $(\!|\lambda x : \tau.\ \check{e}|\!)_{:}$ indicates exactly this error, and we add a final pair of analytic rules:

$$\textsc{MKALAM3}\ \frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \nsim \tau_1 \qquad \Gamma,\ x : \tau_1 \vdash e \hookrightarrow \check{e} \Leftarrow \tau_2}{\Gamma \vdash \lambda x : \tau.\ e \hookrightarrow (\!|\lambda x : \tau.\ \check{e}|\!)_{:} \Leftarrow \tau_3} \qquad \textsc{MALAM3}\ \frac{\tau_3 \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \tau \nsim \tau_1 \qquad \Gamma,\ x : \tau_1 \vdash_{M} \check{e} \Leftarrow \tau_2}{\Gamma \vdash_{M} (\!|\lambda x : \tau.\ \check{e}|\!)_{:} \Leftarrow \tau_3}$$

2.1.5 Applications. In the unmarked language, only a synthesis rule is necessary for applications:

$$\textsc{USAp}\ \frac{\Gamma \vdash_{U} e_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma \vdash_{U} e_2 \Leftarrow \tau_1}{\Gamma \vdash_{U} e_1\ e_2 \Rightarrow \tau_2}$$

Following the same methodology up to this point, we have the following marking and typing rules:

$$\textsc{MKSAp1}\ \frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \hookrightarrow \check{e}_1\ \check{e}_2 \Rightarrow \tau_2} \qquad \textsc{MSAp1}\ \frac{\Gamma \vdash_{M} \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma \vdash_{M} \check{e}_2 \Leftarrow \tau_1}{\Gamma \vdash_{M} \check{e}_1\ \check{e}_2 \Rightarrow \tau_2}$$

Again, to satisfy totality, we must consider the case when τ is not a matched arrow type, such as in the example of Figure 1c. There is no expected type for the argument, so we perform analytic marking on e_2 against the unknown type. In this case, it is not quite right to mark e_1 with the inconsistent type mark; rather than any single type, it is any member of the family of arrow types that is expected. For such a "constrained" synthetic mode, we use a specialized mark $(\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright_{\to}}$, which indicates that $\check{e}$ was expected to be a function—but was not. The output type is unknown, so the entire expression synthesizes the unknown type.

$$\textsc{MKSAp2}\ \frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow ?}{\Gamma \vdash e_1\ e_2 \hookrightarrow (\!|\check{e}_1|\!)^{\Rightarrow}_{\blacktriangleright_{\to}}\ \check{e}_2 \Rightarrow ?} \qquad \textsc{MSAp2}\ \frac{\Gamma \vdash_{M} \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \qquad \Gamma \vdash_{M} \check{e}_2 \Leftarrow ?}{\Gamma \vdash_{M} (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright_{\to}}\ \check{e} \Rightarrow ?}$$

It is natural to extend this approach to other elimination forms that require the handling of unmatched types, such as products. Indeed, the same approach is taken for projections below. Another similar approach might indicate the same kind of error but mark the entire application.

2.1.6 Booleans. The boolean values are similar to numbers:

$$\textsc{USTrue}\ \frac{}{\Gamma \vdash_{U} \mathtt{tt} \Rightarrow \mathtt{bool}} \qquad \textsc{MKSTrue}\ \frac{}{\Gamma \vdash \mathtt{tt} \hookrightarrow \mathtt{tt} \Rightarrow \mathtt{bool}} \qquad \textsc{MSTrue}\ \frac{}{\Gamma \vdash_{M} \mathtt{tt} \Rightarrow \mathtt{bool}}$$

$$\textsc{USFalse}\ \frac{}{\Gamma \vdash_{U} \mathtt{ff} \Rightarrow \mathtt{bool}} \qquad \textsc{MKSFalse}\ \frac{}{\Gamma \vdash \mathtt{ff} \hookrightarrow \mathtt{ff} \Rightarrow \mathtt{bool}} \qquad \textsc{MSFalse}\ \frac{}{\Gamma \vdash_{M} \mathtt{ff} \Rightarrow \mathtt{bool}}$$

Conditionals, however, present a more interesting case. In the unmarked language, we have both explicit synthetic and analytic rules:

USIF
$$\frac{\Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash_{\mathcal{U}} e_2 \Rightarrow \tau_1 \quad \Gamma \vdash_{\mathcal{U}} e_3 \Rightarrow \tau_2}{\Gamma \vdash_{\mathcal{U}} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow \tau_1 \sqcup \tau_2}$$

UAIF
$$\frac{\Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \tau \quad \Gamma \vdash_{\mathcal{U}} e_2 \Leftarrow \tau}{\Gamma \vdash_{\mathcal{U}} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Leftarrow \tau}$$

In synthetic position, conditionals synthesize the *meet* of the branch types τ_1 and τ_2, which we define inductively in Figure 6. We choose the "more specific" type of the two. In analytic position, since there is an expected type for both branches, we shift the blame for any errors to them.

$\boxed{\tau_1 \sqcup \tau_2}$ is a *partial* metafunction defined as follows:

$$\begin{array}{rcl} ? \sqcup \tau & = & \tau \\ \tau \sqcup ? & = & \tau \\ \mathsf{num} \sqcup \mathsf{num} & = & \mathsf{num} \\ \mathsf{bool} \sqcup \mathsf{bool} & = & \mathsf{bool} \\ (\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2') & = & (\tau_1 \sqcup \tau_1') \to (\tau_2 \sqcup \tau_2') \end{array}$$

Fig. 6. Type meet.

Following USIF and UAIF, we may derive the following marking and typing rules.

MKSIF
$$\frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Rightarrow \tau_1 \quad \Gamma \vdash e_3 \hookrightarrow \check{e}_3 \Rightarrow \tau_2}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Rightarrow \tau_1 \sqcup \tau_2}$$

MSIF
$$\frac{\Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Rightarrow \tau_1 \quad \Gamma \vdash_{\mathcal{M}} \check{e}_3 \Rightarrow \tau_2}{\Gamma \vdash_{\mathcal{M}} \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Rightarrow \tau_1 \sqcup \tau_2}$$

MKAIF
$$\frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow \tau \quad \Gamma \vdash e_3 \hookrightarrow \check{e}_3 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Leftarrow \tau}$$

MAIF
$$\frac{\Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \tau \quad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Leftarrow \tau}{\Gamma \vdash_{\mathcal{M}} \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Leftarrow \tau}$$

However, in synthetic position, it may be the case that the two branch types have no meet. This occurs, in fact, when they are inconsistent, motivating the *the inconsistent branches mark* $(\!|\mathsf{if}\ \check{e}\ \mathsf{then}\ \check{e}\ \mathsf{else}\ \check{e}|\!)_{\not\sqcup}$. Then, adding the following rules ensures totality on conditionals:

MKSIncONSISTENTBRANCHES
$$\frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Rightarrow \tau_1 \quad \Gamma \vdash e_3 \hookrightarrow \check{e}_3 \Rightarrow \tau_2 \quad \tau_1 \not\sim \tau_2}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow (\!|\mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3|\!)_{\not\sqcup} \Rightarrow ?}$$

MSIncONSISTENTBRANCHES
$$\frac{\Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Rightarrow \tau_1 \quad \Gamma \vdash_{\mathcal{M}} \check{e}_3 \Rightarrow \tau_2 \quad \tau_1 \not\sim \tau_2}{\Gamma \vdash_{\mathcal{M}} (\!|\mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3|\!)_{\not\sqcup} \Rightarrow ?}$$

As previously mentioned, we do not prescribe any single localization design as "correct", and the framework freely allows for other approaches. For example, as discussed in the introduction, we may choose to regard the first branch as "correct" and localize any errors to the second. The following rule formalizes such a design:

MKSIF'
$$\frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \mathsf{bool} \quad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Rightarrow \tau \quad \Gamma \vdash e_3 \hookrightarrow \check{e}_3 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Rightarrow \tau}$$

2.1.7 Pairs. At this point, the introduction of pairs expressions and the necessary projection operators poses no great challenge. Explicit synthesis and analysis rules govern the former.

$$\textsc{USPair}\ \frac{\Gamma \vdash_{\mathcal{U}} e_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{U}} e_2 \Rightarrow \tau_2}{\Gamma \vdash_{\mathcal{U}} (e_1, e_2) \Rightarrow \tau_1 \times \tau_2} \qquad \textsc{UAPair}\ \frac{\tau \blacktriangleright_{\times} \tau_1 \times \tau_2 \qquad \Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{U}} e_2 \Leftarrow \tau_2}{\Gamma \vdash_{\mathcal{U}} (e_1, e_2) \Leftarrow \tau}$$

In similar fashion to above, we may derive marking rules from these in an intuitive manner:

$$\textsc{MKSPair}\ \frac{\Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Rightarrow \tau_1 \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Rightarrow \tau_2}{\Gamma \vdash (e_1, e_2) \hookrightarrow (\check{e}_1, \check{e}_2) \Rightarrow \tau_1 \times \tau_2} \qquad \textsc{MSPair}\ \frac{\Gamma \vdash_{\mathcal{M}} \check{e}_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Rightarrow \tau_2}{\Gamma \vdash_{\mathcal{M}} (\check{e}_1, \check{e}_2) \Rightarrow \tau_1 \times \tau_2}$$

$$\textsc{MKAPair1}\ \frac{\tau \blacktriangleright_{\times} \tau_1 \times \tau_2 \qquad \Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow \tau_2}{\Gamma \vdash (e_1, e_2) \hookrightarrow (\check{e}_1, \check{e}_2) \Leftarrow \tau} \qquad \textsc{MAPair1}\ \frac{\tau \blacktriangleright_{\times} \tau_1 \times \tau_2 \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Leftarrow \tau_2}{\Gamma \vdash_{\mathcal{M}} (\check{e}_1, \check{e}_2) \Leftarrow \tau}$$

However, as when analyzing a lambda abstraction against a non-matched arrow type, the type against which a pair is analyzed may not match any product type. We solve this by adding an error mark $\langle\!|(\check{e}_1, \check{e}_2)|\!\rangle^{\Leftarrow}_{\blacktriangleright_{\not\times}}$ and the corresponding rules:

$$\textsc{MKAPair2}\ \frac{\tau \blacktriangleright_{\not\times} \qquad \Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow ? \qquad \Gamma \vdash e_2 \hookrightarrow \check{e}_2 \Leftarrow ?}{\Gamma \vdash (e_1, e_2) \hookrightarrow \langle\!|(\check{e}_1, \check{e}_2)|\!\rangle^{\Leftarrow}_{\blacktriangleright_{\not\times}} \Leftarrow \tau} \qquad \textsc{MAPair2}\ \frac{\tau \blacktriangleright_{\not\times} \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_1 \Leftarrow ? \qquad \Gamma \vdash_{\mathcal{M}} \check{e}_2 \Leftarrow ?}{\Gamma \vdash_{\mathcal{M}} \langle\!|(\check{e}_1, \check{e}_2)|\!\rangle^{\Leftarrow}_{\blacktriangleright_{\not\times}} \Leftarrow \tau}$$

As the elimination form for products, projections are handled in a similar manner as applications. In the interest of space, we give only the rules for the left projection operator; right projections are governed analogously.

$$\textsc{USProjL}\ \frac{\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau \qquad \tau \blacktriangleright_{\times} \tau_1 \times \tau_2}{\Gamma \vdash_{\mathcal{U}} \pi_1 e \Rightarrow \tau_1} \qquad \textsc{MKSProjL1}\ \frac{\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_{\times} \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 e \hookrightarrow \pi_1 \check{e} \Rightarrow \tau_1} \qquad \textsc{MSProjL1}\ \frac{\Gamma \vdash_{\mathcal{M}} \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_{\times} \tau_1 \times \tau_2}{\Gamma \vdash_{\mathcal{M}} \pi_1 \check{e} \Rightarrow \tau_1}$$

In the case that the subject of the projection does not synthesize a matched product type, we mark with an error, written $\pi_1 \langle\!|\check{e}|\!\rangle^{\Rightarrow}_{\blacktriangleright_{\not\times}}$:

$$\textsc{MKSProjL2}\ \frac{\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_{\not\times}}{\Gamma \vdash \pi_1 e \hookrightarrow \pi_1 \langle\!|\check{e}|\!\rangle^{\Rightarrow}_{\blacktriangleright_{\not\times}} \Rightarrow \tau_1} \qquad \textsc{MSProjL2}\ \frac{\Gamma \vdash_{\mathcal{M}} \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_{\not\times}}{\Gamma \vdash_{\mathcal{M}} \pi_1 \langle\!|\check{e}|\!\rangle^{\Rightarrow}_{\blacktriangleright_{\not\times}} \Rightarrow ?}$$

2.1.8 Holes. For completeness, we finish the development of the marked lambda calculus on the extended GTLC with the marking of empty holes, which are never marked (see Sec. 4 for marking unfillable holes using constraint solving).

$$\textsc{USHole}\ \frac{}{\Gamma \vdash_{\mathcal{U}} (\!|\,|\!) \Rightarrow ?} \qquad \textsc{MKSHole}\ \frac{}{\Gamma \vdash (\!|\,|\!) \hookrightarrow (\!|\,|\!) \Rightarrow ?} \qquad \textsc{MSHole}\ \frac{}{\Gamma \vdash_{\mathcal{M}} (\!|\,|\!) \Rightarrow ?}$$

2.1.9 Additional Metatheorems. To conclude this section, we present two more metatheorems that help ensure correctness of the system. Though Theorem 2.2 guarantees that marking does not change the syntactic structure of a program, it makes no statement about the presence of error marks in the resulting marked program. Theorem 2.3 established that well-typed expressions are left unmarked and ill-typed expressions have at least one mark.

Theorem 2.3 (Marking of Well-Typed/Ill-Typed Expressions).

(1) (a) *If* $\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau$ *and* $\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \tau$, *then* $\check{e}$ markless.
(b) *If* $\Gamma \vdash_{\mathcal{U}} e \Leftarrow \tau$ *and* $\Gamma \vdash e \hookrightarrow \check{e} \Leftarrow \tau$, *then* $\check{e}$ markless.

(2) (a) *If there does not exist* τ *such that* $\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau$, *then for all* $\check{e}$ *and* τ' *such that* $\Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \tau'$, *it is not the case that* $\check{e}$ markless.
(b) *If there does not exist* τ *such that* $\Gamma \vdash_{\mathcal{U}} e \Leftarrow \tau$, *then for all* $\check{e}$ *and* τ' *such that* $\Gamma \vdash e \hookrightarrow \check{e} \Leftarrow \tau'$, *it is not the case that* $\check{e}$ markless.

Finally, no less importantly, marking is deterministic. This is given by Theorem 2.4.

Theorem 2.4 (Marking Unicity).

(1) *If* $\Gamma \vdash e \hookrightarrow \check{e}_1 \Rightarrow \tau_1$ *and* $\Gamma \vdash e \hookrightarrow \check{e}_2 \Rightarrow \tau_2$, *then* $\check{e}_1 = \check{e}_2$ *and* $\tau_1 = \tau_2$.
(2) *If* $\Gamma \vdash e \hookrightarrow \check{e}_1 \Leftarrow \tau$ *and* $\Gamma \vdash e \hookrightarrow \check{e}_2 \Leftarrow \tau$, *then* $\check{e}_1 = \check{e}_2$.

Together, totality and unicity give that marking may be implemented as a total function. Indeed, given the algorithmic nature of bidirectional typing, it is fairly direct to implement these rules.

2.2 Agda Mechanization

The semantics and metatheory presented above have been fully mechanized in the Agda proof assistant. The mechanization additionally includes the decoupled Hazelnut action semantics described in Section 3.2. Though the mechanization's documentation contains more detailed discussion regarding technical decisions made therein, we highlight some important aspects here.

The standard approach of modeling judgments as inductive datatypes and rules as constructors for those datatypes is taken. By representing marked expressions with implicitly typed terms, we get part of Theorem 2.2 for free from the definition of marking. For the convenience of readers interested in browsing the mechanization, as possible, rule names match those presented here.

2.3 Destructuring Let with Type Annotated Composite Patterns

To ease the use of products and other datatypes, many languages feature destructuring bindings. In a typed setting, we may want to granularly add type annotations in patterns as well. In a bidirectional setting, as it turns out, this combination of features is surprisingly tricky to get right! Let us add let expressions and simple patterns, writing _ for the wildcard pattern, x for a variable pattern, and (p_1, p_2) for a pair:

$$\begin{array}{rrcl} \text{UExp} & e & ::= & \cdots \mid \mathsf{let}\ p = e\ \mathsf{in}\ e \\ \text{UPat} & p & ::= & \_ \mid x \mid (p, p) \mid p : \tau \end{array}$$

Consider the following program:

$$\mathsf{let}\ (a, b) = (1, 2)\ \mathsf{in}\ e$$

To type this, the most obvious approach is to synthesize a type from the pattern and analyze the definition against that type. However, this may run afoul of user expectation, which might reasonably suppose it to be equivalent to the expanded expression $\mathsf{let}\ a = 1\ \mathsf{in}\ \mathsf{let}\ b = 2\ \mathsf{in}\ e$. In the original, the pattern synthesizes the type $? \times ?$, and $(1, 2)$ is analyzed against it, hence 1 and 2 are each analyzed against ?. In the expanded version, however, they are typed in synthetic mode.

Though it is benign in this example, there is a subtle semantic distinction: synthetic mode imposes *no* type constraints, whereas analytic mode imposes a *trivial* type constraint. This manifests when expressions may have internal type inconsistencies, as in the following:

$$\mathsf{let}\ a = \mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ 1\ \mathsf{else}\ \mathtt{ff}\ \mathsf{in}\ e$$

If the pattern a suggests that the conditional is in synthetic position, it will be marked with an inconsistent branches error mark following our development above. If instead it is in analytic position against the unknown type, no mark will be produced.

To remedy this situation and preserve the semantic distinction between synthesis and analysis against the unknown type, we introduce a pattern annotation form, written $p : \tau$, which allows the explicit imposition of typing constraints. That is, the absence of any type annotation on a variable pattern places the corresponding definition in synthetic mode, while the programmer may impose typing constraints—the trivial constraint, if they wish—on the definition.

As illustration, consider the following program:

$$\mathsf{let}\ (a, b : ?) = (\mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ 1\ \mathsf{else}\ \mathtt{ff}, \mathsf{if}\ \mathtt{tt}\ \mathsf{then}\ 2\ \mathsf{else}\ \mathtt{ff})\ \mathsf{in}\ e$$

Since a has no constraint, the left component is in synthetic position, leading to an inconsistent branches error. The type annotation on b puts the right component in analytic mode against the unknown type, leading to no error marks.

This system is achieved by augmenting the type system with a variant of the unknown type, written $?^{\Rightarrow}$, that triggers a "switch" to synthesis and otherwise behaves identically to ?. In the previous example, the unannotated pattern variable a synthesizes $?^{\Rightarrow}$, and the annotated pattern $b : ?$ synthesizes ?. We write $\Gamma \vdash_{\mathcal{U}} p \Rightarrow \tau$ to say that the pattern p synthesizes type τ.

$$\textsc{UASynSwitch}\ \frac{\Gamma \vdash_{\mathcal{U}} e \Rightarrow \tau}{\Gamma \vdash_{\mathcal{U}} e \Leftarrow ?^{\Rightarrow}} \qquad \textsc{USPVar}\ \frac{}{\Gamma \vdash_{\mathcal{U}} x \Rightarrow ?^{\Rightarrow}} \qquad \textsc{USPAnn}\ \frac{\Gamma \vdash_{\mathcal{U}} p \Leftarrow \tau \dashv \Gamma'}{\Gamma \vdash_{\mathcal{U}} p : \tau \Rightarrow \tau}$$

Given by the judgment $\Gamma \vdash_{\mathcal{U}} p \Leftarrow \tau \dashv \Gamma'$, patterns are also typed analytically, in which case they produce an output context Γ', which extends Γ with bindings introduced by the pattern. Note that $?^{\Rightarrow}$ exists entirely to ensure that sub-expressions of the definition are assigned an appropriate typing mode—it is never added to the context, i.e. it cannot escape into the rest of the program and cause body expressions to synthesize accidentally.

$$\textsc{UAPVar}\ \frac{}{\Gamma \vdash_{\mathcal{U}} x \Leftarrow \tau \dashv \Gamma, x : \tau} \qquad \textsc{UAPAnn}\ \frac{\Gamma \vdash_{\mathcal{U}} p \Leftarrow \tau' \dashv \Gamma' \qquad \tau \sim \tau'}{\Gamma \vdash_{\mathcal{U}} p : \tau' \Leftarrow \tau \dashv \Gamma'}$$

To ensure propagation of type information between pattern, definition, and body, the destructuring let requires some extra machinery.

$$\textsc{USLetPat}\ \frac{\Gamma \vdash_{\mathcal{U}} p \Rightarrow \tau_p \qquad \Gamma \vdash_{\mathcal{U}} e_1 \Leftarrow \tau_p \qquad \Gamma \vdash_{\mathcal{U}} e_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{U}} p \Leftarrow \tau_1 \dashv \Gamma' \qquad \Gamma' \vdash_{\mathcal{U}} e_2 \Rightarrow \tau_2}{\Gamma \vdash_{\mathcal{U}} \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \Rightarrow \tau_2}$$

First, to ensure that the pattern and definition types are consistent, the pattern synthesizes a type, against which the definition is analyzed. Then, the definition synthesizes a type, against which the pattern is analyzed. This analysis is guaranteed to succeed; we need the context it produces

for use in the synthesis of the body's type. The analytic rule is similar, with the final premise and conclusion changed to analysis.

The marking rule is directly analogous and utilizes the marking of patterns, which we omit—they may be derived in the same way as those for expressions and are governed by similar metatheorems (again, totality guides the derivation). In the analytic cases, we introduce error marks that parallel to $(\!|\check{e}|\!)_{\star}$ and $(\!|\check{e}|\!)^{\Leftarrow}_{\blacktriangleright\not\star}$.

$$\textsc{ISLetPat}\quad \frac{\Gamma \vdash p \hookrightarrow \check{p} \Rightarrow \tau_p \qquad \Gamma \vdash e_1 \hookrightarrow \check{e}_1 \Leftarrow \tau_p \qquad \Gamma \vdash_{\mathcal{U}} e_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathcal{U}} p \Leftarrow \tau_1 \dashv \Gamma' \qquad \Gamma' \vdash e_2 \hookrightarrow \check{e}_2 \Rightarrow \tau_2}{\Gamma \vdash \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \hookrightarrow \mathsf{let}\ \check{p} = \check{e}_1\ \mathsf{in}\ \check{e}_2 \Rightarrow \tau_2}$$

2.4 Parametric Polymorphism and Richer Judgmental Structures

To further demonstrate that the judgmental structure of the marked lambda calculus may be applied to richer typing features, we now explore an extension of the core language developed above to System F-style parametric polymorphism. Toward this end, we supplement the language in Figure 7 with type abstractions $\Lambda\alpha.\ e$ and applications $e\ [\tau]$. These operate on forall types $\forall\alpha.\ \tau$ and type variables α, which are governed by a well-formedness judgment, written $\Sigma \vdash_{\mathcal{U}} \tau$, in the standard way.

$$\begin{array}{rrcl}
\text{Type} & \tau & ::= & \cdots \mid \forall\alpha.\ \tau \mid \alpha \\
\text{MType} & \check{\tau} & ::= & \cdots \mid \forall\alpha.\ \check{\tau} \mid \alpha \mid (\!|\alpha|\!)_{\square} \\
\text{UExp} & e & ::= & \cdots \mid \Lambda\alpha.\ e \mid e\ [\tau] \\
\text{MExp} & \check{e} & ::= & \cdots \mid \Lambda\alpha.\ \check{e} \mid \check{e}\ [\check{\tau}] \\
 & & \mid & (\!|\Lambda\alpha.\ \check{e}|\!)^{\Leftarrow}_{\blacktriangleright\forall} \mid (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright\forall}\ [\check{\tau}]
\end{array}$$

Fig. 7. Extension of the marked lambda calculus for parametric polymorphism

Crucially, just as expression variables may be free, arbitrary programs may contain free type variables; similar to the introduction of both unmarked and marked patterns in the previous section, we now need separate notions of unmarked and marked types. The latter we denote by the metavariable $\check{\tau}$.

A marking judgment, written $\Sigma \vdash \tau \hookrightarrow \check{\tau}$, then, relates the two sorts. It is parameterized over a context of type variables Σ, and most of the rules recurse straightforwardly. When a free type variable α is encountered, however, we mark it as such, writing $(\!|\alpha|\!)_{\square}$. This gives the following marking and well-formedness rules:

$$\textsc{MKTFree}\quad \frac{\alpha \notin \Sigma}{\Sigma \vdash \alpha \hookrightarrow (\!|\alpha|\!)_{\square}} \qquad \textsc{MTWFFree}\quad \frac{\alpha \notin \Sigma}{\Sigma \vdash_{\mathcal{M}} (\!|\alpha|\!)_{\square}}$$

Just as expression error marks synthesize the unknown type to allow type-checking to continue, these marked types operate identically to the unknown type with respect to consistency and other auxiliary notions. Furthermore, we define analogous notions of mark erasure on types, and, as with pattern marking above, metatheoretic statements ensure their correctness.

The development of marking for type abstractions is similar to that for ordinary lambda abstractions. The synthetic case is straightforward; in analytic position, if the type being analyzed against

is not a matched forall type, an error is marked:

$$\textsc{MKATypeLam2}\ \frac{\check{\tau} \blacktriangleright_{\forall} \qquad \Sigma, \alpha; \Gamma \vdash e \hookrightarrow \check{e} \Leftarrow ?}{\Sigma; \Gamma \vdash \Lambda\alpha.\, e \hookrightarrow (\!|\Lambda\alpha.\, \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\forall}} \Leftarrow \check{\tau}} \qquad \textsc{MATypeLam2}\ \frac{\check{\tau} \blacktriangleright_{\forall} \qquad \Sigma, \alpha; \Gamma \vdash_{\mathbb{M}} \check{e} \Leftarrow ?}{\Sigma; \Gamma \vdash_{\mathbb{M}} (\!|\Lambda\alpha.\, \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\forall}} \Leftarrow \check{\tau}}$$

Likewise, type application mirrors ordinary application: in the case that the expression being applied does not synthesize a matched forall type, an error is marked. With this, totality of marking is satisfied.

$$\textsc{MKSTypeAp2}\ \frac{\Sigma; \Gamma \vdash e \hookrightarrow \check{e} \Rightarrow \check{\tau} \qquad \Sigma \vdash \tau_2 \hookrightarrow \check{\tau}_2 \qquad \check{\tau} \blacktriangleright_{\forall}}{\Sigma; \Gamma \vdash e\ [\tau_2] \hookrightarrow \check{e}\ [\check{\tau}_2] \Rightarrow ?} \qquad \textsc{MSTypeAp2}\ \frac{\Sigma; \Gamma \vdash_{\mathbb{M}} \check{e} \Rightarrow \check{\tau} \qquad \Sigma \vdash_{\mathbb{M}} \check{\tau}_2 \qquad \check{\tau} \blacktriangleright_{\forall}}{\Sigma; \Gamma \vdash_{\mathbb{M}} (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright_{\forall}}\ [\check{\tau}_2] \Rightarrow ?}$$

The case studies above demonstrate that the framework of the marked lambda calculus is suitable to support a variety of extensions to richer judgmental structures. We hope that this spurs language designers to design marked variants of other calculi, following the recipe we have demonstrated: starting with a gradual, bidirectional type system, by considering each possible failure case, one may systematically derive the necessary error marks and marking rules. The metatheorems, particularly totality, ensure that no rules or premises have been missed.

We leave as future work the task of defining marked versions of even more elaborate bidirectional type systems, e.g. Dunfield and Krishnaswami [2013]'s rather substantial formulation of implicit type application in a bidirectional setting (which would have to handle, for example, the situation where no implicit argument can be resolved) or Lennon-Bertrand [2022]'s gradual bidirectional variant of the dependently typed calculus of constructions.

3 INTEGRATING MARKING WITH EDITING

3.1 Hazel Implementation

We have implemented a marking system based on the marked lambda calculus as the foundation for a new version of the Hazel. In particular, we have implemented marking for all of the features in the previous section as well as Hazel's n-tuples, lists, algebraic datatypes, general pattern matching, strings, and explicit polymorphism (all following the same recipe we developed above). Hazel is implemented in OCaml and compiles to Javascript via `js_of_ocaml` [Vouillon and Balat 2014] for use in the browser. The supplemental material contains a pre-built implementation.

The result of this effort is the first full-scale typed functional language equipped with a language server that solves the semantic gap problem: all of Hazel's semantic services are available as long as the program sketch is syntactically well-formed, which include type hints [Potter and Omar 2020], semantic navigation, semantic code completion [Blinn et al. 2022; Potter and Omar 2020], contextualized documentation [Potter et al. 2022], and, because Hazel is able to evaluate programs with empty and non-empty holes due to prior work [Omar et al. 2019], even semantic services that require run-time information, like testing.

Hazel offers both a textual syntax, with explicit empty holes, and more interestingly, a structure editor that is able to additionally offer partial syntactic error recovery. In particular, the `tylr` editor uses a form of structured syntax error recovery that automatically insert empty holes to maintain syntactic well-formedness as long as matching delimiters are placed by the user [Moon et al. 2022, 2023]. Efforts outside of the scope of this paper are underway to define syntactic error recovery mechanisms that can handle unmatched delimiters as well, which would achieve total syntax error recovery. In combination with our contributions, this would achieve the Hazel project's stated goal of ensuring that *there are no meaningless editor states.*

3.2 Fixing Holes in Hazelnut

Earlier versions of Hazel achieved total syntax error recovery by using a term-based structure editor, which maintained delimiter matching by construction, albeit at the cost of some syntactic rigidity. The Hazelnut action calculus introduced by Omar et al. [2017a] specifies such a structure editor formally. Notably, this calculus also supports a form of type error localization and recovery: edit actions can automatically insert non-empty holes around type inconsistencies that appear at the cursor after an edit action. This requires specifying a bidirectionally typed action calculus that propagates type and binding information to the cursor, given by the synthetic action judgment, written $\Gamma \vdash e \Rightarrow \tau \xrightarrow{\alpha} e' \Rightarrow \tau'$, and the analytic action judgment, written $\Gamma \vdash e \xrightarrow{\alpha} e' \Leftarrow \tau$.

A significant issue with Hazelnut is that it does not allow (i.e. it leaves undefined) actions that require *non-local* mark (i.e. non-empty hole, in Hazelnut's parlance) insertion or removal. For example, though the system can wrap an arbitrary expression in an arithmetic operation, it cannot wrap an arbitrary expression e in a lambda abstraction, i.e. construct $\lambda x : ?\,.\, e$ directly from e. The binding of x with the unknown type may shadow a previous declaration and require the removal of error holes in body. The calculus includes manual non-empty hole insertion and removal actions as a workaround.

The fundamental issue is that Hazelnut did not have a total marking system that it could deploy to remark expressions where needed. The marked lambda calculus allows us to solve this problem, in two different ways.

One approach is to define an untyped action calculus, *wholly decoupling* edit actions from typing. Under this simplified design, the action calculus, given by the singular action judgment, written $e \xrightarrow{\alpha} e'$, is concerned only with the manipulation of syntax, and the total marking procedure yields statically meaningful terms with error marks inserted at the appropriate positions as a kind of editor decoration. This solves the problem outlined above; since marking operates on the entire program after each action, arbitrary constructions are permitted.

Alternatively, instead of a wholly untyped action calculus, one may directly integrate re-marking logic into the typed Hazelnut calculus, making use of the type and scoping information being propagated to re-mark only in scopes where it might be necessary. For example, we can wrap $\check{e}$ into $\lambda x : ?\,.\, \check{e}$ by re-marking the body under the context with x binding the unknown type:

$$\textsc{ASEConLam}\ \frac{\Gamma,\ x : ? \vdash \check{e}^{\square} \rightsquigarrow \check{e}' \Rightarrow \tau'}{\Gamma \vdash \rhd \check{e} \lhd \Rightarrow \tau \xrightarrow{\text{construct lam } x} \lambda x : \rhd ? \lhd .\ \check{e}' \Rightarrow ? \to \tau'}$$

The supplemental material contains a complete formal description of both variants, and the untyped action semantics and related metatheory are mechanized in Agda. Because of the simplicity of the untyped version, and because marking is defined separately, this Agda mechanization can serve universally as a baseline for correctness of approaches, like our initial design that aim to make use of type and scope information, and perhaps other analyses, to minimize re-marking. We leave a complete assessment of this and other re-marking optimization approaches to future work, particularly because Hazel is now moving toward the decoupled approach for future development.

4 TYPE HOLE INFERENCE

Local type inference, which has been our focus so far, reduces the number of necessary annotations and induces a local flow of information particularly well-suited to systematic error localization decisions [Dunfield and Krishnaswami 2019; Pierce and Turner 2000b]. In contrast, constraint-based type inference as found in many ML-family languages allows programmers to omit most or all type annotations [Pierce and Turner 2000b]. The trade-off is that type error localization and recovery

become considerably more difficult, because inconsistencies can now arise between constraints derived from many distant locations in a program. Heuristic approaches, e.g. based on manually tuned weights [Pavlinovic et al. 2014; Zhang and Myers 2014] or machine learning methods [Seidel et al. 2017], are necessary to guess which uses of a variable are consistent with the user's intent and which should be marked erroneous.

This situation is reminiscent of the situation considered in Section 2.1.6 of inconsistent branches in a conditional, where biasing one branch over the other requires guessing user intent, whereas the neutral approach is to localize the inconsistency to the conditional expression as a whole. However, there is no such parent expression in the case of inconsistent constraints gathered globally. However, this search for neutrality when there are downstream conflicts motivates the approach we introduce in this section, which *gradually* and *neutrally* harmonizes local and constraint-based type inference. In particular, our *type hole inference* approach generates constraints on unknown types that arise during bidirectional marking process as described in Section 2. After this initial marking is complete, we unify these constraints (using a standard unification algorithm, which we do not detail here [Huet 1976]). When it is discovered that an unknown type is subject to inconsistent constraints, we localize the problem to a hole in the program connected to that unknown type – either to a *type hole* directly, or to an empty or non-empty expression hole from which that unknown type traces its provenance. The key result is that this approach is neutral by construction: it does not attempt to guess at which relevant locations were consistent with the user intent. In Hazel, the user can instead interactively investigate various partially consistent suggestions to clarify their intent, which returns control to the local type inference system.

4.1 Type Hole Inference in Hazel

Unlike in ML-family languages where constraint solving is an obligate part of typing, in Hazel constraint solving is *optional* (i.e. it can be turned off) and is only invoked for solving for unknown types. For example, in the following Hazel program, no constraint solving is necessary: local inference determines that `f : Int` and the error is localized to the application of `f` as a function, as described in Section 2. This coincidentally is also how the corresponding OCaml program would localize the error, but for different reasons: OCaml would detect inconsistent constraints and favor the constraints arising lexically first [McAdam 1998; Odersky et al. 1999; Pottier 2014].

Fig. 8. Error localization of this program in Hazel and OCaml is similar, but for different reasons.

The systems diverge in their capabilities if we explicitly add a type hole annotation to `f`, the bidirectional system treats this as an unknown type and operates gradually, taking this as a signal that the user has not yet determined the intended type of `f`. To help the user fill this type hole, the system generates and attempts to unify the relevant constraints (see Section 4.3 below). In this case, the constraints are inconsistent, i.e. there is no hole filling that satisfies all relevant constraints. Rather than favoring constraints arising first, as in OCaml, the type hole is highlighted in red and marked with a bang (!). The user is informed via the Type Inspector that the type hole cannot be solved due to conflicting information from constraints. Figure 9 shows how the editor temporarily fills the type hole when the user hovers over a constraint, deferring to bidirectional error localization as described in Section 2 from there to cause the error to be marked on the `2` when hovering over the arrow type.

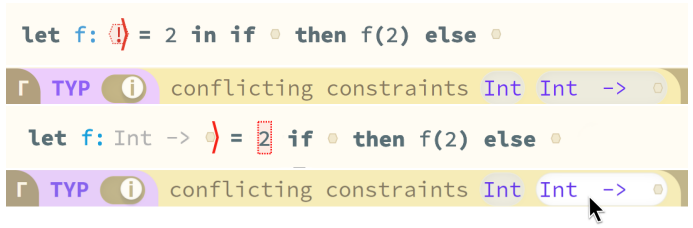


Fig. 9. The error is localized to the conflicted type hole in Hazel. When hovering over the indicated suggestion, the hole is transiently filled, causing the error to be localized to the bound expression, 2.

If, given this feedback, the user deletes the bound expression, 2, then there are no longer conflicting constraints on the type hole and the system displays the inferred type in gray, to indicate that it has been inferred rather than entered as "ground truth" by the user. The user can press the `Enter` key to accept the suggestion (turning the text black). At this point the hole has been filled and localization decisions again become local. Note that there are no constraints on the return type of `f`, so the suggested filling itself contains a type hole (see Section 4.4 below on polymorphic generalization).

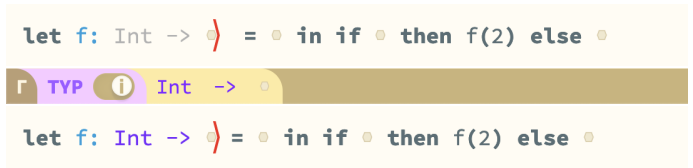


Fig. 10. User removes '2' and accepts the new suggestion

It is also worth considering the situation where no type annotation is present, but the bound expression is an empty expression hole, because this is also a situation where the type of `f` is unknown according to the bidirectional system from Section 2. Type hole inference solves for this unknown type as well, while tracking its provenance as the type of this expression hole. As such, errors due to conflicting constraints can be localized to expression holes in much the same way as described above. When the user hovers over a suggested type, there must be some way of constraining the type of the expression hole, e.g. by finding a suitable variable to annotate with a type or by adding direct ascription syntax to the language (we have not implemented this particular user interface affordance in Hazel as of yet, but there are no fundamental barriers to doing so.) The lightly mocked up process is illustrated in Figure 11.

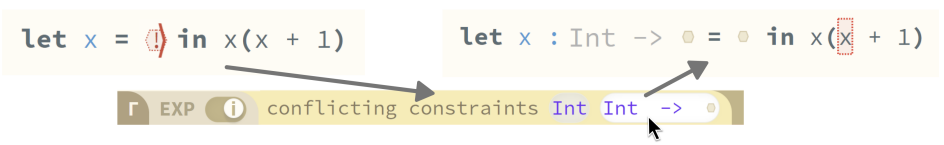


Fig. 11. Localization of errors to expression holes.

4.2 Constraint Generation and Unknown Type Provenance

In order to generate global inference results, we begin with constraint generation. Our approach closely follows the usual approach (cf. [Pierce 2002b]), where we augment the bidirectional type system for the marked lambda calculus with generated constraint sets, C. The new judgement forms are $\Gamma \vdash \check{e} \Rightarrow \tau \mid C$ and $\Gamma \vdash \check{e} \Leftarrow \tau \mid C$.

Constraints, written $\tau_1 \approx \tau_2$, force two incomplete types to be consistent. Consequently, our rules for constraint generation are quite simple. We augment our previous bidirectional typing rules in the marked lambda calculus to accumulate constraints describing necessary type consistencies. For example, when synthesizing the type of an if expression, we constrain the types of expressions in either branch to each other:

MSIf-C

$$\frac{\Gamma \vdash \check{e}_1 \Leftarrow \mathsf{bool} \mid C_1 \qquad \Gamma \vdash \check{e}_2 \Rightarrow \tau_1 \mid C_2 \qquad \Gamma \vdash \check{e}_3 \Rightarrow \tau_2 \mid C_3}{\Gamma \vdash \mathsf{if}\ \check{e}_1\ \mathsf{then}\ \check{e}_2\ \mathsf{else}\ \check{e}_3 \Rightarrow \tau_1 \sqcup \tau_2 \mid C_1 \cup C_2 \cup C_3 \cup \{\tau_1 \approx \tau_2\}}$$

The remaining rules for standard constructs are similarly standard and given in the supplement.

Since unification will be solving for unknown types, we must ensure we are able to distinguish between different type holes based on their associated locus in the program. To this end, we make two modifications to the system from Section 2:

(1) We add a unique id u to all expression holes and type holes that appear directly in the program where we assume that id generation is handled by the editor.
(2) We add provenances p to unknown types. Each provenance links the unknown type to to some (syntactic) hole in the program, perhaps through some intermediate operations. For example, the provenance $exp(3)$ indicates that a type hole must have been synthesized from an expression hole with id 3 in the program.

$$\begin{array}{rrcl} \text{Provenance} & p & ::= & u \mid exp(u) \mid \rightarrow_L(p) \mid \rightarrow_R(p) \mid \times_L(p) \mid \times_R(p) \\ \text{Type} & \tau & ::= & \cdots \mid ?^p \\ \text{MExp} & \check{e} & ::= & \cdots \mid \langle x \rangle^u_{\square} \mid \langle \check{e} \rangle^u_{:} \mid \cdots \mid \langle \check{e} \rangle^{\Rightarrow,\,u}_{\blacktriangleright \not\rightarrow} \mid \langle \check{e} \rangle^{\Rightarrow,\,u}_{\blacktriangleright \not\times} \mid \langle\!|\rangle^u \end{array}$$

Provenance is determined whenever a new unknown type is constructed. For example, we update our matched arrow and product rules to add provenances to outputted type holes and introduce the new matched arrow and product provenances so that every time a matched arrow type is generated for the same hole, it involves the same constituent unknown types without needing to introduce explicit constraints:

TMAHole-C

$$\frac{}{?^p \blacktriangleright_{\rightarrow} ?^{\rightarrow_L(p)} \rightarrow ?^{\rightarrow_R(p)} \mid \{?^p \approx ?^{\rightarrow_L(p)} \rightarrow ?^{\rightarrow_R(p)}\}}$$

TMPHole-C

$$\frac{}{?^p \blacktriangleright_{\times} ? \times ? \mid \{?^p \approx ?^{\times_L(p)} \times ?^{\times_R(p)}\}}$$

Up until now, we have ignored rules that consider marked errors. This leads us to our second point of interest: what do we do with expressions that have been marked with expression holes? A marked expression has already been deemed erroneous. Therefore, when generating constraints, we do not constrain sub-expressions within a marked hole based on types flowing in from outside the hole. As a simple example of this, we present the standard rule for successful subsumption alongside the rule for subsumption upon the failure of the consistency check:

MASubsume-C

$$\frac{\Gamma \vdash \check{e} \Rightarrow \tau' \mid C \qquad \tau \sim \tau' \qquad \check{e}\ \text{subsumable}}{\Gamma \vdash \check{e} \Leftarrow \tau \mid C \cup \{\tau \approx \tau'\}}$$

MAInconsistentTypes-C

$$\frac{\Gamma \vdash \check{e} \Rightarrow \tau' \mid C \qquad \tau \nsim \tau' \qquad \check{e}\ \text{subsumable}}{\Gamma \vdash \langle \check{e} \rangle^u_{:} \Leftarrow \tau \mid C \cup \{\tau \approx ?^{exp(u)}\}}$$

The remaining rules directly follow the intuitions above and are left to the supplemental material.

4.3 Unification and Potential Type Sets

To unify constraints, we use a standard union-find based unification algorithm [Huet 1976; Siek and Vachharajani 2008], which accumulates constraint information in PotentialTypeSets.

A PotentialTypeSet is a recursive data structure representing the potential solutions for an unknown type, inferred from type constraints. A single PotentialTypeSet describes *all* of the potential fillings for its associated type hole. To facilitate this, rather than substituting types during unification, which results in a loss of information, we continuously extend the PotentialTypeSet.

$$\begin{array}{rrcl} \text{PotentialTypeSet} & s & ::= & \mathsf{single}(t) \mid \mathsf{cons}(t, s) \\ \text{PotentialType} & t & ::= & \mathtt{num} \mid \mathtt{bool} \mid ?^p \mid s \to s \end{array}$$

This choice allows us to continue past failures and unify any constraints, yielding an accumulated corpus of information on each type hole's potential solutions and errors through their associated PotentialTypeSets. Each PotentialTypeSet can be scanned by the editor to identify potential solutions or localize errors as illustrated in Figure 9 and Figure 10.

Note again that the novel contribution of this section is not in the particular unification algorithm but rather the architectural decisions that allow us to neatly blend local and constraint-based type inference systems for orthogonal purposes within the same system, and our focus on how to handle inconsistent constraints. As such, we direct the reader to prior work for the algorithmic, formal, and metatheoretic details of unification [Siek and Vachharajani 2008].

4.4 Polymorphic Generalization with Holes

In many type inference systems, unconstrained type inference variables (here, unknown types) are automatically polymorphically generalized, so that functions can be given the most general type. The same approach can be taken with type hole inference, but with one particularly interesting wrinkle. With the inclusion of expression holes in our system, we need to be a bit more careful in when we generalize. Consider the following expression:

$$\lambda x : ?^1 \,.\, (\!|\,|\!)^2$$

We can see that $?^1$ is not constrained. Suppose that we were to suggest the implicitly universally quantified type variable $'a$ as a type hole filling for $?^1$. It is unlikely that the user accepts this suggestion because it is unlikely that the user intends to write the identity function! The fact there are not yet any constraints does not imply that there will not be once the expression hole is filled.

To address this, we need to reason as if there are any number of *unknown constraints* coming from expression holes. This can be represented with a new type of constraint: the `etc` constraint, which we add to our syntax of PotentialTypes below.

$$\text{PotentialType} \quad t \quad ::= \quad \cdots \mid \mathtt{etc}$$

When an expression hole appears, all unknown types in the typing context are constrained to `etc`. This has no impact on unification, but if a type hole $?^p$ is constrained to `etc`, it cannot be generalized.

5 RELATED WORK

The contributions of this paper build directly on the Hazelnut type system [Omar et al. 2017a], which is discussed extensively throughout. Non-empty holes in Hazelnut generalize to marks in this work. In brief, we contribute a total marking procedure (Section 2) and type hole inference scheme (Section 4) for a system based closely on Hazelnut, and use it to fix some expressiveness issues in Hazelnut's edit action calculus (Section 3).

Hazelnut is in turn rooted in gradual type theory [Siek and Taha 2006; Siek et al. 2015], and we make extensive use of (only) the static aspects of gradual typing, namely the universal consistency of the unknown type, to enable recovery from marked errors, which can leave missing type information.

Our focus was exclusively on static typing in this paper, and the results are relevant to the design of language servers for any statically typed language, but it is worth noting that the results in this paper, taken together with Hazel's support for maintaining syntactic well-formedness using structure editing [Moon et al. 2022, 2023] and for running programs with holes and marked errors [Omar et al. 2019], allow our implementation of Hazel to achieve *total liveness*: every editor state is syntactically, statically, and dynamically meaningful, without gaps.

Type error localization is a well-studied problem in practice. This paper is the first to formally support the intuition that, in the words of [Dunfield and Krishnaswami 2019], "bidirectional typing improves error locality". Although there has been considerable folklore around error localization for systems with local type inference, the problem has received little formal attention. We hope that this paper, with its rigorous formulation of type error localization and recovery for bidirectionally typed languages, will provide more rigorous grounding to language server development, much as bidirectional typing has done for type checker development.

For systems rooted in constraint solving, there has been considerable work in improving error localization because it is notoriously well-understood that such systems make error localization difficult, and programmers are often confused by localization decisions [Wand 1986] because they are rooted in *ad hoc* traversal orders [Lee and Yi 1998; McAdam 1998]. More recently, there has been a series of papers on finding the most likely location for an error based on either manual weights [Pavlinovic et al. 2014; Zhang and Myers 2014] or learned weights [Seidel et al. 2017]. While improving the situation somewhat, these remain fundamentally *ad hoc* in their need to guess intent. However, data-driven approaches could perhaps be layered atop type hole inference to improve the ranking or filtering of suggestions.

A more neutral alternative is to derive a set of terms that contribute to an error, an approach known as type error slicing [Haack and Wells 2003; Schilling 2011; Tip and Dinesh 2001]. This creates a large amount of information for the programmer to consume. Our approach is to instead simply report the constraint inconsistencies on a hole in the program and allow for the programmer to interactively refine their intent, so only the local inference system is responsible for identifying particular erroneous expressions. We do not make particular usability claims about the interactive affordances related to type hole inference in this paper, but rather simply claim a novel neutral point in the overall design space that uniquely combines local and global approaches.

Recent work on gradual liquid type inference described an exploratory interface for filling holes in refinement types by selecting from partial solutions to conflicting refinement type constraints [Vazou et al. 2018]. This is similar in spirit to type hole inference as described in this paper, albeit targeting program verification predicates.

The underlying unification algorithm is essentially standard–the novelty is in how the unification results are used and how failures are handled, rather than in the inference itself. In particular, we base our approach on the system described by [Siek and Vachharajani 2008], because it also identifies type inference variables with the unknown type from gradual typing and the union find data structure is useful for computing possible type sets. Garcia and Cimini [2015] similarly present a static implicitly typed language, where users opt into dynamism by annotating an expression with the gradual type "?", and an associated type inference algorithm and accompanying metatheory. By contrast, the Hazelnut type system assigns gradual types to programs that would ordinarily not type-check in a non-gradual system by wrapping them in expression holes. The type inference algorithm presented in Garcia and Cimini [2015] also does not specify what to do if the constraint

set cannot be solved. If a single static type cannot be determined for an expression, its type is simply undefined, whereas our approach provides a list of suggestions derived from any conflicting constraints if a single substitution cannot be determined. As our focus is on failure cases and partially consistent suggestions, the metatheory in this prior work is less relevant in guiding the design of the type hole inference system.

Of note, however, is that type inference approaches that eagerly solve and substitute for type inference variables [McAdam 1998; Odersky et al. 1999; Pottier 2014] are not well-suited to the type hole inference approach, as they lose information necessary for computing partially consistent suggestions.

Our focus in this paper has not been on the error messages displayed on screen, about which we make no specific claims. There has been considerable work on improving explanations for type errors, e.g. by providing sample inputs (dynamic witnesses) that elicit runtime errors. With this approach, one can generate graphs for visualizing the execution of witnesses and heuristically identify the source of errors with around 70% accuracy [Seidel et al. 2016]. These and a variety of other approaches to explaining errors are a fruitful avenue for future integration into the system.

6 CONCLUSION

Nothing will ever be attempted if all possible objections must first be overcome.

- Samuel Johnson

Programming is increasingly a live collaboration between human programmers and sophisticated semantic services. These services need to be able to reason throughout the programming process, not just when the program is formally complete. This paper lays down rigorous type-theoretic foundations for doing just that. Local type inference helps make the localization decisions we make systematic and predictable, and type hole inference shows how local and constraint-based type inference might operate hand-in-hand, rather than as alternatives. We hope that language designers will use the techniques introduced in this paper to consider more rigorously, perhaps even formally, the problems of type error localization and error recovery when designing future languages.

REFERENCES

Djonathan Barros, Sven Peldszus, Wesley KG Assunção, and Thorsten Berger. 2022. Editing support for software languages: implementation practices in language server protocols. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. 232–243.

Andrew Blinn, David Moon, Eric Griffis, and Cyrus Omar. 2022. An Integrative Human-Centered Architecture for Interactive Programming Assistants. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

Edwin C. Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Yair Chuchem and Eyal Lotem. 2019. Steady Typing. *LIVE 2019* (2019).

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 443–455. https://doi.org/10.1145/2837614.2837632

Evan Czaplicki. 2018. An Introduction to Elm. (2018). https://guide.elm-lang.org/. Retrieved Apr. 7, 2018..

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

Jana Dunfield and Neelakantan R Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. *ACM SIGPLAN Notices* 48, 9 (2013), 429–442.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Christian Haack and Joe B. Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2618)*, Pierpaolo Degano (Ed.). Springer, 284–301. https://doi.org/10.1007/3-540-36575-3_20

HaskellWiki. 2014. GHC/Typed holes — HaskellWiki,. https://wiki.haskell.org/index.php?title=GHC/Typed_holes&oldid=58717 [Online; accessed 2-March-2023].

Hazel Development Team. 2023. Hazel. http://hazel.org/. http://hazel.org/

G. Huet. 1976. *Resolution d'Equations dans les langages d'ordre 1, 2, ..., omega.* Ph. D. Dissertation. Université de Paris VII.

Stef Joosten, Klaas van den Berg, and Gerrit van Der Hoeven. 1993. Teaching Functional Programming to First-Year Students. *J. Funct. Program.* 3, 1 (1993), 49–65. https://doi.org/10.1017/S0956796800000599

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a Folklore Let-Polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (1998), 707–723. https://doi.org/10.1145/291891.291892

Meven Lennon-Bertrand. 2022. *Bidirectional Typing for the Calculus of Inductive Constructions. (Typage Bidirectionnel pour le Calcul des Constructions Inductives).* Ph. D. Dissertation. University of Nantes, France. https://tel.archives-ouvertes.fr/tel-03848595

Bruce J McAdam. 1998. On the unification of substitutions in type inference. In *Symposium on Implementation and Application of Functional Languages*. Springer, 137–152.

David Moon, Andrew Blinn, and Cyrus Omar. 2022. tylr: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*. 28–37.

David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. In *To appear, IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2023*.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55.

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. 2017a. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues , Jonathan Aldrich, and Matthew A. Hammer. 2017b. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL)*.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 525–542. https://doi.org/10.1145/2660193.2660230

Benjamin Pierce and David Turner. 2000a. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

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

Benjamin C. Pierce. 2002b. *Types and programming languages*. MIT Press.

Benjamin C. Pierce and David N. Turner. 2000b. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.

Hannah Potter, Ardi Madadi, René Just, and Cyrus Omar. 2022. Contextualized Programming Language Documentation. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 1–15.

Hannah Potter and Cyrus Omar. 2020. Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies. *Human Aspects of Types and Reasoning Assistants (HATRA)* (2020).

François Pottier. 2014. Hindley-milner elaboration in applicative style: functional pearl. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 203–212. https://doi.org/10.1145/2628136.2628145

Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic Syntax Error Reporting and Recovery in Parsing Expression Grammars. *Sci. Comput. Program.* 187, C (feb 2020), 22 pages. https://doi.org/10.1016/j.scico.2019.102373

Thomas Schilling. 2011. Constraint-Free Type Error Slicing. In *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7193)*, Ricardo Peña and Rex L. Page (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually G o Wrong). In *International Conference on Functional Programming (ICFP)*.

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 60:1–60:27. https://doi.org/10.1145/3138818

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-Based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/1408681.1408688

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15 (2013), 475–495.

Arthur Sorkin and Peter Donovan. 2011. LR(1) Parser Generation System: LR(1) Error Recovery, Oracles, and Generic Tokens. *SIGSOFT Softw. Eng. Notes* 36, 2 (mar 2011), 1–5. https://doi.org/10.1145/1943371.1943391

Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM* 24, 9 (1981), 563–573.

Frank Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (2001), 5–55. https://doi.org/10.1145/366378.366379

Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual liquid type inference. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 132:1–132:25. https://doi.org/10.1145/3276502

Jérôme Vouillon and Vincent Balat. 2014. From Bytecode to JavaScript: The Js_of_ocaml Compiler. *Softw. Pract. Exper.* 44, 8 (aug 2014), 951–972. https://doi.org/10.1002/spe.2187

Mitchell Wand. 1986. Finding the Source of Type Errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 38–43. https://doi.org/10.1145/512644.512648

Danfeng Zhang and Andrew C. Myers. 2014. Toward general diagnosis of static errors. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 569–582. https://doi.org/10.1145/2535838.2535870